RoboRebound: Multi-Robot System Defense with Bounded-Time Interaction

Neeraj Gandhi
ngandhi3@seas.upenn.edu
University of Pennsylvania
Philadelphia, Pennsylvania, USA

Yifan Cai
caiyifan@seas.upenn.edu
University of Pennsylvania
Philadelphia, Pennsylvania, USA

Andreas Haeberlen
ahae@seas.upenn.edu
University of Pennsylvania
Philadelphia, Pennsylvania, USA

Linh Thi Xuan Phan
linhphan@seas.upenn.edu
University of Pennsylvania
Philadelphia, Pennsylvania, USA

Abstract

Byzantine Fault Tolerance (BFT) is a classic technique for defending distributed systems against a wide range of faults and attacks. However, existing solutions are designed for systems where nodes can interact only by exchanging messages. They are not directly applicable to systems where nodes have sensors and actuators and can also interact in the physical world – perhaps by blocking each other's path or by crashing into each other.

In this paper, we take a first stab at extending BFT to this larger class of systems. We focus on *multi-robot systems (MRS)*, an emerging technology that is increasingly being deployed for applications such as target tracking, warehouse logistics, and exploration. An MRS can consist of dozens of interacting robots and is thus a bona-fide distributed system. The classic masking guarantee is not practical in a MRS, but we propose a variant called *bounded-time interaction* that can be implemented, and we present an algorithm that achieves it, in combination with a few small hardware tweaks. We built a simulator and prototyped wheeled robots to show that our algorithm is effective, and that it has a reasonable overhead.

CCS Concepts: • **Computer systems organization → Robotics**; **Dependable and fault-tolerant systems and networks**; • **Security and privacy → Embedded systems security**.

Keywords: Security, Robotics, Cyber-physical systems

ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3696079

ACM Reference Format:
Neeraj Gandhi, Yifan Cai, Andreas Haeberlen, and Linh Thi Xuan Phan. 2025. RoboRebound: Multi-Robot System Defense with Bounded-Time Interaction . In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689031.3696079

1 Introduction

A multi-robot system (MRS) is a group of robots that work towards a common goal, such as performing efficient warehouse logistics [16, 43, 45, 53], patrolling a certain geographic area [1, 2, 71, 87], or finding intruders [36, 37, 59]. Robots typically move independently and coordinate over wireless messages. MRS are increasingly deployed commercially; for instance, Ocado is using them in large warehouses, where they have reduced the worst-case order fulfillment time from over three hours to less than 15 minutes [74].

As with other distributed systems, a possible concern is that individual robots could malfunction or be compromised by an adversary. From a distributed systems perspective, it is natural to reach for an existing tool from the fault-tolerance toolbox – say, replicating each robot's controller using Byzantine Fault Tolerance (BFT). However, BFT does not seem like a great fit for MRS, for at least two reasons. First, it is not clear where the replicas should go: they cannot be on the robot itself, since an adversary might gain physical access to some robots and compromise all of their replicas simultaneously, but they also cannot be on other robots because most MRS use limited-range wireless transmitters, so the network topology keeps changing at runtime and does not support the stringent latency requirements of a typical robot control loop. If a robot were to lose contact with its "replicas" even briefly, it could easily crash even without an adversary in the picture.

A second, more fundamental reason is that the robots can directly interact with the physical world, so incorrect behavior is not the only concern: an adversary can use a compromised robot to cause considerable destruction. For instance, Ocado's robots resemble "washing machines on wheels" [74] and can move at $8\frac{m}{s}$ [93], so one could imagine a compromised robot, e.g., knocking over shelves, smashing products that contain dangerous chemicals, or crashing into other robots. This risk does not exist in most classical application scenarios for, say, BFT: although the replicated system might certainly control security-critical actuators (say, in a chemical plant), the replicas *themselves* are compute nodes and cannot cause physical destruction on their own.

Although the robotics community has started to explore MRS-specific security solutions, most of the existing work has been done in isolation, without much input from the distributed-systems community. Just like the early solutions in our area, current MRS defenses tend to focus on specific, known attacks: for instance, [32, 33, 57, 104] only consider Sybil attacks and certain types of lying, while [103] focuses on physical masquerade attacks. There are some consensus protocols [81–83, 91, 113, 114], but they focus on lying about the *inputs* and not on misbehavior during the protocol itself. We think that it would be useful – and an interesting challenge! – to translate some of the general solutions from the distributed-systems literature to this new setting.

This paper is our first step in this direction: we present RoboRebound, which is, to our knowledge, the first general-purpose MRS defense for the fully-Byzantine threat model. RoboRebound is based on two key insights. The first is that it is critical to define the right goal: although the traditional fault-tolerance guarantee – fully masking the effects of a limited number of faults – would be hard to achieve in MRS, it also does not seem strictly necessary: since the robots have limited speed, they cannot cause destruction instantly. Thus, it would already be useful to catch malfunctions reasonably quickly. For instance, if a robot veers off the safe track towards a shelf or another robot, we can typically prevent damage as long as the robot can be shut down right away. We call this property *bounded-time interaction (BTI).*

At first glance, it seems possible to provide BTI with a solution such as bounded-time recovery (BTR) [28], which can detect and isolate faulty nodes within milliseconds. However, BTR assumes that a) it is possible to tell what a node's inputs were, and that b) success means getting the other nodes to stop paying attention to the faulty node. Neither is true in an MRS: no robot can tell what another robot's sensors are showing, and a robot can cause damage simply by controlling its *own* actuators. However, our second key insight is that two very small and simple pieces of trusted hardware can solve both problems. The situation is roughly analogous to TrInc [50], which similarly showed a huge security benefit from an extremely simple piece of trusted hardware – in TrInc's case, an increment-only counter.

We have implemented a prototype of RoboRebound (in simulation) and on a variant of our lab's mobile-robot platform, SecBot, and we have evaluated it with a combination of simulations and microbenchmarks, using a standard flocking protocol (Olfati-Saber [68]) as a case study. Our results show that 1) RoboRebound is effective against a variety of attacks and that it has a reasonable overhead, and 2) the trusted components can be implemented with just a few lines of code, on tiny MCUs that cost about €3 a piece. In summary, our contributions are: (*i*) the bounded-time interaction property (Section 2); (*ii*) the design of RoboRebound (Section 3); (*iii*) a prototype implementation (Section 4); and (*iv*) an experimental evaluation (Section 5).

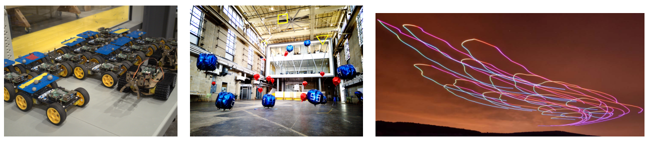


Figure 1. Examples of practical MRS: multi-tooled ground robots, cooperative blimp robots, and a long-exposure shot of a flying-robot flock. (Images by Jiawei Xu (left [107], middle [106]) and from [99]; reproduced with permission.)

2 Overview

Since MRS are relatively new to this community, we begin with a brief overview. The MRS landscape is highly heterogeneous: the robots can be wheeled ground robots, swimming robots, or flying robots, and the system might contain hundreds of them, or just a few. Figure 1 shows three examples. A common characteristic is that the robots act *individually*: each has its own controller, its own array of sensors and actuators, and its own radio. Many MRS are fully distributed; action is completely regulated by decentralized communication. Generally, robots can be programmed with different, mission-dependent protocols.

The hardware used varies widely; e.g., sensors might include photodiodes, cameras, GNSS, or gyroscopes [11, 30, 78, 97, 99]. Robots often *do* have fairly substantial computation power: for instance, even the small e-puck 2 [30] has a 32-bit MCU that runs at 60MHz, while the higher-end Starling [97] has a 64-bit CPU that runs at 2.15GHz. Available RAM ranges from a few kilobytes [30] to gigabytes [77, 97]; flash memory is more plentiful, ranging from 144kB to 128MB. MRS typically rely on a time-varying ad-hoc wireless network for communication. Data rates range from a few Kbps to tens of Mbps [44, 97]. Thus, the robots typically do have resources available for adding a security technique, including some simple cryptographic operations.

2.1 What do MRS protocols look like?

MRS use cases vary widely, so there are many different coordination protocols. For concreteness, we sketch three common example applications next.

In *Multi-Agent Path Finding (MAPF)*, each robot has its own destination; the goal is to generate paths that achieve efficient, collision-free motion. This can be used in, e.g., warehouses [43, 47, 105], service robots [100], and airport surface

Algorithm 1 Flocking Protocol for a Robot i, from [68].

```
Let $g$ be an 'agent' representing the global rendezvous point
for each control algorithm period do
    $\mathbf{u}_i \leftarrow \mathbf{0}$        /* Initialize control (acceleration) vector */
    /* Factor in attraction/repulsion from nearby robots */
    for each neighbor $j$ of $i$ do
        $\mathbf{u}_i \leftarrow \mathbf{u}_i +$ NbrSpring($i$, $j$) + NbrDamp($i$, $j$)
    /* Account for the closest nearby (convex) obstacles */
    for each nearest-point on a convex obstacle $k$ do
        $\mathbf{u}_i \leftarrow \mathbf{u}_i +$ ObsSpring($i$, $k$) + ObsDamp($i$, $k$)
    /* Account for the global 'goal' or destination */
    $\mathbf{u}_i \leftarrow \mathbf{u}_i +$ SysGoalSpring($i$, $g$) + SysGoalDamp($i$, $g$)
    applyNewAcceleration($\mathbf{u}_i$)
```

operations [66]. As with flocking, the robots exchange information about their locations and current destinations.

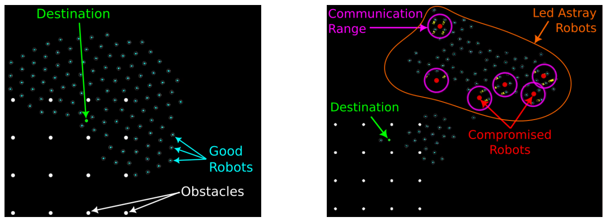
In *exploration*, the robots need to explore an unknown area – perhaps to map an unknown building, to locate a radiation source, or to find people who may need to be rescued – and they split up the region to cover it more quickly as a group. They coordinate infrequently to ensure that their subregions do not overlap, and that no area is missed.

A *flocking protocol*, such as [68], enables the robots to move as a group, towards a common destination. This can be useful, e.g., for target tracking [57]. Typically, the robots exchange information about their current positions and about obstacles to avoid collisions. To make things concrete, Algorithm 1 shows simplified pseudocode for [68]: it resembles a spring-damper mechanism, in which a robot is attracted or repelled by the destination, obstacles, and nearby robots. Notice that each robot's acceleration vector depends on the state of the neighbors, which robots exchange over a wireless network. We will use this algorithm as a case study, but our solution is general and covers other MRS protocols as well.

2.2 Threat model

We focus on an adversary that can compromise up to $f_{\max}$ robots, e.g., because they briefly have physical access to the area where robots are stored, or because they can capture a few robots during the mission. The adversary can reprogram these nodes, but we assume that they cannot carry out attacks that would take a lot of time or specialized equipment, such as altering the hardware or applying sophisticated side-channel attacks (differential power analysis etc.). Thus, *components with fixed functionality (e.g., an MCU with a program in ROM) will operate correctly after the attack*, and their secrets will remain secret, as long as they do not contain bugs the adversary can exploit. Notice, however, that this cannot include the main controller on each robot because, as we have pointed out in Section 2, this needs to be programmable to accommodate different missions.

We do not aim for perfect security: one could certainly imagine stronger adversaries who are able to tamper with the trusted nodes, given enough time and the right equipment. For instance, an adversary with a soldering iron and some



(a) No robots compromised **(b)** 10/125 robots compromised

Figure 2. Effects of a small-scale attack on an MRS that performs flocking using the Olfati-Saber protocol [68].

extra components could modify or replace a robot's logic board, and an adversary with acid and an electron microscope could decap chips and read out their secrets. However, these invasive attacks seem considerably more difficult than the ones we are considering.

2.3 What can go wrong?

How might an adversary attack MRS fitting this model? We observe that MRS protocols have a few common features: 1) each robot reports some local information, such as its current position or the presence of an obstacle; 2) this information is often hard to verify by other robots; and 3) the group makes decisions based on the aggregate information. Here, compromised robots could "poison" the overall state, and to affect the entire MRS (beyond the compromised robots), by reporting false information and/or by equivocating.

This vulnerability is not unique to flocking – it also exists in MAPF and exploration, as well as in other MRS applications. It is also not unique to a specific protocol. Rather, it is fundamental to MRS protocols, which must control the actions of the entire group based on information provided by individual robots. We surveyed 34 protocols from the MRS literature, and found variants of this vulnerability in every single protocol we examined. In transportation systems [12, 15, 48, 66, 72], compromised robots can lead good robots astray, damaging cargo and increasing operational cost. In warehouses [16, 24, 43, 45, 47, 52, 53, 85, 95, 105, 109], compromised robots can delay getting objects to destinations, block other robots' paths, or put objects in incorrect places. In target tracking [35, 98, 115–117], compromised robots can mislead robots into looking elsewhere while the target escapes. In perimeter defense [1, 2, 36, 37, 59, 71, 86–88], compromised robots can open a breach in the perimeter by lying about incoming adversary positions. In surveillance and mapping [3, 34, 75, 112], compromised robots can lie about the objects of surveillance to misinform users or cause resources to be allocated incorrectly.

2.4 How bad can it get?

To illustrate the potential impact on MRS performance, we show the effects of an attack in Figure 2. We simulated an MRS with 125 robots running our example "application" (flocking, using Olfati-Saber [68]). Robots were to move to a destination located amidst a grid of obstacles. Figure 2a shows a snapshot of a no-attack case; as expected, all robots move around the obstacles as a group to reach the target.

This contrasts with Figure 2b; here, 10 compromised robots masquerade as other robots and incorrectly report the positions of the other robots as between a correct robot and the destination. Correct robots stay away from the destination to avoid a crashing into the positions of robots forged by the adversary. Since all robots want to avoid crashes, even correct robots not communicating with compromised robots are affected by the misinformed actions of those that the compromised robots can communicate with.

2.5 Challenges

In traditional distributed systems, a problem like this would usually be solved by replicating the relevant components, e.g., using PBFT [13]. However, as discussed earlier, this approach is not a good fit for MRS: if the replicas are on the same robot, an adversary could likely compromise them all, and if they are on other robots, it would be difficult to maintain the tight latency requirements of the robots' control loops. Thus, fault tolerance is not likely to be a good solution.

Post-hoc fault detection seems more promising: we can let each robot make latency-sensitive decisions independently, but then 1) have other robots check that these decisions were correct, e.g., by auditing [42], and 2) isolate any robots that have made incorrect decisions. This approach works for classical distributed systems, where the inputs and outputs are messages from and to other nodes, but it seems difficult for MRS, where each robot interacts with its local sensors and actuators. A compromised robot could evade detection by reporting inputs that would be consistent with its intended actions (say, claiming a strong wind from the right if it wants to move to the left), and/or by reporting to auditors that it is taking the actions the protocol prescribes, while actually doing something else. And even if one robot could detect that another was compromised, it is not clear how the latter could be isolated from the rest of the system.

However, we observe that fault detection *mostly* works, even for MRS: at a high level, all that is missing is a way to verify some simple statements about each robot's inputs and outputs. (As we shall see in a moment, it is a bit more subtle than this, but not much more!) It is also general and widely applicable [42]. Thus, it seems possible to get a big security gain if we are willing to trust a very simple primitive – much simpler than the entire auditing system, and certainly much simpler than the entire MRS functionality!

2.6 Our approach

In this paper, we propose just such a primitive, along with a system, RoboRebound, that leverages it. Specifically, we add a pair of small, trusted hardware components called the s-node and a-node to each robot, which interpose on sensor and actuator communication, respectively – hence the names. Unlike the main control algorithm on the control node (c-node), which is complex and mutable, the s- and a-node are simple (a few lines of pseudocode) and immutable; their software is meant to be burned into ROM on cheap MCUs. Thus, under our threat model, they would survive an attack, even if the c-node is compromised.

Once the s-node and a-node certify the sensor data and control vector, robots can perform PeerReview [42]-style auditing. Each robot logs its inputs and outputs to enable deterministic replay [19]; other robots download and inspect this log regularly to check for correctness. If the replay outputs match the log-recorded outputs [42], then the robot is correct; otherwise, it is compromised.

Since a compromised robot may try to evade detection by ignoring audit requests or by moving out of communication range, we enforce a timeout mechanism in the a-node: each c-node must periodically obtain *tokens*, representing successful audits, from $f_{\max} + 1$ other c-nodes, and present them to its a-node. If the a-node does not receive enough tokens, it can put the robot into safe mode by, e.g., triggering a kill switch.

The three key challenges with this approach are 1) keeping the complexity of the s- and a-node low enough to make them trustworthy, 2) detecting whether a robot has logged its actions correctly, and 3) deciding whether a given log represents correct behavior. Section 3 describes how RoboRebound overcomes these challenges.

2.7 Bounded-time interaction

The security property provided by this approach is clearly weaker than masking. A compromised robot has a small (but bounded) window of opportunity to misbehave and/or send incorrect information, until its a-node times out and disables it. We call this *bounded-time interaction (BTI).* Whether BTI is strong enough depends on the use case. In Olfati-Saber, it seems sufficient: even if the spring-damper mechanism is briefly distorted by disinformation from compromised robots, it will quickly return to its expected state once these robots have been disabled. In other use cases, BTI's window of opportunity could be enough for a compromised robot to cause lasting damage – e.g., by crashing into a good robot. But even in such cases, a few extra precautions could help, such as keeping some distance between the robots.

3 Design

In this section, we propose RoboRebound and explain how it addresses the challenges from above.

3.1 Additional assumptions

We assume the existence of a cryptographic hash function H and a message authentication code MAC with compact, efficient, trustworthy embedded system implementations. We *do not* require public-key cryptography. We assume that

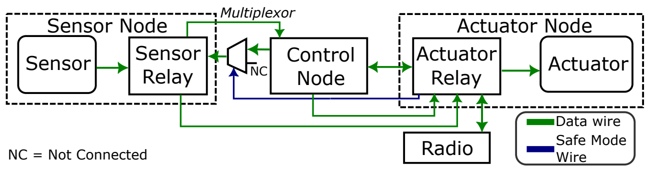


Figure 3. Wiring diagram in a single robot.

the robot has a *Safe Mode* that can be triggered when its controller is compromised or malfunctions. This is a standard feature in many robots and can range from using a simpler controller [102] to a kill switch [29] that disconnects the robot from its actuators. Having a Safe Mode is necessary for any system that can physically affect its surroundings to safeguard against faulty behavior, even in the non-adversarial case [94]. In fact, in some jurisdictions, safety-critical AI and robotic systems are even required by regulation to have a Safe Mode – see, e.g., Article 14(4)(e) of the EU's AI Act of 2024 [21].

3.2 *s*-nodes and *a*-nodes

As per Section 2.6, RoboRebound requires two small trusted components on each robot. Figure 3 shows how these components are connected. The *control node (c-node)* of a robot is a CPU that runs the control algorithm and any other software the robot would need. Instead of sensor data going directly to the *c*-node, here it goes to the *sensor node (s-node)* instead, which forwards the data to the *c*-node along with some extra information (see below). Similarly, instead of the *c*-node directly controlling actuators, commands are relayed through the *actuator node (a-node).* The *a*-node interposes on all communication with the robot's radio; it can see, and potentially drop, all messages the *c*-node sends or receives. The electronics can be wired via a printed circuit board, making it difficult for an adversary to bypass the trusted nodes, as that would require (de)soldering wires, which is time-consuming, noticeable, and requires bulky equipment. Due to the limited API available to the *c*-node and the lack of a wire from the *c*-node to reset pins on either trusted node, an adversary cannot restart the trusted nodes to reset their clocks.

The general *c*-node software can be quite complex, so it is hard to trust; but the *s*- and *a*-node run very simple protocols, which we assume can be trusted. To ensure that they cannot be affected by *c*-node bugs, they run on small, separate processors; e.g., a PIC or an MSP430. The only special features we need are 1) a local timer, and 2) a way to prevent reprogramming. The first is standard on most MCUs, and the second can be done, e.g., on the MSP430, by the "JTAG fuse" [96, §2.2.4.5].

The *s*- and *a*-node each implement a small set of functions, which the *c*-node can invoke via an RPC-like message. Pseudocode for these functions is in Algorithms 2, 3, and 4. We will explain each function below. Here, we note that the main source of complexity is a cryptographic hash and the MAC.

Algorithm 2 Functions shared by *s*-nodes and *a*-nodes.

```
flash var robId = 0                          // Unique ID for each robot
flash var masKey = 0, keySeq = 0             // Master key (set only once)
var key = 0                                  // Mission key
var buffer = [], batchCtr = 0                // Buffer for batching commitments
var tHash = 0                                // Hash at the top of the hash chain
// Private functions (cannot be invoked by the c-node)
function FLUSHBUFFER
    tHash ⟵ H(tHash || buffer)
    (batchCtr,buffer) ⟵ (0, [])
function APPENDTOCHAIN($m$)
    buffer.append($m$)
    if (++batchCtr) = batchSize then FLUSHBUFFER()
// Functions the c-node can invoke
function LOADMASTERKEY($m$, $id$)
    if masKey = 0 then (masKey, robId) ⟵ ($m$, $id$)
function LOADMISSIONKEY($k$, $r$, $s$, $h$)
    if MAC(MKEY || k || r || s || masKey) = h and s > keySeq then
        (keySeq, key) ⟵ (s, $k \oplus$ H(r || masKey))
function MAKEAUTHENTICATOR
    if (batchCtr > 0) then FLUSHBUFFER()
    return (tHash, robId, MAC(AUTH || tHash || robId || key))
```

Algorithm 3 *s*-node-specific functions.

```
function POLLSENSORS
    if key = 0 then return
    s ⟵ SENSORINPUT()
    return APPENDTOCHAIN(INPUT || s.len || s)
function CHECKAUTHENTICATOR(h, id, mac)
    return MAC(AUTH || h || id || key) = mac
```

Since the *a*-node is between the *c*-node and the actuators, we assume that Safe Mode is implemented here.

3.3 Key distribution

Each *s*-node and *a*-node has a one-time programmable (with LOADMASTERKEY) *master key* that is shared by all the robots in a given MRS. The MRS owner should keep this key in a safe place, just like a private signing key. The *a*-nodes could technically use the master key to authenticate each other's messages, but then a compromised *c*-node might be able to capture and replay messages at the adversary's convenience; e.g., future missions. To prevent this, each *s*-node and *a*-node has a separate, in-RAM *mission key* that is zero at power-up, and can be set once per mission via LOADMISSIONKEY. This requires a MAC with the master key, so that the adversary cannot set the mission key arbitrarily if she has compromised the *c*-node; a monotonically increasing sequence number *s*, so the adversary cannot use an old mission key intercepted at a previous power-up; and the message type bit MKEY. The mission key *k* is blinded with a hash of the master key and a random number *r*. The *a*-node only starts forwarding actuator commands once it has received a mission key, so, if a compromised *c*-node withholds the mission key, the robot will remain disabled and can be spotted easily.

Algorithm 4 *a*-node-specific functions.

```
var tkMap = ⊥                                    /* Active tokens from auditors */
var lastBktUpdate = 0, bktLvl = 0                /* For audit rate-limiting */
function checkTokens                             /* Runs periodically */
    nVal=|{$n$ ∈ tkMap.keys | tkMap.get(n) ≥ localTimer() − $T_{val}$}|
    if nVal < $f_{\max}$ + 1 then key ← 0; invokeSafeMode()
function recvWireless($m$)                       /* Triggered on packet reception */
    if key=0 then return
    forwardToCnode($m$)
    if (m.type ≠ audit then appendToChain(recv || $m$.len || $m$)
/* Below: Functions the c-node can invoke */
function sendWireless($m$)
    if key=0 then return
    forwardToNIC($m$)
    if (m.type ≠ audit) then appendToChain(send || $m$.len || $m$)
function actuatorCmd($m$)
    if key=0 then return
    forwardToActuator($m$)
    appendToChain(acmd || $m$.len || $m$)
function makeTokenRequest(dest)
    t ← localTimer()
    bktLvl ← min(bucketCapacity, bktLvl+$\rho$·(t-lastBktUpdate))
    bucketLastUpdated ← t
    if bktLvl < minPerToken then return ⊥
    bktLvl ← bktLvl - minPerToken
    return t, mac(treq || t || robId || dest || key)
/* Shorthands: 'tor' = auditor and 'tee' = auditee */
function issueToken(tee, $t$, $mac$, $h_{ckpt}$)
    if tee ≠ robId and mac(treq||$t$||tee||robId||key) = $mac$ then
        return mac(token || robId || tee || $t$ || $h_{ckpt}$ || key)
    return ⊥
function isTokenValid(tor, $t$, $h_{ckpt}$, mac)
    return mac(token || tor || robId || $t$ || $h_{ckpt}$ || key) = mac
function installToken(tor, $t$, $h_{ckpt}$, mac)
    if isTokenValid(tor, $t$, $h_{ckpt}$, mac) then tkMap.put(tor, $t$)
```

Thus, at the beginning of a mission, the s-nodes and a-nodes have a freshly initialized state, including a shared mission key. Since we have assumed that the adversary cannot use differential power analysis or similar hardware attacks, the adversary does not know this key.

3.4 Logging and authenticators

At runtime, each c-node keeps a local log of its nondeterministic inputs and outputs, such as sensor/actuator data and wireless messages it sent or received. During an audit (Section 3.7), this log is made available to other robots, which can verify it by deterministic replay.

A compromised c-node may omit, modify, or fabricate inputs and outputs. To make this detectable, the s-node and the a-node each maintain a *hash chain* of the inputs (e.g., state vectors) and outputs (e.g., motor commands) they forward. The hash chain starts at $h_0 := 0$ after power-up, and, when some inputs or outputs d_i are added as the i^{th} entry, the new top-level hash value becomes $h_i := H(h_{i-1} \,||\, d_i)$. Upon the c-node's request, they produce an *authenticator* for the top-level hash value; an authenticator α_i is a tuple $\alpha_i := (h_i, id, mac)$, where h_i is the hash of entry i, id is the robot's ID, and mac is a MAC. Given an authenticator α_j, a sequence of entries $d_i, .., d_j$, and the hash value h_i for the first entry, a trusted node can verify the existence and position (i.e., ordering w.r.t. other messages) in the chain by recomputing the hashes.

Since the log is transmitted wirelessly to auditors, the a-node cannot naïvely log all outgoing messages; otherwise the log would grow exponentially. Thus, wireless messages have a type bit that is set only for audit-related messages. If a message has this bit set, it is not logged, but the bit is included in the message, so the recipient will not confuse the message with a regular message. Hence, a compromised c-node cannot use this feature to bypass logging.

3.5 Tokens

Each robot must periodically request nearby robots to audit its log. Each request includes a message from the robot's a-node with the current value of its local timer; the c-node can obtain this message by calling makeTokenRequest on the a-node. If the audit is successful, the auditor returns a *token* (s, d, t, h_{ckpt}, mac) to the auditee, which includes the IDs of the auditor and the auditee (s and d), the timestamp t of the auditee's a-node, and the hash h_{ckpt} of the checkpoint at the end of the log segment that has been audited. The auditee then installs the token in its a-node with installToken.

The a-node periodically checks the timestamps in the installed tokens; if there are fewer than $f_{\max} + 1$ valid tokens whose timestamps are $\leq T_{val}$ old, the a-node triggers Safe Mode (Section 3.1). This incentivizes each robot to request audits regularly: if a compromised robot fails to do so, it is shut down after time T_{val} to enforce BTI. Since each robot requires $f_{\max} + 1$ valid tokens, at least one must be from a correct robot; thus, we only need to ensure that a correct node never declares a misbehaving node's log to be correct.

To ensure BTI, tokens must be time-specific, but the time has to be that of the auditee (to avoid synchronized clocks); this is why we include explicit audit requests issued by the a-node. Each token must be specific to both the auditor and the auditee: otherwise, a compromised robot could sign its own tokens or share tokens with other compromised robots.

3.6 Checkpointing

Naïvely, a robot's log would grow steadily over time, and auditing costs proportionally grow. To avoid this, RoboRebound does incremental auditing: each audit covers only a recent log segment. An auditor needs to know the c-node's state at the beginning of a segment it is auditing, so the c-node checkpoints its current state periodically; an audit can cover any segment between two consecutive checkpoints. To ensure that the entire log is audited, RoboRebound maintains the following invariant: the c-node's log always starts either a) at boot-up time, or b) at a checkpoint c_i for which

$f_{\max}+1$ tokens are available. The c-node records a new checkpoint c_{i+1} whenever it requests audits, and, once it collects $f_{\max}+1$ tokens for c_{i+1}, it can discard log segments up to c_i. Since audits need to be performed at least every T_{val}, the amount of storage the c-node needs for the log is constant.

3.7 Auditing

Each robot periodically asks other robots to audit its log, as that is the only way it can acquire tokens to avoid being put into Safe Mode. This is done in intervals of length $T_{audit} < T_{val}$, since audits can take a bit of time to complete. When a robot requests an audit from another robot, it sends the auditor a) the log segment to be audited; b) the checkpoint c_1 at the beginning of the segment; c) the authenticators immediately before c_1 and at the end of the log segment; and d) the tokens that cover c_1. The auditor uses isTokenValid to check whether the tokens are valid, and it verifies that they all cover c_1 and are from $f_{\max}+1$ different robots. If any check fails, the auditor ignores the audit request; there is no need to take further action because no correct auditor will agree to such a request, so the requestor's existing tokens will expire and cause it to be shut down. If an auditee is concerned about not hearing auditor responses, it can request additional requests, because extra tokens cause no harm, to the extent allowed by the leaky bucket rate limiter (see below).

Next, the auditor checks the log using deterministic replay [19]. It initializes a copy of the auditee's state machine with c_1, replays the inputs from the log into the state machine, and verifies that its produced outputs match those in the log. Since the state machine of a correct robot is deterministic, the outputs should be identical; if they are not, then the auditee has misbehaved and thus must be compromised. In that case, the auditor ignores the audit request.

During replay, the auditor also tracks the auditee's s-node and a-node hash chains. It knows the start of the hash chain from the authenticators before c_1, and it can update the hash chains whenever the s-node or a-node would have done so. If the final hashes match those of the auditee-provided authenticators, then the auditor can verify *all* the inputs and outputs of the log by simply verifying the two authenticators.

The auditor invokes issueToken and returns the resulting token to the auditee c-node only if all of the above checks pass. Since a compromised auditor could return an invalid token, the auditee uses isTokenValid to verify that the token is valid. If valid, it forwards the token to its a-node; if invalid, or if no token is received, the auditee requests an audit from another nearby robot.

3.8 Refinements

As described, the s-node and a-node would have to update their hash chain for every sensor input, actuator output, or wireless message observed, which could be costly for small MCUs. Batching can reduce this cost: the s- and a-node can compute authenticators and hash-chain entries over groups of inputs and outputs. This is safe because this information is only used during audits; there is no immediate need to check whether an authenticator exists for an input or output. In Figure 2, batching is implemented in appendToChain.

We also need to limit the rate at which robots can request audits; otherwise, compromised robots can run denial-of-service attacks by requesting lots of audits and thus make it difficult for correct robots to get fresh tokens. The function makeTokenRequest addresses this by including a leaky-bucket rate limiter, which caps the request rate at ρ, but still lets the robots request audits in batches.

3.9 Limitations

RoboRebound cannot directly detect when a robot is compromised; it can only detect when a compromised robot *misbehaves*, that is, its sequence of actuator commands and transmitted messages differs from the one a correct robot would have produced. This a fundamental limitation [41]. However, another way of saying this is that a compromised robot can only avoid detection by behaving as a correct robot would, so the adversary still gets only a limited time window in which she can change the robot's behavior, as per BTI.

Since deterministic replay is at the heart of the technique, RoboRebound will not work for nondeterministic control systems. However, deterministic systems are not uncommon [14] and deterministic control algorithms are widely deployed [49, 54, 55, 61, 68, 78, 80, 99]; RoboRebound can provide meaningful security to these systems.

To some extent, our technique relies on having enough correct robots nearby. Colluding adversarial robots could potentially surround good robots one-by-one to prevent them from acquiring enough tokens. However, typically, robots are not spaced so far apart that simply being surrounded by some robots would prevent communication with robots that are further afield. For example, Vásárhelyi et al. [99] reported near-perfect inter-robot wireless transmissions in an outdoor setting at distances within 20m.

Tiny trusted MCUs like PICs would have trouble interposing on data flows from data-rich sensors like LIDAR. This does not prevent the use of BTI, but the trusted nodes may need to be more complex devices, which in turn may require us to formally verify the trusted nodes.

Our solution involves adding special hardware for the s- and a-node, which is much easier to do for newly designed robots than for existing ones. On some robots it may be possible to use existing trusted-hardware features, such as TEEs, but these are much more complex than our two simple devices, so this would mean a larger attack surface [22, 67, 73] and possibly additional implementation challenges, depending on how Safe Mode is implemented. For us, a key goal was to identify a minimal hardware primitive that is sufficient for BTI, and easy to trust.

3.10 Security argument

We observe that the adversary cannot learn the mission key or the master key. We assumed the functions in the *s*-node and *a*-node to be trustworthy, and none of them leak the key – they return only MACs. We also assumed in Section 2.2 that the adversary cannot extract the keys through other means; e.g., differential power analysis or the JTAG interface (see Section 3.2 for the assumed countermeasure). Although the mission key is loaded at a time when the robot could already be compromised, the key is blinded in loadMissionKey, and the adversary cannot unblind it without knowing the master key. Thus, since token requests, tokens, and authenticators all contain MACs that cover all of the fields, as well as a constant type identifier and the mission key, the adversary cannot forge any of them.

Since authenticators cannot be forged, auditors know the top of the auditee's hash chains at the time of the audit. And because H is a cryptographic hash function, the auditor can tell if the auditee has omitted, forged, or tampered with any sensor input, actuator output, or incoming/outgoing wireless message: all of these are recorded in the hash chain, so the adversary cannot make changes without computing a second preimage. The auditor can also tell if the provided checkpoint is not authentic; this has to be either the starting state at power-up or a checkpoint whose hash is contained in $f_{\max} + 1$ tokens. But the latter must contain at least one token issued by a correct node, and a correct node would only issue such a token once replay succeeds.

If an auditor has a log segment and valid checkpoint for the start of that segment, deterministic replay allows it to determine if the auditee produced the right outputs given its inputs. A correct auditor issues a token only after this check succeeds, so compromised robots cannot get tokens from correct robots once they "misbehave" by doing anything deviating from their installed protocol.

Since the *a*-node only considers tokens valid for an interval T_{val}, any pre-misbehavior valid tokens would expire. But a compromised, misbehaving robot can only get up to $f_{\max}$ valid tokens (from other compromised robots it colludes with), so it cannot avoid having its Safe Mode triggered at most T_{val} time after the first misbehavior. This is the BTI property from Section 2.7 that RoboRebound is meant to provide.

4 Implementation

We built two prototypes for our experiments: an end-to-end simulation in `ns-3` [76] to run experiments with many robots under controlled conditions, and a hardware implementation in our lab's mobile-robot platform.

Simulation. Our `ns-3` simulation implements the s-, c-, and a-nodes, and the radio interface as `ns-3 Applications` that run atop a `ns-3 Node`, which corresponds to a single robot. Wireless propagation loss is modeled on the transmitter in

Figure 4. Three SecBot robots

the ESP32, used, e.g., by the e-puck 2 [20, 30], paired with a 2dBi antenna. The ESP32 and antenna combination results in signal loss of 36.05 dBm at 1m. Thus, we set the `ns-3` propagation loss model to `LogDistancePropagationLossModel` with reference distance of 1m, reference loss of 36.05, and reference exponent to three (the reference exponent used was the default value). We set the standard to `802.11ax`, the MCS value to three, the channel width to 20MHz, and the guard interval to 1600ns. We used the `ConstantRateWifiManager` with the `DataMode` set to `HeMcs3` and `ControlMode` set to `OfdmRate24Mbps`. The `MpduBufferSize` setting was set to sixty-four. The system was set to operate on the 6GHz band. The WiFi MAC high model was set to the `AdhocWifiMac`. All transmissions were done as either UDP broadcasts or unicasts. We pre-populated the ARP cache and the IPv4 routing table for all robots. Robot acceleration magnitude was capped at $5\frac{m}{s^2}$ per dimension. We wrote 9,579 lines of C++ code for the simulator.

Workload. We used our example "application" (flocking) for our experiments – specifically, the Olfati-Saber protocol. Our implementation follows [68], except for one a small modification: the controller runs every 0.25s and the state is wirelessly broadcast every 1.5s, whereas [68] set these two values as equal. This not only saved bandwidth but also led to better performance in our setting. There are many other parameters; we list them in Appendix A.

Baseline. We compare the unprotected implementation to one that is protected by RoboRebound. There is no existing solution we could fairly compare to: BFT protocols from the distributed-systems literature do not fit our threat model (Section 2.5) and would in any case perform poorly because of the intermittent network connectivity; solutions from the MRS literature focus on consensus (Section 6) and thus apply only to applications that use this primitive.

Hardware platform. We also added an *s*-node and an *a*-node to SecBot, our lab's mobile-robot platform. SecBot is a wheeled robot that is powered by a 2200mAh, 50C lithium-polymer battery. It has a variety of sensors; the one that is relevant here is a PA1010D global navigation satellite system (GNSS) module, which we use for navigation. The robots communicate using a half-duplex, 915MHz radio module (RFM69HCW), and they are controlled by a TinkerBoard S, which has a Rockchip Quad-Core ARM Cortex-A17 MPCore RK3288 CPU running at 1.8GHz, as well as 2GB of RAM. This board serves as the *c*-node in our design.

For the *s*-node and *a*-node, we added two single-core, MIPS32-based PIC32MX130F064B MCUs. At the time of writing, these cost about €3.16 each; they have 64KB of program memory and 20KB of data memory, and they run at 50MHz. We used SPI, I2C, and PWM to interface these MCUs to the rest of the system – that is, to the *c*-node and the sensors (*s*-node) and to the *c*-node, the motors, and the radio module (*a*-node). For details, please see [26] and Appendix B.

Cryptography. RoboRebound uses two cryptographic primitives: hashes and MACs. We used SHA-1 for hashing, which is a bit cheaper than SHA-2 and seems fine for the duration of a mission, which would be on the order of hours. (The 2017 collision attack on SHA-1 [90] took 6,500 CPU years and 100 GPU years.) If this is a concern, a different hash algorithm could be used. Our design does *not* require "heavyweight" primitives, such as signatures or public-key encryption. For MACs, we use LightMAC, which is optimized for resource-constrained systems. We configure LightMAC to use the recommended 80-bit keys and 64-bit tags [56].

5 Evaluation

Our experiments were designed to answer four questions: 1) Can the trusted-node portion of the protocol work in real, simple, embedded hardware?; 2) What are the costs of RoboRebound?; 3) How effective is RoboRebound at providing the bounded-time interaction property?; and 4) How does our system scale as the number of robots in the flock grows? We answer the first question with experiments on a SecBot and the last three questions with simulations in ns-3.

5.1 *s*-node and *a*-node

We begin with a look at our implementations of the *s*-node and *a*-node algorithms on the PIC.

Complexity. One key question is whether the algorithms are simple enough for the implementation to be trusted. We believe that they are: as discussed in Section 4, the two nodes need only 106 and 145 lines of C code, respectively. This amount of code is small enough to be carefully audited or perhaps even formally verified.

Hashes. Our next question is whether the PIC is fast enough to handle the necessary hashes. This is a particularly important question because robotic security papers often do not consider or consider as impractical cryptographic solutions [31–33, 81–83]. We ran the MAC and hash functions on the PIC with varying-length arguments; we report the mean value of 100 trials for each length. Figure 5a shows the results. Although we use a standard hash function (SHA-1) on an inexpensive PIC, it is still possible to hash a ten-message batch of our biggest chain entry – Olfati-Saber's 27-byte state message – in around 144 μs. If we assume that a robot

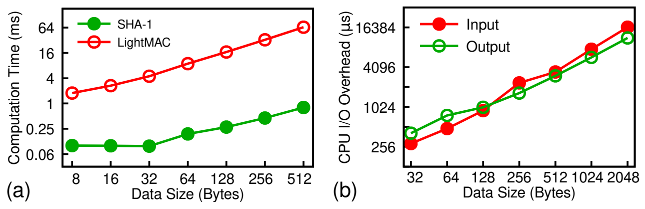


Figure 5. (a) Speed of MACs and hashing on the PIC. (b) CPU overhead for I/O, which grows linearly when data size goes beyond 512 bytes.

sends 10 of these per second, receives status messages from 10 other robots within its range, and issues 4 actuator commands per second, its *a*-node would have to hash 4 batches per second, which would consume about 0.056% of its computation power. The *s*-node's load is even lower because it only handles sensor inputs; if we assume that there are 4 such inputs per second, this would still only yield 0.4 batches and thus consume 0.0056% of the *s*-node's computation power. Further improvements are possible with hashes optimized for embedded processors, e.g., PHOTON-Beetle [8, 9].

MACs. The *s*- and *a*-node also need to compute and verify MACs. We begin with a microbenchmark: we computed MACs with arguments of different sizes, and we report the mean time for 100 runs as another line in Figure 5a.

How often might an *a*-node have to compute a MAC? MACs are used only during audits, so we have to consider only two cases: auditing, and being audited. Assume T_{audit} = 4*s*, the worst case for being audited is that for each second, the *a*-node has to make one authenticator, $2f_{\max} + 1$ token requests, and then validate all $2f_{\max} + 1$ tokens that are returned. (To be conservative, we ignore the obvious possibility of reusing the same authenticator, and making token requests in batches.) If we assume $f_{\max} = 3$, this would involve $2 \cdot (2f_{\max}+1)+1 = 15$ MACs over a maximum length of 39B, which would take about 5.36ms each, or 80.40ms total per audit period (4s), which turns into 2.01% CPU load. The load on the *s*-node would be lower, since it only makes authenticators, not token requests.

A robot also has to audit at least an equal number of other robots. During each audit, the *a*-node may need to issue a token, and the *s*-node has to check the authenticator. (The hashing needed to check the chains would happen on the *c*-node during the audit, which is presumably more powerful and not the bottleneck.) If we conservatively assume that a robot may agree to $6f_{\max}$ audit requests per token validity interval and all arrive at the same moment, the corresponding $6f_{\max}$ MACs on each of the two PICs, with $f_{\max} = 3$, would consume 96.48ms per audit period (or 2.412% CPU).

I/O Overhead. Besides computation, I/O will also add extra overhead to the CPU. The CPU needs to copy the data from or to the corresponding registers when it is available. We measured the I/O overhead by running the same program

Primitive (computation)	ms/op	ops/s	Load
makeAuthenticator	12.0	0.25	0.30%
isTokenValid	11.0	2.00	2.20%
makeTokenRequest	11.0	2.00	2.20%
sendWireless (state and token, <40B)	1.1	3.17	0.35%
sendWireless (audit, <2kB)	20.1	0.25	0.50%
recvWireless (state and token, <40B)	1.1	9.17	1.01%
recvWireless (audit, <2kB)	20.1	2.50	5.03%
actuatorCmd	1.1	4.00	0.44%
issueToken	21.0	2.50	5.25%
Total			17.28%

Table 1. Worst-case load on the *a*-node.

Primitive (computation)	ms/op	ops/s	Load
pollSensors	1.1	4.00	0.44%
makeAuthenticator	12.0	0.25	0.30%
checkAuthenticator	21.0	2.50	5.25%
Total			5.99%

Table 2. Worst-case load on the *s*-node.

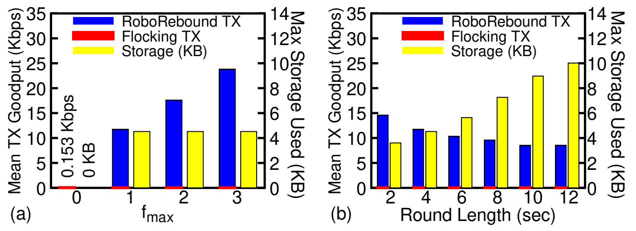


Figure 6. Bandwidth and storage required of a single robot with RoboRebound.

with and without I/O, starting from a state where the I/O buffer is empty, and calculated the time difference. Figure 5b shows the results. A small message of 32B takes the CPU 0.29ms to receive and 0.41ms to send. A medium message of 512B takes 3.47ms to receive and 3.03ms send. A large message of 2kB takes 16.46ms to receive and 11.34ms to send. For most trusted functions, only small messages are exchanged; the sendWireless and the recvWireless may involve large messages. Large messages are only transmitted when auditing happens. The message sizes are all less than 2kB, and they are transmitted at a rate no larger than 3 messages per second. Therefore, the worst-case processing time for them is smaller than 50ms, or 5% of the CPU utilization.

Worst-case overall load. We next show the worst-case overall load of trusted nodes with the previous configurations of auditing period $T_{audit} = 4s$, state exchange period $T_{state} = 1.5s$, and control period $T_{control} = 0.25s$ (this matches the ns-3 setup we use next). We set $f_{max} = 3$ and each robot to be connected to 10 peers. Tables 1 and 2 show each of the primitives, the cost of a single invocation, and a worst-case number of invocations for both trusted nodes. Based on our results, we set 1ms to be the worst-case execution time for a SHA-1 operation for 270B, and 10ms to be the worst-case execution time for a MAC operation over 40B. The worst-case values are conservative and unlikely to occur; 99.9% of the results are within ±1% of the mean values. For I/O, we assume that in the worst case, a small message takes 1ms to process, and large messages take 20ms. The CPU loads are thus 17.28% on the *a*-node and 5.99% on the *s*-node, well within the PIC's capabilities. Results with other configurations show that the utilization is approximately linear to T_{audit} and the number of other robots that one has connection with, while it is not sensitive to f_{max} or $T_{control}$.

5.2 *c*-node

Next, we evaluate the c-node costs in ns-3, which allows us to simulate multiple robots. We use two setups: *i*) a 25-robot flock with desired inter-robot space 4m, and *ii*) 16–324 robots (square arrangements with 4–18 robots per edge) arranged with desired inter-robot distance 4m–64m. Unless otherwise stated, experiments ran for 50s, created new audit requests every 4s, and set the target location to $(500m, 500m)$. Sensors are polled and a new control vector is generated every 0.25s, and wireless state broadcasts occur every 1.5s. Auditees wait 50ms after sending $f_{\max} + 1$ audit requests before requesting additional audits (e.g., in case one of their selected auditors is no longer a neighbor). Across our experiments, no robots crashed, and no correct robots were put into Safe Mode. We collected data from each robot and present the results.

Bandwidth. Figure 6 shows our results for bandwidth given setup (*i*) from above. The robots use a small, fixed amount of bandwidth for Olfati-Saber's state broadcasts and, for $f_{\max} > 0$, a larger amount of bandwidth for RoboRebound's audits, which increases with $f_{\max}$. This is because each node needs to stream its log to $f_{\max}+1$ auditors. The logs grow at a rate of about 0.8kBps (see below), but there is also some overhead for the audit responses (e.g., tokens). In relative terms, RoboRebound uses more bandwidth than Olfati-Saber, but its cost is still small in absolute terms, and well within the capacity of the wireless channel we used. The cost does *not* vary significantly with the auditing interval. This is because auditing effectively transfers each log entry to $f_{\max} + 1$ auditors. Short and long auditing intervals cause this transfer to happen in smaller or larger increments, respectively, but the overall cost is still roughly the same.

Mean transmitted goodput per node decreases as the audit period increases. This is due to less overhead for each token that is requested, and due to the fact that audit requests can be spaced apart in time better, resulting in less contention over the wireless channel.

Storage size. On the *s*- and *a*-node, the costs can easily be seen in Algorithms 2–4: since we concentrated the more expensive functionality on the c-node, both nodes only need a few bytes. Recall from Section 3.4 that a *c*-node stores a) a few recent checkpoints, and b) the log entries since that checkpoint. Since logs are periodically truncated when a new round of tokens has been collected, the storage cost is essentially that of the most recent few checkpoints and of the log segment. Consider again Figure 6, which uses

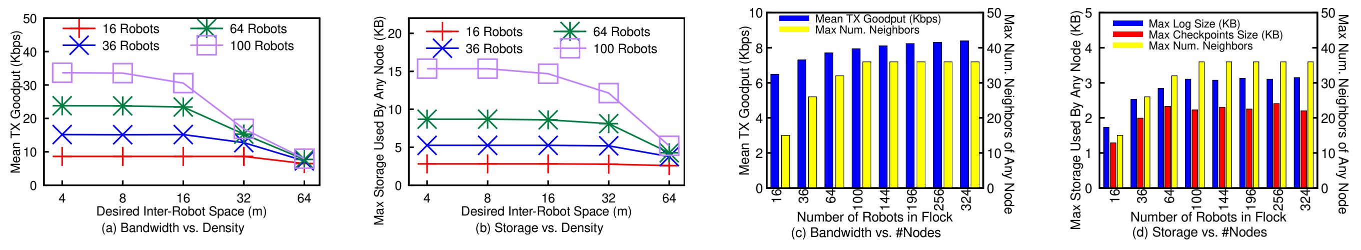


Figure 7. Scalability for variable density (a+b) and for variable number of robots at constant density (c+d).

setup (*i*) from above. Storage requirements do not grow with $f_{\max}$; the contents of checkpoints and logs are independent of the number of auditors. Log size grows linearly with the auditing interval; with the chosen Olfati-Saber parameters, it is about 0.8 kB/s. This seems seems more than reasonable; e.g., SecBot's *c*-node, the TinkerBoard S, has 2GB of RAM.

What exactly goes into these 0.8kB/s? The log contains each message that the robot has sent or received over the wireless network (excluding the audit-related messages), as well as the sensor inputs and actuator outputs. Sensor log entries take 34B and actuator log entries take 26B. Local sensing happens every 250ms, consuming 136B/s; likewise, actuator commands then consume 104B/s. Nodes broadcast state every 1.5s, consuming 566.67B/s (twenty-four received states and one sent). This means the log could grow at up to 806.67B/s. Note that this cost changes as the frequency of its three components changes.

A *c*-node also stores, at most, the latest three checkpoints. Each checkpoint contains the time at which it was generated (8B), the position and velocity of the robot (8B each), the current top hashes as tracked by the *c*-node for both the *a*- and the *s*-node (20B each), the number of neighbors (2B), and a set of neighbor information entries, which each contain the neighbor's ID (2B), the time when the last message from that neighbor was received (8B), and the neighbor's position and velocity vectors (8B each). One checkpoint could thus consume up to 690B (if all 24 robots are neighbors). A robot might store up to three checkpoints before old checkpoints get pruned down to the most recent two.

Scalability. Next, we examine how these costs change with MRS scale. There are two ways to scale an MRS: i) increase the density of the system, by packing a fixed number of robots into less space, or ii) keep the density constant and increase the number of robots. We examine both.

Figures 7(a) and (b) show how the bandwidth and storage costs change with the inter-robot distance while the number of robots is constant; each figure shows lines for 16, 36, 64, and 100 robots. At smaller distances (higher densities), the costs for the larger systems are higher than those for the smaller ones because the wireless transmission range is fixed, so higher densities cause each robot to "hear" more transmissions from peers, which it then has to store in its log and transmit to auditors later on. But at larger distances (lower densities), this effect disappears because the entire MRS no longer fits into a single transmission range, so the number of neighbors per robot levels off. In absolute terms, the bandwidth is quite low, even at the higher densities.

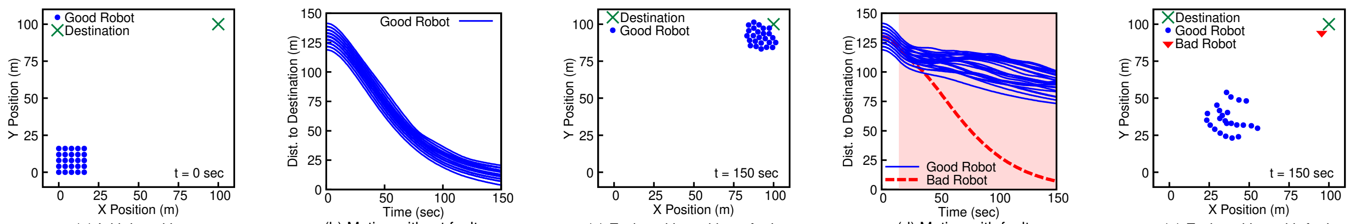
Figures 7(c) and (d) use the same metrics but keep inter-robot distance constant at 64m and vary MRS size. After an initial increase, the transmission ranges fill up and the per-robot costs remain roughly constant. This is expected because all the interactions are local between a robot and its direct neighbors. There is still a small increase due to boundary effects: robots at the edge of the flock will have fewer neighbors and thus a lower cost, but the edge grows linearly with the diameter of the flock, while the overall area grows quadratically. The fraction of robots at the edge decreases, thereby increasing the average.

5.3 Example attack

Our final experiment shows how RoboRebound acts during an example attack. It is *not* meant to prove that RoboRebound is secure: for that, we would have to try out all combinations of systems and attacks, which is impossible. However, we have already presented our security argument in Section 3.10, so this is mostly meant as an illustration.

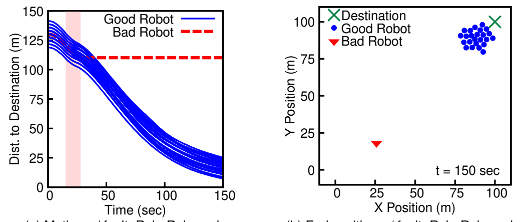
Attack method. A compromised robot spoofs robots to cause correct robots to stay away from the destination. This works because, in [68], the robots broadcast their position and speed, and each robot uses these broadcasts to compute its own direction and speed. In the absence of RoboRebound, the state broadcasts are not verifiable, so a compromised robot can easily lie. Note that correct robots *i* and *j* continue to broadcast their own states during this time, so periodically *i* gets the correct state of *j*. However, since *i* cannot differentiate real and spoofed messages, the adversary just broadcasts spoofed packets faster than correct *c*-nodes.

Let $\mathbf{x}$ be a robot's position and $\mathbf{d}$ be the destination. For each correct robot *i*, the compromised robot selects, and claims, on behalf of another robot *j* that *j* is at $\mathbf{x}_{j,spoof} = \left(\mathbf{x}_i - \frac{\mathbf{x}_i - \mathbf{d}}{||\mathbf{x}_i - \mathbf{d}||_2}\right) m$ if $||\mathbf{x}_i - \mathbf{d}||_2 \leq z$, where z is the maximum distance the adversary wants to keep between correct robots and the destination. If $||\mathbf{x}_i - \mathbf{d}||_2 > z$, then $\mathbf{x}_{j,spoof} = \left(\mathbf{d} + (z - \varepsilon)\frac{\mathbf{x}_i - \mathbf{d}}{||\mathbf{x}_i - \mathbf{d}||_2}\right) m$. Here, $z = 150$m and $\varepsilon = 2$m.



(a) Initial position (b) Motion without faults (c) End position without faults (d) Motion with fault (e) End position with fault

Figure 8. Starting position (a), execution with correct robots (b+c), and execution with one compromised robot and RoboRebound disabled (d+e). We show each robot's distance to the goal (b,d) and the final position (c,e).



(a) Motion w/ fault+RoboRebound (b) End position w/ fault+RoboRebound

Figure 9. Execution with one compromised robot and RoboRebound enabled. The compromised node is only able to act in the shaded time span.

The adversary could also make claims about the velocity of j; e.g., the adversary claims that j is moving directly away from the destination, which spurs i to move away from the destination to avoid a crash. For our attack, it was sufficient for the spoofed messages to claim that the velocity was $c = 1$ times the unit direction vector: $\mathbf{x}_{j,spoof} = \left(c\frac{1}{sec}\right)\left(\frac{\mathbf{x}_i - \mathbf{d}}{||\mathbf{x}_i - \mathbf{d}||_2} m\right)$. Since this is not a particularly sophisticated attack, the results should be interpreted as a lower bound on what a smart, determined adversary could achieve.

Setup and results. We placed 25 robots (blue dots) in a 100m×100m arena, as shown in Figure 8a. Robots use Olfati-Saber's flocking protocol to reach the destination (green 'x') while avoiding crashing. In Figure 8b, we show each robot's distance to the target (thin blue line). The robots smoothly move towards the destination, reaching it near the end of the 150s. Figure 8c shows the final state at $t = 150$s: the robots are still together and clustered near the destination.

Figures 8d and 8e show what happens without RoboRebound; a robot is compromised at $t = 15$s. The compromised robot keeps sending spoofed messages. Correct robots believe their path is blocked and remain away from the target to avoid crashing into the spoofed robots they believe to be real. The shaded region represents when the attack is active. When RoboRebound is enabled, the scenario unfolds as in Figures 9a and 9b. The compromised robot briefly disturbs the formation of the correct robots, but its audits start failing immediately, its tokens expire, and it is disabled quickly, as shown by the much smaller shaded region. The correct robots continue on their path, reaching roughly the same final state as in the baseline scenario without an adversary.

6 Related Work

As we argued in Section 2.5, solutions from the distributed systems literature are difficult to apply to MRS. Existing robot security work tends to be specific to a particular types of algorithms or attacks. We are not aware of an existing, general solution for the fully-Byzantine threat model.

Resilient consensus. There is a substantial literature on solving consensus in MRS under an adversarial model [38–40, 69, 70, 81, 82, 91, 92, 113, 114]. However, the threat model typically focuses on robots that attempt to sway the consensus outcome. Saulnier et al. [83] presents a resilient flocking algorithm that that can withstand a few compromised robots; however, compromised robots are assumed to move according to the consensus, and consensus requires $4f_{\max} + 1$ nodes, due to network topology constraints [38]. Franceschelli et al. [23] detects compromised robots by executing "motion probes", in which some correct nodes briefly deviate from the normal control algorithm, to see whether the other nodes will respond as expected. Unlike RoboRebound, all of these solutions are problem-specific and not designed to work for a more general class of algorithms.

Sybil attacks. Another line of work defends MRS against Sybil attacks [31, 33, 57, 110] by deploying a fingerprinting approach based on synthetic aperture radar. To create fingerprints, robots must spin in a circle to collect wireless signals and then construct fingerprints based on how signals have reflected and refracted. This approach is largely orthogonal to ours, and the solutions are specific to Sybil attacks.

Byzantine-tolerant routing. Some work focuses on multi-hop routing tolerant of Byzantine faults. E.g., [101] uses hierarchical clustering for Byzantine-resilient routing in ad-hoc networks. Other work [5–7] uses adaptive probing to find faulty links caused by Byzantine nodes, and routes packets to avoid these links. BSMR [18] uses a reliability metric computed by comparing advertised and observed data rates of nodes in an ad-hoc wireless network to avoid routing packets through adversarial links. However, it is not clear whether they could be generalized beyond their specific applications.

Byzantine Fault Tolerance. There is an extensive work on Byzantine Fault Tolerance (BFT) [13, 46, 51, 89, 111], but most of it focuses on traditional distributed systems and not CPS, let alone MRS. Solutions often make assumptions that do not apply to MRS: e.g., most assume the asynchronous model (with a few notable exceptions [4, 17]), and almost all assume a fully-connected network. BFT++ [62] is a rare example of a BFT protocol that is designed specifically for CPS, but it is constrained by the need to work with legacy systems. Turqois [64, 65] is designed for wireless ad-hoc networks, but it cannot detect when a compromised robot lies about its sensor inputs or issues different actuator commands.

Auditing. Some research tries to *detect* compromised nodes, rather than merely masking their presence. PeerReview [42] uses tamper-evident logs and auditing to detect observable Byzantine faults [41]. However, as most BFT protocols, most auditing techniques assume asynchrony and a fully connected network, making them a poor fit for MRS. Bounded-Time Recovery [27, 28] can recover a system once a faulty node is found, but assumes a static and reliable network.

Trusted hardware. RoboRebound's approach of using a few simple trusted components to boost a protocol's power has been used before. E.g., TrInc [50], uses a simple attestation-enabled counter to prevent equivocation in BFT protocols. However, TrInc is specific to BFT and would not help with MRS, partly because it cannot be used to secure inputs or outputs. Individual devices have used trusted components to certify the accuracy of sensor inputs – the classical example is trustworthy cameras [25], which can certify that their pictures are genuine – but we are not aware of any solution that uses this to detect compromised nodes.

As discussed in Section 3.9, it should be possible to implement RoboRebound using trusted hardware primitives that are already available in a CPU – e.g., with TEEs such as ARM TrustZone or Intel's SGX [10, 108]. However, TEEs tend to have a much richer functionality than our *s*- and *a*-node, so their attack surface tends to be larger [22, 67, 73].

Simplex. Simplex [63, 84] uses a a complex controller and a simple safety controller that can take over when the former is compromised. This approach has been applied to multi-agent systems [60], but it requires a trustworthy implementation of the system's specific algorithm for the safety controller. In contrast, RoboRebound only requires trust in the *s*- and *a*-node, which are protocol-agnostic and much simpler.

7 Conclusion

RoboRebound shows that a little bit of trusted hardware can go a very long way. Although the *s*- and *a*-node in RoboRebound implement minimal functionality, they massively enhance RoboRebound's ability to detect a misbehaving compromised robot – even in cases where the robot lies about its sensor inputs or actuator commands – and they force each robot to periodically undergo auditing. This enables RoboRebound to be a general technique applicable to a wide range of MRS applications under the fully-Byzantine threat model. To our knowledge, this is a first: we are not aware of any other MRS defense that can do this.

And MRS do urgently need a general defense: common protocols tend to relay critical information from robot to robot, with little or no checking, so protocols may require robots to simply believe what other robots say. This could cause myriad vulnerabilities, of which this paper barely scratches the surface; this could be fertile ground for a more comprehensive study. In addition, we hope that this paper will raise awareness of the general problem, and ideally lead to the development of alternative approaches that are more secure.

Acknowledgments

We thank our shepherd, Jia-Ju Bai, and the anonymous reviewers for their thoughtful comments and suggestions. We thank David Saldaña for his help with the theory in [68], and Jiawei Xu for sharing pictures of multi-robot systems being used at Lehigh University. This work was supported in part by NSF grants CNS-1750158, CNS-1955670, CNS-2111688 and CCF-2326606.

References

[1] N. Agmon, G. A. Kaminka, and S. Kraus. Multi-robot adversarial patrolling: Facing a full-knowledge opponent. *J. Artif. Intell. Res.*, 42:887–916, 2011.

[2] N. Agmon, S. Kraus, and G. A. Kaminka. Multi-robot perimeter patrol in adversarial settings. *2008 IEEE International Conference on Robotics and Automation*, pages 2339–2345, 2008.

[3] M. Ahmadi and P. Stone. A multi-robot system for continuous area sweeping tasks. *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2006.

[4] Y. Amir, B. A. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8:564–577, 2011.

[5] B. Awerbuch, R. Curtmola, D. Holmer, C. Nita-Rotaru, and H. Rubens. Mitigating Byzantine attacks in ad hoc wireless networks. Technical report, 2004. https://people.cs.ksu.edu/~danielwang/Investigation/Wireless_Network/mitigate.pdf.

[6] B. Awerbuch, R. Curtmola, D. Holmer, C. Nita-Rotaru, and H. Rubens. ODSBR: An on-demand secure Byzantine resilient routing protocol for wireless ad hoc networks. *Trans. Inf. Syst. Secur.*, 10:6:1–6:35, 2008.

[7] B. Awerbuch, D. Holmer, C. Nita-Rotaru, and H. Rubens. An on-demand secure routing protocol resilient to Byzantine failures. In *Proc. ACM workshop on Wireless security (WiSE)*, 2002.

[8] Z. Bao, A. Chakraborti, N. Datta, J. Guo, M. Nandi, T. Peyrin, and K. Yasuda. Photon-beetle authenticated encryption. https://www.isical.ac.in/~lightweight/beetle/index.html.

[9] Z. Bao, A. Chakraborti, N. Datta, J. Guo, M. Nandi, T. Peyrin, and K. Yasuda. Photon-beetle authenticated encryption and hash family. *NIST Lightweight Compet. Round*, 1:115, 2019.

[10] J. Behl, T. Distler, and R. Kapitza. Hybrids on steroids: SGX-based high performance BFT. In *Proc. EuroSys*, 2017.

[11] F. Berlinger, M. Gauci, and R. Nagpal. Implicit coordination for 3d underwater collective behaviors in a fish-inspired robot swarm. *Science Robotics*, 6, 2021.

[12] G. A. Cardona, D. S. D'antonio, C. I. Vasile, and D. Saldaña. Nonprehensile manipulation of cuboid objects using a catenary robot. In *Proc. IROS*, 2021.

[13] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. OSDI*, 1999.

[14] Y. Chen, S. Zhang, Q. Guo, L. Li, R. Wu, and T. Chen. Deterministic replay. *ACM Computing Surveys (CSUR)*, 48:1 – 47, 2015.

[15] P. Cheng, J. R. Fink, S. Kim, and V. R. Kumar. Cooperative towing with multiple robots. In *Proc. Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2008.

[16] Christoph Roser. The Amazon robotics family: Kiva, Pegasus, Xanthus, and more... https://www.allaboutlean.com/amazon-robotics-family/, March 2020.

[17] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proc. NSDI*, 2009.

[18] R. Curtmola and C. Nita-Rotaru. BSMR: Byzantine-resilient secure multicast routing in multihop wireless networks. *Transactions on Mobile Computing*, 8:445–459, 2009.

[19] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. OSDI*, 2002.

[20] Espressif. ESP32 datasheet. https://projects.gctronic.com/epuck2/doc/esp32_datasheet_en.pdf.

[21] European Union. Regulation (EU) 2024/1689 of the EUROPEAN parliament and of the council. https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=OJ:L_202401689, 2024.

[22] S. Fei, Z. Yan, W. Ding, and H. Xie. Security vulnerabilities of SGX and countermeasures: A survey. *ACM CSUR*, 54(6), July 2021.

[23] M. Franceschelli, M. Egerstedt, and A. Giua. Motion probes for fault detection and recovery in networked control systems. *Proc. American Control Conference*, 2008.

[24] Fraunhofer Society. Research news: Swarming and transporting. https://www.fraunhofer.de/en/press/research-news/2012/march/swarming-and-transporting.html.

[25] G. Friedman. The trustworthy digital camera: restoring credibility to the photographic image. *IEEE Transactions on Consumer Electronics*, 39(4):905–910, 1993.

[26] N. Gandhi. *Bounded-Time Detection and Recovery from Faults in Large-Scale Distributed Cyber-Physical Systems.* PhD thesis, University of Pennsylvania, 2024.

[27] N. Gandhi, E. Roth, R. Gifford, L. T. X. Phan, and A. Haeberlen. Bounded-time recovery for distributed real-time systems. In *Proc. RTAS*, 2020.

[28] N. Gandhi, E. Roth, B. Sandler, A. Haeberlen, and L. T. X. Phan. REBOUND: Defending distributed systems against attacks with bounded-time recovery. In *Proc. EuroSys*, Apr. 2021.

[29] M. Gault. Hacker finds kill switch for submachine gun–wielding robot dog, 2022. https://www.vice.com/en/article/akeexk/hacker-finds-kill-switch-for-submachine-gun-wielding-robot-dog?utm_source=reddit.com.

[30] GCtronic. e-puck2. https://www.gctronic.com/doc/index.php/e-puck2.

[31] S. Gil, C. Baykal, and D. Rus. Resilient multi-agent consensus using Wi-Fi signals. *Control Systems Letters*, 3:126–131, 2019.

[32] S. Gil, S. Kumar, M. Mazumder, D. Katabi, and D. Rus. Guaranteeing spoof-resilient multi-robot networks. In *Proc. RSS*, 2015.

[33] S. Gil, S. Kumar, M. Mazumder, D. Katabi, and D. Rus. Guaranteeing spoof-resilient multi-robot networks. *Autonomous Robots*, 41:1383–1400, 2017.

[34] A. Gonzalez-Ruiz and Y. Mostofi. Cooperative robotic structure mapping using wireless measurements—a comparison of random and coordinated sampling patterns. *IEEE Sensors Journal*, 13:2571–2580, 2013.

[35] B. Grocholsky, J. F. Keller, V. R. Kumar, and G. J. Pappas. Cooperative air and ground surveillance. *IEEE Robotics & Automation Magazine*, 13:16–25, 2006.

[36] L. Guerrero-Bonilla and D. V. Dimarogonas. Perimeter surveillance based on set-invariance. *IEEE Robotics and Automation Letters*, 6:9–16, 2021.

[37] L. Guerrero-Bonilla, C. Nieto-Granda, and M. Egerstedt. Robust perimeter defense using control barrier functions. In *Proc. International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*, 2021.

[38] L. Guerrero-Bonilla, A. Prorok, and V. R. Kumar. Formations for resilient robot teams. *RA-L*, 2:841–848, 2017.

[39] L. Guerrero-Bonilla, D. Saldaña, and V. R. Kumar. Design guarantees for resilient robot formations on lattices. *RA-L*, 4:89–96, 2019.

[40] L. Guerrero-Bonilla, D. Saldaña, and V. R. Kumar. Dense r-robust formations on lattices. In *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2020.

[41] A. Haeberlen and P. Kuznetsov. The fault detection problem. In *Proc. OPODIS*, 2009.

[42] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. SOSP*, 2007.

[43] W. Hönig, S. Kiesel, A. Tinka, J. W. Durham, and N. Ayanian. Persistent and robust execution of mapf schedules in warehouses. *IEEE Robotics and Automation Letters*, 4:1125–1131, 2019.

[44] IEEE Spectrum. Kilobot. https://robots.ieee.org/robots/kilobot/.

[45] A. Kamagaew, J. Stenzel, A. Nettsträter, and M. ten Hompel. Concept of cellular transport systems in facility logistics. In *Proc. 5th International Conference on Automation, Robotics and Applications*, 2011.

[46] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *Trans. Comput. Syst.*, 27:7:1–7:39, 2010.

[47] J. Kottinger, S. Almagor, and M. Lahijanian. Conflict-based search for explainable multi-agent path finding. *ArXiv*, abs/2202.09930, 2022.

[48] F. Laurent, M. Schneider, C. V. Scheller, J. D. Watson, J. Li, Z. Chen, Y. Zheng, S.-H. Chan, K. Makhnev, O. Svidchenko, V. Egorov, D. Ivanov, A. Shpilman, E. Spirovska, O. Tanevski, A. M. S. nikov, R. Grunder, D. Galevski, J. Mitrovski, G. Sartoretti, Z. Luo, M. Damani, N. Bhattacharya, S. Agarwal, A. Egli, E. Nygren, and S. P. Mohanty. Flatland competition 2020: MAPF and MARL for efficient train coordination on a grid world. In *Proc. NeurIPS*, 2020.

[49] T. Lee, M. Leok, and N. H. McClamroch. Geometric tracking control of a quadrotor UAV on SE(3). *Proc. IEEE Conference on Decision and Control (CDC)*, 2010.

[50] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proc. NSDI*, 2009.

[51] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proc. NSDI*, 2007.

[52] Z. Li, A. V. Barenji, J. Jiang, R. Y. Zhong, and G. Xu. A mechanism for scheduling multi robot intelligent warehouse system face with dynamic demand. *Journal of Intelligent Manufacturing*, 31:469–480, 2020.

[53] L. Liu, W. Li, T. Xia, and Z. Wan. Design of warehouse cooperative robot system based on ZigBee. *Proc. International Conference on Computer Engineering and Networks*, 2021.

[54] Y. K. Lopes, S. M. Trenkwalder, A. B. Leal, T. J. Dodd, and R. Groß. Supervisory control theory applied to swarm robotics. *Swarm Intelligence*, 10:65–97, 2016.

[55] C. E. Luis and J. L. Ny. Design of a trajectory tracking controller for a nanoquadcopter. *ArXiv*, abs/1608.05786, 2016.

[56] A. Luykx, B. Preneel, E. Tischhauser, and K. Yasuda. A MAC mode for lightweight block ciphers. *IACR Cryptol. ePrint Arch.*, 2016:190, 2016.

[57] F. Mallmann-Trenn, M. Cavorsi, and S. Gil. Crowd vetting: Rejecting adversaries via collaboration with application to multirobot flocking. *Transactions on Robotics*, 38:5–24, 2022.

[58] M. McCauley. RadioHead packet radio library for embedded microprocessors. https://www.airspayce.com/mikem/arduino/RadioHead/, 2023. Accessed: 2024-05-17.

[59] T. G. McGee and J. K. Hedrick. Guaranteed strategies to search for mobile evaders in the plane. In *Proc. American Control Conference (ACC)*, 2006.

[60] U. Mehmood, S. Stoller, R. Grosu, S. Roy, A. Damare, and S. Smolka. A distributed simplex architecture for multi-agent systems. *ArXiv*, abs/2012.10153, 2020.

[61] D. Mellinger and V. R. Kumar. Minimum snap trajectory generation and control for quadrotors. In *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2011.

[62] J. S. Mertoguno, R. Craven, M. Mickelson, and D. Koller. A physics-based strategy for cyber resilience of CPS. In *Defense + Commercial Sensing*, 2019.

[63] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo. S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *Proc. ACM international conference on High Confidence Networked Systems*, 2013.

[64] H. Moniz, N. Neves, and M. Correia. Turquois: Byzantine consensus in wireless ad hoc networks. In *Proc. DSN*, 2010.

[65] H. Moniz, N. Neves, and M. Correia. Byzantine fault-tolerant consensus in wireless ad hoc networks. *Transactions on Mobile Computing*, 12:2441–2454, 2013.

[66] R. Morris, C. S. Pasareanu, K. S. Luckow, W. A. Malik, H. Ma, T. K. S. Kumar, and S. Koenig. Planning, scheduling and monitoring for airport surface operations. In *Proc. AAAI Workshop on Planning for Hybrid Systems*, 2016.

[67] A. Muñoz, R. Ríos, R. Román, and J. López. A survey on the (in)security of trusted execution environments. *Computers & Security*, 129, 2023.

[68] R. Olfati-Saber. Flocking for multi-agent dynamic systems: algorithms and theory. *IEEE Transactions on Automatic Control*, 51:401–420, 2006.

[69] H. Park and S. Hutchinson. A distributed robust convergence algorithm for multi-robot systems in the presence of faulty robots. In *Proc. IROS*, 2015.

[70] H. Park and S. Hutchinson. An efficient algorithm for fault-tolerant rendezvous of multi-robot systems with controllable sensing range. In *Proc. ICRA*, 2016.

[71] F. Pasqualetti, A. Franchi, and F. Bullo. On cooperative patrolling: Optimal trajectories, complexity analysis, and approximation algorithms. *IEEE Transactions on Robotics*, 28:592–606, 2012.

[72] Peloton Technology. Peloton announces its vision for the trucking industry: Drivers lead, and technology follows. https://peloton-tech.com/peloton-announces-its-vision-for-the-trucking-industry-drivers-lead-and-technology-follows/, July 2019.

[73] S. Pinto and N. Santos. Demystifying ARM TrustZone. *ACM Computing Surveys (CSUR)*, 51:1 – 36, 2019.

[74] J. Prisco. Why online supermarket Ocado wants to take the human touch out of groceries. https://www.cnn.com/2021/04/26/world/ocado-supermarket-robot-warehouse-spc-intl/index.html, May 2021.

[75] A. Renzaglia, L. Doitsidis, A. Martinelli, and E. B. Kosmatopoulos. Multi-robot three-dimensional coverage of unknown areas. *The International Journal of Robotics Research*, 31:738 – 752, 2012.

[76] G. F. Riley and T. R. Henderson. The ns-3 network simulator. In *Modeling and Tools for Network Simulation*, 2010.

[77] C. Robotics. Turtlebot 4. https://clearpathrobotics.com/turtlebot-4/.

[78] M. Rubenstein, C. Ahler, and R. Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. In *Proc. ICRA*, 2012.

[79] F. Rusu. LowPowerLab RFM69 library. https://github.com/LowPowerLab/RFM69, 2023. Accessed: 2024-05-17.

[80] D. Saldaña, B. Gabrich, G. Li, M. Yim, and V. Kumar. ModQuad: The flying modular structure that self-assembles in midair. In *Proc. ICRA*, 2018.

[81] D. Saldaña, A. Prorok, M. F. M. Campos, and V. Kumar. Triangular networks for resilient formations. In *Proc. International Symposium on Distributed Autonomous Robotic Systems (DARS)*, 2016.

[82] D. Saldaña, A. Prorok, S. Sundaram, M. Campos, and V. R. Kumar. Resilient consensus for time-varying networks of dynamic agents. In *Proc. American Control Conference (ACC)*, 2017.

[83] K. Saulnier, D. Saldaña, A. Prorok, G. J. Pappas, and V. R. Kumar. Resilient flocking for mobile robot teams. *IEEE Robotics and Automation Letters*, 2:1039–1046, 2017.

[84] L. R. Sha. Using simplicity to control complexity. *IEEE Softw.*, 18:20–28, 2001.

[85] K. Sharma and R. Doriya. Coordination of multi-robot path planning for warehouse application using smart approach for identifying destinations. *Intelligent Service Robotics*, 14:313–325, 2021.

[86] D. Shishika and V. R. Kumar. Local-game decomposition for multiplayer perimeter-defense problem. In *Proc. IEEE Conference on Decision and Control (CDC)*, 2018.

[87] D. Shishika, J. Paulos, M. R. Dorothy, M. A. Hsieh, and V. R. Kumar. Team composition for perimeter defense with patrollers and defenders. In *Proc. IEEE Conference on Decision and Control (CDC)*, 2019.

[88] D. Shishika, J. Paulos, and V. R. Kumar. Cooperative team strategies for multi-player perimeter-defense games. *IEEE Robotics and Automation Letters*, 5:2738–2745, 2020.

[89] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent Byzantine-fault tolerance. In *Proc. NSDI*, 2009.

[90] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full sha-1. In *Proc. CRYPTO*, 2017.

[91] V. Strobel, E. Castelló Ferrer, and M. Dorigo. Managing Byzantine robots via Blockchain technology in a swarm robotics collective decision making scenario. In *Proc. International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2018.

[92] V. Strobel, E. C. Ferrer, and M. Dorigo. Blockchain technology secures robot swarms: A comparison of consensus protocols and their resilience to Byzantine robots. *Frontiers in Robotics and AI*, 7, 2020.

[93] S. C. Stuart. Inside the online-only grocer where robots get the job done. https://www.pcmag.com/news/inside-the-online-only-grocer-where-robots-get-the-job-done, Dec 2018.

[94] S. Sun, G. Cioffi, C. de Visser, and D. Scaramuzza. Autonomous quadrotor flight despite rotor failure with onboard vision sensors: Frames vs. events. *IEEE Robotics and Automation Letters*, 6:580–587, 2021.

[95] Swarm Logistics. Demo of a decentralized & serverless fleet control system and the metaverse economy in logistics. https://www.youtube.com/watch?v=UTVBYGwIEyU, March 2022.

[96] Texas Instruments. MSP430 programming with the JTAG interface. SLAU320AJ, May 2021.

[97] D. Thakur, Y. Tao, R. Li, A. Zhou, A. Kushleyev, and V. R. Kumar. Swarm of inexpensive heterogeneous micro aerial vehicles. In *Proc. International Symposium on Experimental Robotics (ISER)*, 2020.

[98] P. Tokekar, E. Branson, J. V. Hook, and V. Isler. Tracking aquatic invaders: Autonomous robots for monitoring invasive fish. *IEEE Robotics & Automation Magazine*, 20:33–41, 2013.

[99] G. Vásárhelyi, C. Virágh, G. Somorjai, T. Nepusz, A. E. Eiben, and T. Vicsek. Optimized flocking of autonomous drones in confined environments. *Science Robotics*, 3, 2018.

[100] M. M. Veloso, J. Biswas, B. Coltin, and S. Rosenthal. Cobots: Robust symbiotic autonomous mobile service robots. In *Proc. IJCAI*, 2015.

[101] S.-S. Wang, K. Yan, and S.-C. Wang. An optimal solution for Byzantine agreement under a hierarchical cluster-oriented mobile ad hoc network. *Comput. Electr. Eng.*, 36:100–113, 2010.

[102] X. Wang, N. Hovakimyan, and L. Sha. L1simplex: Fault-tolerant control of cyber-physical systems. *Proc. ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2013.

[103] K. Wardega, R. Tron, and W. Li. Resilience of multi-robot systems to physical masquerade attacks. In *Proc. IEEE Security and Privacy Workshop (SPW)*, 2019.

[104] T. Wheeler, E. Bharathi, and S. Gil. Switching topology for resilient consensus using wi-fi signals. In *Proc. ICRA*, 2019.

[105] P. R. Wurman, R. D'Andrea, and M. Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Mag.*, 29(1), 2008.

[106] J. Xu. MochiSwarm blimps from AIRLab at Lehigh University. Image, 2024.

[107] J. Xu. Multi-tool ground robots used to teach cooperative robotics by David Saldaña, Subhrajit Bhattacharya, and Cristian-Ioan Vasile at Lehigh University. Image, 2024.

[108] W. Xu and R. Kapitza. RATCHETA: Memory-bounded hybrid Byzantine consensus for cooperative embedded systems. In *Proc. IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2018.

[109] F. Xue, H. Tang, Q. Su, and T. Li. Task allocation of intelligent warehouse picking system based on multi-robot coalition. *KSII Trans. Internet Inf. Syst.*, 13:3566–3582, 2019.

[110] M. Yemini, A. Nedic, A. J. Goldsmith, and S. Gil. Characterizing trust and resilience in distributed consensus for cyberphysical systems. *Transactions on Robotics*, 38:71–91, 2022.

[111] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. SOSP*, 2003.

[112] X. Yu and M. A. Hsieh. Synthesis of a time-varying communication network by robot teams with information propagation guarantees. *RA-L*, 5:1413–1420, 2020.

[113] X. Yu, D. Saldaña, D. Shishika, and M. A. Hsieh. Resilient consensus in robot swarms with periodic motion and intermittent communication. *IEEE Transactions on Robotics*, 38:110–125, 2022.

[114] X. Yu, D. Shishika, D. Saldaña, and M. A. Hsieh. Modular robot formation and routing for resilient consensus. In *Proc. American Control Conference (ACC)*, 2020.

[115] U. Zengin and A. Dogan. Real-time target tracking for autonomous uavs in adversarial environments: A gradient search algorithm. *IEEE Transactions on Robotics*, 23:294–307, 2007.

[116] L. Zhou and V. R. Kumar. Robust multi-robot active target tracking against sensing and communication attacks. In *Proc. American Control Conference (ACC)*, 2022.

[117] L. Zhou, V. Tzoumas, G. J. Pappas, and P. Tokekar. Distributed attack-robust submodular maximization for multi-robot planning. In *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2020.

A Olfati-Saber Parameters

Table 3 lists the values of parameters we used for our `ns-3`-based implementation of the Olfati-Saber flocking protocol. Broadly, these cover parameters related to the network communications and control algorithm that runs on each robot in a group of robots that are performing Olfati-Saber flocking. We provide basic descriptions, but for a more in-depth treatment, please see the original paper [68].

Table 3. Parameters used for Olfati-Saber flocking.

Name	Value	Note
d	Varies	Desired inter-robot spacing [68, Eq. 5].
r	$1.2d$	This is the wireless range in the original paper [68, Eq. 3], but is not really used since we use the more realistic signal loss models of `ns-3`. $\kappa = 1.2$ is from [68, Section VIII].
κ	d/r	Distance-to-wireless-range ratio [68, Def. 1]
ϵ	0.1	Parameter for the σ-norm [68, Eq. 8].
a	5.0	Parameter for 'action function' [68, Eq. 15].
b	5.0	Parameter for 'action function' [68, Eq. 15].
c	$\frac{a-b}{\sqrt{4ab}}$	Parameter for 'action function' [68, Eq. 15].
h_{ϕ_a}	0.2	Boundary of one of the piecewise 'bump functions' [68, Eq. 10].
h_{ϕ_b}	0.9	Boundary of one of the piecewise 'bump functions' [68, Eq. 10].
d'	$0.5\kappa d$	Used to describe robot-to-obstacle algebraic constraints [68, Eq. 50].
r'	$\kappa d'$	Defines the *interaction range* w.r.t. obstacles [68, Eq. 47].
C_1^α	0.005	Akin to a 'spring' constant for control gain w.r.t. other robots [68, Eq. 59].
C_2^α	0.05	Akin to a 'damper' constant for control gain w.r.t. other robots [68, Eq. 59].
C_1^β	0.0	Akin to a 'spring' constant for control gain w.r.t. obstacles [68, Eq. 59]. This is set to zero because our evaluation does not use obstacles.
C_2^β	0.0	Akin to a 'spring' constant for control gain w.r.t. obstacles [68, Eq. 59]. This is set to zero because our evaluation does not use obstacles.
C_1^γ	−0.001	Akin to a 'spring' constant for control gain w.r.t. destination [68, Eq. 59].
C_2^γ	−0.060	Akin to a 'damper' constant for control gain w.r.t. destination [68, Eq. 59].

B SecBot Hardware Design

A SecBot consists of three key pieces: the software running on the two trusted nodes and one untrusted node, the electronics that connect these nodes to one another, the sensors, the actuators, and the radio, and the mechanical chassis that holds everything together. This appendix will primarily focus on the design of the electronics such that SecBot can be a usable robot. There are six modules that the circuit board is broken into: the (1) sensor module, (2) sensor node module, (3) compute node module, (4) actuator node module, (5) actuator module, and (6) radio module.

Each robot provides slots for a sensor suite: one Global Navigation Satellite System (GNSS), one Inertial Measurement Unit (IMU), and two encoders. Data from sensors gets channeled to the s-node, which is a PIC32MX130F064B MCU with 20KB data memory and 64KB of program memory. The s-node, once it does BTI-level processsing of sensor data (such as appending sensor data to its hash chain), it forwards the data to the c-node module. The c-node module runs a control algorithm and determines what control signals to send to the a-node module. The a-node module uses Pulse Width Modulation (PWM) to control the speed of two DC brush motors bolted to the chassis. The a-node also interfaces to a 915 MHz half-duplex radio transceiver. To prevent issues with having to trust third-party drivers, all software for trusted nodes MCU was custom-written. The only external file used was `xc.h`, which provides mappings from register addresses to human-readable register names.

The *c*-node on a SecBot is a TinkerBoard S, which has a Quad-Core 32-bit ARM Cortex A-17 MPCORE RK3288 CPU running at 1.8GHz, and 2GB of DDR3 RAM. The *s*-node and *a*-node both are implemented in MIPS32-based PIC32MX130F064B MCUs running at 50MHz, with each having 64KB of program memory and 20KB of data memory.

Sensors, actuators, and radio module. SecBot contains an Adafruit Mini GPS PA1010D module and an Adafruit TDK InvenSense ICM-20948 9-DoF IMU that are both controlled over a shared Inter-Integrated Circuit (I2C) bus. There are also two pins reserved to be used by encoders that are attached to the axles of each of the two wheels; encoders are used to get accurate estimations of the amount a wheel has rotated. For actuation, SecBot includes a L298N motor driver that controls two DC brush motors. A SecBot robot can communicate with other robots over a RFM69HCW 915MHz radio. This radio is connected over an SPI bus to the *a*-node. This ensures that the *c*-node never directly communicates with outside agents.

Sensor node. The *s*-node controls the I2C bus that talks to the GNSS and IMU modules at a bus rate of 100kHz. The *s*-node also uses four pins for SPI communication with the *c*-node. This setup allows for the *s*-node to interpose on the communications line between the sensors and the untrusted *c*-node. The speed of this bus is configurable using a setting in the *c*-node, and has been tested at 500kHz-1.5MHz.

Control node. The control node, or *c*-node, is complex and therefore untrustable. SecBot uses a TinkerBoard S as its *c*-node. The *c*-node uses its onboard SPI0 SPI module to talk to the *s*-node and SPI2 to talk to the *a*-node. The communications rely on using the `spidev` kernel module. Access to the radio comes through the *a*-node, and the radio module has a FIFO buffer limited to 66B. As a result, large packets are fragmented and re-assembled by the receiver.

Actuator node. SecBot uses the same PIC32MX130F064B MCU for its *a*-node as it does for its *s*-node. Since the *a*-node has a few more responsibilities than the *s*-node, it has a slightly more complex setup. The *a*-node also uses an SPI bus to talk to the *c*-node, but it has a second SPI module for communicating with the RFM69HCW 915MHz ISM-band radio module; this is necessary since a trusted node needs to interpose on all external communications that the *c*-node has. The *a*-node handles this additional responsibility by using an addressable latch. This chip allows the *a*-node to control eight outputs with only five pins (as opposed to eight pins otherwise).

For the RoboRebound protocol, the *a*-node needs to have a (local) sense of time. It uses Timer 4 and Timer 5 together as a 32-bit system clock that is initialized to 0 at startup. Each tick of this clock represents 5.12ms.

The *a*-node uses two Output-Compare modules (OC2 and OC4) to produce Pulse Width Modulation signals to control the speed of two DC brush motors via the L298N motor driver. Two of the addressable latch outputs are used to control the direction that each motor spins.

The *a*-node software includes a radio driver that was written specifically for SecBot, but based on the drivers available from RadioHead [58] and LowPowerLabs [79]. The radio driver implements a simple version of Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). It does not implement CSMA/CA acks; instead, it simply checks whether the channel is busy before attempting to send.

Received 22 May 2024; accepted 21 Aug 2024