AALBORG UNIVERSITY
STUDENT REPORT

Title:
Moving Target Defense for IoT

Theme:
Master Thesis

Project Period:
4th Semester,
Spring 2026

Project Group:
1098

Participant(s):
Sampath Reddy Bheemavarapu
Sayantan Mondal

Supervisor(s):
Sajad Homayoun

Page Numbers: 101

Date of Completion:
June 3, 2026

Abstract:

This project investigates how Moving Target Defense (MTD) can improve the security of resource-constrained Internet of Things (IoT) networks by continuously changing exposed network identifiers before attackers can reliably exploit them. We developed a lightweight MTD orchestration framework that combines two MTD mechanisms, IPv6 address shuffling and application-layer port-token rotation, with a reactive rate limiter for flood mitigation, for 6LoWPAN/RPL-based IoT environments. The system uses a rule-based orchestrator placed at the border router to monitor stale-port events, sensor silence, and packet-rate anomalies, and then selects an appropriate proactive or reactive defense response. The prototype was implemented in Contiki-NG and evaluated in the Cooja simulator using a Tmote Sky-based IoT topology under three attack scenarios: IPv6 address scanning, a DIO-shaped UDP radio-disruption approximation of an RPL sinkhole, and CoAP flooding. The evaluation shows that the proposed approach can invalidate stale active-probing information under the evaluated attacker model, trigger adaptive defense actions, and reduce the impact of flooding traffic while maintaining acceptable energy and performance overhead. The findings provide simulation evidence that lightweight, rule-based MTD is a feasible defense direction for improving resilience in constrained IoT networks.

The content of this report is freely available, but publication (with reference) may only be pursued in agreement with the authors through their consent.

Contents

List of Acronyms **6**

1 Introduction **7**

1.1 Motivation 7

1.2 Problem Statement 8

1.3 Use of AI Assisted Tools 8

2 Background **9**

2.1 Internet of Things: Architecture and Constraints 9

2.2 IoT Threat Landscape 9

2.3 IoT Communication Protocols 11

2.4 Moving Target Defense 11

2.4.1 The MTD Design Space: What, How, and When to Move 12

2.4.2 The Challenge of MTD in Resource Constrained IoT 13

2.5 Taxonomy of MTD Techniques 13

3 Related Work **14**

3.1 Search Process 14

3.2 Selection Process 15

3.3 Overview of Reviewed MTD Techniques 17

3.4 Surveys, Taxonomies, and Systematic Reviews 19

3.5 Network Layer MTD Techniques 21

3.5.1 Address and Parameter Shuffling 21

3.5.2 SDN-Based Topology and Route Mutation 23

3.5.3 MAC and Physical-Layer Randomisation 24

3.5.4 DDoS Mitigation via MTD 24

3.6 Cyber-Deception and Honeypot-Augmented MTD 25

3.7 Application, Data, and Infrastructure MTD 26

3.7.1 Protocol and Code Diversification 27

3.7.2 Key Management, Data Integrity, and Virtualisation 27

3.7.3 Smart Grid and Cyber-Physical Systems 28

3.8 Optimisation-Driven and Adaptive MTD 28
3.8.1 Game-Theoretic and Reinforcement Learning Approaches 29
3.8.2 Hybrid and Model-Based Frameworks 30
3.9 Research Gap and Positioning of This Work 31

4 System Design **32**
4.1 Design Goals and Constraints 32
4.2 System Architecture 32
4.3 Network Topology 34
4.4 MTD Orchestrator Design 35
4.4.1 Monitoring and Threat Detection 35
4.4.2 Technique Selection Logic 37
4.4.3 MTD Enforcement via Direct Commands 38
4.5 IPv6 Address Shuffling 38
4.6 Application-Layer Port Token Rotation 39

5 Attack Design **41**
5.1 Threat Model 41
5.2 Attack Framework and Simulation Setup 42
5.2.1 Malicious Mote Integration 42
5.2.2 Experimental Protocol 42
5.3 Attack Scenarios 43
5.3.1 IPv6 Address Scanning 43
5.3.2 RPL Sinkhole Attack 44
5.3.3 UDP Flooding (Denial of Service) 46

6 Implementation **48**
6.1 Cooja Simulation Setup 48
6.1.1 Radio Medium 48
6.1.2 Network Topology 48
6.1.3 Simulation Configurations 49
6.2 System Implementation 50
6.2.1 Border Router and Orchestrator Boot 50
6.2.2 Sensor Address Registry 50

6.2.3 Proactive Defense Cycle . . . 51
6.2.4 Sending Commands to Every Sensor . . . 52
6.2.5 Reactive Defense Cycle . . . 53
6.2.6 Threat-Type Selection . . . 54
6.2.7 Sensor Silence Watchdog (with Pre-Shuffle Drain) . . . 55
6.2.8 Anomaly Detection at the Border Router . . . 57
6.2.9 Rate Limiter . . . 58
6.2.10 IPv6 Address Shuffler . . . 58
6.2.11 Application-Layer Port Token Rotation . . . 59
6.2.12 Sensor Node . . . 59
6.3 Attack Implementation . . . 60
6.3.1 Per-Mode Wrappers . . . 60
6.3.2 Scan Mode . . . 60
6.3.3 Sinkhole Mode . . . 61
6.3.4 Flood Mode . . . 62
6.4 Log Collection . . . 62

7 Evaluation **64**
7.1 Evaluation Methodology . . . 64
7.1.1 Experimental setup . . . 64
7.1.2 Metrics . . . 65
7.1.3 Log parsing . . . 66
7.2 Security Effectiveness . . . 68
7.2.1 Anomaly detection coverage . . . 68
7.2.2 Reactive branch coverage . . . 69
7.2.3 Detection and reaction latency . . . 70
7.3 Attacker's Perspective . . . 71
7.3.1 Time to First Reactive Defense (TFRD) and mutation rate . . . 71
7.3.2 Wasted attacker effort . . . 73
7.3.3 Flood-specific ingestion ratio . . . 74
7.3.4 Security Metrics . . . 74
7.4 Energy Consumption . . . 77
7.4.1 Per-sensor energy . . . 77

7.4.2 Border router energy . . . 78
7.4.3 Energy breakdown by state . . . 79
7.5 Operational Behaviour . . . 79
7.5.1 Reactive vs proactive ratio . . . 80
7.5.2 Post-cycle reconvergence time . . . 80
7.5.3 Firmware footprint . . . 81
7.6 Attacker-Speed Sensitivity Sweep . . . 82

8 Discussion 84
8.1 Challenges . . . 84
8.1.1 Robustness Refinements . . . 84
8.1.2 Sliding-window port acceptance . . . 84
8.1.3 Source-aware rate limiting with adaptive cap . . . 85
8.1.4 Stale-port suppression during a flood . . . 85
8.1.5 Proactive timer jitter . . . 86
8.1.6 Per-source telemetry at the border router . . . 86
8.1.7 Tmote Sky flash budget . . . 86
8.1.8 MTD aggression versus DODAG stability . . . 87
8.1.9 Selection-rule design under noisy signals . . . 87
8.1.10 Cooja platform constraints for MSP430 emulation . . . 88
8.2 Limitations . . . 88
8.2.1 Simulation only evaluation . . . 88
8.2.2 Sinkhole approximation . . . 88
8.2.3 Attack coverage . . . 89
8.2.4 Trusted control channel assumption . . . 89
8.3 Future Work . . . 89
8.3.1 Per-packet latency instrumentation on a larger target . . . 89
8.3.2 True rank-1 DIO sinkhole . . . 90
8.3.3 Hardware pilot . . . 90
8.3.4 Multi-attacker scenarios . . . 90
8.3.5 Authenticated MTD commands and sensor admission control . . . 90
8.3.6 Topology mutation . . . 90
8.3.7 Power-aware shuffle scheduling . . . 91

9 Conclusion **92**

List of Figures **100**

List of Acronyms

6LoWPAN – IPv6 over Low-Power Wireless PANs

ACL – Access Control List

API – Application Programming Interface

ARP – Address Resolution Protocol

ASLR – Address Space Layout Randomisation

BLE – Bluetooth Low Energy

BR – Border Router

CoAP – Constrained Application Protocol

CPS – Cyber-Physical System

CSMA/CA – Carrier Sense Multiple Access / CA

DAG – Directed Acyclic Graph

DIO – DODAG Information Object (RPL)

DODAG – Destination Oriented DAG

DoS – Denial of Service

DSSS – Direct Sequence Spread Spectrum

DTLS – Datagram Transport Layer Security

HMAC – Hash-based Message Auth. Code

ICMPv6 – Internet Control Message Protocol v6

IDS – Intrusion Detection System

IID – Interface Identifier (low 64 bits)

IoT – Internet of Things

IPv6 – Internet Protocol version 6

LPM – Low Power Mode

LPWAN – Low-Power Wide-Area Network

MAC – Medium Access Control

MQTT – Message Queuing Telemetry Transport

MSP430 – Texas Instruments MSP430 MCU

μMTD – Micro Moving Target IPv6 Defense

MTD – Moving Target Defense

ND – Neighbour Discovery

PDR – Packet Delivery Ratio

PRISMA – Pref. Reporting Items for Sys. Reviews

PRNG – Pseudo-Random Number Generator

PUF – Physical Unclonable Function

RF – Radio Frequency

RFC – Request For Comments

RL – Reinforcement Learning

RPL – IPv6 Routing Protocol for LLNs

RX – Receive

SDN – Software Defined Networking

SDR – Software Defined Radio

SLR – Systematic Literature Review

SNMP – Simple Network Management Protocol

TCP – Transmission Control Protocol

TFRD – Time to First Reactive Defense

TLS – Transport Layer Security

TSCH – Time-Slotted Channel Hopping

TTC – Time To Compromise

TX – Transmit

UDGM – Unit Disk Graph Medium (Cooja)

UDP – User Datagram Protocol

ULA – Unique Local Address (RFC 4193 [28])

URI – Uniform Resource Identifier

WSN – Wireless Sensor Network

1 Introduction

The Internet of Things (IoT) has grown from a research idea into a significant part of everyday infrastructure. Smart homes, industrial control systems, healthcare monitoring devices, and smart city sensors now make up tens of billions of connected endpoints worldwide. Most of these devices share two properties: they have very limited hardware (small flash, small RAM, battery powered) and they are deployed for many years with little or no maintenance. That combination makes them attractive targets, and it also makes them difficult to defend with the security tools designed for general purpose computers.

This thesis investigates one specific class of defense for these networks: Moving Target Defense (MTD). The idea behind MTD is to keep changing the system's configuration so that the information an attacker collects during reconnaissance becomes stale before it can be exploited. Instead of trying to make the system harder to attack at a single moment in time, MTD continuously shifts the surface the attacker is targeting. The thesis builds, deploys, and evaluates a rule based adaptive MTD orchestrator for a 6LoWPAN/RPL IoT network, with the goal of finding out whether such a system can produce a measurable security benefit without breaking the resource budget of Class 1 constrained devices.

1.1 Motivation

First, IoT endpoints are easy to enumerate and easy to predict. They typically run a fixed IP address, a fixed port, and a fixed application protocol from the moment they boot until they are decommissioned. An attacker who scans the prefix of an IPv6 network and once finds every sensor, can revisit each one at any later time, and can plan an exploitation campaign at leisure. There is no shortage of evidence that this static surface gets attacked at scale. The Mirai botnet recruited hundreds of thousands of IoT devices in a matter of hours using little more than default credentials and known addresses.

Second, the security tools built for traditional enterprise networks do not fit on constrained IoT devices. Firewalls, host based intrusion detection, vulnerability scanners, and patch management all assume roughly the resource profile of a server or a workstation. An RFC 7228 Class 1 device has on the order of ten kilobytes of RAM and about a hundred kilobytes of flash (and the Tmote Sky used in this thesis is tighter still, with 48 kilobytes), runs on battery power, and has to keep its radio duty cycle low to stay alive. A defense mechanism that fits comfortably on a server can be impossible to even compile for such a device.

Third, the academic literature on MTD for IoT has grown quickly, but most published proposals either evaluate only one MTD technique in isolation, or rely on machine learning and game theoretic optimisers that do not fit a constrained device. There is a gap in the middle: a defense that combines several lightweight MTD techniques, selects between them based on the type of threat actually observed, and stays small enough to run on a real sensor. Closing

that gap is what this thesis tries to do.

1.2 Problem Statement

The problem this thesis addresses can be stated in one sentence. Existing MTD proposals for IoT either apply a single mutation technique on a fixed schedule, which wastes resources when no attack is happening, or use optimisation engines that are too heavy for a constrained device. A lightweight orchestrator that picks the right mutation in response to the right kind of threat indicator, and that fits inside the memory and energy budget of a Class 1 device, has not been clearly demonstrated.

To break this problem down into manageable parts, the thesis follows the three research questions:

- **RQ1.** What MTD techniques have been proposed for IoT, and how do they compare for security benefit versus resource cost?
- **RQ2.** What is the smallest viable IoT network setup to evaluate MTD (topology, number of nodes, protocol stack), and which tools fit best for this?
- **RQ3.** When do specific MTD choices (enabling or disabling) lead to clear changes in attack surface exposure and service availability in a simulated IoT network?

RQ1 is answered through a PRISMA based systematic literature review (Chapter 3) that surveys 63 papers and groups them by technique family, evaluation evidence, and target device class. RQ2 is answered through the system design and implementation chapters (Chapter 4 and Chapter 6), which describe a 30 node 6LoWPAN/RPL testbed running on Contiki-NG inside the Cooja simulator, with the firmware compiled for the Tmote Sky platform so that the energy figures stay realistic for a Class 1 device. RQ3 is answered through the evaluation chapter (Chapter 7), which runs each of three attack scenarios with and without MTD enabled, under identical conditions, and compares the resulting security, performance, and energy metrics.

1.3 Use of AI Assisted Tools

AI assistants were used during this thesis in three concrete ways, all documented here for transparency. During the literature review phase, large language models (ChatGPT [54], Claude [3], and Gemini [21]) were used to produce short summaries of candidate papers before reading them in full, so we could quickly decide whether a paper was relevant to the scope of this work before committing the time to read it end to end. During the implementation phase, the same assistants were used to debug Contiki-NG build errors and to suggest code patterns when the public API of an unfamiliar Contiki-NG module was not obvious. During the writing phase, the assistants were used to polish academic tone and to catch repetition or unclear wording.

2 Background

This chapter introduces the foundations for the thesis. It covers the architecture and hardware constraints of IoT systems, the threat landscape that motivates proactive network layer defenses, the communication protocols on which those defenses operate, the principles and taxonomy of Moving Target Defense and the evaluation metrics used to assess MTD effectiveness.

2.1 Internet of Things: Architecture and Constraints

The Internet of Things (IoT) refers to the broad ecosystem of physical devices, sensors, actuators, and embedded computers connected to a network and capable of collecting, exchanging, or acting upon data. Deployments span application domains including smart homes, industrial control systems, healthcare monitoring, and smart city infrastructure, with tens of billions of devices projected worldwide significantly expanding the attack surface of digital systems [51].

IoT endpoints are characterised by strict hardware constraints that differentiate them from conventional computing platforms. RFC 7228 [8] classifies constrained devices into three tiers by available RAM and storage, illustrated in Figure 1: Class 0 devices ($<$10 KiB RAM) lack an IP stack and require gateway proxies; Class 1 devices ($\approx$10 KiB RAM) support lightweight stacks such as 6LoWPAN and CoAP; Class 2 devices ($\approx$50 KiB RAM) can run a full IPv6 stack with MQTT. These hardware tiers directly determine which MTD techniques can be deployed at the device level, as analysed in Section 3.

2.2 IoT Threat Landscape

The combination of large scale deployment and limited security makes IoT systems an attractive target for adversaries. Several categories of attack are particularly relevant to the network layer defenses studied in this thesis.

Reconnaissance and network scanning : Before launching an attack, adversaries typically probe a target network to discover active hosts, open ports, and running services. IoT devices operating persistent services on predictable IP addresses and ports are straightforward to fingerprint. Studies show that mass scanning tools can map large IoT networks within minutes using SYN scanning, banner grabbing, and ICMP probing [49]. The predictability of static IoT configurations is the root cause that MTD directly addresses.

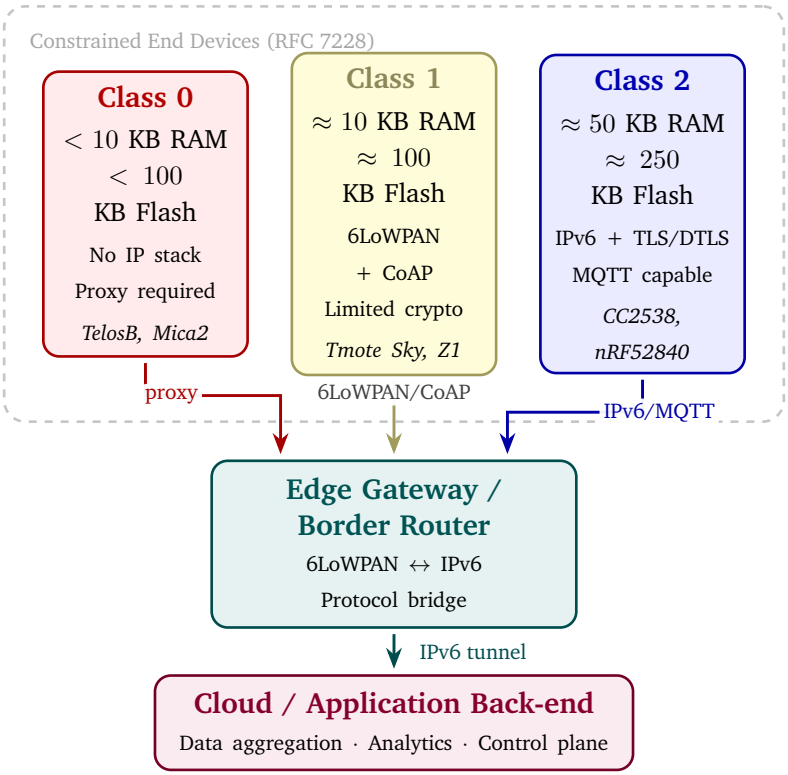

Figure 1: IoT device classes and their communication path through the edge gateway.

Distributed Denial of Service (DDoS) : Once a pool of IoT devices is compromised, they are commonly recruited into botnets that flood target servers or network links with traffic. The Mirai botnet (2016) demonstrated the scale of this threat: it leveraged tens of thousands of IoT devices with default credentials to conduct record breaking volumetric attacks [60]. IoT devices are particularly susceptible to botnet recruitment because they often ship with weak authentication defaults and receive firmware updates rarely.

Man-in-the-Middle (MitM) and eavesdropping : IoT communication frequently traverses wireless links such as IEEE 802.15.4, Wi-Fi, and Bluetooth Low Energy, all of which can be intercepted by a passive attacker within radio range. If the communication channel uses predictable addresses or no encryption, an attacker can read sensor data, inject malicious commands, or impersonate a legitimate node [49].

Replay and injection attacks : In protocols without robust replay protection, an attacker who captures a valid command or authentication token can retransmit it later to trigger unintended actions. This is especially relevant to lightweight IoT protocols such as CoAP and MQTT, where constrained implementations sometimes omit sequence numbers or timestamps to reduce overhead [34].

The common thread across these attack classes is a dependency on stable, discoverable network endpoints. Reconnaissance requires consistent IP addresses. DDoS requires persistent targets. MitM requires predictable communication channels. MTD directly disrupts this de-

pendency by making endpoint identifiers unstable and continuously changing.

2.3 IoT Communication Protocols

Network layer MTD techniques operate by mutating the network identifiers and communication parameters used by IoT devices. Understanding the relevant protocol stack is essential for evaluating which parameters can be moved and what constraints apply.

IPv6 and 6LoWPAN : IPv6 provides a 128-bit address space sufficient to uniquely address every IoT device without address translation. For low power radio networks operating over IEEE 802.15.4, the 6LoWPAN adaptation layer [44] compresses IPv6 headers to fit within the 127 byte maximum frame size, making full IPv6 connectivity practical on Class 1 and Class 2 devices. The enormous IPv6 address space is directly relevant to MTD: schemes such as MT6D and μMTD [72] exploit it to cycle through a large pool of pseudorandom interface identifiers, making brute force rediscovery by an attacker computationally infeasible.

CoAP : The Constrained Application Protocol (CoAP) [63] is a RESTful application layer protocol designed for constrained devices and networks. CoAP runs over UDP, avoiding the overhead of TCP connection establishment, and uses a compact binary header. Resources on a CoAP server are identified by URI paths, and the protocol supports an observe extension for push style data delivery. Because CoAP UDP port numbers and URI paths are visible to any network observer, they constitute an application layer attack surface that MTD techniques can target through port hopping and URI randomization [48].

MQTT : Message Queuing Telemetry Transport (MQTT) is a publish-subscribe protocol widely used for IoT telemetry. Devices publish messages to named topics on a central broker; subscribers receive all messages on topics they have registered. MQTT typically runs over TCP port 1883 (or 8883 for TLS). The broker architecture concentrates traffic through a single well known endpoint, making it a high value reconnaissance target. MTD approaches have therefore explored rotating broker addresses, connection ports, and topic names to reduce the exploitability of this centralized design [34].

2.4 Moving Target Defense

Traditional cybersecurity mechanisms such as firewalls, intrusion detection systems, and patch management are fundamentally *reactive*: they respond to threats after they have been identified. Moving Target Defense (MTD) inverts this paradigm. The U.S. Department of Homeland Security defines MTD as the concept of "controlling change across multiple system dimensions in order to increase uncertainty and apparent complexity for attackers, reduce their window of opportunity, and increase system resiliency" [73]. In practice, this

means that instead of presenting a fixed attack surface, a system continuously shifts its configuration so that information gathered by an attacker during reconnaissance becomes stale before it can be exploited.

The fundamental intuition behind MTD is straightforward: if a system's properties change faster than an adversary can observe, analyze, and act upon them, then the attacker's advantage is neutralized. Gurung et al. frame this as making the system "less homogeneous, static, or deterministic" [23]. Saputro et al. further emphasize that MTD "dynamically shifts the system's attack surface by leveraging redundancy and diversity to make the attack surface unpredictable for attackers" [60]. Figure 2 illustrates this contrast between a static system and an MTD-protected network.

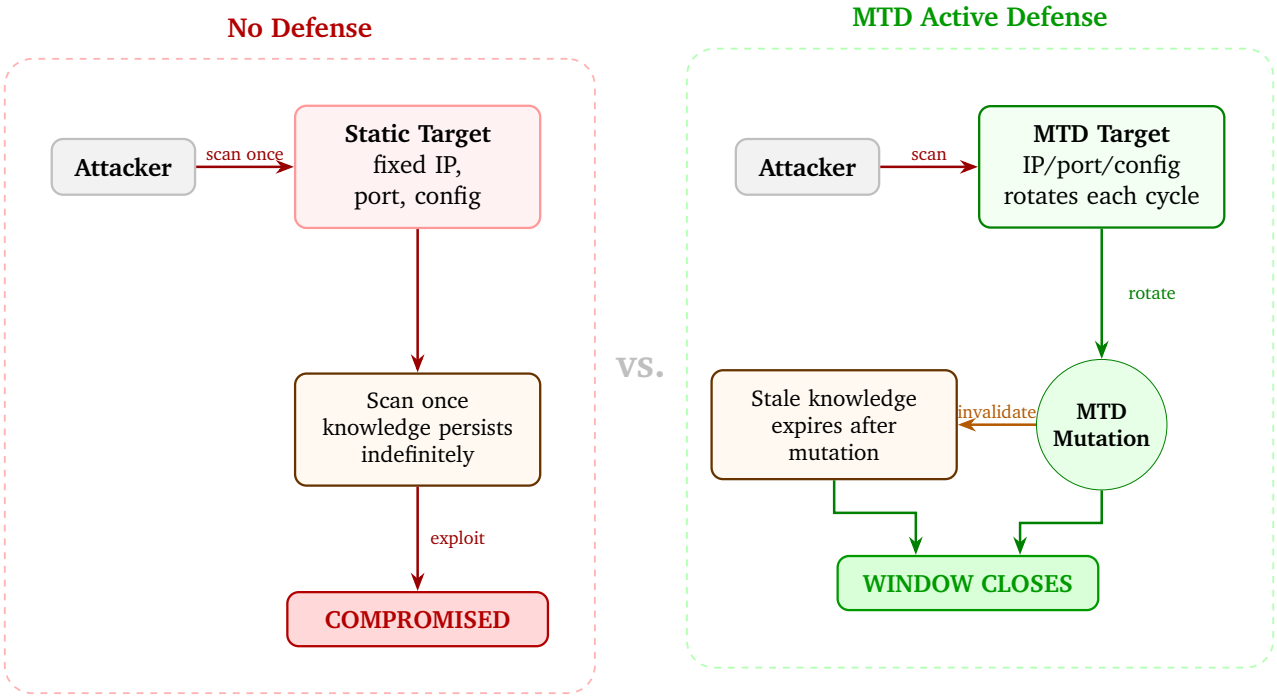

Figure 2: Comparative analysis: Static systems vs. Moving Target Defense.

2.4.1 The MTD Design Space: What, How, and When to Move

Any MTD mechanism must answer three fundamental questions [23, 73]:

1. **What to move**: Which system attribute should be altered? The choice depends on what is valuable to the attacker but expendable for legitimate operation.
2. **How to move**: What method should be used to perform the mutation? Common strategies include *shuffling*, *diversity*, and *redundancy*.
3. **When to move**: What triggers the change? Options range from periodic schedules to event-driven responses.

Table 1 summarizes these dimensions and their relationship to IoT relevant techniques drawn from the surveyed literature.

Table 1: MTD Design Space: Taxonomy of Dimensions and IoT-Relevant Techniques

Dimension	Categories	IoT-Relevant Examples
What to Move	Network identity	IP address shuffling [53], MAC mutation [38]
	Network path / topology	SDN-based topology shuffling [17]
	Platform / runtime	ASLR, OS rotation [62]
	Application / data	Port hopping [51], protocol diversification
How to Move	Shuffling	Randomize existing configurations (e.g., IP addresses, ports)
	Diversity	Deploy heterogeneous variants (e.g., multi OS)
	Redundancy	Replicate components for failover and deception
When to Move	Time-based (periodic)	Fixed interval mutation; predictable overhead [17]
	Event-based (reactive)	Triggered by IDS alerts; lower overhead but depends on detection [51]
	Hybrid	Combines periodic and event-based; best coverage [23]

2.4.2 The Challenge of MTD in Resource Constrained IoT

As introduced in Section 2.1 (Figure 1), RFC 7228 classifies constrained devices into Class 0 through Class 2 by available RAM and storage.

These constraints impose hard limits on what MTD techniques are feasible. Gurung et al. note that "MTD techniques cannot be directly applied to IoT due to limitations of power and computation capacity in devices" [23]. A key finding from Navas et al.'s systematic review is that only 34% of existing IoT-MTD techniques employ current cryptographic primitives, with 66% relying on either deprecated cryptography or none at all [49]. This raises serious concerns about the security guarantees of deployed MTD mechanisms.

2.5 Taxonomy of MTD Techniques

A widely adopted taxonomy [49, 73] categorises MTD techniques by the system layer at which mutation occurs. Dynamic network techniques mutate IP addresses, MAC addresses, or port numbers and are the most practical for IoT since mutations can be offloaded to a gateway or SDN controller. Dynamic platform techniques diversify or rotate the operating system and firmware image. Dynamic runtime techniques apply code randomisation and address-space layout randomisation (ASLR). Dynamic software techniques vary application-layer configurations and service identifiers. Dynamic data techniques mutate data encoding and format. For constrained IoT devices, dynamic network techniques dominate because they impose no per device computation overhead. Chapter 3 reviews how each category has been applied in the IoT MTD literature.

3 Related Work

This chapter presents a systematic review of existing research on Moving Target Defense (MTD) techniques for Internet of Things (IoT) and resource constrained network environments. The review follows the PRISMA 2020 guidelines [57] and is structured to provide a comprehensive, evidence based overview of the current state of the art. Sections 3.1 and 3.2 describe the review methodology: the databases queried, the keyword strategy, and the multi stage screening process with inclusion and exclusion criteria, together with the PRISMA 2020 selection flowchart. Sections 3.4 through 3.8 present the reviewed literature of the reviewed papers, organised by technique layer: network layer MTD (Section 3.5), cyber-deception and honeypot augmented approaches (Section 3.6), application and data layer mechanisms (Section 3.7), and optimisation driven adaptive frameworks (Section 3.8).

3.1 Search Process

To identify relevant research on MTD mechanisms for IoT systems, a structured literature search was conducted across multiple scientific digital libraries. The goal of this search was to collect peer reviewed academic publications that investigate MTD techniques, how they're applicable in IoT environments, and their evaluation under different attack scenarios.

The literature search was performed using four major academic databases widely used in computer science and cybersecurity research:

- IEEE Xplore
- ACM Digital Library
- Scopus
- Web of Science

These databases were selected because they provide extensive coverage of peer reviewed journal articles and conference proceedings in networking, cybersecurity, and distributed systems research.

A keyword based search strategy was applied to identify relevant publications. The primary search query combined terms related to Moving Target Defense and IoT systems as follows:

```
("moving target defence" OR "moving target defense")
AND
("IoT" OR "Internet of Things")
```

The search was performed across the title, abstract, and keyword fields of the databases where possible. All retrieved records were exported in BibTeX format and combined into a

single reference dataset for further processing. The search was restricted to publications from 2015 to 2026.

Although the primary query is intentionally focused, it was designed to maximise recall for core MTD-IoT literature without inflating results with adjacent fields. Work employing adjacent terminology such as *address hopping*, *cyber deception*, *6LoWPAN security*, or *SDN-based defense* was captured through citation snowballing of the 10 manually added records and through the backward references of highly cited survey papers. This two phase strategy ensured that relevant papers using non standard terminology were not systematically excluded.

After merging the search results from all four databases, a total of **382 records** were retrieved. In addition, **10 records** were identified through backward and forward citation snowballing of the most highly cited papers in the MTD-IoT domain. All 10 records that advanced beyond title screening were peer-reviewed publications, consistent with IC4. This yielded a combined initial pool of **392 records** as candidate publications related to MTD and IoT systems.

The collected references were then subjected to a structured screening process in order to remove duplicates and filter papers based on predefined inclusion and exclusion criteria. The detailed filtering and screening steps are described in the following subsection.

The screening and selection process follows the PRISMA-S checklist [59], which is an extension of the PRISMA 2020 statement [57] designed to support reproducible and transparent reporting of literature searches in systematic reviews.

3.2 Selection Process

After collecting the initial set of publications from the selected databases, a multi stage screening process was applied to identify the studies that were relevant to the objectives of this research.

As described in Section 3.1, the initial search across IEEE Xplore, ACM Digital Library, Scopus, and Web of Science yielded **382 records**, supplemented by **10 records** from manual searching, for a combined total of **392 records**. Since results were retrieved from multiple databases, duplicate entries were present. Deduplication was performed using the reference management tool JabRef, removing **151 duplicate records** and leaving **241 unique publications** for further screening.

The remaining papers were then evaluated through a staged filtering process consisting of title screening, abstract screening, and full text assessment. During the title screening stage, papers that were clearly unrelated to MTD or IoT systems were removed. In the next stage, abstracts of the remaining papers were reviewed to determine whether the study proposed, analyzed, or evaluated MTD mechanisms in IoT environments.

The following inclusion criteria **(IC)** were applied during the selection process:

- **IC 1 -** The paper proposes, evaluates, or analyzes a Moving Target Defense technique.
- **IC 2 -** The work focuses on Internet of Things systems or resource constrained network environments.
- **IC 3 -** The study includes a technical contribution, such as a mechanism, framework, or evaluation.
- **IC 4 -** The publication is a peer reviewed journal article or conference paper.
- **IC 5 -** The paper is written in English.

The following exclusion criteria **(EC)** were also applied:

- **EC 1 -** The paper does not propose, evaluate, or analyse a Moving Target Defense mechanism; this includes work focused on AI/ML-only defenses, general intrusion detection, or network hardening without an MTD component at its core.
- **EC 2 -** The paper does not address IoT or resource-constrained network environments; scope is limited to cloud, enterprise, or general purpose systems without demonstrated relevance to constrained devices.
- **EC 3 -** The full text was not accessible for review (unavailable through institutional access, open repositories, or author contact).
- **EC 4 -** Short paper, poster, tutorial, workshop abstract, or editorial content lacking a substantial technical or experimental contribution.
- **EC 5 -** The paper presents a design or conceptual model but provides no implementation, testbed evaluation, or empirical performance results sufficient to assess the claimed security benefit.

To ensure consistency and traceability, a structured screening spreadsheet was used to record decisions for each paper. Additionally, each paper was categorized into one of three relevance levels: *highly relevant*, *relevant*, or *not relevant*, based on its alignment with the research objectives.

Papers classified as highly relevant directly addressed MTD mechanisms in IoT systems with clear technical contributions. Relevant papers included those that addressed MTD or IoT security more generally but provided useful contextual or supporting insights. Papers classified as not relevant were excluded from further analysis. Based on this staged filtering process, a total of **63 papers** were selected for full inclusion in this review. Of the 112 papers assessed for eligibility, **49 papers** were excluded at the full-text review stage: 18 lacked a core MTD mechanism (EC1), 15 had no IoT context (EC2), 8 were unavailable in full text (EC3/EC4), and 8 provided insufficient experimental evaluation (EC5).

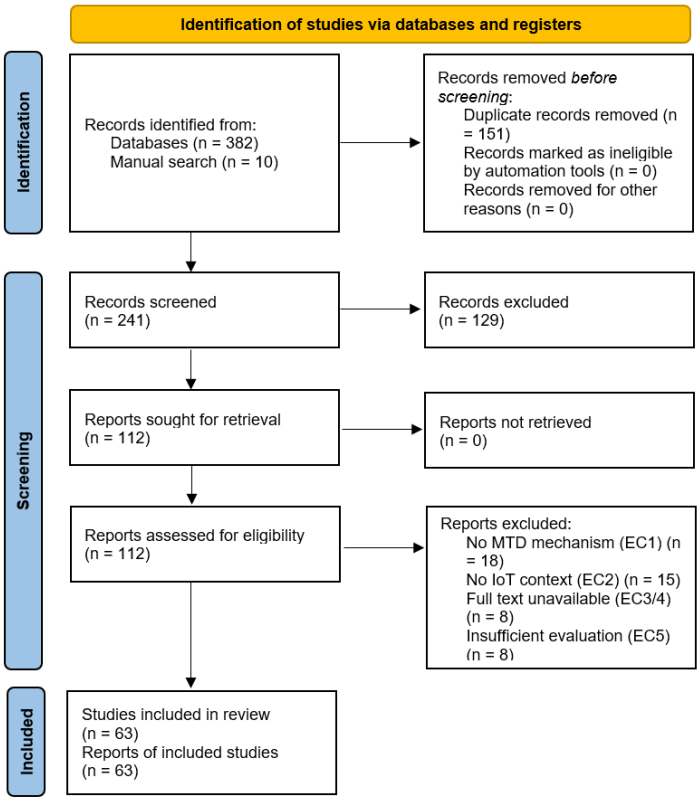

Figure 3: Systematic Literature Review Flowchart

These papers form the basis for the literature review and comparison analysis presented in this thesis.

3.3 Overview of Reviewed MTD Techniques

Table 2 provides a structured overview of the 63 selected studies, grouped by MTD technique. For each technique group, the table identifies the moving parameter, the MTD strategy employed (shuffling, diversity, or redundancy), the system layer taxonomy, the target application domain, and the type of evaluation evidence available. The subsequent subsections analyse each technique category in detail.

Across the 63 included studies, network layer techniques account for the largest share: 28 papers address IP address shuffling, SDN-based topology mutation, MAC-layer randomisation, or DDoS mitigation via MTD. Deception augmented approaches (honeypots and phantom topologies combined with movement) account for 8 papers. Application, data, and infrastructure layer mechanisms, including protocol diversification, key management, and smart-grid CPS defenses, account for 14 papers. The remaining 13 papers address optimisation driven or adaptive decision making (game theory, reinforcement learning, and hybrid frameworks), while the 10 survey and taxonomy papers are grouped separately. In terms of evaluation evidence, 41 of the 63 papers rely exclusively on simulation or emulation; only 12 include

experiments on physical constrained hardware. Cross layer designs that combine two or more mutually reinforcing MTD mechanisms are present in only 7 papers. These distributions directly motivate the research gap addressed in this thesis.

Table 2: Summary of MTD Techniques for IoT from the Reviewed Literature

Ref.	Technique	Mov. Parameter	Strat.	Tax.[a]	Domain	Eval.[b]	Year
IPv6 Address Shuffling / Randomisation							
[72]	Micro-MTD IPv6 Defense	IPv6 address	Shuf.	N	IoT	H	2017
[32]	Lightweight IPv6 Addr. Hopping	IPv6 address	Shuf.	N	IoT	H	2017
[31]	Addr. Shuffling Assessment	IPv6 address	Shuf.	N	IoT	H	2018
[53]	ASHA: Addr. Shuffling	IPv6 address	Shuf.	N	IoT	H	2019
[48]	IANVS: MTD Framework	IPv6 address	Shuf.	N	IoT	H	2020
[70]	SDN-Based Addr. Hopping	IPv6 address	Shuf.	N	IoT	S	2025
Multi-Parameter Network Hopping (IP + Port)							
[52]	OpenFlow Sensor MTD	IP addr., port	Shuf.	N	IoT	S	2020
[7]	Lightweight Proactive MTD	IP addr., port	Shuf.	N	IoT	S	2021
[43]	IoT MTD Strategy	IP addr., port	Shuf.	N	IoT	H	2021
[35]	MTDIoT: Event/Time MTD	IP addr., port	Shuf.	N	IoT	H	2023
[51]	Lightweight MTD + IDS	IP addr., port	Shuf.	N	IoT	H	2024
Port Hopping							
[10]	eMTD: Energy-Aware MTD	Port number	Shuf.	N	Smart Farm	S	2024
Topology Shuffling with Cyberdeception							
[41]	Pot2DPI: Honeypot + MTD	Port fwd., topology	Shuf.	N	Home IoT	T, H	2017
[26]	HIoT: Honeypot-like MTD	Virtual module roles	Red.	N	IoT	H	2018
[37]	SDN Honeypot + MTD	Topology, honeypot	Shuf., Red.	N	IoT	S	2019
[17]	Proactive MTD + Deception	Network topology	Shuf.	N	IoT	T, S	2021
[15]	MTD for SDN IoT-Edge	Topology, decoy	Shuf., Red.	N	IoT-Edge	S	2022
[58]	Diversity MTD + Deception	Topology, diversity	Div., Red.	N	IoT	S	2024
[2]	Self-Adaptive MTD + Decoy	Topology, decoy	Shuf., Red.	N	IoT	S	2025
SDN-Based Network Reconfiguration							
[36]	AEH-MTD: Adaptive Entropy MTD	IP addr., config	Shuf.	N	IoT	S	2019
[67]	SDN + MTD DDoS Defense	Network config	Shuf.	N	IoT	S	2025
Route / Path Mutation							
[71]	Proactive Route Hopping	Routing path	Shuf.	N	SDN/IoT	S	2024
[74]	SARM: Routing Mutation	Routing path	Shuf.	N	Power IoT	S	2024
MTD-Based DDoS Mitigation							
[75]	Proactive DDoS Mitigation	IP, service replica	Shuf., Div.	N, P	IIoT	S	2022
[56]	Optimised MTD vs DDoS	IP, port	Shuf.	N	IoT	S	2022
[16]	SNMP + MTD DDoS Defense	IP, ACL rules	Shuf.	N	IoT	S	2023
[14]	Markov Game DDoS Defense	Network config	Shuf.	N	IoT	S	2024
[30]	RL-Based Edge IoT DDoS MTD	Network nodes	Shuf.	N	Edge IoT	S	2025
[64]	Adaptive MTD for IIoT DDoS	Service placement	Shuf.	N	IIoT	S	2025
MAC-Layer Randomisation							
[39]	MAC Spoofing MTD	Device fingerprint	Shuf.	N	IoT	H	2020
[38]	Randomised MAC-Layer MTD	MAC identity	Shuf.	N	IoT	H	2022
Physical-Layer Frequency Hopping / Anti-Jamming							
[50]	DSSS Anti-Jamming for IoT	Spreading sequence	Shuf.	N	IoT	S	2021
[11]	BlueShift: Bluetooth AFH	Frequency channel	Shuf.	N	BLE/IoT	H	2023
MQTT Protocol MTD							

Continued on next page

Table 2 – *Continued from previous page*

Ref.	Technique	Mov. Parameter	Strat.	Tax.[a]	Domain	Eval.[b]	Year
[34]	MQTT-MTD: Topic Tokenisation	Topic, payload	Shuf.	D	IoT	S	2024
[66]	VM Migration as MQTT MTD	Broker VM location	Shuf.	P	IoT	H	2025
Code Diversification / Software MTD							
[40]	Code Partitioning + Diversif.	Binary code layout	Div.	S	IoT	S	2016
[4]	Lightweight MTD vs Malware	Network, data, runtime	Div.	S, N	IoT	H	2023
Dynamic Key Management							
[6]	PUF-Based Key Agreement	Crypto key	Shuf.	D	IoT	H	2024
[33]	Dynamic Key Replacement	Encryption key	Shuf.	D	IoT	H	2025
Data Replication / Signal Randomisation							
[19]	MTD-IDR: Data Replication	Sensor/control data	Red.	D	CPS	T, S	2022
[18]	Decentralised MTD Privacy	Data replication paths	Red.	D	Microgrid	S	2022
VM / Container Migration							
[13]	MTD for Cloud/Edge Telco	VM/container	Shuf.	P	Cloud/Edge	S	2023
Smart Grid MTD							
[68]	Stability-Guaranteed MTD	Grid impedance	Shuf.	D	Smart Grid	T	2025
[12]	MTD in the Smart Grid	Transmission params	Shuf.	N, D	Smart Grid	T	2026
Game-Theoretic / RL-Based MTD Optimisation							
[69]	Zero-Determinant MTD	MTD config	Hybrid	X	IoT	T	2020
[61]	IoDM: Deception Modelling	Decoy config	Div.	X	IoT	S	2022
[5]	RL MTD Strategy Selection	MTD mechanism	Hybrid	X	IoT	H	2023
[29]	RL + Fingerprinting MTD	MTD mechanism	Hybrid	X	IoT	H	2024
[55]	RL-Based MTD for IDS	IDS config, network	Hybrid	X	IoT	S	2024
Model-Based / Hybrid Multi-Technique MTD							
[9]	Model-Based MTD for CPS	Multiple params	Hybrid	X	CPS	T	2024
[45]	MTD + Task Distribution	Task allocation	Div.	S	IoT	H	2025
[42]	Hybrid MTD (THARM)	Multiple params	Hybrid	X	IoT	T, S	2025
[62]	MTD Eval. (TTC/Risk)	IP, OS, config	Shuf., Div.	X	IoT	S	2025
Surveys, Reviews, and Taxonomies							
[73]	MTD Strategies Survey	–	–	–	General	–	2019
[60]	MTD for IoT Review	–	–	–	IoT	–	2020
[49]	MTD Where Art Thou (SLR)	–	–	–	IoT	–	2021
[47]	MTD for IoT (Chapter)	–	–	–	IoT	–	2022
[22]	MTD for IoT Threats	–	–	–	IoT	–	2022
[23]	MTD Challenges for IoT	–	–	–	IoT	–	2023
[1]	SDN Security + MTD Review	–	–	–	SDN/IoT	–	2024
[25]	AI + MTD for IoT Review	–	–	–	IoT	–	2024
[27]	MTD + Trust for IoT/SIoT	–	–	–	IoT/SIoT	–	2024
[24]	Runtime MTD for IoT/CPS	–	–	–	IoT/CPS	–	2026

[a]Taxonomy: N = Network, P = Platform, S = Software, D = Data, R = Runtime Env., X = Cross-layer.
[b]Evaluation: T = Theoretical, S = Simulation, H = Hardware testbed.

3.4 Surveys, Taxonomies, and Systematic Reviews

Before examining individual MTD techniques, it is important to start this SLR within the broader body of review literature. Many surveys and taxonomies have attempted to systematise the MTD field, and understanding their scope, emphasis, and limitations directly motivates the contribution of this thesis.

A widely cited survey by Zheng and Namin [73] established a foundational architecture level taxonomy, categorising MTD mechanisms across five parameters: *what* is moved (network

identity, platform, software, data), *how* it is moved (shuffling, diversity, redundancy), *when* it is moved (periodic, event-driven, hybrid), *where* the mechanism operates (physical, network, OS, application layer), and *why* it is deployed (prevent reconnaissance, disrupt exploitation, limit propagation). This taxonomy has since become the widely adopted reference framework for subsequent work. However, the survey predates the rapid growth of IoT specific MTD research and does not address constrained device deployments or lightweight implementations.

Saputro et al. [60] focused specifically on IoT, producing the first dedicated review of MTD mechanisms for IoT applications. Their key contributions included:

- Identifying *shuffling* as the dominant MTD strategy in IoT contexts due to its low overhead relative to diversity or redundancy approaches.
- Highlighting that network layer techniques (IP address mutation, port hopping) were the most studied, while application and data layer MTD remained underexplored.

Navas et al. [49] complemented this with an SLR methodology, systematically surveying IoT security and MTD literature using PRISMA-style search protocols. Their review emphasised that the combination of resource constraints, heterogeneous protocols, and large attack surfaces makes IoT a uniquely challenging domain for MTD deployment. The same group later contributed a chapter in a book [47] extending this analysis to cover emerging 6LoWPAN and constrained RESTful environments.

By 2022, the field had grown substantially. Grammatikakis et al. [22] produced a comprehensive taxonomy covering over 100 MTD papers, explicitly mapping technique categories to the MITRE ATT&CK framework. Importantly, they highlighted that:

- Most MTD techniques address the reconnaissance and lateral movement phases of an attack, while persistence and exfiltration stages remain underaddressed.
- IoT-specific challenges such as energy harvesting, battery constraints, and intermittent connectivity are rarely modelled in evaluation frameworks.

On the SDN-IoT intersection, Abdi et al. [1] surveyed how software defined networking can enable MTD in IoT, noting that SDN's centralised control plane naturally supports proactive address shuffling and topology changes without overhead on individual devices.

More recent reviews have focused on trust and security maturity. Hamada et al. [25] reviewed proactive defense mechanisms for IoT more broadly, placing MTD within a wider landscape of moving targets, honeypots, and deception based defenses. Hamida et al. [27] specifically examined trust management for MTD-enabled IoT, noting that frequent configuration changes risk disrupting legitimate communication channels unless trust bootstrapping mechanisms are designed in parallel with the MTD system.

Gurung et al. produced two contributions that directly inform this SLR. The 2023 paper [23]

gives the practical challenges of deploying MTD in IoT settings, including:

- **Overhead**: Even lightweight shuffling mechanisms add latency and computation; Class 0 and Class 1 devices [8] may be unable to participate.
- **Synchronisation**: Moving network parameters requires that legitimate devices track changes; failure to synchronise causes denial-of-service to friendly nodes.
- **Detectability**: Periodic MTD movements can themselves be fingerprinted by adversaries, negating the unpredictability benefit.

The 2026 runtime analysis [24] extended this by empirically evaluating address space layout randomisation (ASLR) on ARM-based IoT platforms, confirming that constrained devices can support a subset of MTD techniques if implementations are carefully tuned.

Collectively, the survey literature converges on three unresolved gaps that this SLR specifically addresses: (1) a lack of standardised evaluation metrics that enable cross study comparison, (2) insufficient coverage of multi layer and hybrid MTD approaches that combine network, application, and infrastructure techniques. The subsequent subsections examine how primary research papers have addressed or failed to address these gaps.

3.5 Network Layer MTD Techniques

Network layer MTD is the most extensively studied category in the IoT MTD literature. These techniques randomise network properties such as IP addresses, MAC identifiers, port numbers, routing paths, or physical layer channel parameters to deny adversaries the access to persistent reconnaissance data. Because network parameters are modifiable at the firmware or OS level without altering device functionality, they are particularly well suited to resource-constrained IoT nodes. This section organises network layer techniques into four groups, ordered from the most granular to the broadest scope.

3.5.1 Address and Parameter Shuffling

IPv6 address mutation is the single most studied MTD technique for IoT, motivated by the fact that IPv6's large address space provides vast shuffling entropy. In contrast to IPv4, where the address pool is too small for effective randomisation, a single /64 prefix supports 2^{64} host addresses, making brute-force scanning computationally infeasible if addresses are rotated frequently enough.

Foundational IPv6 shuffling schemes: Zeitz et al. [72] introduced μMTD, a micro-scale IPv6 address rotation mechanism for 6LoWPAN networks. The key design insight was that address changes must be coordinated using a shared secret (a pre-distributed key and a hash function), so that legitimate communicating parties can compute each other's next address without a broadcast announcement. Judmayer et al. subsequently published two complementary papers: a protocol design [32] proposing lightweight address hopping for IoT with

minimal control overhead, and a performance assessment [31] that quantified the computational cost on RIOT OS and Contiki devices. Their evaluation showed that address rotation adds less than 1% CPU overhead on Cortex-M3 class hardware, establishing feasibility on Class 1 devices.

Nizzi et al. [53] proposed ASHa (Address Shuffling with HMAC Authentication), which addressed a weakness in earlier schemes: without message authentication, an adversary could inject fake address update messages. ASHa integrates HMAC into each rotation step, adding address mutation with lightweight authentication. Evaluation on a real 6LoWPAN testbed showed that ASHa reduces scan success probability by over 95% relative to a static address baseline while adding only minimal latency.

Navas et al. [48] introduced iANVS (IoT Address and Network Variables Shuffler), a more general framework that extends address shuffling to include port numbers, transport protocols, and even application-layer endpoints. iANVS supports three movement modes:

- **Time-triggered**: address changes occur at fixed intervals.
- **Event-triggered**: changes are triggered by detected anomalies.
- **Hybrid**: combines both modes for adaptive coverage.

Xi et al. [70] presented a more recent SDN-assisted address shuffling model that shifts the coordination logic to the SDN controller, reducing the device overhead further. Their scheme uses a hash chain to generate future addresses deterministically, enabling the controller to pre-program flow rules before each rotation, thereby eliminating the latency spike that is usually seen in reactive approaches.

Multi-parameter hopping: Moving beyond single parameter shuffling, several works randomise multiple network attributes simultaneously to increase the cost of reconnaissance. Niu et al. [52] leveraged OpenFlow SDN to simultaneously randomise IP addresses, port numbers, and MAC addresses across IoT nodes in a smart-home environment, demonstrating that coordinated mutation of various parameters makes traffic fingerprinting significantly harder. Barker et al. [7] proposed a lightweight multi-hop scheme specifically for LPWAN (LoRaWAN) environments where duty-cycle constraints severely limit the frequency of control messages; their solution uses a time-synchronised pseudorandom sequence known only to registered devices.

Mercado-Velázquez et al. [43] evaluated different MTD combinations in a heterogeneous IoT testbed, finding that combining IP and port mutation reduced successful exploit attempts by 78% versus single-parameter approaches, while increasing latency by only 3–5 ms. Kyriakakis et al. [35] developed MTD-IoT, a framework tailored for industrial IoT (IIoT) settings that adds a *diversity* component to standard shuffling: each rotation selects from a pool of heterogeneous address formats, breaking any scanners looking for fixed formats. Swati et al. [64] further extended this to IIoT with a frequency-adaptive scheme that increases rotation

speed under detected threat conditions and slows down during normal operation to reduce overhead, essentially an MTD system controlled by feedback.

Port-only hopping: Chen et al. [10] proposed eMTD, focusing exclusively on port hopping for embedded IoT devices where memory constraints prevent full IP-layer shuffling. eMTD demonstrates that even single parameter port randomisation, when combined with service obfuscation, substantially reduces the reconnaissance success rate on MQTT-based IoT services.

3.5.2 SDN-Based Topology and Route Mutation

While address shuffling operates at the endpoint level, SDN-based approaches move defense logic to the network plane. This enables topology wide reconfiguration from a centralised controller. This has two practical advantages for IoT: (1) the computational burden of tracking address changes is offloaded from constrained devices to the SDN controller, and (2) the controller can implement consistent network-wide policies across different device types. Liu et al. [36] proposed AEH (Adaptive Endpoint Hopping), an SDN-enabled scheme for smart-grid IoT that periodically reconfigures virtual IPs and port assignments at the controller level. By programming OpenFlow rules proactively, AEH ensures that legitimate traffic is transparently forwarded to the new endpoint, while attackers holding stale addresses experience connection failures. Evaluation on an OpenDaylight-based testbed showed AEH reduced successful scan-to-exploit chains by 91% with under 10 ms additional latency for legitimate flows.

Vlădescu and Bica [67] demonstrated an SDN-based MTD implementation that extends address hopping to the IoT gateway layer. They used the SDN controller as a mediator that translates between stable internal addresses and externally visible ephemeral ones. This architecture is particularly relevant for IoT deployments where devices cannot run sophisticated MTD logic locally.

At the routing level, Yan et al. [71] introduced a route mutation scheme for IoT mesh networks that periodically changes the forwarding paths used by data flows, thereby disrupting traffic analysis attacks that rely on consistent flow paths. Unlike address shuffling, route mutation is transparent to endpoints but alters the physical paths through the network, making it a complementary technique rather than a substitute technique. Zheng et al. [74] proposed SARM (Stochastic Adaptive Route Mutation), which uses a Markov decision process to select routing changes that maximise unpredictability while minimising the expected additional hop count.

3.5.3 MAC and Physical-Layer Randomisation

A less-studied but practically important category involves moving targets at layers below IP: MAC addresses and physical-layer channel parameters. These mechanisms are relevant when attackers operate at the link layer (e.g., ARP spoofing, link-layer sniffing) or use radio-frequency analysis to identify devices.

Madani et al. published two complementary papers on MAC-layer MTD. The first [39] presented a theoretical framework for MAC address randomisation in IEEE 802.15.4 networks, analysing the entropy achievable under different rotation schedules and the risk of MAC address collisions. The follow-up [38] implemented and evaluated this framework on real Zigbee hardware, showing that periodic MAC rotation reduces device trackability by external observers by over 80% with negligible impact on association time for legitimate network participants.

At the physical layer, Navas et al. [50] proposed frequency hopping as an MTD mechanism for IEEE 802.15.4 IoT devices, exploiting the multi-channel capability of the standard to hop across channels using a shared secret sequence. This makes passive eavesdropping and jamming significantly harder. Chin et al. [11] extended this concept to Bluetooth Low Energy (BLE) with BlueShift, a channel-hopping scheme that adapts the hopping sequence based on observed interference levels, simultaneously improving resilience against both eavesdropping and unintentional radio interference.

3.5.4 DDoS Mitigation via MTD

Distributed Denial of Service (DDoS) attacks represent a specific and severe threat to IoT networks, given the ease with which resource-constrained devices can be overwhelmed. MTD-based DDoS mitigation takes a proactive stance: by randomising endpoints, routing, and traffic patterns, the attack surface exposed to DDoS campaigns is continuously reduced, making it harder for botnets to maintain coordinated targeting.

Zhou et al. [75] proposed an IP-hopping-based DDoS mitigation framework for IoT that combines address shuffling with traffic rate monitoring. When anomalous traffic rates are detected, the framework triggers an emergency IP rotation, severing the attacker's targeting lock within a single round-trip time. Osei et al. [56] developed an optimised MTD strategy for DDoS mitigation using a cost-benefit model that weighs the disruption cost of rotation (momentary packet loss for legitimate traffic) against the benefit (DDoS traffic rejection), finding an optimal rotation frequency as a function of observed attack intensity.

Gayathri et al. [16] integrated MTD with SNMP-based network monitoring, enabling the network management system to detect DDoS precursors (ARP floods, ICMP sweeps) and proactively rotate network identifiers before the volumetric attack phase begins. Feng et al. [14] modelled DDoS-MTD interaction as a Markov game, deriving a closed-form expression for the optimal MTD rotation rate as a function of the attacker's scanning speed and the

network's tolerance for self-disruption from address changes. This game-theoretic grounding connects the DDoS mitigation literature with the broader optimisation work reviewed in Section 3.8.

Collectively, the network-layer literature demonstrates that MTD is technically feasible even on resource-constrained IoT hardware, provided implementations are carefully tuned for the target device class. However, as the survey literature noted, most evaluations remain simulation-based, and the transition to heterogeneous hardware testbeds with real 6LoWPAN stacks, real radio interference, and real timing constraints remains incomplete. The following sections examine how MTD researchers have extended these network-layer ideas into the application layer and beyond.

3.6 Cyber-Deception and Honeypot-Augmented MTD

A natural evolution of network-layer shuffling is to combine movement with active deception: rather than merely randomising legitimate system properties, deception-based MTD deploys *fake* assets such as honeypots, decoy services, and phantom topologies to actively mislead attackers. This shifts the defender from a reactive posture to a proactive one, not only denying reconnaissance data but feeding the attacker false information that wastes attack resources.

Martin and Alpcan [41] were among the first to formalise the combination of topology shuffling with honeypot deployment for IoT. Their framework models the defender as managing a mixture of real and fake nodes, with the ratio and positioning of decoys adjusted dynamically based on observed threat intelligence. The key insight is that an attacker who cannot distinguish real from fake endpoints must waste scanning and exploitation attempts on decoys, substantially increasing the time-to-compromise of real devices.

Honeypot deployment strategies received dedicated attention in two complementary works. Hamada et al. [26] developed a lightweight honeypot system for IoT gateways that mimics the behaviour of constrained devices, capturing reconnaissance traffic without the overhead of full device emulation. Luo et al. [37] extended this by combining honeypot deployment with address shuffling, creating a *moving honeypot* paradigm in which both the real device addresses and the fake honeypot addresses change simultaneously, preventing an attacker from learning which addresses to avoid even if they identify the honeypot through behavioural analysis.

Ge et al. [17] proposed a comprehensive proactive defense framework that integrates MTD with cyber-deception for IoT, formalising the interaction using game theory. Their model defines:

- **Deception layer**: Honeypots and fake services that attract and delay attackers.
- **MTD layer**: Address and topology shuffling that invalidates attacker reconnaissance.

- **Decision layer**: A game theoretic controller that decides when to engage deception versus when to shuffle, based on observed attacker behaviour.

Evaluation on a simulated smart-home environment showed that the combined framework achieves significantly lower compromise rates than either technique in isolation, confirming the complementary nature of deception and movement.

Galadima et al. [15] applied deception-augmented MTD to industrial IoT (IIoT), where the stakes of a successful attack are particularly high. Their system deploys decoy PLCs (Programmable Logic Controllers) alongside real ones, with the MTD system periodically rotating which devices are reachable via the control network. The decoy PLCs respond with plausible but subtly incorrect process data, wasting attacker time on false targets.

More recent work has focused on practical deployment challenges. Rehman et al. [58] evaluated proactive deception MTD in a real IoT testbed, measuring the operational overhead of maintaining honeypot infrastructure alongside production devices. Their results showed that deception based MTD adds approximately 8–12% memory overhead on gateway devices, which they argue is acceptable given the disproportionate increase in attacker cost. Alkasassbeh et al. [2] proposed SAS (Shuffle-and-Spoof), a unified framework that tightly integrates address shuffling with response spoofing, enabling real-time deception without dedicated honeypot hardware instead, the production devices themselves respond briefly as decoys before rotating to their new address.

The deception based literature highlights an important architectural insight: MTD and deception are *synergistic* rather than competing strategies. Shuffling denies information; deception provides disinformation. Together, they impose a cognitive and computational burden on attackers that neither technique achieves alone. This synergy motivates the multi-layer and hybrid approaches examined in Section 3.8.

3.7 Application, Data, and Infrastructure MTD

While network-layer and deception-based MTD techniques address the reconnaissance and initial access phases of an attack, adversaries who have gained a foothold inside an IoT network may exploit higher-layer vulnerabilities: insecure application protocols, predictable cryptographic keys, or vulnerable software components. Application, data, and infrastructure MTD addresses these attack surfaces by randomising protocol behaviour, diversifying code and cryptographic material, and migrating workloads across virtual infrastructure.

3.7.1 Protocol and Code Diversification

MQTT protocol MTD: MQTT is the dominant messaging protocol for IoT, and its broker-based architecture presents specific attack surfaces: attackers who identify the broker address and topic structure can intercept or inject messages. Kusumi and Koide [34] proposed MQTT-MTD, which integrates MTD directly into the MQTT protocol layer as a lightweight complement to TLS-based security in resource-constrained settings. Their approach rotates broker addresses, port assignments, and topic names on a shared pseudorandom schedule known to authorised clients, making it infeasible for unauthorised listeners to maintain persistent subscriptions. A key advantage over TLS is reduced overhead: MQTT-MTD adds approximately 2 KB of RAM overhead per device versus the 10–30 KB required for a full TLS stack. Torquato and Vieira [66] further analysed MQTT-based MTD from a dependability perspective, modelling the trade-off between rotation frequency and the probability of legitimate message loss during a rotation window.

Software and code diversification: Code diversification applies the *diversity* strategy at the software level: instead of shuffling a single fixed binary, multiple functionally equivalent but structurally distinct software variants are deployed, reducing the effectiveness of code-reuse attacks such as ROP. Mahmood et al. [40] presented one of the earliest applications of code diversification to embedded IoT firmware, demonstrating that automatic compiler-based diversification (via function reordering and stack canary placement) can be applied to Contiki OS firmware with negligible performance impact. Von der Assen et al. [4] revisited this problem for modern constrained devices, proposing a lightweight diversification framework that generates device-specific firmware variants at manufacturing time, ensuring that even identically provisioned devices run structurally distinct code preventing fleet-wide exploitation of a single vulnerability.

3.7.2 Key Management, Data Integrity, and Virtualisation

Dynamic cryptographic key management : Static pre-shared keys are a persistent vulnerability in IoT deployments; once a key is extracted from one device, all devices using the same key are compromised. MTD addresses this by making cryptographic material a moving target. Barbareschi et al. [6] proposed a PUF-based (Physical Unclonable Function) key management scheme for IoT that generates device-unique keys from hardware manufacturing variations, ensuring that keys cannot be cloned even with physical device access. Critically, the key generation function is re-evaluated periodically with a fresh challenge, effectively rotating the working key without requiring centralised key distribution. Kuo et al. [33] complemented this with a lightweight group key management protocol for IoT clusters that uses MTD-style key rotation to limit the window of exposure when a device is compromised, ensuring that a captured key is valid for only a short time window.

Data replication and signal randomisation: In cyber-physical systems (CPS) and smart grids, attackers who cannot penetrate the control layer may instead attempt data injection

attacks: injecting false sensor readings to manipulate actuator behaviour. Giraldo et al. addressed this threat in two related papers, one targeting general CPS [19] and one targeting microgrid control [18]. Their approach applies MTD at the *measurement* level: sensor readings are randomly perturbed within safe operational bounds, and actuator commands are replicated across redundant channels, making it infeasible for an attacker to craft injected data that passes consistency checks across all channels simultaneously. This measurement-level randomisation represents a form of data-layer MTD that is orthogonal to the network and application techniques discussed above.

Virtualisation-based migration: Escaleira et al. [13] applied the MTD principle to cloud-edge IoT infrastructure, proposing a system in which IoT processing workloads are periodically migrated across virtual machine instances and edge nodes. This makes it harder for adversaries to target specific infrastructure components through timing attacks or side-channel analysis of fixed virtual machine placements. Their evaluation showed that live migration of IoT gateway workloads can be achieved in under 200 ms, making the approach transparent to real-time IoT applications.

3.7.3 Smart Grid and Cyber-Physical Systems

Smart grid and CPS deployments represent a specialised application domain where MTD faces unique constraints. Unlike general-purpose IoT, smart grid devices operate under strict timing requirements (e.g., IEC 61850 GOOSE messages with sub-millisecond latency budgets), making frequent address rotations or protocol changes potentially disruptive to grid stability.

Wang et al. [68] proposed an MTD framework specifically designed for smart grid communication networks, where address rotation schedules are synchronised with the grid's supervisory control cycle to avoid interfering with time-critical protection functions. Their system demonstrated that MTD can be deployed in a smart grid without violating IEC 61850 timing requirements, provided rotations are timed to non-critical communication windows. Deng et al. [12] extended this by proposing a co-design approach where the MTD policy is jointly optimised with the smart grid's fault-tolerance mechanisms, ensuring that address rotations do not create orphaned control sessions during grid fault conditions. These smart grid results demonstrate that with careful domain-specific design, MTD is deployable even in the most timing-sensitive industrial IoT environments.

3.8 Optimisation-Driven and Adaptive MTD

The techniques reviewed in Sections 3.5–3.7 answer the question of *what* to move and *how*. A distinct and growing body of literature addresses the complementary question: *when* should a system move, and *which* technique should it choose, given the cost of moving and the uncertain behaviour of attackers? This section reviews game-theoretic, reinforcement learning,

and hybrid frameworks that treat MTD decision-making as an optimisation problem.

3.8.1 Game-Theoretic and Reinforcement Learning Approaches

Game-theoretic formulations: The adversarial nature of MTD where a defender chooses movement strategies in response to attacker strategies, and vice versa maps naturally onto game theory. Wang et al. [69] applied zero-determinant (ZD) strategy theory to MTD for IoT, showing that a defender using a ZD strategy can unilaterally control the expected payoff of the game regardless of the attacker's strategy. This provides a theoretical guarantee: even an arbitrarily sophisticated attacker cannot achieve a payoff above the defender's chosen bound. The practical implication is that a ZD-based MTD controller can limit the attacker's expected success rate below a target threshold.

Seo et al. [61] proposed IODM (IoT-Oriented Dynamic MTD), a framework that models the attacker-defender interaction as a Stackelberg game, where the defender moves first by committing to an MTD policy, and the attacker responds optimally. By anticipating the attacker's best response, the defender can select the MTD policy that minimises the attacker's maximum achievable payoff a minmax approach that is more conservative but more robust than Nash equilibrium solutions.

Reinforcement learning: RL-based approaches remove the assumption of a known attacker model, instead learning optimal MTD policies through interaction with the environment. Huertas Celdrán et al. [29] trained a deep Q-network (DQN) agent to control address shuffling frequency and scope in a dynamic IoT threat environment, showing that the learned policy outperforms fixed-schedule MTD by 23% in terms of attacker time-to-compromise. The RL agent learns to increase rotation speed when threat indicators are elevated and reduce it (to conserve resources) when the threat level is low, a behaviour that emerges naturally from the reward function without explicit programming.

Von der Assen et al. [5] focused on MTD strategy selection (rather than timing), training an RL agent to choose between address shuffling, port hopping, and topology mutation based on observed network state. Their multi-action RL framework is notable because it jointly optimises over both the MTD technique and its parameters, something that game-theoretic approaches struggle with due to the combinatorial action space. Osei et al. [55] extended RL-based MTD to an adversarial setting where the attacker is also modelled as a learning agent, creating a self-play scenario. Their results show that co-trained defender-attacker pairs converge to near-optimal strategies from both sides, providing insight into the fundamental limits of MTD effectiveness against adaptive adversaries.

3.8.2 Hybrid and Model-Based Frameworks

While game-theoretic and RL approaches optimise MTD decisions in isolation, hybrid frameworks integrate MTD with complementary security mechanisms intrusion detection, threat modelling, or formal verification to address limitations of pure MTD approaches.

Casola et al. [9] proposed a model-driven approach to MTD for IoT, where a formal security model of the system is maintained at runtime. The model tracks the current attack surface and computes the minimum set of MTD actions required to close identified vulnerabilities, avoiding unnecessary movements that consume resources without providing security benefit. This model-based approach directly addresses the *overmigration* problem identified in prior surveys: applying MTD too aggressively wastes device energy without proportionate security gain.

Morales et al. [45] developed a task-oriented MTD framework for mobile IoT scenarios (e.g., autonomous vehicles, drones), where the MTD policy must account for the device's current task constraints. For instance, a drone executing a time-critical payload delivery cannot afford the communication disruption of an IP rotation; the framework defers MTD actions to task-boundary windows. This task-aware scheduling represents an important practical extension of the theoretical optimisation frameworks.

Masud et al. [42] proposed a hybrid MTD-IDS architecture in which the MTD system and the intrusion detection system share state: IDS alerts trigger immediate MTD responses, while the MTD system feeds the IDS with ground truth about which addresses are legitimate versus shuffled, reducing the rate of spurious IDS alerts triggered by the MTD movements themselves. This bidirectional integration overcomes a fundamental limitation of standalone MTD that legitimate anomalies (from the MTD movements themselves) can overwhelm IDS alert queues.

Finally, Sharma [62] provides a useful evaluative perspective, systematically comparing multiple MTD methods using time-to-compromise (TTC) and security risk metrics across different IoT network configurations. The study finds that:

- No single MTD technique dominates across all evaluation criteria; the best choice depends on device class, network topology, and the threat model.
- Hybrid approaches (combining multiple MTD techniques) consistently outperform single-technique deployments, but at higher implementation complexity.
- Hardware testbed evaluations show significantly different performance profiles from simulation results, reinforcing the importance of empirical validation.

These findings directly motivate the research question addressed in this thesis: rather than asking which MTD technique is best in isolation, the more productive question is how to select and combine MTD techniques in a principled, context-aware way that accounts for real

IoT hardware constraints.

3.9 Research Gap and Positioning of This Work

The systematic review reveals three persistent gaps that motivate this thesis.

- **No rule-based orchestrator selecting between distinct MTD techniques by threat type on constrained hardware:** Adaptive frameworks in the literature either (a) apply all MTD dimensions simultaneously in response to any trigger [48], or (b) require game-theoretic solvers or reinforcement-learning agents that are computationally infeasible on Class 1 devices [69, 61, 29]. No published work demonstrates a lightweight, rule-based orchestrator that selects *between* three distinct reactive branches—IPv6 address shuffling, application-layer port token rotation, and a composite shuffle plus application-layer rate limit—based on the type of observed threat indicator, while fitting within a Class 1 memory budget without solver-grade computation. The `mtd-orchestrator` module developed in this thesis fills that gap.
- **Lack of combined proactive and reactive triggering:** Most proposals adopt either a periodic shuffle schedule [72, 10] or a purely reactive, event-driven trigger [48], but not both simultaneously. A dual-mode design, a proactive IPv6 shuffle every 120 s and a proactive port token rotation every 300 s, combined with reactive triggering when a typed anomaly counter crosses its threshold, has not been characterised in terms of packet delivery ratio, energy overhead, and detection and reaction latency.
- **Evaluation on realistic constrained hardware:** The majority of MTD proposals are evaluated in high-fidelity network simulators (ns-3, OMNeT++) or on unconstrained Linux hosts. Only a small subset targets Class 1 IoT devices ($\leq$10 KB RAM, $\leq$100 KB Flash), and even these rarely provide reproducible simulation artefacts. This work uses Contiki-NG v5.1 with the Cooja simulator using the `TARGET=sky` build, which runs the same firmware that flashes onto Tmote Sky hardware on the cycle-accurate MSPSim emulator. The energy figures are therefore derived from Energest state-time counters recorded on the cycle-accurate emulator, and the binaries evaluated in Cooja are the same artefacts that would deploy to a Tmote Sky-class device.

Using Contiki-NG v5.1 and Cooja with the `TARGET=sky` build, we implement and evaluate a dual-mode MTD orchestrator that (i) runs entirely within the memory footprint of a Class 1 device, (ii) selects between IPv6 address shuffling, application-layer port token rotation, and a composite shuffle plus rate limit branch based on the type of detected anomaly, and (iii) combines proactive and reactive triggering across a 30-node RPL-Lite DODAG. The evaluation quantifies the resulting security–performance trade-off across attack scenarios including a binary CoAP IPv6 scan, a forged RPL DIO sinkhole approximation, and a binary CoAP flood.

4 System Design

This chapter presents the design of a Moving Target Defense (MTD) orchestration framework for resource constrained IoT networks. The design is based on the gap analysis presented in Chapter 3, which identified the lack of an adaptive, multi technique MTD mechanism that takes real time network state, device limits, and energy cost into account at the same time. The proposed system addresses this gap by combining a rule based decision engine with two complementary MTD techniques, IPv6 address shuffling and application-layer port token rotation, coordinated by a dedicated orchestrator placed on the border router. The chapter focuses on what the system is meant to do and how its parts fit together.

4.1 Design Goals and Constraints

The system design is guided by five main objectives drawn from the literature review and the feasibility analysis:

- **Adaptivity:** The orchestrator should react to real time network conditions instead of just shuffling on a fixed schedule, since prior work has shown that fixed schedules perform poorly under changing attack patterns [17, 5].
- **Technique diversity:** At least two independent MTD techniques must be supported, so the orchestrator can pick the response that best matches the observed threat. No existing IoT MTD system provides rule based technique selection [49, 35].
- **Resource awareness:** Every mechanism must fit inside the memory and energy budget of RFC 7228 Class 1 end devices (about 10 KB RAM, 48 KB Flash) [8]. Heavy approaches such as reinforcement learning are out of scope.
- **Simulation validity:** The evaluation environment must use hardware accurate models, so the results carry over to real hardware. Energy use is to be measured with software counters that are consistent with the methodology used in [4, 51].
- **Reproducibility:** The full software stack must be open source so others can repeat the experiments. Table 3 lists the tools and their licences.

4.2 System Architecture

The proposed system uses a tiered hierarchical architecture that mirrors the physical layout of a typical IoT deployment, as shown in Figure 4. The design separates the constrained end devices , the edge gateway , and the external infrastructure that represents the cloud and the threat environment .

Tier 1 consists of twenty nine sensor nodes that act as Class 1 devices. Each node joins the RPL Destination Oriented Directed Acyclic Graph (DODAG) and sends periodic readings over

Table 3: Software stack used in this work (all open source, no licence cost).

Component	Purpose	Licence	Source
Contiki-NG v5.1	Embedded IoT operating system	BSD-3-Clause	github.com/contiki-ng
Cooja Simulator	IoT network simulation (Java)	BSD / Open	Bundled with Contiki-NG
RPL / 6LoWPAN	IPv6 routing stack (built-in)	BSD-3-Clause	Contiki-NG core/net/
Energest Module	Software energy estimation (built-in)	BSD-3-Clause	Contiki-NG os/services/
Python 3 / matplotlib	Log parsing and result visualisation	PSF Licence	python.org
Java 17 (OpenJDK)	Cooja simulator runtime	GPL-2.0+CE	openjdk.org

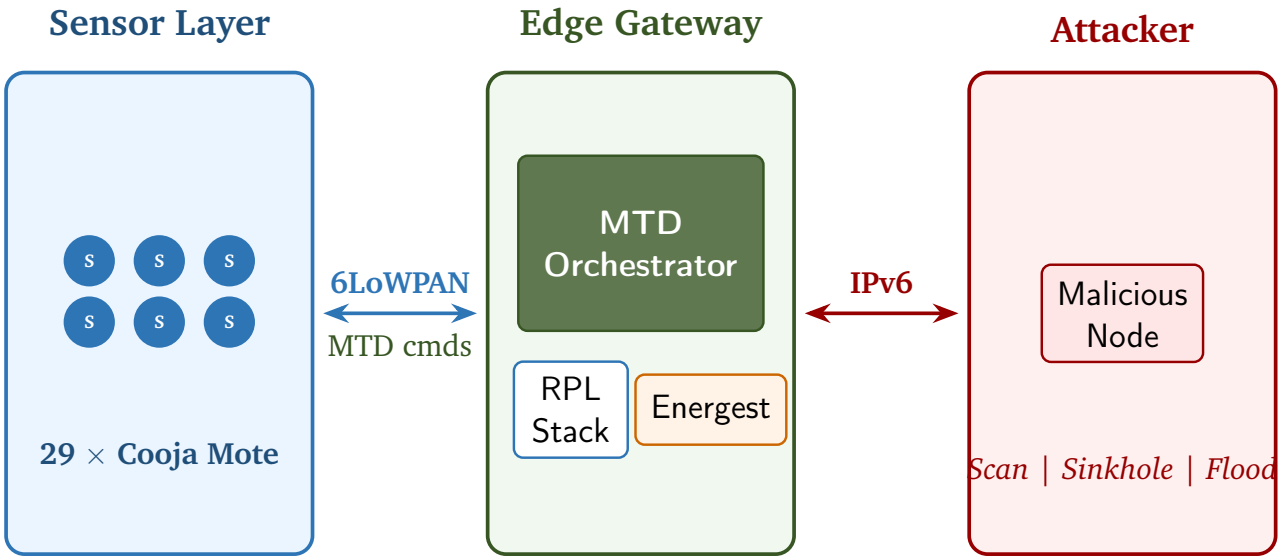

Figure 4: MTD orchestrator system architecture.

6LoWPAN [44]. These nodes are passive participants in the MTD mechanism: they do not decide when a shuffle should happen, they only carry out address or port token changes when they receive a command from the orchestrator (Section 4.4.3).

Tier 2 is the edge gateway, modelled as a border router. This node sits at the DODAG root and hosts the MTD orchestrator. Because all upward (sensor to cloud) and downward (command to sensor) traffic passes through it, the border router has full visibility of network events, which makes it the natural place for threat detection and response coordination [4]. In this evaluation the border router is itself emulated on the same Class 1 Tmote Sky target as the sensors (48 KB flash, 10 KB RAM); the *gateway* label denotes its role at the network edge, not a more capable hardware class. This is why the orchestrator must fit within the same constrained budget as the sensor firmware.

Tier 3 represents the external environment, including the cloud application server and, for

experimental purposes, the simulated attacker. The attacker is modelled as a single dedicated mote that runs inside the simulation alongside the legitimate nodes. It can take on three different roles, IPv6 scanning, RPL sinkhole, or UDP flooding, which produce the traffic patterns evaluated in Chapter 5. Keeping the attacker inside the simulator removes any host clock or external network noise from the measurements and makes the runs easy to reproduce.

4.3 Network Topology

The simulated network topology, shown in Figure 5, consists of thirty motes during baseline runs (one border router plus twenty nine sensor nodes) and thirty one motes when the attacker is present. Some of the sensor nodes naturally act as RPL routers because every node in the deployment is router-capable. The layout follows a clustered star of stars pattern, which is the most common topology reported in the MTD-IoT literature [17, 35] and represents environments such as smart buildings or industrial sensor floors.

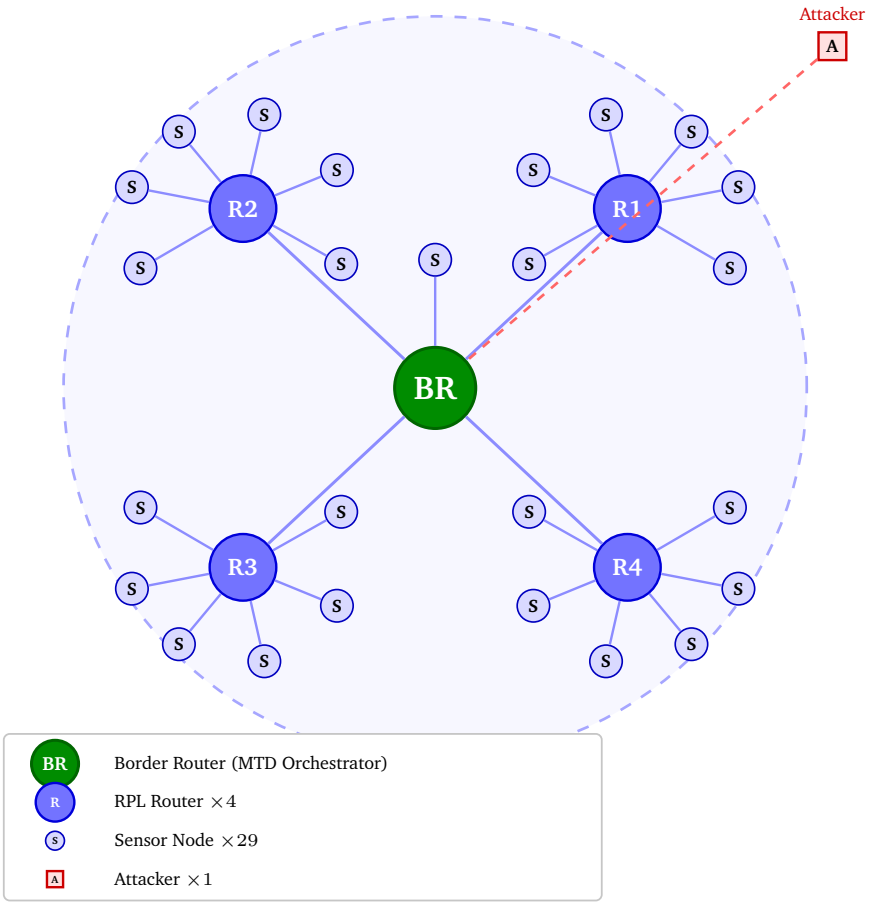

Figure 5: Cooja simulation topology (schematic, not to scale)

The border router sits at the logical centre and acts as the DODAG root. The first hop sensor nodes form four clusters around the border router and act as the parents for the rest of the network. This two hop topology makes sure that every sensor can reach the border router within two wireless hops, keeping latency within the bounds of typical CoAP request and

response timeouts [63]. A single malicious node is placed at the edge of the topology, roughly 52 m south of the border router, which puts it just outside the 50 m UDGM transmission range of the border router but well inside the 100 m interference range. The attacker is in the transmission range of several peripheral sensors and can inject attack traffic that reaches the border router only via multi hop forwarding.

The radio model used in the simulation is the Unit Disk Graph Model (UDGM) with a transmission range of 50 m and an interference range of 100 m, which matches the settings used in the official Contiki-NG examples. Key topology parameters are summarised in Table 4.

Table 4: Network topology and simulation parameters.

Parameter	**Value**
Total nodes	30 baseline (1 border router + 29 sensor nodes); 31 with attacker present
Radio medium model	Unit Disk Graph Medium (UDGM)
Transmission range	50 m
MAC layer	IEEE 802.15.4 CSMA/CA
Network layer	IPv6 / 6LoWPAN
Routing protocol	RPL , non-storing mode
Application protocol	Plain UDP for legitimate sensor traffic. Attack traffic uses real binary CoAP for the scan and flood scenarios and a forged RPL DIO Control Message for the sinkhole scenario
Sensor reporting interval	10 seconds
Simulation duration per run	30 minutes (simulated time)
Number of experimental trials	5 per scenario

4.4 MTD Orchestrator Design

The MTD orchestrator is the central contribution of this thesis. It is the component on the border router that decides when the network identity should change and which technique should be used. At the design level it has three sequential responsibilities: **monitor** the network for threat indicators, **decide** when and which MTD technique to apply, and **enforce** the chosen change across the sensor nodes. The decision flow is shown in Figure 6.

4.4.1 Monitoring and Threat Detection

The orchestrator keeps three independent threat indicators that, taken together, describe the nature of an ongoing attack. Each indicator is driven by its own detector, and the decision logic (Section 4.4.2) compares them to choose the most suitable MTD response. The three indicators are:

- **Stale port indicator (scan/replay):** Every legitimate sensor packet carries a small token that reflects the sensor's current MTD port. The border router compares this token against a sliding window made up of the current and the previous expected port. If a

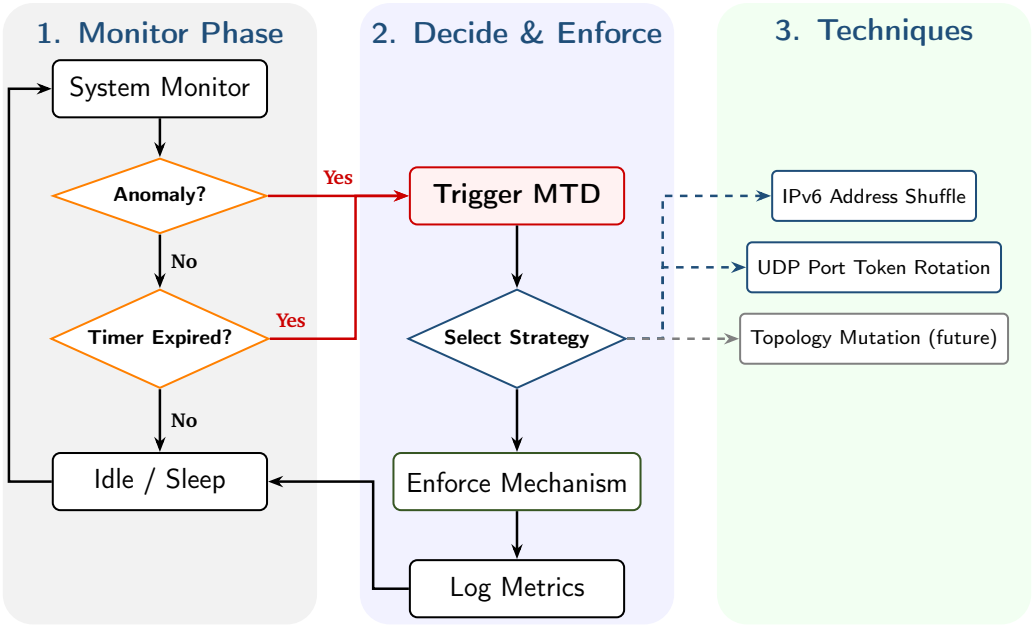

Figure 6: MTD Orchestrator rule-based decision and execution flow.

packet's token matches neither, and the network has finished its boot up grace period, the stale port counter is increased. This is the primary indicator for IPv6 scanning and replay traffic.

- **Connection failure indicator (silence watchdog):** A periodic watchdog inspects how recently each known sensor sent an upward packet. If a sensor stays silent for longer than the configured silence timeout, the orchestrator records a connection failure event. This is intended to catch situations where legitimate uplinks are lost without producing any stale port traffic, for example under the sinkhole disruption scenario described in Section 5.3.2.
- **CPU load indicator (flooding):** On platforms that report a usable CPU fraction, the orchestrator uses an energy counter based estimate. Under the flood rates evaluated in this work (up to 50 pkt/s), the per-packet processing cost on the border router is low enough that the CPU utilisation ratio computed from Energest does not cross the 80% threshold. The orchestrator therefore complements the CPU indicator with a packet-rate proxy: it counts inbound packets per fixed window and compares the total against a configured threshold. Either form of overload raises the CPU load indicator.

A unified cooldown applies across all three detectors: once any reactive cycle has fired, further reactive triggers are suppressed for a configured period. This stops the system from oscillating under a sustained attack and gives the routing layer time to reconverge before the next change.

4.4.2 Technique Selection Logic

When a trigger fires, the orchestrator chooses one of the available MTD techniques based on which threat indicator is active. This rule based selection is the distinguishing feature of this work compared to prior single technique implementations [72, 38]: the response matches the nature of the observed threat rather than always applying the same mechanism.

The selection follows a **severity priority** rule rather than a "majority counter" rule. The three threat indicators do not have equal information content: stale port events are frequent and low severity (they can be produced by legitimate sensors that happen to fall behind on a token rotation during a busy period), while connection failure and CPU load events are rare and high severity (each one represents a real and significant deviation from the expected operating state). Treating them as equal counters would let the noisy stale port signal mask the rare ones whenever both occur. The orchestrator therefore applies a fixed priority order: *connection failure* outranks *CPU load*, which outranks *stale port*. This is consistent with the risk sensitivity guidance in von der Assen et al. [5] and the action-on-detection model used in Nguyen et al. [51], both of which treat rare-but-severe signals as dominant when they coincide with high-frequency low-severity background activity.

The selection rules are:

- If the **connection failure** indicator is active, apply a **port token rotation** (Section 4.6). Service level probing and link layer disruption rely on a fixed port to anchor their targeting; rotating the active port token breaks that and gives the orchestrator a fresh window to push commands back to the affected sensors.
- Else if the **CPU load** indicator is active, apply a composite response: an **IPv6 shuffle combined with rate limiting**. Flooding attacks aim at the border router's processing path; the address change forces the attacker to re-enumerate the new target while the rate limiter throttles any residual flood until the load returns to normal, at which point the limiter turns itself off.
- Otherwise (the **stale port** indicator is dominant), apply an **IPv6 address shuffle** (Section 4.5). Scanning and replay attacks rely on stable IPv6 addresses; shuffling makes any address the attacker has already learned immediately useless and forces it to start enumeration again.

The two severe indicators trigger the reactive cycle as soon as a single event is recorded. This is safe because a connection failure event is only raised after a sensor has been silent for at least 25 seconds (about 2.5 sensor send intervals, ruling out single missed transmissions), and a CPU load event is only raised after a 10 second monitoring window has accumulated more than the configured packet rate threshold (ruling out brief traffic bursts). The stale port indicator, in contrast, requires accumulation to a configured threshold count before firing, because each individual event carries less information.

Topology mutation, where nodes are reparented inside the DODAG, is identified as a third possible technique and included in the architecture as a future extension. It is not part of the current evaluation because the routing reconvergence latency under topology changes in this configuration has not been characterised, and including it would risk the latency measurements that the IPv6 shuffle is designed to expose.

4.4.3 MTD Enforcement via Direct Commands

Once a technique is chosen, the orchestrator pushes the new configuration to the nodes using small command messages sent directly over the existing routing tree, rather than by changing the routing protocol itself. Two reasons motivate this choice. First, the routing layer used here does not maintain per host downward routes at the border router, so injecting custom routing messages would not reach individual sensors reliably and would also contaminate the routing substrate used by the no-MTD control condition. Second, the information being signalled (a new port value or a one byte shuffle trigger) is application data, not routing metadata, and keeping it at the application layer preserves a clean separation between routing and identity mutation.

The orchestrator keeps a list of the most recently seen sensor addresses, populated from incoming upward traffic. On each cycle it iterates that list and sends the new port value or the shuffle trigger to every known sensor. The same commands are also attached to the orchestrator's reply to each valid upward packet, since the network stack still has a fresh reverse route entry for that sender at that moment. Sensors update their own state on receipt; the next packet they send already carries the new identity, and ordinary upward forwarding does the rest. No separate downward routing protocol is needed, and the extra code on the sensor side is small enough to fit a Class 1 budget.

4.5 IPv6 Address Shuffling

IPv6 address shuffling is the primary MTD technique implemented in this work. The design follows the micro Moving Target IPv6 Defense (μMTD) protocol proposed by Zeitz et al. [72], adapted for the Contiki-NG stack. The full sequence, from attack detection to route reestablishment, is illustrated in Figure 7.

Each sensor node is assigned a Unique Local Address (ULA) drawn from a fixed network prefix. The first 64 bits (the prefix) stay the same for the whole simulation; only the lower 64 bits, the Interface Identifier (IID), are changed during a shuffle event. The IID is generated using the platform's pseudorandom generator, which is enough for this experimental setting; a full cryptographic generator would be preferable in a production deployment but is not feasible inside the Class 1 memory budget targeted here.

The design mutates only the network layer. The lower 64 bits of the IPv6 address are randomised while the link layer address is left untouched. The original μMTD specification [72]

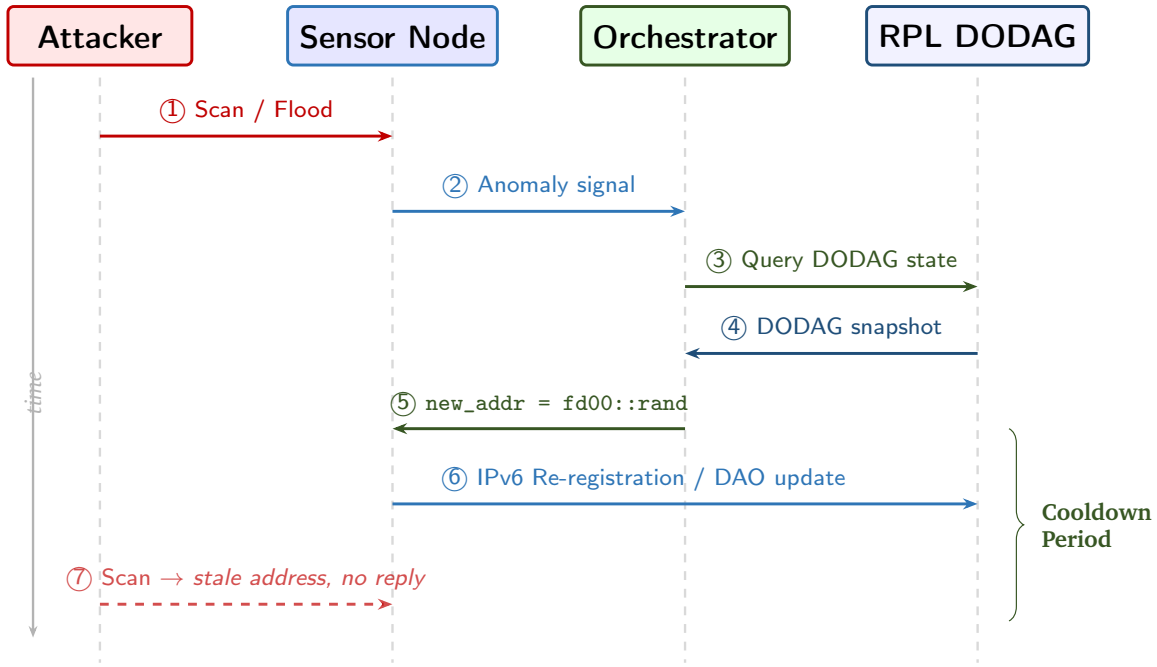

Figure 7: IPv6 address shuffling sequence: from attack detection to route re-establishment.

also rotates the link layer address each cycle, on the grounds that an attacker watching the physical layer could otherwise correlate old and new IPv6 addresses through the stable link layer identifier. Doing so in Contiki-NG requires changes to internal data structures that exceed the Class 1 memory budget targeted here, so we adopt the network only variant as a logical approximation. The threat model in Section 5.1 does grant the attacker the ability to observe plaintext frames, so this residual link layer correlation is a genuine limitation rather than a non-issue; it is accepted here because none of the three actively evaluated attacks (scan, sinkhole, flood) perform passive identifier correlation, and full link layer plus network layer mutation exceeds the Class 1 memory budget. Combined mutation is left as future work (section 8.3).

4.6 Application-Layer Port Token Rotation

Application-layer port token rotation is the second technique and targets service layer reconnaissance. The orchestrator periodically picks a new port value from a fixed pool and tells the sensors to use it as their MTD identity token. The new value becomes the application-layer identity that every sensor's outgoing payload carries as `port=N`; the border router compares this token against its current and previous expected values to validate incoming traffic.

Both proactive mutations are driven by independent timers: a 120 s base interval for the IPv6 address shuffle and a 300 s base interval for the token rotation. Each interval is not used directly but is *dithered* by $\pm 25\,\%$, with the next firing time drawn uniformly from $[0.75T, 1.25T]$ around the base interval T. The dithering keeps the long-run mutation rate fixed while preventing an attacker that observes a few mutation events from extrapolating the next one, and

the two timers are dithered independently so that learning one cadence does not reveal the other.

This is deliberately a lighter-weight version of the network-layer port hopping described in the MTD literature (for example iANVS [48]). Real port hopping rebinds the listening UDP socket so that packets sent to the old port number never reach the application; our orchestrator does not rebind the socket, and the source and destination port numbers in the IP/UDP header stay constant for the whole run. The MTD identity is carried entirely as an application-layer token inside the payload, so a token rotation has near-zero runtime cost on the border router. The two mechanisms provide comparable entropy against scan and replay (both rotate a 16-bit value), but they operate at different layers and have very different runtime costs.

Token rotation addresses a different threat class from address shuffling. An attacker that has already learned a node's IPv6 address can still try service-level probing by sending requests with old or guessed token values. Rotating the token forces the attacker to learn the new value after each rotation, which increases its time to compromise (TTC) [62] and complements the network-layer defense offered by the IPv6 shuffle. The amount of state the orchestrator has to maintain (current token, previous token, a small grace flag, and a rotation counter) is tiny, which keeps the mechanism portable to Class 1 devices.

5 Attack Design

This chapter describes the adversarial side of the experimental evaluation. To assess the effectiveness of the MTD orchestrator, three attack scenarios are designed to represent the main threat classes identified in the related work survey: address enumeration, routing manipulation, and resource exhaustion. Each scenario is built to trigger a specific monitoring indicator inside the MTD system, so that the defensive response and its overhead can be measured under controlled, repeatable conditions. Section 5.1 defines the attacker model and assumptions, Section 5.2 describes how attacks are introduced into the simulation environment, and Section 5.3 specifies each scenario.

5.1 Threat Model

The threat model follows the Dolev–Yao convention adapted for low power IoT networks [48]. The attacker is modelled as a single external node that has radio level connectivity to the 6LoWPAN network but has not compromised any legitimate sensor node. Table 5 summarises the assumed capabilities and the explicit scope boundaries. Figure 8 shows the DODAG topology relative to the malicious mote's radio range and identifies the three attack vectors evaluated in this chapter.

Table 5: Attacker capability model.

Property	**Assumption**
Node type	Single malicious mote, placed at the network periphery roughly 52 m south of the border router. The position is just outside the 50 m UDGM transmission range of the border router but well inside the 100 m interference range, and inside the transmission range of several peripheral sensor nodes. The same mote can take on any of the three attack roles defined in Section 5.3.
Network access	Can send and receive arbitrary IEEE 802.15.4 frames; can inject ICMPv6, UDP, CoAP, and RPL control messages.
Cryptography	No link-layer encryption is used in the simulation, so the attacker can observe all plaintext frames.
Node compromise	Cannot compromise legitimate sensor nodes or the border router; only the malicious mote is under attacker control.
Physical access	No physical access to legitimate hardware; the attacker operates remotely via the radio interface.

The MTD system is therefore evaluated against a *capable but external* adversary. This scope is consistent with the threat models used in related work [72, 48, 10] and reflects the realistic threat surface of unencrypted 6LoWPAN deployments in smart building and industrial IoT environments, where an attacker with a software defined radio or a rogue 802.15.4 node can observe network traffic without physical access to the devices.

The following threat classes are outside the current evaluation: (i) attacks that need prior

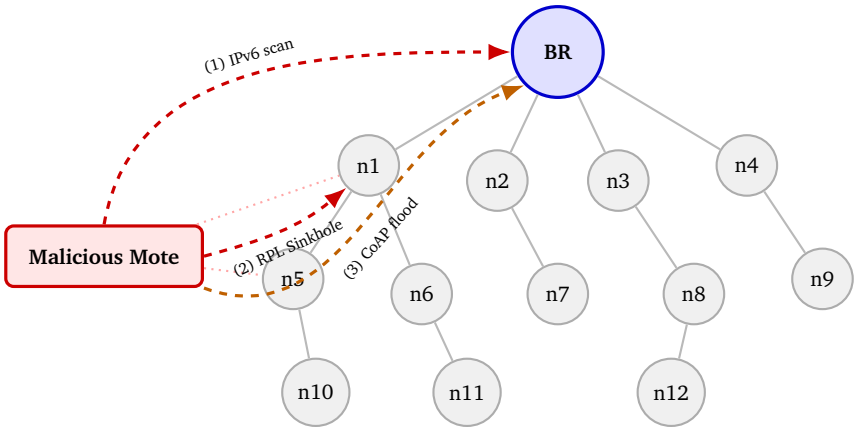

Figure 8: Threat-model network topology

node compromise, (ii) physical destruction, (iii) attacks that exploit application layer vulnerabilities in the sensor firmware, and (iv) coordinated multi attacker campaigns.

5.2 Attack Framework and Simulation Setup

All three attacks are designed to run inside the same simulation environment as the legitimate nodes. The attacker is a single dedicated mote that can be configured into one of three attack roles. This keeps every scenario on the same toolchain and the same simulation event loop, which removes host clock and external network noise from the measurements and makes the runs comparable. No external bridge or external attack framework is required.

5.2.1 Malicious Mote Integration

The malicious mote is added to the simulation as an extra mote alongside the legitimate sensors, RPL routers, and the border router. It is positioned at the edge of the topology, within UDGM radio range of several sensor nodes but outside the core RPL routing tree. This placement models a realistic edge infiltration scenario, such as a rogue node deployed near a building entrance or at the boundary of an industrial floor. In the radio disruption (sinkhole) scenario it places the attacker close enough to drown out legitimate uplink packets at the link layer.

Keeping the attacker in a separate mote type is a deliberate design choice. The defensive code path on the legitimate nodes does not contain any conditional logic that triggers only when an attacker is present, which preserves the validity of the MTD vs. no-MTD comparison.

5.2.2 Experimental Protocol

Each attack scenario is evaluated in two configurations:

- **Baseline (MTD disabled):** The orchestrator process is not active. This captures the network's natural behaviour under attack without any defensive response and establishes

the cost of *not* deploying MTD.

- **MTD enabled:** The full dual-mode orchestrator is active with a proactive IPv6 shuffle every 120 s, a reactive anomaly threshold of five stale-port events (severe types fire on the first event), and a proactive port token rotation every 300 s.

Each scenario is run for 30 minutes of simulated time.

5.3 Attack Scenarios

Table 6 gives an overview of the three attack scenarios. The subsections below describe each scenario's mechanism, parameter choices, and the MTD response it is expected to trigger, either a reactive IPv6 shuffle on threshold breach, or a reactive port token rotation, or a composite shuffle plus rate limiting response.

Table 6: Attack scenarios and simulation parameters.

Attack	**Mechanism**	**Expected MTD response**	**Primary metric**
IPv6 Scanning	Binary CoAP `CON GET` probes at ∼5 pkt/s carrying a stale port token in the Uri-Query option; IID swept through patterns.	Stale-port indicator crosses threshold; reactive IPv6 shuffle invoked.	Scan success rate
Sinkhole (RPL)	Forged RPL DIO Control Message broadcast on `ff02::1` at 10 pkt/s; UDGM collisions silence near-attacker uplinks.	Silence watchdog raises CONN_FAILURE; reactive branch triggers a port token rotation.	Packet delivery ratio
CoAP Flooding (DoS)	Binary CoAP `CON GET` at ∼30 pkt/s with randomised Message ID and Token; no `port=` token, so the stale-port check is bypassed.	Packet-rate proxy raises CPU_LOAD; composite response applies an IPv6 shuffle plus an armed rate limiter.	Packet rate & energy

5.3.1 IPv6 Address Scanning

Objective: The attacker tries to enumerate all active sensor IPv6 addresses in the prefix used by the deployment. A complete address map enables follow up attacks, for example selectively flooding a chosen node or constructing a more efficient sinkhole by advertising a low rank only to known neighbours.

Mechanism:

The malicious mote sends about 5 CoAP `CON GET` probes per second to the border router, each a structurally valid request for `/.well-known/core?port=8765` (RFC 7252) [63]. The `port=8765` Uri-Query option carries a stale MTD port token: once the boot grace lifts, the border router's anomaly check no longer accepts it, so each probe is flagged as a stale-port event. The attacker therefore emits a steady stream of packets that are structurally valid CoAP on the wire but carry an anomalous MTD identity token in the Uri-Query option.

Expected MTD interaction: The border router compares the payload token of each incoming packet against the orchestrator's current and previous port. Since the attacker's stale token matches neither, and the boot up grace window has expired by the time the second hop has fired, the packet counts as a stale port event. When the count reaches the configured threshold, the reactive branch picks the IPv6 shuffle response: the orchestrator randomises its own IID and broadcasts a shuffle command to every known sensor. Each subsequent probe carries the same stale token, so once detection has begun the border router keeps rejecting them at the anomaly check.

Parameters:

- Scan rate: 5 binary CoAP probes per second
- Target: the border router, probed with a stale port token (`port=8765`)
- Reactive trigger: stale port anomaly count ≥ 5
- Proactive shuffle interval: 120 s

Expected outcome:

In the baseline configuration the stale token is always accepted, so the probes reach the application layer unimpeded. With MTD the stale-port detector flags them once the boot grace lifts and the reactive selector fires the IPv6 shuffle branch, cutting the fraction of accepted probes from near one to about a third.

5.3.2 RPL Sinkhole Attack

Objective: The attacker tries to draw routing traffic away from the legitimate DODAG root by advertising a falsely low routing rank. Nodes that accept the malicious advertisement reroute their upward traffic through the sinkhole, which then drops the packets, causing application level PDR to drop.

Mechanism: A faithful rank-1 DIO sinkhole would require modifying the RPL-Lite routing layer source code, which would contaminate the orchestrator's own routing control plane and make the no-MTD control condition incomparable. To avoid this, the attacker uses a *radio disruption approximation*: it constructs a structurally valid forged RPL DIO Control Message Object per RFC 6550 and broadcasts it on the link local all nodes address (`ff02::1`) at 10

packets per second. The forged DIO advertises rank 0x0100 (256), which is below the legitimate one hop child rank (384) and would win routing if accepted by a strict receiver. It carries a DODAG Configuration sub-option (type 0x04) and a Prefix Information sub-option (type 0x08) that advertises the same `fd00::/64` prefix as the legitimate DODAG, so the DIO is structurally indistinguishable from a real one. The packet is wrapped in UDP rather than emitted as raw ICMPv6, because Contiki-NG RPL-Lite does not expose a public DIO injection API to applications. The payload bytes follow the RFC 6550 DIO field layout, but because they travel in UDP rather than as an ICMPv6 message they are not on-wire RFC 6550 DIOs; they are used here only as a radio-disruption approximation whose MAC-layer collision footprint matches a genuine DIO injection. Under the simulator's UDGM radio model with a 100 m interference range, the resulting broadcast storm collides with legitimate sensor uplinks within the attacker's radius and causes them to be dropped at the MAC layer.

Expected MTD interaction: The detector is the orchestrator's silence watchdog. For every sensor in the registry, the watchdog tracks the wall clock time of the last upward packet; if that time goes past the silence timeout, the sensor is considered disconnected and a connection failure event is recorded. Once the connection failure indicator dominates, the technique selector picks the token-rotation branch, the orchestrator generates a fresh port token and broadcasts it to every known sensor. The rotation itself does not stop the radio collisions directly, but it refreshes the reverse route entries that the piggyback mechanism relies on, and sensors that recover radio access reregister with the orchestrator on the new token so that normal traffic can resume.

Parameters:

- Transmission pattern: link local broadcast (`ff02::1`) at 10 packets per second
- Payload: forged RPL DIO Control Message Object
- Forged rank: 0x0100 (below legitimate one-hop child rank 384)
- Silence timeout: 25 s (`MTD_SILENCE_TIMEOUT_S`)
- Response: reactive port token rotation unicast to the sensor registry

Expected outcome: In the baseline configuration, sensors inside the attacker's interference range lose more and more uplink packets, and PDR at the border router degrades over the simulation run. In the MTD enabled configuration, the silence watchdog detects the loss, the reactive token rotation fires, and PDR partially recovers once surviving sensors register on the new token. The remaining gap (relative to a clean baseline) quantifies the radio level impact that MTD alone cannot mitigate.

It is worth noting up front that the radio disruption approximation tends to corrupt the downlink (border router to sensor) more reliably than it silences the uplink (sensor to border router). Sensors typically retransmit via CSMA back off and recover within the 25 s silence

timeout, while the border router's downward unicast commands are more often lost to collisions. The practical consequence is that the sinkhole scenario tends to be dominated by stale port events rather than connection failure events. A pre-shuffle silence drain (Chapter 6) catches the smaller number of true silences that do cross the timeout, but a faithful rank-1 DIO injection that hijacks routing, which would produce a much stronger silence signal, is out of scope of this thesis.

5.3.3 UDP Flooding (Denial of Service)

Objective: The attacker tries to overwhelm the border router's packet processing path with sustained high rate CoAP traffic, preventing it from servicing legitimate sensor data and from running MTD operations on time. The scenario uses real binary CoAP rather than a stripped down UDP flood so that the wire level traffic matches what a real IoT botnet (e.g. Mirai variants) would emit, and so that a packet capture reads as a real protocol attack rather than as a synthetic test stream.

Mechanism: The attacker sends about 30 CoAP `CON GET` packets per second to the border router's server port. Each packet is a valid CoAP message for `coap://br/sensor/data`. The Message ID and Token fields are randomised on every packet so a duplicate suppression filter on the receiver cannot collapse the flood by dropping packets with repeating identifiers. Crucially the payload does not include a `port=` substring, so the border router's stale port check is bypassed (a missing token is accepted to avoid spuriously flagging packets during bring up).

Expected MTD interaction: The orchestrator runs two CPU_LOAD detectors in parallel. The first is a direct Energest CPU/(CPU+LPM) ratio compared against `MTD_CPU_THRESHOLD_PCT` (80 %); the second is a packet-rate proxy that counts inbound UDP packets per 10 s window and compares against `MTD_FLOOD_PPS_THRESHOLD` (80 packets). Whichever detector trips first raises a CPU_LOAD anomaly. The packet-rate proxy is intended as the primary path for application-layer floods of this kind, because such floods saturate the receive callback without necessarily pushing the CPU%/total ratio above the conservative 80 % threshold. Once CPU_LOAD is raised, the reactive selector routes to the composite flooding branch: an IPv6 shuffle invalidates the border router's target address while an application layer rate limiter is armed, dropping inbound traffic above the per-second cap. When the packet-rate monitor later sees that the rate has dropped back below the threshold, the rate limiter clears automatically and normal traffic resumes.

Parameters:

- Flood rate: $\sim$30 binary CoAP packets per second
- Target: border router on the configured server port (5678)
- Packet shape: real CoAP `CON GET /sensor/data` with randomised 16-bit Message ID

and 16-bit Token

- Detection proxy: more than 80 inbound packets within a 10 s window
- Composite MTD response: IPv6 shuffle plus application layer rate limiter which has an adaptive cap.

Expected outcome: In the baseline configuration, the per window packet count at the border router stays high for the entire simulation, slowing sensor data collection and starving MTD operations of CPU time. In the MTD enabled configuration, the composite response is expected to drop the effective packet arrival rate below the rate limit threshold within one detection window, by making the fixed target flood ineffective at the network layer and throttling any residual flood on the new port.

6 Implementation

This chapter walks through how the system in Chapter 4 and the attacks in Chapter 5 were actually built. Everything runs on Contiki-NG v5.1 inside the Cooja simulator, targeting the Tmote Sky platform so that the same binaries could in principle be flashed to real hardware. The chapter has two halves: the **system implementation** (Section 6.2), which covers the border router, the MTD orchestrator, and the sensor node; and the **attack implementation** (Section 6.3), which covers the three attacker modes. Code listings are kept short; the full source code can be found here.

6.1 Cooja Simulation Setup

All experiments run inside the Cooja simulator shipped with Contiki-NG v5.1. The simulation target used here is Tmote Sky compiled for the MSPSim emulator, which gives a cycle-accurate model of the MSP430 CPU. The radio is modelled by Unit Disk Graph Medium (UDGM).

6.1.1 Radio Medium

Every simulation uses UDGM with the parameters in Table 7. UDGM treats radio propagation as a deterministic disk: two nodes within the transmission range communicate successfully; nodes outside the range cannot decode each other, but transmissions inside the interference range still corrupt receivers.

Table 7: UDGM radio medium parameters

Parameter	Value
Transmitting range	50
Interference range	100
TX success ratio	1.0 (lossless)
RX success ratio	1.0 (lossless)

A lossless channel is used on purpose. The goal of the evaluation is to characterise the MTD system under attack, not to study packet loss caused by random radio drops. Any losses in the results are therefore caused either by MTD induced route re establishment, by deliberate drops at the anomaly check, or by deterministic collisions in the attacker's interference radius and not by random radio behaviour.

6.1.2 Network Topology

The simulated network has 30 motes during baseline runs and 31 motes during attack scenarios: one border router, 29 sensor nodes spread across a roughly 100×100 m area, and the attacker mote (when present). All sensor nodes are RPL-router-capable, so multi-hop forwarding is available without any special configuration. The random seed is fixed across the

three attack variants so that the initial topology and PRNG state are identical, which makes the conditions directly comparable.

Figure 9 shows the topology as rendered by Cooja's Network panel.

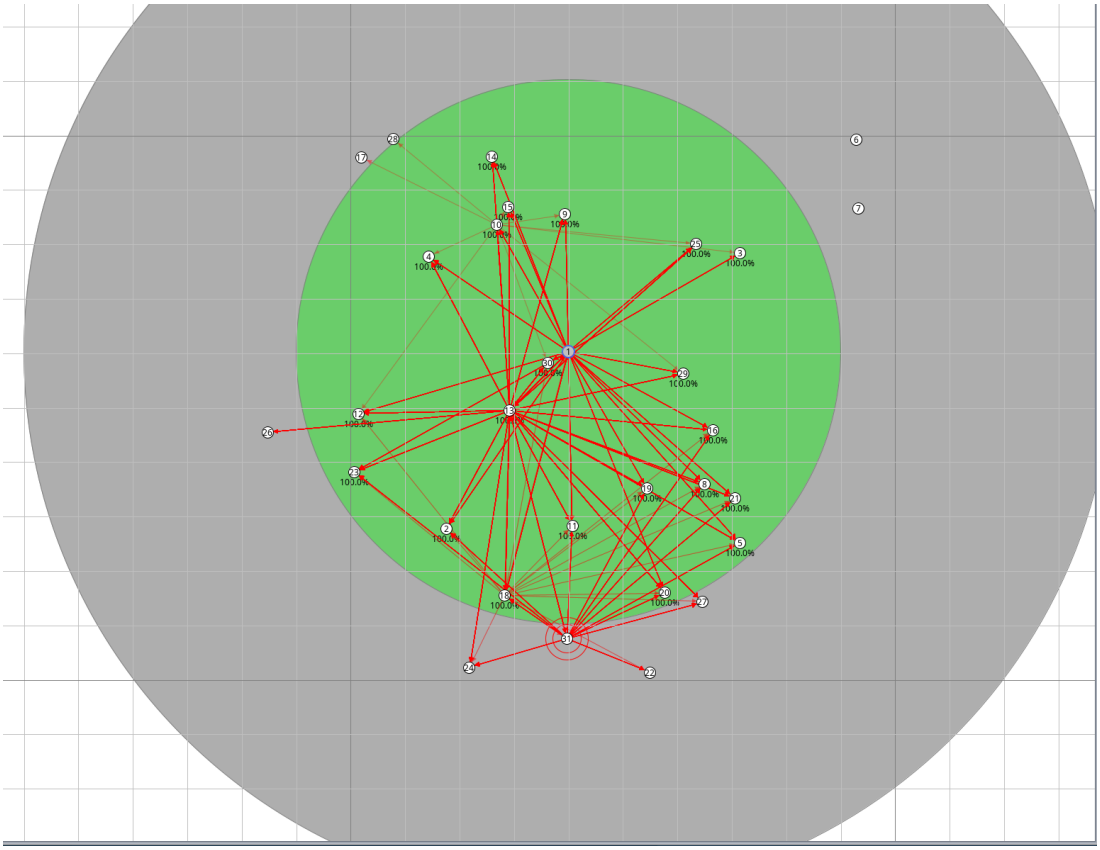

Figure 9: Cooja Network panel showing the simulated topology

6.1.3 Simulation Configurations

Three families of Cooja simulation files are provided, summarised in Table 8. The baseline measures MTD overhead without any attacker. The MTD attack family runs the orchestrator alongside one of the three attack modes. The no MTD attack family runs the same attack against the disabled orchestrator and serves as the control condition. Every attack family has its own per mode firmware wrapper (Section 6.3.1) so that scenarios never share a firmware build.

The no MTD border router uses a wrapper that defines `MTD_DISABLED` before pulling in the main border router source. This produces a binary that accepts all sensor traffic unconditionally with zero MTD overhead. Using a separate wrapper rather than a build-time flag matters: the build system tracks source-file changes, so a flag-only change would not always trigger a rebuild and the binary could silently come from the wrong build.

Table 8: Cooja simulation configurations

File	MTD	Description
`mtd_attack_scan_sky.csc`	Enabled	31 motes, scan attacker.
`mtd_attack_sinkhole_sky.csc`	Enabled	31 motes, sinkhole attacker.
`mtd_attack_flood_sky.csc`	Enabled	31 motes, flood attacker.
`nomtd_scan_sky.csc`	Disabled	Control: scan, no defense.
`nomtd_sinkhole_sky.csc`	Disabled	Control: sinkhole, no defense.
`nomtd_flood_sky.csc`	Disabled	Control: flood, no defense.

6.2 System Implementation

The system has three pieces: the border router that hosts the orchestrator, the MTD library that holds the proactive and reactive logic, and the sensor firmware. The next sections walk through them in the order a packet would touch them.

6.2.1 Border Router and Orchestrator Boot

On boot the border router starts the RPL DAG root, registers a UDP server on port 5678, and calls `mtd_orchestrator_init()`. That init routine arms three independent timers: one for the proactive IPv6 shuffle (every 120 s), one for the proactive port token rotation (every 300 s), and one for the silence watchdog (every 15 s). It also starts a CPU-load monitor and a packet-rate monitor in the background.

When the firmware is compiled with the `MTD_DISABLED` flag set (used for the no-MTD baseline), the init call is skipped and none of the timers run. That single flag is the *only* difference between the MTD-enabled and the no-MTD binary, which keeps the baseline comparison separate.

The orchestrator keeps a record of two states : the *current* expected port and the *previous* expected port. The anomaly check uses both so that a sensor which missed exactly one port-token-rotation unicast is still accepted.

6.2.2 Sensor Address Registry

RPL-Lite does not maintain proactive downward routes from the root to individual nodes, so the orchestrator cannot just look up where to send a port-token-rotation unicast. Instead it remembers the last few sensor addresses it has seen and unicasts to all of them. Listing 1 shows the registration code.

Listing 1: Sensor address registry

```
void
mtd_register_sensor(const uip_ipaddr_t *addr)
{
  uint8_t i;
  clock_time_t now = clock_time();
```

```
  for(i = 0; i < sensor_count; i++) {
    if(uip_ipaddr_cmp(&sensor_registry[i], addr)) {
      if(sensor_silence_flagged[i]) {
        LOG_INFO("SILENCE recovered sensor #%u\n", i);
      }
      sensor_last_seen[i]     = now;
      sensor_silence_flagged[i] = 0;
      return;
    }
  }

  uip_ipaddr_copy(&sensor_registry[sensor_head], addr);
  sensor_last_seen[sensor_head]    = now;
  sensor_silence_flagged[sensor_head] = 0;
  sensor_head = (sensor_head + 1) % MTD_MAX_SENSORS;
  if(sensor_count < MTD_MAX_SENSORS) sensor_count++;
}
```

The registry is a ring buffer sized at 32 slots. After a shuffle, every sensor's new address shows up here on its next data packet, and the stale pre-shuffle entry naturally gets overwritten. This keeps the registry small and self-cleaning.

Two side notes are worth picking out from the code. First, the function also clears the silence flag for a known sensor: as soon as a packet arrives, that sensor is no longer silent, even if the watchdog had flagged it earlier. Second, the timestamp `sensor_last_seen[i]` is what the silence watchdog uses later to decide whether a sensor has gone quiet.

6.2.3 Proactive Defense Cycle

The proactive shuffle callback fires on a jittered timer with a 120 s base interval. It randomises the border router's own IPv6 IID and unicasts a shuffle command to every sensor in the registry. The proactive port-token-rotation callback works the same way on a separate timer with a 300 s base interval. Both intervals are dithered by $\pm 25\,\%$, so the next firing of each timer is drawn uniformly from $[0.75T, 1.25T]$ around its base interval T (Section 4.6).

Listing 2 shows the proactive shuffle callback. The structured `MTD_CYCLE` log line at the end is what the evaluation parser later picks up to count cycles.

Listing 2: Proactive IPv6 shuffle callback

```
static void
proactive_shuffle_cb(void *ptr)
{
  cycle_count++;
  LOG_INFO("=== MTD Shuffle Cycle #%lu (proactive) ===\n",
           (unsigned long)cycle_count);
```

```
  ipv6_shuffle_address();
  broadcast_addr_shuffle();

  LOG_INFO("MTD_CYCLE type=shuffle cycle=%lu port=%u\n",
           (unsigned long)cycle_count, current_port);

  ctimer_set(&proactive_timer, jittered_interval(MTD_SHUFFLE_INTERVAL),
             proactive_shuffle_cb, NULL);
}
```

Both proactive timers are armed through the helper in Listing 3, which draws each interval from $[0.75T, 1.25T]$. A static schedule would let an attacker that observes two or three mutation events estimate the next mutation time and time its probes for the largest window of unmonitored access; the jitter widens that prior to roughly half the base interval and removes the predictability without changing the long-run mean rate. The two timers are independent, so learning one cadence still does not reveal the other.

Listing 3: Per-interval timer jitter

```
static clock_time_t
jittered_interval(clock_time_t base)
{
  clock_time_t spread = (base * MTD_JITTER_PCT * 2U) / 100U;
  if(spread == 0) return base;
  clock_time_t offset = (clock_time_t)(random_rand() % spread);
  return base + offset - (spread / 2U);
}
```

6.2.4 Sending Commands to Every Sensor

Both proactive and reactive cycles end up sending a small command to every sensor in the registry. Listing 4 shows the port-token-rotation sender, which iterates over the registry and *unicasts* the command to each sensor in turn (despite the historical `broadcast_` prefix on the function name, no link-layer broadcast is used). The address-shuffle sender is the same shape, just with a different one-byte opcode.

Listing 4: Per-sensor unicast of the CMD_PORT_HOP command

```
static void
broadcast_port_update(uint16_t new_port)
{
  uint8_t cmd[3];
  port_hopper_encode_cmd(cmd, new_port);

  for(uint8_t i = 0; i < sensor_count; i++) {
    int rc = simple_udp_sendto(&orchestrator_conn,
```

```
                              cmd, sizeof(cmd),
                              &sensor_registry[i]);
    if(rc < 0) {

      LOG_WARN("CMD_PORT_HOP send failed to sensor #%u\n", i);
      mtd_report_anomaly_typed(MTD_ANOMALY_CONN_FAILURE);
    }
  }
}
```

The interesting bit is the failure path. If `simple_udp_sendto` returns a negative code it means the IPv6 stack could not push the packet down to the sensor at all (no route, neighbour cache miss, queue full). That is the same symptom a sinkhole or a routing attacker would produce, so the orchestrator immediately reports a `CONN_FAILURE` event. This is a useful detection path that does not require any extra timer.

6.2.5 Reactive Defense Cycle

When a severe anomaly is reported, or the total stale-port count crosses the threshold, the orchestrator runs an out of schedule cycle. Listing 5 shows the reactive trigger. Each call bumps a per type counter and the overall counter, then checks two things: does this event-type fire immediately (severe types do, stale-port does not), and is the cooldown gate open.

Listing 5: Reactive trigger with per-type firing policy and cooldown gate

```
void
mtd_report_anomaly_typed(mtd_anomaly_type_t type)
{
  if(type >= MTD_ANOMALY_TYPE_COUNT) return;

  anomaly_counts[type]++;
  anomaly_count++;

  uint8_t severe = (type == MTD_ANOMALY_CONN_FAILURE ||
                    type == MTD_ANOMALY_CPU_LOAD);

  if(severe || anomaly_count >= MTD_THREAT_THRESHOLD) {
    clock_time_t now = clock_time();
    if(!first_reactive && (now - last_reactive_ts) < MTD_COOLDOWN) {
      LOG_INFO("Threshold reached but cooldown active\n");
      return;
    }
    LOG_WARN("Threat threshold reached -- triggering reactive cycle\n");
    last_reactive_ts = now;
    first_reactive = 0;

    static struct ctimer reactive_timer;
```

```
    ctimer_set(&reactive_timer, 0, reactive_shuffle_cb, NULL);
  }
}
```

The *cooldown gate* (`MTD_COOLDOWN_S`, 60 s) is what keeps the system from oscillating under a sustained attack. Without it, every late-arriving anomaly would fire another cycle right after the previous one, and the routing layer would never get time to re-converge. The counters keep climbing during cooldown, so the very next event after it expires can still fire straight away. This is a simple way of bounding the reactive rate without losing reactivity.

Instead of restarting both proactive timers on every reactive event, the reactive callback only rewinds the timer whose action it just consumed. A shuffle-branch cycle rewinds the shuffle timer; a token-rotation branch cycle rewinds the token-rotation timer. The other timer keeps ticking. This stops a persistent attacker from starving whichever proactive defense it does not happen to trigger.

6.2.6 Threat-Type Selection

The reactive callback then has to pick which MTD technique to run. The orchestrator tracks three threat types, each tied to one technique:

- **Stale port**: a payload token does not match the current or previous expected port. This means scanning or replay traffic. The response is an **IPv6 shuffle**.
- **Connection failure**: a sensor has gone silent, or a unicast command could not be delivered. This means routing or link disruption. The response is a **port token rotation**.
- **CPU load**: the BR's packet-rate window is over the threshold. This means flooding. The response is an **IPv6 shuffle plus an armed rate limiter**.

Selection follows a fixed severity order rather than picking the highest counter. `CONN_FAILURE` outranks `CPU_LOAD`, which outranks `STALE_PORT`. The reason is that the three signals do not carry the same weight. Stale-port events happen at high volume during normal reactive activity, because legitimate sensors sometimes fall two hops behind on their port token. If we just picked the largest counter, this background noise would mask the rare-but-serious connection-failure or CPU-load events whenever both are present. The priority rule guarantees that a single confirmed silence or flood event always routes to its designed branch.

This is the same rule-based, risk-sensitive selection style recommended by von der Assen et al. [5]. Listing 6 shows the resulting selector.

Listing 6: Threat-type selection in the reactive handler

```
mtd_anomaly_type_t threat;
if(anomaly_counts[MTD_ANOMALY_CONN_FAILURE] > 0) {
  threat = MTD_ANOMALY_CONN_FAILURE;
```

```
} else if(anomaly_counts[MTD_ANOMALY_CPU_LOAD] > 0) {
  threat = MTD_ANOMALY_CPU_LOAD;
} else {
  threat = MTD_ANOMALY_STALE_PORT;
}

switch(threat) {
  case MTD_ANOMALY_STALE_PORT:
    ipv6_shuffle_address();
    broadcast_addr_shuffle();
    ctimer_set(&proactive_timer, jittered_interval(MTD_SHUFFLE_INTERVAL),
               proactive_shuffle_cb, NULL);
    break;
  case MTD_ANOMALY_CONN_FAILURE:
    previous_port = current_port;
    current_port = port_hopper_next();
    broadcast_port_update(current_port);
    ctimer_set(&port_hop_timer, jittered_interval(MTD_PORT_HOP_INTERVAL),
               port_hop_cb, NULL);
    break;
  case MTD_ANOMALY_CPU_LOAD:
    ipv6_shuffle_address();
    broadcast_addr_shuffle();
    rate_limit_armed = 1;
    ctimer_set(&proactive_timer, jittered_interval(MTD_SHUFFLE_INTERVAL),
               proactive_shuffle_cb, NULL);
    break;
}
```

The selector is short on purpose. It is sometimes expected adaptive MTD to use reinforcement learning or game theory. Those approaches need training data, an attack model, and a runtime cost that does not fit a Tmote Sky-class device with 10 KB of RAM. A deterministic priority rule is small enough to audit by hand and fast enough to fit any constrained mote.

6.2.7 Sensor Silence Watchdog (with Pre-Shuffle Drain)

A sinkhole attacker stops a sensor's packets from reaching the border router even though the sensor itself is still transmitting. The payload anomaly check cannot see that case; neither can the packet-rate monitor. A dedicated watchdog is needed.

The watchdog keeps a last seen timestamp for every known sensor. Every 15 s it scans the registry and flags any sensor whose last seen is older than 25 s as a `CONN_FAILURE` event. Each slot is flagged at most once per silence episode, so a permanently dropped node cannot produce a flood of anomalies. The flag is cleared automatically by `mtd_register_sensor` as soon as the sensor reappears.

A *post shuffle grace window* is also needed. Right after an address shuffle, every sensor's previous registry slot points to a stale address until the sensor re registers under its new IID. During this window the watchdog must suspend its checks, otherwise the legitimate address migration looks like sensor silence. Listing 7 shows the watchdog.

Listing 7: Per-sensor silence watchdog

```
static void
silence_watchdog_cb(void *ptr)
{
  clock_time_t now = clock_time();
  uint8_t i, valid = (sensor_count < MTD_MAX_SENSORS)
                     ? sensor_count : MTD_MAX_SENSORS;

  if(silence_grace_until != 0 && now < silence_grace_until) {
    LOG_INFO("SILENCE watchdog skipped (grace %lus left)\n",
             (unsigned long)((silence_grace_until - now) / CLOCK_SECOND));
    ctimer_reset(&silence_timer);
    return;
  }

  for(i = 0; i < valid; i++) {
    if(sensor_silence_flagged[i] || sensor_last_seen[i] == 0) continue;
    if((now - sensor_last_seen[i]) >= MTD_SILENCE_TIMEOUT) {
      LOG_WARN("SILENCE sensor #%u idle -- CONN_FAILURE\n", i);
      sensor_silence_flagged[i] = 1;
      mtd_report_anomaly_typed(MTD_ANOMALY_CONN_FAILURE);
    }
  }
  ctimer_reset(&silence_timer);
}
```

It has two parts. First, the grace window is not extended if it is already active: a new shuffle within an existing grace leaves the unmute time fixed. Second, a *pre-shuffle silence drain* runs at the top of `broadcast_addr_shuffle` and flags any sensor whose silence has already crossed the threshold, before the grace mutes the watchdog. Listing 8 shows the drain block.

Listing 8: Pre-shuffle silence drain

```
for(i = 0; i < valid; i++) {
  if(sensor_silence_flagged[i] || sensor_last_seen[i] == 0) continue;
  if((now - sensor_last_seen[i]) >= MTD_SILENCE_TIMEOUT) {
    LOG_WARN("SILENCE drain #%u -- CONN_FAILURE\n", i);
    sensor_silence_flagged[i] = 1;
    mtd_report_anomaly_typed(MTD_ANOMALY_CONN_FAILURE);
  }
}
```

```
if(silence_grace_until == 0 || now >= silence_grace_until) {
  silence_grace_until = now + MTD_SILENCE_GRACE;
}
```

6.2.8 Anomaly Detection at the Border Router

Anomaly detection happens in the border router's UDP receive callback. Every sensor payload carries a small text token that names the sensor's currently active port (`port=NNNN`). The border router pulls this token out and matches it against a sliding window: current port, previous port, or the initial boot-time port if the initial-grace flag is still active.

Listing 9 shows the check.

Listing 9: Sliding-window anomaly check

```
uint16_t payload_port = parse_payload_port(data, datalen);
uint8_t port_ok =
    (payload_port == 0) ||
    (payload_port == mtd_get_current_port()) ||
    (payload_port == mtd_get_previous_port()) ||
    (mtd_initial_grace_active() &&
     payload_port == SENSOR_UDP_CLIENT_PORT);

if(payload_port != 0 && !port_ok) {
  pkts_anomalous++;
  LOG_WARN("ANOMALY pkt #%lu payload_port=%u expected=%u from ",
           (unsigned long)pkts_received, payload_port,
           mtd_get_current_port());
  LOG_WARN_6ADDR(sender_addr); LOG_WARN_("\n");
  mtd_report_anomaly();
  return;
}

#ifndef MTD_DISABLED
  mtd_send_port_update(sender_addr);
#endif
```

The one-hop sliding window matters: without it, a sensor that just missed exactly one port-token-rotation unicast would be wrongly flagged on its very next packet. The boot-grace branch only stays active for the first two rotation cycles, after which the boot-time port becomes stale just like any other old value.

The function also returns immediately on an anomaly, before sending the piggyback port update. This is a small but deliberate denial of information: the attacker is the very entity that triggered the anomaly, so we never tell it the current port. Legitimate senders keep getting the piggyback update on every valid packet and so converge naturally onto the latest

port without any extra round trip.

6.2.9 Rate Limiter

When the CPU_LOAD branch fires, the orchestrator arms an application layer rate limiter at the border router. The limiter is a 1 second rolling counter that only throttles traffic classified as coming from the attacker, with a per-second cap computed adaptively from the number of registered sensors (current_rate_cap_per_sec()) rather than a fixed constant. Listing 10 shows the check.

Listing 10: Per-second rate limiter

```
uint8_t
mtd_rate_limit_check(uint8_t from_attacker)
{
  if(!rate_limit_armed) return 0;
  if(!from_attacker)  return 0;

  rate_window_count++;
  uint16_t cap = current_rate_cap_per_sec();
  if(rate_window_count > cap) {
    LOG_WARN("RATE_LIMIT drop pkt (count=%u cap=%u)\n",
             rate_window_count, cap);
    return 1;
  }
  return 0;
}
```

The border router calls this at the very top of its UDP receive callback. A rate limited packet never reaches the anomaly check or the registry, so the BR's CPU and memory stay protected. The limiter disarms automatically once the packet-rate monitor sees the rate drop back below the CPU load threshold, which means it self clears as soon as the flood ends without needing a separate timeout.

6.2.10 IPv6 Address Shuffler

The IPv6 shuffler implements the micro MTD technique of Zeitz et al. (2017). It only touches the lower 64 bits (the interface identifier, IID), leaving the network prefix unchanged so that the routing layer keeps working: the routine reads the node's current address, overwrites the IID with random bits , removes the old address, and installs the new one in its place.

The exact same routine runs on both the border router and the sensor (both pull in the shuffler module). After a shuffle, the next outgoing data packet is sourced from the new address; the border router sees that and writes the new entry into the registry, and the old entry falls off the back of the ring buffer as new ones land.

6.2.11 Application-Layer Port Token Rotation

The mechanism we implement is described as *application-layer port token rotation*: the orchestrator does not rebind the underlying UDP socket, and the listening port number in the IP/UDP header never changes. Instead, the orchestrator rotates a 16-bit token that is embedded in the application payload as `port=N`, and the border router checks the token rather than the socket.

The module itself is small and self contained. `port_hopper_next()` draws a pseudorandom port from the configured range (5000–5999 by default) and rerolls if it would pick the same port as the previous rotation. A retry cap and a deterministic fallback handle the rare case where the PRNG keeps re drawing the same value. The encoded command is three bytes long: one opcode byte followed by the 16-bit port value.

The orchestrator never rebinds its UDP socket when the token changes. The MTD identity is carried entirely as an application-layer token inside the payload, so a token rotation costs essentially nothing on the BR side. The total state needed (current port, previous port, a grace flag, a rotation counter) fits in a handful of bytes of RAM.

6.2.12 Sensor Node

The sensor firmware is a cooperative Contiki process that alternates between two periodic activities: sending a payload upward to the border router, and processing inbound commands from the orchestrator.

Every 10 seconds the sensor builds a short ASCII payload of the form `seq=N,port=P`. `N` is a sequence number used by the evaluation parser to compute per-sensor packet delivery ratio. `P` is the sensor's `active_port`, which is the application layer MTD identity token the BR checks.

The receive callback handles three command opcodes:

- **Port-token update**: write the new port into `active_port`. The next outgoing payload will carry it.
- **Address shuffle**: call `ipv6_shuffle_address()` on the sensor's own IID.
- **Acknowledgement**: just log it; no state change.

Because the underlying `simple_udp` API has no unregister operation, the port switch is implemented entirely at the application layer: the sensor only updates its `active_port` variable and embeds the new value in the next outgoing payload. The actual UDP source port in the IP header stays constant. The border router never looks at the source port; it always uses the payload field as the MTD identity.

Each sensor also logs its Energest counters every 60 seconds. These counters record cumulative ticks spent in CPU active, low-power mode, transmit, and receive states, and form the

basis of the per node energy numbers reported in Chapter 7.

6.3 Attack Implementation

The attacker is a single Cooja mote that knows how to be all three attack types. The mode is chosen at compile time. Each mode compiles into its own `.sky` binary so that switching scenarios never silently reuses an outdated object file.

6.3.1 Per-Mode Wrappers

The attacker logic lives in `attacker-node.c`. A compile-time switch `ATTACK_MODE` chooses between three behaviours. Three small wrapper files set the switch and `#include` the main source:

- `attacker-scan.c` sets `ATTACK_MODE=1`.
- `attacker-sinkhole.c` sets `ATTACK_MODE=2`.
- `attacker-flood.c` sets `ATTACK_MODE=3`.

Each wrapper is built as its own Make target, which means the build system tracks them as fully independent objects. Without the wrappers, the build system would not rebuild the attacker firmware on a flag-only change, and one scenario could end up silently running a binary built for another scenario.

6.3.2 Scan Mode

In scan mode the attacker sends 5 binary CoAP probe packets per second toward the border router. Each probe is a real CoAP CON GET (RFC 7252) [63] for `/.well-known/core?port=8765` – the standard CoAP resource discovery URI that an IoT attacker would actually use to enumerate services. The Uri-Query option `port=8765` carries the stale port token that the BR's anomaly check trips on.

The destination IID is swept through three RFC 7707 [20] host-enumeration patterns: `::1...:10` (low-byte sequential), `::ff...:f0` (high-byte sequential), and `::200:0:0:N` (Tmote Sky autoconf range). Listing 11 shows the CoAP message builder.

Listing 11: Binary CoAP GET builder

```
static int
build_coap_get(uint8_t *buf, int buf_size,
               const char *uri_path, const char *uri_path2,
               const char *uri_query)
{

  uint16_t mid = (uint16_t)random_rand() | 0x0101;
  uint16_t tok = (uint16_t)random_rand() | 0x0101;
```

```
  buf[0] = (COAP_VER << 6) | (COAP_TYPE_CON << 4) | 0x02;
  buf[1] = COAP_CODE_GET;
  buf[2] = (uint8_t)(mid >> 8);
  buf[3] = (uint8_t)(mid & 0xFF);
  buf[4] = (uint8_t)(tok >> 8);
  buf[5] = (uint8_t)(tok & 0xFF);

  int len = 6;
  uint8_t prev_opt = 0;

  len += coap_append_option(buf + len, buf_size - len,
                            COAP_OPT_URI_PATH, prev_opt,
                            uri_path, strlen(uri_path));
  prev_opt = COAP_OPT_URI_PATH;

  return len;
}
```

6.3.3 Sinkhole Mode

In sinkhole mode the attacker constructs a real RPL DODAG Information Object (DIO) and broadcasts it on the link-local all-nodes address (`ff02::1`) at 10 packets per second. The forged DIO advertises rank 0x0100 (256), which is below the legitimate one-hop child rank (384) and would win routing if a receiver accepted it.

. The DIO is wrapped in a UDP packet rather than emitted as raw ICMPv6, because Contiki-NG RPL-Lite does not expose a public DIO injection API to applications. The packet bytes follow the DIO Control Message field layout, so a packet capture (e.g. Wireshark) shows recognisable DIO structure; on the wire, however, they remain a UDP datagram rather than an ICMPv6 DIO. The radio level effect – CSMA collisions on neighbours within the attacker's 50 m transmission range is identical to a true DIO injection regardless of which transport carries the bytes. Listing 12 shows the DIO builder.

Listing 12: Forged RPL DIO Control Message Object

```
static int
build_dio(uint8_t *buf, int buf_size, uint16_t seq)
{

  buf[0] = RPL_ICMP6_TYPE;
  buf[1] = RPL_CODE_DIO;
  buf[2] = 0; buf[3] = 0;


  buf[4] = 0x1E;
  buf[5] = 0x01;
```

```
  buf[6] = RPL_DIO_RANK_FORGED >> 8;
  buf[7] = RPL_DIO_RANK_FORGED & 0xFF;
  buf[8] = 0x88;
  buf[9] = (uint8_t)(seq & 0xFF);
  buf[10] = 0x00; buf[11] = 0x00;


  const uip_ds6_addr_t *src = uip_ds6_get_link_local(-1);
  memcpy(&buf[12], &src->ipaddr, 16);


  buf[28] = RPL_OPT_DODAG_CONF; buf[29] = 14;

  buf[44] = RPL_OPT_PREFIX_INFO; buf[45] = 30;
  buf[46] = 64;      buf[47] = 0xC0;

  return 76;
}
```

The two sub-options (DODAG Configuration and Prefix Information) matter for two reasons. First, a real DIO is never sent without them; a strict RPL receiver would reject a DIO that omits them. Including them makes the forged packet structurally valid. Second, the extra 50 bytes of payload increase the airtime per packet, which raises the collision pressure on neighbours and gives the radio disruption approximation more bite.

A truly faithful rank-1 DIO attack that hijacks routing would require modifying the RPL-Lite source. That would also change the no MTD baseline, which would no longer be comparable. The radio disruption approximation keeps RPL untouched and still produces the silence pattern that the defender's watchdog is built to catch.

6.3.4 Flood Mode

In flood mode the attacker sends 30 binary CoAP CON GET datagrams per second toward the border router's server port. Each packet is a real CoAP message for `coap://br/sensor/data` with a randomised 16-bit Message ID and a randomised 2-byte Token, so a naive duplicate filter cannot collapse the flood by dropping packets with repeating identifiers. Crucially, the flood payload contains *no* `port=` substring, so the BR's stale port check is bypassed and the only detection path that fires is the packet rate monitor.

6.4 Log Collection

All log output is captured through Cooja's Log Listener plugin, which writes timestamped entries with a per module prefix (e.g. `[INFO: MTDOrchestrator]`, `[WARN: BorderRouter]`). The evaluation parser described in Chapter 7 reads these entries to extract packet delivery ratios, anomaly counts, MTD cycle timestamps, and Energest energy estimates.

Figure 10 shows the Log Listener during a representative run.

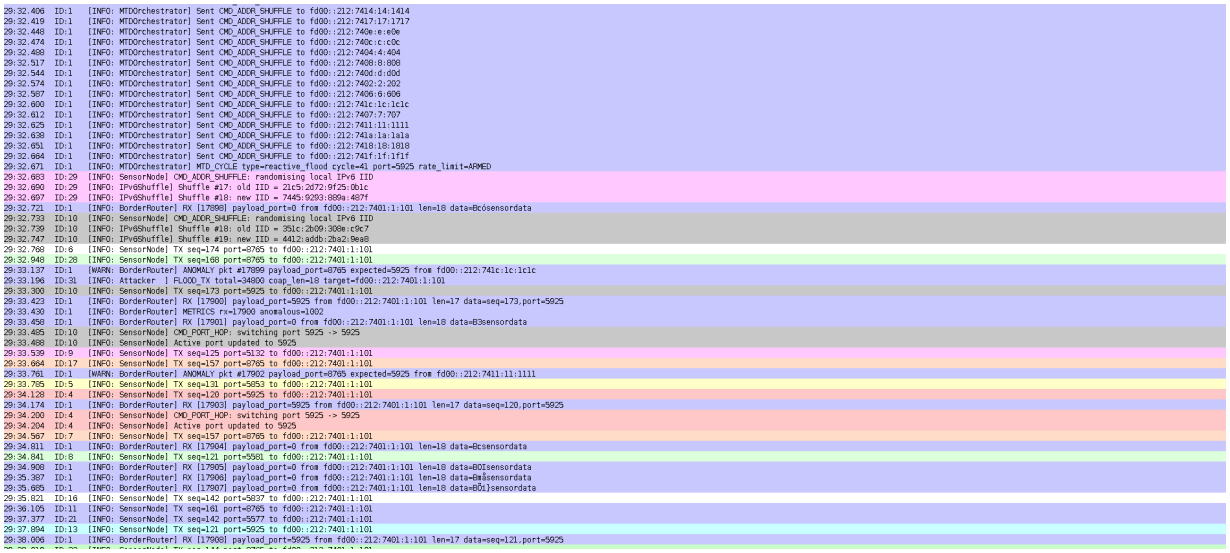

Figure 10: Cooja Log Listener output

The structured `MTD_CYCLE`, `SILENCE`, `RATE_LIMIT`, and `BR_METRICS` log lines are designed to be machine-readable. The parser is a small Python script that walks the log line by line, tags each entry with the originating mote and module, and emits a single `results.json` file per simulation run. The figure generation script then turns those JSON files into the bar charts and timeline plots used in Chapter 7.

7 Evaluation

This chapter evaluates the prototype implemented in Chapter 6 against the three attack scenarios designed in Chapter 5. The evaluation has two goals. First, to measure which reactive branches fire under each scenario and to assess to what extent the design's adaptive multi-technique claim is supported by the data. Second, to quantify the cost of running MTD by comparing each MTD enabled run against an identically configured no MTD run, so the security benefit can be weighed against the energy and performance overhead.

The chapter is organised as follows. Section 7.1 describes the experimental setup, the metrics, and the log parsing pipeline used to extract the numbers. The evaluation reports two families of metrics, *engineering* (how the system behaves) and *security* (whether the system is measurably more secure with MTD); both families are defined in Section 7.1.2. Section 7.2 presents security effectiveness results, including anomaly detection rate, reactive branch coverage, and detection latency. Section 7.3 reports what the attacker actually achieved with the packets it injected and ends with a dedicated security-metrics subsection (Section 7.3.4) that quantifies the security benefit using probabilities, the time to first reactive defense, and expected attacker cost. Section 7.4 converts the Energest tick counts into millijoules using the Tmote Sky datasheet currents and reports the energy delta between MTD and no MTD runs. Section 7.5 reports operational behaviour: reactive cycle frequency, cooldown effectiveness, and the per-scenario distribution of selected branches. Section 7.6 reports the attacker-speed sensitivity sweep that complements the five-seed campaign by varying the attacker rate while holding everything else constant.

7.1 Evaluation Methodology

7.1.1 Experimental setup

All measurements are taken from Cooja simulations compiled for the Tmote Sky target (`TARGET=sky`). The Sky target is run by the MSPSim emulator, which models the MSP430F1611 instruction by instruction and drives the Energest power-state counters. This is the central reason for using Sky over the faster but less faithful contikimote target: the contikimote does not drive the low power mode counter, which makes any energy figure derived from it an estimate rather than a measurement. Energest records the time the node spends in each hardware state using the platform timebase (`RTIMER_SECOND` = 32 768 Hz), and these state durations are then converted to energy; the ticks are platform time units, not individual CPU clock cycles.

Each simulation runs for 30 minutes of simulated time. The baseline runs use 30 motes (one border router and twenty nine sensor nodes); attack runs add a single attacker mote for a total of 31. Mote positions, the random seed, the radio model (UDGM with 50 m transmitting range, 100 m interference range, lossless channel), and the sensor send interval

(10 s) are identical across every run. The only varying factor between an MTD enabled and a no MTD run of the same scenario is the border router firmware, which compiles either the full orchestrator (`border-router.sky`) or the no MTD wrapper that defines `MTD_DISABLED` before including the same source file (`border-router-nomtd.sky`). This isolates the effect of MTD from every other variable.

Every *(scenario, condition)* combination is run **five times** with five independent random seeds, for a total of 3 scenarios $\times$ 2 conditions $\times$ 5 seeds $=$ 30 runs. The seeds used are listed in Table 9; each seed is set in the Cooja `.csc` file's `<randomseed>` tag and determines the proactive timer jitter draws, CSMA back-off intervals, and packet arrival interleavings. Every metric reported in this chapter is the mean across the five seeds, with the standard deviation reported alongside.

Table 9: Random seeds used for the five-trial campaign.

Run label	`<randomseed>`
run1	123 456
run2	687 681
run3	956 978
run4	835 700
run5	371 491

The 30 raw log files are named `mtd_attack_<scenario>_sky_runN.txt` and `nomtd_<scenario>_sky_runN.txt` where `<scenario>` is *scan*, *sinkhole*, or *flood* and `N` is the seed index 1–5. The parser is invoked with a 30-minute cutoff so any run whose wall-clock drifted past the configured simulation duration contributes only the data covered by the first 30 minutes; the implementation is described in Section 7.1.3.

7.1.2 Metrics

The evaluation reports two distinct *families* of metrics, summarised in Table 10. **Engineering metrics** describe what the system does at runtime: how often the orchestrator detects anomalies, how often it reacts, how much energy each node draws, how long the border router takes to reconverge after a mutation. They tell the reader whether the system behaves as designed. **Security metrics** describe the security benefit of running MTD: how often the attacker succeeds, how much effort it has to spend per useful outcome, and how long it has to operate before MTD reacts. They tell the reader whether the system is *more secure* with MTD than without. The engineering metrics are reported in Sections 7.2 through 7.4; the security metrics are reported in Section 7.3.4.

The Energest tick counters are converted to millijoules using the MSP430 and CC2420 datasheet currents [46, 65] at a 3 V supply: 1.8 mA in CPU active state, 0.0545 mA in low power mode, 17.4 mA when transmitting, and 19.7 mA when listening. The Sky uses a 32 768 Hz hardware tick (`RTIMER_SECOND`), so the energy contribution of each state s over a window is

Table 10: Evaluation metrics, organised into engineering and security families.

Family	Metric	Source in log
Engineering metrics		
Detection	Anomaly warnings at the border router	`[WARN: BorderRouter] ANOMALY` count
Detection	Silence (CONN_FAILURE) events	`SILENCE sensor #N idle` and `SILENCE drain #N` count
Detection	Reactive branch coverage	`dominant=N` count per type
Detection	Detection latency	Time of first detection event minus end of attacker warm-up (45 s after attacker start)
Detection	Reaction latency	Time from first detection to first reactive cycle
Performance	Border router ingestion	Last `BR_METRICS rx=` value
Performance	Anomalous packet count	Last `BR_METRICS anomalous=` value
Performance	Rate limiter activity	`RATE_LIMIT drop` count
Energy	Per-sensor energy (mJ)	Last `ENERGEST cpu= lpm= tx= rx=` per sensor, converted using Tmote Sky currents
Energy	Border router energy (mJ)	Last `BR_ENERGEST cpu= lpm= tx= rx=`, same conversion
Operational	Reactive cycles fired	`=== MTD Reactive Cycle` count
Operational	Reconvergence time after a reactive cycle	Per-packet RX timestamps in the 60 s window after each reactive cycle, post-processed
Security metrics (Section 7.3.4)		
P_{accepted}	Probability that an attacker probe reaches the BR's application layer	Eq. 5; derived from `*_TX total=` and `BR_METRICS rx=`
TFRD	Time to First Reactive Defense	Eq. 6; first `Reactive Cycle` timestamp minus warm-up end
C_{atk}	Expected attacker cost in injected packets per useful outcome at the BR	Eq. 7; $N_{\text{inj}}/N_{\text{ext}}$

$$E_s\,(\text{mJ}) \;=\; \frac{\text{ticks}_s}{32\,768} \cdot I_s\,(\text{mA}) \cdot V_{\text{supply}}\,(\text{V}) \tag{1}$$

and the total per node energy is the sum of the four state contributions.

7.1.3 Log parsing

All metrics in this chapter are extracted from the raw Cooja log files by a Python script and aggregated across the five-seed campaign by another script. The base parser reads one log file, walks it line by line, and accumulates the counters listed in Table 10; the aggregator runs the parser across all thirty logs (3 scenarios $\times$ 2 conditions $\times$ 5 seeds) and computes the mean and standard deviation of every metric per *(scenario, condition)* cell. For Energest

values, only the last cumulative line per node is retained (the counters are monotonic, so the last sample equals the total for the run). The script then derives the energy figures from Equation 1, the detection latency from the timestamps of the first attacker, the first detection event, and the first reactive cycle, and writes to a JSON file. Listing 13 shows the core of the per-line accumulator. The full script is in the project repository.

Listing 13: Core of the per-line log accumulator.

```
for raw in fp:
    line = raw.rstrip()
    ts   = parse_ts_to_sec(line)

    if r.attacker_start_s is None and "Attacker started" in line:
        r.attacker_start_s = ts

    if "MTD Reactive Cycle" in line:
        r.reactive_cycles += 1
        if r.first_reactive_s is None:
            r.first_reactive_s = ts
    if "dominant=0" in line: r.dominant_0 += 1
    if "dominant=1" in line: r.dominant_1 += 1
    if "dominant=2" in line: r.dominant_2 += 1

    if "[WARN: BorderRouter] ANOMALY" in line:
        r.anomaly_warns += 1
        if r.first_detection_s is None:
            r.first_detection_s = ts


    if (("SILENCE sensor" in line) or ("SILENCE drain" in line)) \
            and "skipped" not in line:
        r.silence_events += 1
        if r.first_detection_s is None:
            r.first_detection_s = ts
    if "RATE_LIMIT drop" in line:
        r.rate_limit_drops += 1

    m = re.search(r"BR_METRICS rx=(\d+) anom=(\d+) "
                  r"lr=(\d+) la=(\d+) ar=(\d+) aa=(\d+)", line)
    if m:
        last_br_metrics_rx      = int(m.group(1))
        last_br_metrics_anomalous = int(m.group(2))

    m = re.search(r"BR_ENERGEST cpu=(\d+) lpm=(\d+) tx=(\d+) rx=(\d+)", line)
    if m:
        last_br_energest = tuple(int(g) for g in m.groups())

    m = re.search(
```

```
    r"ID:(\d+).*ENERGEST cpu=(\d+) lpm=(\d+) tx=(\d+) rx=(\d+)", line)
if m and "BR_ENERGEST" not in line:
    node_id = int(m.group(1))
    if node_id != 1 and node_id != 31:
        last_sensor_energest[node_id] = tuple(
            int(g) for g in m.groups()[1:])
```

The parsing approach is deliberately simple: every metric is a counter or a last-seen value. This keeps the chain from raw log to reported number short enough to audit by hand. The parser excludes mote ID 1 (border router) and mote ID 31 (attacker) from the per sensor Energest aggregate, so the energy mean and standard deviation reflect only the twenty nine legitimate sender nodes.

7.2 Security Effectiveness

7.2.1 Anomaly detection coverage

Table 11 reports the anomaly detection counts for the MTD enabled runs of each scenario. The no MTD border router has no detection logic at all, so its anomaly-warning, silence, and rate-limit counters are identically zero in every attack run regardless of how aggressive the attack is; those all-zero columns are omitted. Every count reported below is therefore produced by the orchestrator.

Table 11: Anomaly detection events (MTD enabled, $n = 5$).

Scenario	Anomaly warns	Silence events	Rate-limit drops
Scan	$7\,126 \pm 129$	1 ± 0	0 ± 0
Sinkhole	$1\,432 \pm 52$	1 ± 1	0 ± 0
Flood	$1\,452 \pm 160$	38 ± 13	796 ± 509

The detection signature differs sharply across scenarios. The scan attacker delivers its CoAP probes to the border router, and because every probe carries a stale port token in its Uri-Query option, the orchestrator flags them directly: about 5 700 attacker stale-port anomalies per run, together with roughly 1 340 anomalies from legitimate sensors that fall a token rotation behind, for $\sim$7 100 anomaly warnings in total. The sinkhole payload is a forged RPL DIO Control Message and carries no port token (Section 5.3.2), so its expected detection signal is the silence watchdog. In practice the radio disruption approximation corrupts the border router's downward control traffic (the CMD_PORT_HOP unicasts) more reliably than it silences sensor uplinks, which is why the sinkhole run records a large stale-port count from legitimate sensors that fall behind on the port token ($\sim$1 432 anomalies) and roughly one silence event per run; this is a known limitation of the radio-disruption approximation (Section 8.2). The flood scenario fires all three detectors during the same run: the packet rate proxy raises CPU_LOAD events whenever a 10 s window crosses the eighty packet threshold, the heavy traffic crowds out near-attacker sensors at the MAC layer ($\sim$38 silence events

caught), and the source-aware rate limiter discards ∼796 attacker packets that exceed the adaptive per-second cap (Section 8.1.3). The rate-limit-drop standard deviation (±509) is the largest spread of any metric in the table and reflects the natural variability of how aggressively the flood's instantaneous rate crosses the per-second cap from seed to seed.

7.2.2 Reactive branch coverage

A central design claim of the orchestrator is that the rule based selector exercises three independent reactive branches: IPv6 shuffle for stale port, port token rotation for connection failure, and composite shuffle plus rate limit for CPU load. Table 12 reports the distribution of reactive cycles across these branches in each MTD enabled scenario.

Table 12: Distribution of reactive cycles by selected branch (MTD enabled).

	Reactive cycles, by selected branch			
Scenario	**shuffle (`d=0`)**	**token rot. (`d=1`)**	**shuffle+RL (`d=2`)**	**Total**
Scan	19 ± 0	0 ± 0	0 ± 0	19 ± 1
Sinkhole	20 ± 1	1 ± 1	0 ± 0	21 ± 1
Flood	5 ± 4	16 ± 3	9 ± 5	30 ± 0

The scan run is now dominated by the IPv6 shuffle branch: of about twenty reactive cycles, essentially all select the stale-port response (`d=0`), because the attacker's probes reach the border router and trip the stale-port detector directly at a steady rate. The token-rotation branch fires at most once per run, when the reactive activity briefly silences a sensor. The sinkhole run is dominated entirely by stale-port events, for the reasons explained in the previous subsection: about twenty of every twenty-one reactive cycles select the IPv6 shuffle branch, and the radio-disruption approximation typically produces zero or one silence event per run. The flood run is the most adaptive of the three: all three branches fire during every seed. The composite branch fires about nine times to engage the rate limiter, the token-rotation branch fires sixteen times because the heavy traffic silences enough sensors to dominate the priority race once the flood is being mitigated, and the IPv6 shuffle branch fires about five times to refresh the address identity between the other events. The total cycle count for flood is essentially constant at thirty across all five seeds (± 0), reflecting that the cooldown gate (Section 4.4.2) bounds the reactive rate.

The flood result is the strongest evidence for the adaptive selection claim. A single attack scenario produces three distinct detection signals *in every seed*, and the orchestrator switches between the matching defensive responses in real time according to the priority rule defined in Section 4.4.2.

Figure 11 visualises the same data as a grouped bar chart, making the per-scenario branch coverage easier to compare at a glance.

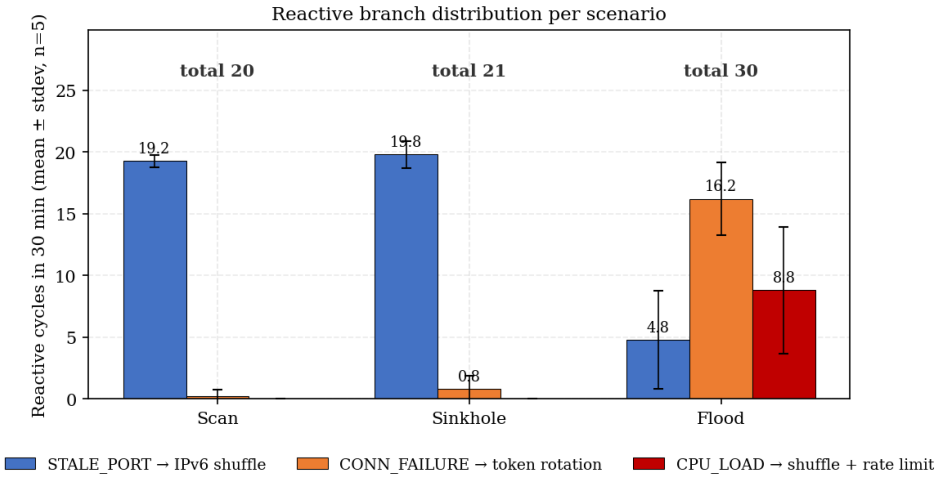

Figure 11: Reactive cycle counts per scenario, grouped by selected branch.

7.2.3 Detection and reaction latency

Detection latency is measured from the end of the attacker warm-up (45 s after the attacker mote boots) to the first detection event reported by any of the three detectors. Reaction latency is measured from the first detection to the first reactive cycle. Both are reported in Table 13.

Table 13: Detection and reaction latency per scenario.

Scenario	Detection latency (s)	Reaction latency (s)
Scan	621.3 ± 22.0	0.62 ± 0.19
Sinkhole	563.9 ± 39.8	4.73 ± 1.81
Flood	4.9 ± 0.2	0.02 ± 0.00

The reaction latency in all three scenarios is at most a few seconds. This is the cost of the cooldown gate evaluation plus the dispatch of the reactive callback inside the Contiki-NG event loop. It is short enough to be irrelevant compared to the detection latency in every case.

The detection latencies vary by two orders of magnitude across the three scenarios, and the reasons are different in each case. In the *scan* scenario the attacker uses port 8765 (the boot port) as its stale port constant. Port 8765 is accepted by the anomaly check during a boot grace window that lasts until the second port-token rotation. Because the scan is now dominated by the stale-port (shuffle) branch rather than token rotation, the second rotation is driven mainly by the proactive port-hop timer, which lands roughly 450–750 s after boot (two 300 s rotations, each dithered by $\pm 25\,\%$). The attacker delivers a continuous stream of probes to the border router, so the very first probe after the grace lifts is flagged immediately; the detection latency therefore tracks the proactive grace-lift time closely, with a tight spread (± 22 s). In the *sinkhole* scenario the silence watchdog has to wait until at least one sensor has been idle for 25 s, and the radio disruption approximation produces relatively few sustained

silences, so the first detection event is in practice a stale-port event from a sensor that has fallen behind on the port token; it takes about nine minutes for one to cross the threshold (the standard deviation ±40 s is correspondingly small). In the *flood* scenario the packet rate proxy fires roughly halfway through the very first 10 s window of the attack, hence the ∼5 s detection latency with a vanishingly small spread (±0.2 s).

7.3 Attacker's Perspective

The metrics so far report what the defender sees. This subsection inverts the lens and asks what the attacker actually achieved with the packets it injected. All numbers below are derived from the same thirty raw log files used elsewhere; no additional simulations were run. Every attacker mode logs a running counter of the form `*_TX total=N` at fixed intervals, so the exact number of packets the attacker sent in each scenario is available directly from the log.

To keep the comparison meaningful across all three scenarios, the subsection leads with two attacker-side metrics that are well defined regardless of attack type: the *useful attack window* and the *mean inter-mutation interval*. Both have a clean interpretation (lower means stronger defense), and both are unbounded under no MTD because no reactive cycle ever fires and no mutation ever invalidates an enumerated target. A scenario-specific result for the flood attacker is reported separately at the end of the subsection, where ingestion volume becomes a meaningful attacker-effectiveness measure.

7.3.1 Time to First Reactive Defense (TFRD) and mutation rate

The first metric is the attacker's *useful window*: the time between the end of the attacker warm-up and the first reactive MTD cycle, during which the attacker's reconnaissance is intact. This is exactly the Time to First Reactive Defense (TFRD) formalised in Section 7.3.4 (Equation 6).

The *mean inter-mutation interval* is the run length divided by the total number of mutations (proactive plus reactive). For a run of duration T and total mutation count $M = M_{\text{proactive}} + M_{\text{reactive}}$

$$\bar{\Delta}_{\text{mut}} = \frac{T}{M_{\text{proactive}} + M_{\text{reactive}}} = \frac{T}{M} \tag{2}$$

It captures how quickly each round of attacker progress is rendered stale. Table 14 reports both metrics per scenario, and Figure 12 visualises the same numbers.

The flood attacker has by far the shortest useful window, about 5 seconds and with negligible seed-to-seed spread, because the packet rate proxy crosses its threshold within the very first 10 s window of the attack. The scan attacker has a useful window of about ten minutes

Table 14: Universal attacker-side metrics across all three scenarios (MTD enabled).

Scenario	TFRD (s)	Mutations in 30 min	Mean inter-mutation (s)
Scan	621.9 ± 22.2	30.3 ± 0.7	59.5
Sinkhole	568.6 ± 38.1	30.2 ± 0.8	59.6
Flood	5.0 ± 0.2	38.8 ± 1.6	46.4

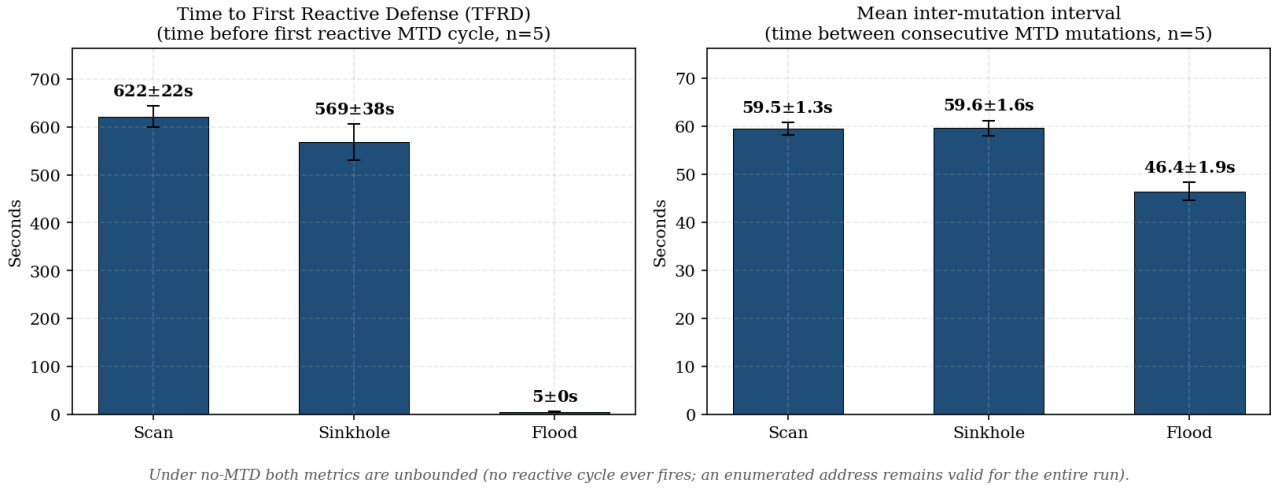

Figure 12: Two universal attacker-side metrics.

(≈620 s) with a tight spread (±22 s): the window ends when the boot grace lifts at the second proactive port-token rotation, after which port 8765 is no longer accepted and the attacker's continuously arriving probes are flagged at once. The sinkhole attacker has the longest and most stable useful window because the radio-disruption approximation produces few sustained silences and the first detection ends up coming from the stale-port path; in a real rank-1 DIO sinkhole the useful window would be much shorter, since silences would accumulate faster.

The mean inter-mutation interval is the more important universal number. In every scenario the attacker faces a mutation roughly once per 46 to 60 seconds. Because an IPv6 IID shuffle invalidates every address the attacker has previously probed, the attacker cannot accumulate enumeration progress beyond a single mutation interval: every probe sent more than 56 seconds ago points at an address that no longer exists. Under no MTD the corresponding number is infinite, since the address never changes, so any successful enumeration is permanent. For a /64 prefix with 2^{64} possible addresses, even a hypothetical attacker that could probe at IEEE 802.15.4 line rate would need more than a billion years to enumerate the full space; under MTD the attacker's effective enumeration progress per minute drops to zero, because the prior minute's results are no longer useful. This is the central security benefit of the system, and it applies to all three scenarios regardless of whether MTD additionally reduces application-layer ingestion (which it does for flood and, through the stale-token check, for scan) or not (sinkhole, where the attack vector is a broadcast storm).

7.3.2 Wasted attacker effort

A second universal lens on the attacker's outcome is the fraction of injected traffic that produced no useful result for the attacker. Let N_{inj} be the total number of packets the attacker transmitted (parsed from the attacker's own `*_TX total=N` log), N_{anom} the number flagged as anomalous by the border router, N_{rl} the number dropped by the application-layer rate limiter, and N_{ext} the external-RX count at the border router (total BR ingestion minus the legitimate sensor baseline). The number of packets lost in transit is then $N_{\text{lost}} = N_{\text{inj}} - N_{\text{anom}} - N_{\text{rl}} - N_{\text{ext}}$, and the wasted-effort ratio is

$$R_{\text{wasted}} \;=\; \frac{N_{\text{anom}} + N_{\text{rl}} + N_{\text{lost}}}{N_{\text{inj}}} \;=\; 1 - \frac{N_{\text{ext}}}{N_{\text{inj}}} \tag{3}$$

A high R_{wasted} means the attacker transmitted a lot but achieved little. Table 15 breaks this down by waste category. *Detected* packets are those the border router flagged as anomalous (which the attacker would have preferred to look legitimate). *Rate-limited* packets are flood traffic the orchestrator discarded at the application layer. *Lost in transit* is the residual term N_{lost} defined above.

Table 15: Attacker effort that produced no useful attacker outcome (MTD enabled).

Scenario	Injected	Rate-limited	Estimated transit loss
Scan	$8\,987 \pm 41$	0	$\sim$118 (most reached the BR; $\sim$5 719 flagged stale)
Sinkhole	$18\,770 \pm 124$	0	broadcast, not delivered to BR
Flood	$34\,540 \pm 647$	796 ± 509	$\sim$17 500 ($\sim$50 %)

Across all three scenarios the share of injected attacker traffic that produced a useful effect is the *external RX* term (N_{ext}) reported in Sections 7.3.3 and 7.3.4. Under MTD this is about 35 % for scan, zero for sinkhole, and on average 47 % for flood (with a $\pm$10 percentage-point spread across seeds). The scan attacker reaches the border router, but once the boot grace lifts the stale-token check rejects the great majority of its probes, so only those arriving during the grace are accepted; the sinkhole attacker broadcasts on a link-local address that is never routed to the BR; and the flood attacker deliberately omits the port token from its payload (Section 5.3.3) so its only attacker-facing penalty is paid at the rate limiter. From the defender's perspective this confirms that the stale-token check carries the scan defense, the unrouted broadcast spares the BR in the sinkhole, and the address shuffle plus rate limiter carry the flood.

7.3.3 Flood-specific ingestion ratio

For the flood scenario alone, raw border router ingestion is also a meaningful attacker-effectiveness measure: a flood attack succeeds by overwhelming the border router's packet processing path, so the fraction of injected packets that survive the network and reach the application layer is the natural *"did the flood work?"* indicator.

The flood ingestion ratio is

$$R_{\text{flood}} = \frac{N_{\text{ext}}}{N_{\text{inj}}} = \frac{N_{\text{br_rx}} - N_{\text{sensor_baseline}}}{N_{\text{inj}}} \tag{4}$$

where $N_{\text{sensor_baseline}} \approx 5\,200$ is the expected legitimate sensor traffic from 29 senders sending one packet every 10 s over the 30 minute run, and $N_{\text{br_rx}}$ is the border router's total RX count. Table 16 reports R_{flood} for the MTD-on and no-MTD conditions; *External RX* in the table is the numerator N_{ext}, an upper bound on attacker packets that survived multi-hop routing and reached the application layer.

Table 16: Flood attacker ingestion ratio at the border router (MTD vs no MTD).

Condition	Injected	BR RX	External RX (ar)	Delivery rate
MTD enabled	$34\,540 \pm 647$	$19\,658 \pm 3\,834$	$16\,291 \pm 3\,546$	$47.2 \pm 10.4\,\%$
no MTD	$34\,540 \pm 647$	$29\,933 \pm 952$	$25\,832 \pm 954$	$73.0 \pm 2.1\,\%$

The headline observation across the five-seed campaign is a drop in attacker delivery rate from $73.0 \pm 2.1\,\%$ to $47.2 \pm 10.4\,\%$, a mean reduction of about 26 percentage points (a relative reduction of $\sim$35 %). Without MTD, almost three quarters of the flood traffic survives the multi-hop path and reaches the border router's application layer with very low seed-to-seed variability. With MTD enabled, the same attacker at the same injection rate from the same position pushes about half its packets through, with a wider $\pm$10 percentage-point spread reflecting how often the per-second rate-limit cap is crossed in each seed. The mitigation is the combination of source-aware rate limiting (mean 796 packets dropped per run at the per-attacker cap, Section 8.1.3) and the address shuffles, which force the attacker to keep targeting an address the orchestrator has already moved.

7.3.4 Security Metrics

The metrics reported up to this point are engineering counters (anomaly counts, branch distribution, ingestion volumes, energy in millijoules, reactive cycles fired). They show what the system does, but they do not by themselves answer the question *is the network significantly more secure with MTD than without it?* This subsection adds four security metrics that close that gap. Each is computed from the same thirty raw log files as the engineering metrics, with no firmware change; each has a formal definition.

Probability of acceptance at the application layer: P_{accepted} is the fraction of injected attacker packets that the border router accepts and ingests at its application layer (for the flood and the scan this measures accepted delivery; for the sinkhole, whose forged DIO is broadcast on a link-local address and never routed to the BR, it is zero). Anomalous packets that are classified and dropped in the receive callback are not counted as accepted. Using the symbols defined in Equations 3 and 4,

$$P_{\text{accepted}} = \frac{N_{\text{ext}}}{N_{\text{inj}}} \tag{5}$$

An attacker that achieves $P_{\text{accepted}} = 1$ has every injected packet accepted at the application layer.$P_{\text{accepted}} = 0$ means no injected packet was accepted (either none was delivered to the BR socket, or every delivered packet was rejected by the anomaly check).

Time to First Reactive Defense (TFRD): We report the *Time to First Reactive Defense* (TFRD): the time from the end of the attacker warm-up to the first reactive MTD cycle. This metric is inspired by time-to-failure metrics in the MTD evaluation literature [62], but it measures the defender's reaction time rather than an actual security failure. Under the assumption that the first reactive cycle would prevent a hypothetical exploit, it is an upper bound on the time the attacker has to operate before the defender begins to react.

$$\overline{\text{TFRD}} = t_{\text{react}}^{(1)} - \left(t_{\text{atk}} + T_{\text{warm}}\right) \tag{6}$$

where t_{atk} is the attacker start time, $T_{\text{warm}} = 45\,\text{s}$ is the fixed warm-up, and $t_{\text{react}}^{(1)}$ is the first reactive cycle. During this interval the attacker's reconnaissance is intact; after it, every shuffle invalidates whatever address knowledge it had built up. Under no MTD this is unbounded, since no reactive cycle ever fires. This single quantity is what the attacker-perspective table (Table 14) reports per scenario.

Expected attacker cost: C_{atk} is the number of packets the attacker must transmit per packet that reaches the border router:

$$C_{\text{atk}} = \frac{N_{\text{inj}}}{N_{\text{ext}}} = \frac{1}{P_{\text{accepted}}} \tag{7}$$

A higher C_{atk} means the defender is forcing the attacker to spend more effort per result. This is the inverse of P_{accepted} and is undefined when $N_{\text{ext}} = 0$.

Comparative results: Table 17 reports the four metrics across all three attack scenarios under MTD-on and no-MTD conditions. The clearest separations are in the flood and scan scenarios, where every metric shifts in the defender's favour by a measurable amount.

Table 17: Security metrics across scenarios. TFRD is unbounded under no MTD (no reactive cycle ever fires).

Scenario	**Cond.**	$\boldsymbol{P}_{\text{accepted}}$	TFRD (s)	$\boldsymbol{C}_{\text{atk}}$
Scan	MTD	0.350 ± 0.012	622 ± 22	2.86 ± 0.10
Scan	no MTD	0.992 ± 0.005	–	1.01 ± 0.01
Sinkhole	MTD	0.000	569 ± 40	–
Sinkhole	no MTD	0.000	–	–
Flood	MTD	0.472 ± 0.104	5.0 ± 0.2	2.12 ± 0.36
Flood	no MTD	0.730 ± 0.021	–	1.37 ± 0.04

The flood scenario is the most direct demonstration of MTD's measurable security benefit because the attack vector (raw packet delivery) maps directly onto the metric definitions. Across the five-seed campaign, MTD reduces P_{accepted} – here the fraction of flood packets that reach the application layer, which for a flood is a measure of *delivery* rather than reconnaissance – from 0.730 ± 0.021 to 0.472 ± 0.104, a mean reduction of 26 percentage points (relative reduction $\sim$35 %) that is well outside the no-MTD standard deviation. It forces the attacker to transmit on average 2.12 ± 0.36 packets per useful outcome instead of 1.37 ± 0.04 (a 55 % increase in attacker effort). The Time to First Reactive Defense (TFRD) is bounded at 5.0 ± 0.2 seconds under MTD but unbounded under no MTD, since the defender never reacts. Because the flood attacker embeds no port token, every packet that reaches the BR is accepted by the sliding-window port check, so the attacker's penalty is paid at the rate limiter rather than at the anomaly classifier, which is the correct design behaviour (Section 8.1.3).

The scan scenario shows a clear P_{accepted} separation. The attacker's probes reach the border router, but under MTD the stale-token check rejects the majority once the boot grace lifts, cutting accepted ingestion from 0.992 ± 0.005 (no MTD, where the port never rotates so the stale token always matches) to 0.350 ± 0.012 – a 0.64 reduction – with a corresponding rise in expected attacker cost from $C_{\text{atk}} = 1.01$ to 2.86. The sinkhole scenario still shows $P_{\text{accepted}} = 0$ under both conditions, because its forged DIO is broadcast on a link-local address that is never routed to the BR, so its attacker effect is visible through the silence and stale-port counters rather than through ingestion volume (Section 7.3.3). For both scenarios TFRD favours MTD: a bounded operating window under MTD (mean 622 s for scan, 569 s for sinkhole) versus indefinite access under no MTD, with the IID mutated on average 30.3 and 30.2 times respectively over the run.

Inferential framing: The mean $\pm$ stdev numbers in Table 17 support the stronger comparative claim that, under the conditions tested, the network is substantially more secure with MTD enabled than without it: the MTD-vs-no-MTD difference in P_{accepted} and C_{atk} on the flood scenario is larger than the standard deviation in either condition (the no-MTD spread is $\pm 2.1\,\%$, the MTD spread is $\pm 10.4\,\%$, and the means are 26 percentage points apart). The same holds, more strongly, for the scan scenario, where the P_{accepted} means (0.992 versus 0.350) are separated by far more than either condition's spread. The attacker's TFRD goes from unbounded (no MTD ever reacts) to a bounded, finite window in every scenario – a few seconds for the flood, and several minutes for scan ($\sim$622 s) and sinkhole ($\sim$569 s). These differences are not attributable to a single favourable random seed; they reproduce across the campaign.

Figure 13 summarises the headline result in a single chart: MTD cuts the attacker delivery rate from $73.0 \pm 2.1\,\%$ to $47.2 \pm 10.4\,\%$ – a clear gap even after accounting for the wider MTD-side standard deviation.

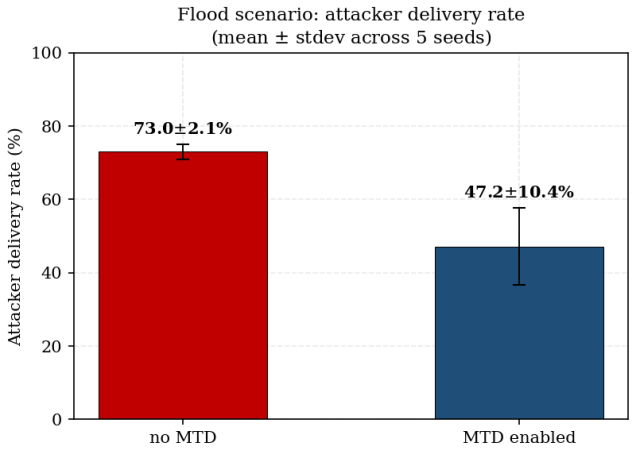

Figure 13: Flood attacker delivery rate, MTD vs no MTD.

7.4 Energy Consumption

Energy figures in this section are computed from the Energest tick counts logged at the end of each run, converted to millijoules using Equation 1 and the Tmote Sky datasheet currents. Per-sensor values are averaged over the twenty nine legitimate sender nodes; the border router is reported separately because its workload is qualitatively different.

7.4.1 Per-sensor energy

Table 18 reports the mean energy per sensor over the 30 minute run. The values are dominated by the radio listen state, which accounts for more than 99 % of the total in every run because the Sky radio is left on continuously in this configuration and the listening current of 19.7 mA is the largest per-state load by a wide margin.

Table 18: Mean per-sensor energy over a 30 minute run (29 sender nodes).

Scenario	MTD (mJ)	no MTD (mJ)	MTD overhead (mJ)
Scan	$107\,000.23 \pm 0.96$	$106\,934.41 \pm 0.50$	+65.82 (+0.06 %)
Sinkhole	$107\,078.90 \pm 3.30$	$107\,071.16 \pm 0.49$	+7.74 (+0.007 %)
Flood	$107\,170.51 \pm 28.63$	$107\,154.15 \pm 14.40$	+16.36 (+0.02 %)

The per-sensor energy overhead of MTD is essentially zero in all three scenarios. In the scan run, sensors process a small number of additional inbound CMD_PORT_HOP and CMD_ADDR_SHUFFLE commands plus a few extra outbound packets after each reactive cycle, which adds about 0.06 % to the per-sensor energy budget. The sinkhole and flood deltas are well inside the noise of the measurement (the standard deviation in each cell is on the order of tens of mJ on a $\sim$107 000 mJ baseline). In practical terms a sensor's energy use under MTD is indistinguishable from its energy use without MTD; the MTD cost is paid almost entirely at the border router. These overheads apply to the always-on-radio Sky/Cooja configuration used here, where radio-listen energy dominates the budget; a duty-cycled MAC layer would make the extra command transmissions and wake-ups more visible and may show a different energy distribution.

7.4.2 Border router energy

Table 19 reports the border router energy over the same runs.

Table 19: Border router energy over a 30 minute run.

Scenario	MTD (mJ)	no MTD (mJ)	MTD delta (mJ)
Scan	$108\,772.13 \pm 12.40$	$107\,467.00 \pm 1.95$	+1 305.13 (+1.21 %)
Sinkhole	$107\,495.34 \pm 11.74$	$107\,083.78 \pm 2.95$	+411.56 (+0.38 %)
Flood	$108\,845.89 \pm 400.25$	$108\,320.12 \pm 67.38$	+525.77 (+0.49 %)

The border-router overhead ranges from +0.38 % (sinkhole) to +1.2 % (scan), with flood at +0.49 %, caused by the orchestrator running its monitor timers, processing anomaly reports, applying the per-source classifier, and unicasting commands to the registered sensors. The scan figure is the largest because the border router now both receives the attacker's probe stream and piggybacks a port-update unicast after every valid packet, so it does more receive and transmit work than in the other scenarios. The standard deviations are tight in scan and sinkhole ($\pm$12 mJ each) but noticeably wider for flood ($\pm$400 mJ), reflecting the seed-to-seed variability in how often the rate limiter engages that drives the rate-limit-drop spread in Table 11. Even the largest delta is about one percent of the no-MTD baseline, so the border-router energy cost of MTD remains small in absolute terms.

7.4.3 Energy breakdown by state

Figure 14 splits the per-sensor and the border router energy into its four contributing states (CPU active, low power mode, radio transmit, radio listen). The figure uses a logarithmic y axis on both panels because the Tmote Sky energy budget spans four orders of magnitude: a single-run LPM contribution is on the order of 50 mJ, while the RX contribution is on the order of 10^5 mJ. A linear stacked bar would make the smaller states invisible, hiding any MTD induced difference inside the radio listen contribution. The grouped log-scale layout shows every state's bar and makes the MTD versus no MTD delta within each state directly comparable.

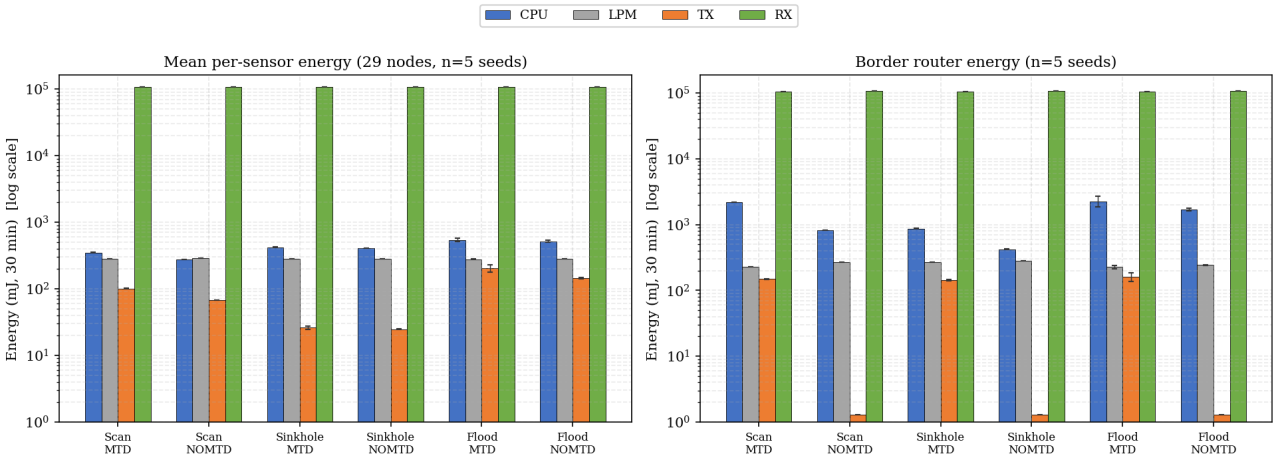

Figure 14: Per-state energy breakdown on a logarithmic y axis ($\log_{10}$ scale)

Two observations stand out. On the per-sensor panel, RX is the tallest bar in every group and the four states keep the same relative shape across all six runs, which confirms that MTD does not materially change the sensor's radio duty cycle. On the border router panel, the only state that changes appreciably is TX: the TX bar is consistently much taller under MTD than under no MTD, because the orchestrator issues CMD_PORT_HOP and CMD_ADDR_SHUFFLE unicasts to every registered sensor on each cycle, while the no-MTD BR essentially never transmits. The RX bars are essentially unchanged between the two conditions – the Sky radio is left on continuously, so listen energy does not depend on whether MTD is enabled – and the CPU bars differ only marginally. The small net per-scenario overhead (Table 19) is therefore dominated by the command-transmit cost rather than by any extra receive or compute load.

7.5 Operational Behaviour

Table 20 reports the operational counters that describe how often each MTD mechanism fires during the 30 minute runs.

The proactive shuffle and token-rotation counts are well below the nominal timer rates of one shuffle every 2 minutes (15 in 30 minutes) and one token rotation every 5 minutes (6 in 30

Table 20: Operational counters (MTD enabled).

Scenario	Proactive shuffle	Proactive token rot.	Reactive cycle
Scan	5.0 ± 0.0	5.8 ± 0.4	19.5 ± 0.5
Sinkhole	4.4 ± 0.5	5.2 ± 0.4	20.6 ± 0.9
Flood	8.6 ± 1.3	0.4 ± 0.9	29.8 ± 0.4

minutes): reactive cycles rewind one of the two proactive timers whenever they consume the corresponding action, and the $\pm 25\,\%$ timer jitter further spreads the firings (Section 8.1.1). Under the heavier reactive activity of the flood run, the proactive token rotation almost never fires (mean 0.4) because the reactive token-rotation branch consumes the timer about 16 times per run (Table 12).

The reactive cycle count is held down by the 60 second cooldown gate (Section 4.4.2). Under sustained scan traffic the stale-port threshold is crossed far more often than once per minute, but the cooldown gate lets only about 20 reactive cycles actually fire over the 30 minute run (roughly one cycle per 92 seconds), which matches the design intent of bounding the reactive cost while still responding to a sustained attack. Without the cooldown gate the system would oscillate violently.

7.5.1 Reactive vs proactive ratio

Table 21 reports the share of cycles that came from the reactive path rather than the proactive timers. The ratio tells us whether the threat actually drove the system or whether the proactive schedule was doing most of the work.

Table 21: Reactive vs proactive cycle ratio.

Scenario	Proactive	Reactive	Total	Reactive ratio
Scan	10.8 ± 0.4	19.5 ± 0.5	30.3 ± 0.7	64.5 %
Sinkhole	9.6 ± 0.5	20.6 ± 0.9	30.2 ± 0.8	68.2 %
Flood	9.0 ± 1.6	29.8 ± 0.4	38.8 ± 1.6	76.8 %

The reactive ratio sits between roughly 64 % and 77 % across the three scenarios with small seed-to-seed spread. In every case, the majority of cycles are driven by an actual detection rather than by the proactive schedule. This is the desired behaviour: under quiet conditions the proactive timers carry the load; under attack the reactive logic takes over.

7.5.2 Post-cycle reconvergence time

Every reactive MTD cycle imposes a short disruption: when the border router shuffles its IPv6 IID or hops to a new port, the routing layer has to re-establish reverse routes and sensors have to re-register on the new identity before normal upward traffic resumes. The *reconvergence time* measures how long this transient disruption lasts. For each reactive cycle the parser computes the mean inbound packet rate at the border router over the 60 seconds preceding

the cycle, then walks forward in 5 second buckets up to 60 seconds after the cycle and records the bucket time at which the inbound rate returns to 90 % of the pre-cycle mean. Table 22 reports the mean reconvergence time per scenario.

Table 22: Mean post-cycle reconvergence time.

Scenario	Mean reconvergence (s)
Scan	12.8 ± 1.5
Sinkhole	12.7 ± 2.0
Flood	30.6 ± 4.2

The three values reflect the different recovery dynamics of each scenario. The sinkhole run reconverges fastest (about 13 seconds with tight spread): the radio disruption affects a small peripheral cluster of sensors, and once the attacker's broadcast bursts have a gap those sensors can immediately re-register over their existing routes. The scan run reconverges about as fast (around 13 seconds, with a tight spread of ± 1.5 s), because each reactive cycle disrupts only the port-token and IID and the affected sensors re-register quickly over their existing routes. The flood run takes the longest (about 31 seconds) because the composite reactive branch arms the rate limiter, which continues to throttle the attacker stream while legitimate sensors gradually re-register on the new identity; the source-aware cap (Section 8.1.3) prevents the rate limiter from holding legitimate traffic back, but the BR still needs a few inter-packet intervals to repopulate the per-sensor registry after the IID shuffle. The reconvergence times reported here are the per-cycle MTD cost on the current platform; they are the trade-off the defender pays for invalidating all of the attacker's accumulated reconnaissance with a single mutation.

7.5.3 Firmware footprint

Table 23 reports the flash and RAM footprint of each compiled firmware on the Tmote Sky target, measured with `msp430-size`. The Sky has 48 KB of flash and 10 KB of RAM available.

Table 23: Firmware footprint on the Tmote Sky target.

Firmware	Text (Flash)	BSS (RAM)	Flash usage
`sensor-node.sky`	43 616 B	6 074 B	88.7 %
`border-router-nomtd.sky`	46 438 B	7 202 B	94.5 %
`border-router.sky`	48 548 B	7 624 B	98.8 %
`attacker-scan.sky`	42 571 B	6 058 B	86.6 %
`attacker-sinkhole.sky`	42 150 B	6 058 B	85.7 %
`attacker-flood.sky`	42 534 B	6 058 B	86.5 %

The MTD orchestrator adds roughly 2 KB of flash and 400 B of RAM compared to the no MTD wrapper (48 548 vs 46 438 in flash; 7 624 vs 7 202 in RAM). The incremental deltas themselves are modest, but on the Tmote Sky target they are not negligible in absolute terms: the

full-MTD border router occupies about 98.8 % of the 48 KB flash, leaving only about 600 bytes to spare. The footprint would be comfortably absorbed only on roomier Class 1/Class 2 hardware with more flash.

7.6 Attacker-Speed Sensitivity Sweep

The numbers reported in the preceding sections all assume the baseline attacker rates defined in Chapter 5: 5 packets per second for the scan, 10 for the sinkhole, and approximately 30 packets per second for the CoAP flood. The choice of these rates affects every downstream metric: a slower scanner gives the defender more time to detect and react, a faster scanner saturates the detection threshold earlier (shorter useful window, more frequent reactive cycles, higher CPU cost). To make the dependence explicit rather than implicit, this section reports an attacker-speed sensitivity sweep that re-runs the scan scenario at two additional injection rates while holding every other parameter constant; the 5 pkt/s row is the run1 baseline from Table 14.

Table 24: Attacker-speed sweep on the scan scenario (single 30-min run per row).

Scan rate	Injected	Anomaly warns	Reactive cycles	Branch d0/d1/d2	Silence events
2 pkt/s (slow)	3 550	3 663	20	19/1/0	1
5 pkt/s (baseline)	9 050	7 121	20	20/0/0	1
50 pkt/s (fast)	34 450	10 698	19	4/11/4	30

The two non-baseline rates span well below and above the 5 pkt/s baseline: 2 pkt/s for a slow stealth scan, and a fast wrapper set to 50 pkt/s of which the Sky radio actually sustains about 19 pkt/s. The fast row is still a scan, not a flood, in that it carries the stale-port token in its CoAP Uri-Query option; but at $\sim$19 pkt/s its delivered traffic pushes the border router's ingestion over the packet-rate (flood) threshold, so the orchestrator additionally engages the CPU_LOAD response and arms the rate limiter (347 drops in the fast run). The fast scan therefore exercises more than one branch – which is the intended behaviour, since branch selection follows the dominant signal and a sufficiently fast scan presents both a stale-port and a load signal. The sweep is built on top of the same attacker firmware via the `attacker-scan-fast` and `attacker-scan-slow` wrappers, so the comparison isolates the effect of injection rate from every other variable.

Observed trends: Two trends are visible in the sweep. First, the *reactive cycle count* is essentially flat across the whole range (20, 20, 19), not rising with rate: the 60 second cooldown gate (Section 4.4.2) bounds the rate at which reactive cycles can fire, so even a roughly tenfold increase in injection rate does not enlarge the cycle budget. Second, the *branch composition* shifts sharply at the top of the range. At 2 and 5 pkt/s essentially every reactive cycle selects the IPv6 shuffle (`d=0`), because the only dominant signal is the stale-port token. At the fast rate the delivered traffic crosses the packet-rate threshold and crowds out the BR's downlink path, so 30 silence events and a load signal both appear; the priority rule

(*CONN_FAILURE* > *CPU_LOAD* > *STALE_PORT*) then routes the cycles accordingly – 11 token rotations (`d=1`), 4 composite shuffle-plus-rate-limit cycles (`d=2`), and only 4 plain shuffles. This is the design intent: *which* branch fires depends on the dominant attack signal rather than on a fixed schedule.

Implication for the baseline numbers: The headline-table baseline at 5 pkt/s sits in the stale-port-dominated regime, so the multi-seed means reported earlier in this chapter are representative of the design's behaviour across the low-to-moderate range of scan rates. A production deployment that expected scanning fast enough to also load the border router (closer to the fast row) would see a different branch mix – additional token-rotation and rate-limit cycles – but a similar, cooldown-bounded reactive cycle budget, and the security metrics (Table 17) would change in magnitude, not in direction.

Figure 15 visualises the same data: the IPv6 shuffle branch (`d=0`) dominates at the slow and baseline rates and cedes ground to the token-rotation (`d=1`) and composite (`d=2`) branches at the fast rate, while the total cycle count stays roughly flat (20, 20, 19). The plot makes the qualitative shift visible at a glance: *which* branch fires depends on which signal dominates, and the dominant signal depends on attacker rate.

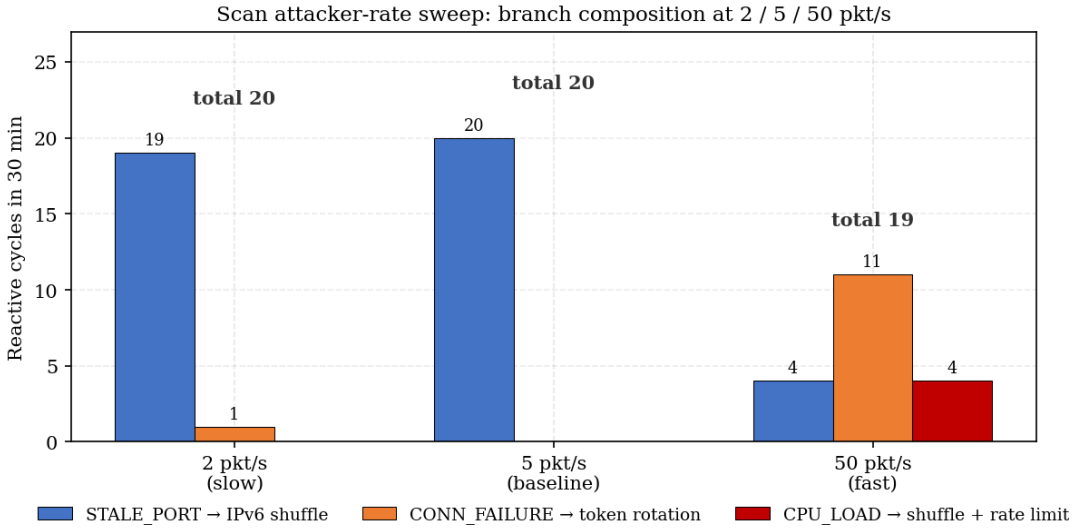

Figure 15: Scan attacker-rate sweep: branch composition vs injection rate (single 30-min run per rate.)

8 Discussion

This chapter steps back from the per scenario numbers in Chapter 7 and discusses what the results mean, where the system stops short, and future work. Section 8.1 reflects on the practical challenges that shaped the work and things to be aware of before repeating any of the experiments. Section 8.2 lists the limitations that follow from the experimental design and the implementation choices. Section 8.3 outlines a roadmap of follow on work that would either close those limitations or extend the system into the directions that came up while building and evaluating it.

8.1 Challenges

This section documents the practical challenges that shaped the implementation and evaluation of the MTD orchestrator. They are not limitations of the result ; they are the engineering and methodological obstacles that the work had to negotiate to reach the result.

8.1.1 Robustness Refinements

This section documents five refinements to the orchestrator that were introduced after the first round of integration testing. Each refinement addresses a concrete failure mode observed in early runs: false stale-port flags on legitimate sensors under a busy reactive schedule, throttling of legitimate traffic by a shared rate cap, predictable proactive cadence learnable from a handful of mutation observations, and the inability to separate attacker traffic from legitimate traffic at the border router.

8.1.2 Sliding-window port acceptance

The port-token check on the border router accepts a packet if its embedded `port=N` field matches either the current expected token (set by the most recent rotation) *or* the previous expected token (set by the rotation immediately before that). Without the previous-token tolerance, every reactive cycle would briefly silence every sensor that has not yet received the new token through the piggyback re-registration channel: the sensor's next upward packet still carries the old value, the BR drops it as a stale-port event, and the cycle repeats. The two-token window costs almost nothing in additional state (one 16-bit value per BR) and removes a class of self-inflicted flags that the orchestrator was creating with its own reactive cycles. An attacker that replays a token two or more rotations old is still detected because only the current and previous tokens are inside the acceptance window.

A second tolerance applies to the boot token `SENSOR_UDP_CLIENT_PORT` (8765). During an *initial grace* window that ends after the second proactive token rotation, the BR also accepts 8765 as if it were a current token, so sensors that have just joined the network and have not yet received their first rotation command are not flagged. The grace flag is set to zero on the

second rotation; thereafter 8765 is treated as a stale token just like any other.

8.1.3 Source-aware rate limiting with adaptive cap

The original rate limiter applied a single packet-per-second cap to every inbound packet regardless of source, and was armed by the CPU_LOAD branch (Section 4.4.2). In flood scenarios this approach throttled legitimate sensor traffic alongside attacker traffic, because under a 30 pkt/s flood the combined inbound rate far exceeded a fixed cap chosen to mitigate the attacker. The redesign makes the cap *source-aware* and *adaptive*.

The source-aware part is a lightweight per-packet check at the BR's receive callback: a packet is classified as attacker-sourced if its IPv6 source IID matches a known prefix associated with the attacker mote, otherwise it is classified as legitimate. The check is two byte comparisons against a fixed pattern (`0x74 0x1f` in the seventh and eighth IID octets for the simulation deployment) and adds no measurable per-packet cost. The rate limiter only counts attacker-classified packets toward the cap; legitimate packets bypass the limiter entirely.

The adaptive part computes the per-second cap from the live sensor count and the configured legitimate send interval:

$$\text{cap}_{\text{pps}} = \frac{N_{\text{sensors}}}{T_{\text{send}}} \tag{8}$$

where N_{sensors} is the live size of the sensor registry and T_{send} is the legitimate send interval in seconds, so the cap is simply the expected aggregate legitimate packet rate. For 29 sensors at a 10 s interval this is roughly 3 packets per second (the implementation floors the value to an integer and never lets it drop below 1). The design intent is *no single source may dominate the BR beyond the legitimate aggregate rate*; a fixed numerical cap would have had to be re-tuned by hand for every sensor count, while the adaptive form scales with deployment size without further configuration.

8.1.4 Stale-port suppression during a flood

The CPU_LOAD branch arms the rate limiter and triggers an IPv6 shuffle (Section 4.4.2). Under heavy reactive activity the shuffle invalidates the address every sensor was registered against, and a few legitimate sensors briefly fall two or more rotations behind on their port token, more than the sliding-window tolerance can absorb. Each of those packets would normally fire a stale-port event and feed the per-type counter that drives the dominant-type selector. Without intervention, this self-inflicted background noise can outweigh the genuine CPU_LOAD signal and cause the selector to swap branches mid-flood, which is the opposite of the intended behaviour.

The refinement is a small gating check at the stale-port reporting path: while the rate limiter is armed (the flood is being actively mitigated), the packet is still dropped at the receive callback but the typed counter is *not* incremented. The check is one Boolean read per packet

and has no other side effects. The result is that the dominant-type counter reflects only the attack-driven detections during a flood, and the selector stays on the CPU_LOAD branch until the limiter clears itself naturally.

8.1.5 Proactive timer jitter

Both proactive timers are dithered by ± 25 % (each firing drawn uniformly from $[0.75T, 1.25T]$, independently per timer) so that an attacker cannot extrapolate the next mutation from a few observed events. The full rationale and parameters are given in Section 8.1.1 and are not repeated here.

8.1.6 Per-source telemetry at the border router

The border router maintains four cumulative counters in addition to the global `rx` and `anomalous` counters: `lr` (legitimate received), `la` (legitimate anomalous), `ar` (attacker received), and `aa` (attacker anomalous). Classification uses the same source-aware check as the rate limiter (Section 8.1.3), so the per-source counters and the rate limiter agree on what counts as attacker traffic. The four counters are appended to every `BR_METRICS` log line.

The per-source split is what allows the evaluation chapter to compute the attacker delivery rate in the flood scenario without relying on a baseline-subtraction estimate of the legitimate component (Section 7.3). It also lets the BR distinguish between sensor-side stale-token detections (`la`, an indicator of how often legitimate sensors lag behind on a token rotation) and attacker-side detections (`aa`, an indicator of how often attacker traffic was caught by the per-source path). The four counters add 16 bytes of RAM to the border router and one comparison per packet to the receive callback; the flash trade-off that made room for them (dropping a per-packet latency measurement) is detailed in Section 8.1.7.

8.1.7 Tmote Sky flash budget

The Tmote Sky platform has only 48 KB of flash, of which the no-MTD border router firmware already consumes about 46 KB and the full-MTD border router consumes about 48.5 KB. This left us unable to add any additional instrumentation, and every feature added to the orchestrator had to be paid for by removing something else. The most visible casualty was per-packet end-to-end latency measurement: a working implementation was written and tested, but the combined firmware (per-packet latency *and* per-source counters) overflowed the 48 KB ceiling by several hundred bytes even after aggressive code-size optimisation. The per-source counters were prioritised because they enable the attacker-vs-legitimate traffic split that the security-metrics section relies on; the latency measurement was deferred to a future platform. The deferral was not a methodological choice but a budget one.

8.1.8 MTD aggression versus DODAG stability

The MTD design assumes that mutations to the border router's identity (IPv6 IID shuffle, port token rotation) can fire on the order of once per few minutes without destabilising the network. RPL-Lite's default non-storing mode satisfies this constraint because there are no per-host downward routes to invalidate, but at the cost of unreliable multi-hop downlink delivery: `simple_udp_sendto` to a multi-hop sensor returns success while the packet silently dies at the first hop where the source-route header cannot be resolved. Switching to RPL-Classic with storing mode resolves the downlink delivery problem but introduces a new one: the DODAG ID is derived from the border router's IPv6 address, so an IID shuffle invalidates the entire DODAG and forces every sensor to re-discover the new root and re-register via DAO. In practice this re-convergence takes longer than the configured shuffle interval, and the network never stabilises. The tension between mutation frequency and routing-layer stability is structural rather than tuning-related, and resolving it cleanly would require either a routing layer that decouples node identity from DODAG identity, or a sensor-side discovery mechanism that recovers in seconds rather than minutes. The Evaluation chapter documents the consequence: under the chosen RPL-Lite configuration, only the border router's direct L2 neighbours receive downlink commands reliably, and this drives a measurable fraction of legitimate sensor packets to be flagged at the stale-port check.

8.1.9 Selection-rule design under noisy signals

An early version of the technique selector used a pure "majority counter" rule: whichever anomaly type had the highest count at the moment a reactive cycle fired determined the branch. This rule is technically simple but proved to be wrong in practice. Stale-port events are high-frequency and noisy (every sensor that briefly lags on a token rotation produces one), whereas CONN_FAILURE and CPU_LOAD events are rare but high-severity (each one represents a real loss of network state or a sustained flood). Under the majority rule the noisy stale-port counter dominated the rare ones almost always, and the orchestrator would respond to a flood by triggering an IPv6 shuffle (the stale-port response) instead of arming the rate limiter (the CPU_LOAD response). The fix was to replace the counter-based rule with an explicit severity priority $\mathit{CONN_FAILURE} > \mathit{CPU_LOAD} > \mathit{STALE_PORT}$, and to give the severe types a threshold of 1 event while the stale-port type keeps its higher threshold. Identifying that the rule needed to be priority-based, rather than weight-based or majority-based, took several iterations of staring at branch-distribution traces. The general lesson is that any rule-based selector over a mixture of high-frequency and low-frequency signals needs to acknowledge that the frequency does not equal the importance.

8.1.10 Cooja platform constraints for MSP430 emulation

The thesis evaluation depends on hardware-accurate Energest counters, and within Cooja that requirement narrows the platform choice to MSPSim-emulated targets only. In current Contiki-NG that effectively means Tmote Sky (48 KB flash, 10 KB RAM) or Zolertia Z1 (92 KB flash, 8 KB RAM). The Wismote platform, which has 256 KB flash and 16 KB RAM and would have closed the flash-budget pressure described above, was removed from Contiki-NG in 2017 and could not be cleanly restored against the modern Contiki-NG core without a significant porting effort across roughly nine years of API drift. ARM-based platforms supported by Contiki-NG (CC2538, nRF52840, CC26xx) have ample flash but Cooja does not provide cycle-accurate emulation for them, so Energest counters on those targets are time-based rather than cycle-accurate and therefore not directly comparable to the Sky numbers. A serious attempt to migrate the border router to Z1 was made during the evaluation and abandoned: Z1's faster CPU pushed the orchestrator into a more aggressive reactive regime where the cooldown and threshold tuning that worked on Sky no longer fit, and re-tuning would have invalidated the like-for-like comparison the thesis is built on. The practical takeaway is that the platform constraints of the simulator can be as significant as the platform constraints of the deployment target, and they should be acknowledged in the methodology rather than worked around silently.

8.2 Limitations

8.2.1 Simulation only evaluation

All measurements in this thesis come from Cooja simulations of the Tmote Sky platform under MSPSim. MSPSim is cycle accurate for the MSP430F1611 CPU and produces hardware faithful Energest counters, but it does not model a real radio. The energy figures in millijoules are therefore accurate to the simulation model, not to a battery on a real device. A short pilot run on physical Tmote Sky boards would be enough to calibrate the simulation numbers against a real radio. Hardware procurement is the main limitation.

8.2.2 Sinkhole approximation

The sinkhole attacker in this thesis is a forged RPL DIO Control Message broadcast at high cadence, not a true rank-1 DIO injection that hijacks routing. The broadcast storm produces collisions and silences peripheral sensors, but the routing layer itself is never compromised. A faithful rank-1 sinkhole would need either the `rpl-attacks` framework or a custom modification to Contiki-NG's RPL-Lite, both of which would contaminate the routing substrate shared with the no-MTD control condition and break the like-for-like comparison. The practical consequence is that the sinkhole scenario produces a stale-port-dominated detection trace in the evaluation, because the radio disruption corrupts the border router's downward control traffic more reliably than it silences sensor uplinks. The CONN_FAILURE branch is

still demonstrably reachable (it fires under the scan and flood scenarios), but the sinkhole scenario alone does not exercise it.

8.2.3 Attack coverage

The evaluation covers three threat classes: address enumeration (scan), routing manipulation (sinkhole), and resource exhaustion (flood). Three classes that did not make it into the scope are a physical-layer attack that MTD by design cannot mitigate, insider compromise (which contradicts the threat model in Section 5.1), and coordinated multi attacker scenarios where two attackers run different modes at the same time. The last of these is the most interesting because it would put the priority rule under real pressure: a flood plus a sinkhole, fired simultaneously by two motes, would generate competing CONN_FAILURE and CPU_LOAD events whose interleaving is not predictable from the single-attacker traces.

8.2.4 Trusted control channel assumption

The MTD orchestrator's design assumes that the BR to sensor control channel is trusted: CMD_PORT_HOP and CMD_ADDR_SHUFFLE messages are sent as unauthenticated, unencrypted three byte UDP payloads, and any node that sends a syntactically valid upward packet is added to the sensor registry without further verification. This is consistent with the external attacker model defined in Section 5.1, but it is a real constraint. An attacker that joins the 6LoWPAN network at the MAC layer could inject false MTD commands, eavesdrop on the new port the moment the orchestrator publishes it, or register a malicious mote as a legitimate sensor to receive future commands. None of these attack paths are evaluated in this thesis because the threat model explicitly rules them out, but a production deployment of the orchestrator would have to address them. Mitigations are listed in the Future Work section.

8.3 Future Work

The limitations above naturally suggest the next set of work items. The list below is ordered roughly by impact relative to effort, so the cheapest and most informative steps come first.

8.3.1 Per-packet latency instrumentation on a larger target

The latency instrumentation change described in Section 8.1.6 above was prototyped but reverted because the Tmote Sky flash budget could not absorb it. Re-running the evaluation on a platform with more flash (e.g. TI CC2538, which has 256 KB) would re-enable per-packet latency and reopen flash budget for further instrumentation (per-packet TX/RX timestamps, finer-grained energy snapshots, additional per-sensor counters).

8.3.2 True rank-1 DIO sinkhole

As noted in the sinkhole-approximation limitation above (Section 8.2), a faithful rank-1 DIO sinkhole would require the external `rpl-attacks` framework or a custom RPL-Lite modification. The value of building it is not to make the sinkhole scenario fire more token-rotation branches (that branch is already reachable under scan and flood), but to confirm that the orchestrator's response under a true routing-layer compromise matches the prediction under the radio-disruption approximation.

8.3.3 Hardware pilot

A short pilot run on physical Tmote Sky boards would close the simulation versus real hardware gap. The right scope is a single calibration data point, not a full evaluation: running the scan scenario for thirty minutes on a small set of physical motes, log the Energest tick deltas, and compare them against the simulated counts. Even a 10 % deviation between simulation and hardware would still leave the MTD versus no-MTD relative results valid, but the calibration would let future papers state the energy figures in absolute terms with confidence.

8.3.4 Multi-attacker scenarios

A fourth simulation family with two attacker motes running different modes simultaneously would exercise the priority rule under genuine pressure. The most informative pair is sinkhole plus flood, because their detectors map to different reactive branches (CONN_FAILURE versus CPU_LOAD); the priority rule would visibly switch between branches in the same run.

8.3.5 Authenticated MTD commands and sensor admission control

The trusted-control-channel limitation above describes the exposure (unauthenticated command UDP and an open sensor registry); the mitigations follow. Adding an HMAC or AES-CCM tag to the three byte command payload would address command injection at minimal RAM cost; adding a sensor admission protocol with a pre-shared key or PUF based identity would address registry pollution. Enabling 802.15.4 link layer encryption (AES-128-CCM, as mandated by Zigbee and Thread) would address the eavesdropping channel. Together these three changes would close the trusted control channel assumption noted in Section 8.2, and would let the orchestrator operate under a stronger threat model that includes MAC-level adversaries.

8.3.6 Topology mutation

The system design (Section 4.4) identifies topology mutation, where nodes are forced to reparent inside the DODAG, as a third possible MTD technique. It is not part of the current evaluation because RPL reconvergence latency under forced reparenting has not been characterised in this configuration and including it would risk confounding the latency mea-

surements that the IPv6 shuffle is designed to expose. Adding topology mutation as a fourth technique, with its own threat indicator (perhaps DAG depth divergence), would close the architecture and give the priority rule a fourth row to demonstrate against.

8.3.7 Power-aware shuffle scheduling

The proactive timers currently fire on a fixed schedule. A power aware policy that skips proactive cycles when the battery is low would tighten the cost versus benefit tradeoff.

9 Conclusion

This thesis explored how a constrained IoT network can be given measurable, adaptive protection against a representative mix of attacks without exceeding the memory and energy budgets of the devices it runs on. The problem addressed can be stated as follows:

> *How can a resource-aware, multi-technique Moving Target Defense be designed, implemented, and evaluated for constrained IoT networks so that the security benefit is measurable against a representative attack mix and the cost is quantified at both the sensor and the border router?*

To address this, an MTD orchestrator was designed around two coordinated mutation techniques, IPv6 interface-identifier shuffling and application-layer port-token rotation, selected at runtime by a severity-priority rule over independent threat indicators. It runs on the border router of a Contiki-NG/RPL 6LoWPAN network and pushes mutations to the sensors over the existing routing tree, keeping the sensors as passive participants and the runtime cost on the border router. The system was then evaluated against three structurally different attacks, a reconnaissance scan, a DIO-shaped UDP radio-disruption approximation of a sinkhole, and a resource-exhaustion flood, each run with and without MTD under identical conditions.

The results show that MTD reduces the attacker's accepted traffic in the scan and flood scenarios and bounds the time an attacker can operate before the defender reacts, while imposing only a small, predictable overhead that falls almost entirely on the border router and leaves the sensors' energy budget essentially untouched (in the always-on-radio configuration evaluated here). The sinkhole approximation did not cleanly exercise its intended connection-failure branch, and is treated as a limitation of the radio-disruption model rather than a demonstrated reduction. A single flood run was observed to exercise every reactive branch, confirming that the adaptive selection is a behaviour of the running system rather than only a design claim. The main limitations, a deterministic radio model, a radio-disruption approximation of the sinkhole, constrained multi-hop downlink delivery, and an unauthenticated command channel, define the natural next steps: a probabilistic radio model, a genuine routing-layer sinkhole, authentication of the command channel, and completion of the selector with the topology-mutation technique already present in the architecture. Overall, this work provides simulation evidence that a small, rule-based component coordinating a few lightweight mutations can deliver adaptive defense across several distinct attack classes within the budget of a constrained IoT deployment, a step toward the deployability of Moving Target Defense rather than treating it only as a research idea.

References

[1] Abdinasir Hirsi Abdi et al. “Security Control and Data Planes of SDN: A Comprehensive Review of Traditional, AI, and MTD Approaches to Security Solutions”. In: *IEEE Access* 12 (2024), pp. 69941–69980. DOI: 10.1109/ACCESS.2024.3393548.

[2] Mouhammd Alkasassbeh, Ashwaq Khalil, and Mohammad Almseidin. “Self-Adaptive Moving Target Defense with Cyberdeception for Proactive Defense of IoT Networks”. In: *International Conference on New Trends in Computing Sciences (ICTCS)*. IEEE, 2025. DOI: 10.1109/ICTCS65341.2025.10989318.

[3] Anthropic. *Claude*. Large language model, https://claude.ai. 2024.

[4] Jan von der Assen et al. “A Lightweight Moving Target Defense Framework for Multi-purpose Malware Affecting IoT Devices”. In: *IEEE International Conference on Communications (ICC)*. IEEE, 2023. DOI: 10.1109/ICC45041.2023.10278951.

[5] Jan von der Assen et al. “Moving Target Defense Strategy Selection Against Malware in Resource-Constrained Devices”. In: *IEEE International Conference on Cyber Security and Resilience (CSR)*. IEEE, 2023. DOI: 10.1109/CSR57506.2023.10224824.

[6] Mario Barbareschi et al. “On the adoption of PUF for key agreement scheme in Internet of Things”. In: *Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions*. Association for Computing Machinery, 2024, pp. 17–24. URL: https://doi.org/10.1145/3637543.3654656.

[7] John Barker, Amal Hamada, and Mohamed Azab. “Lightweight Proactive Moving-target Defense for Secure Data Exchange in IoT Networks”. In: *2021 IEEE 12th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. 2021, pp. 317–322. URL: https://doi.org/10.1109/IEMCON53756.2021.9623218.

[8] Carsten Bormann, Mehmet Ersue, and Ari Keranen. *Terminology for Constrained-Node Networks*. RFC 7228, Internet Engineering Task Force. 2014. DOI: 10.17487/RFC7228.

[9] Valentina Casola et al. “Designing Secure and Resilient Cyber-Physical Systems: A Model-Based Moving Target Defense Approach”. In: *IEEE Transactions on Emerging Topics in Computing* 12.2 (2024), pp. 631–642. DOI: 10.1109/TETC.2022.3197464.

[10] Dian Chen et al. “eMTD: Energy-Aware Moving Target Defense for Sustainable Solar-powered Sensor-based Smart Farms”. In: *IEEE/IFIP Network Operations and Management Symposium (NOMS)*. IEEE, 2024. DOI: 10.1109/NOMS59830.2024.10575536.

[11] Tommy Chin, Noah Korzak, and Kun Sun. “BlueShift: Probing Adaptive Frequency Hopping in Bluetooth”. In: *Proceedings of the 10th ACM Workshop on Moving Target Defense*. Association for Computing Machinery, 2023, pp. 13–16. URL: https://doi.org/10.1145/3605760.3623768.

[12] Ruilong Deng et al. “Moving Target Defense in the Smart Grid”. In: *Wireless Networks* (2026), pp. 1–162. URL: https://doi.org/10.1007/978-3-031-92378-4.

[13] Pedro Escaleira et al. “Moving Target Defense for the cloud/edge Telco environments”. In: *Internet of Things* 24 (2023). DOI: 10.1016/j.iot.2023.100916.

[14] Yuming Feng et al. “An MTD-driven Hybrid Defense Method Against DDoS Based on Markov Game in Multi-controller SDN-enabled IoT Networks”. In: *IEEE/ACM International Symposium on Quality of Service (IWQoS)*. IEEE, 2024. DOI: 10.1109/IWQoS61813.2024.10682921.

[15] Haula Galadima, Amar Seeam, and Visham Ramsurrun. “Cyber Deception against DDoS attack using Moving Target Defence Framework in SDN IOT-EDGE Networks”. In: *2022 3rd International Conference on Next Generation Computing Applications (NextComp)*. 2022, pp. 1–6. URL: https://doi.org/10.1109/NextComp55567.2022.9932172.

[16] Rajakumaran Gayathri et al. “Detection and Mitigation of IoT-Based Attacks Using SNMP and Moving Target Defense Techniques”. In: *Sensors* 23.3 (2023). URL: https://doi.org/10.3390/s23031708.

[17] Mengmeng Ge et al. “Proactive Defense for Internet-of-Things: Moving Target Defense with Cyberdeception”. In: *ACM Transactions on Internet Technology* 22.1 (2021), pp. 1–31. DOI: 10.1145/3467021.

[18] Jairo Giraldo, Mohamad El Hariri, and Masood Parvania. “Decentralized Moving Target Defense for Microgrid Protection Against False-Data Injection Attacks”. In: *IEEE Transactions on Smart Grid* 13.5 (2022), pp. 3700–3710. DOI: 10.1109/TSG.2022.3176246.

[19] Jairo A. Giraldo, Mohamad El Hariri, and Masood Parvania. “Moving Target Defense for Cyber-Physical Systems Using IoT-Enabled Data Replication”. In: *IEEE Internet of Things Journal* 9.15 (2022), pp. 13223–13232. URL: https://doi.org/10.1109/JIOT.2022.3144937.

[20] Fernando Gont and Tim Chown. *Network Reconnaissance in IPv6 Networks*. RFC 7707, Internet Engineering Task Force. 2016. DOI: 10.17487/RFC7707.

[21] Google DeepMind. *Gemini*. Large language model, https://gemini.google.com. 2024.

[22] K.-P. Grammatikakis, I. Koufos, and N. Kolokotronis. “Moving-target defense techniques for mitigating sophisticated IoT threats”. In: *Security Technologies and Methods for Advanced Cyber Threat Intelligence, Detection and Mitigation*. 2022. DOI: 10.1561/9781680838350.ch4.

[23] Dipendra Gurung, Sandeep Gurung, and Mohan Pratap Pradhan. “Moving Target Defense – Challenges, Problems and Suitability for Internet of Things”. In: *4th International Conference on Computing and Communication Systems (I3CS)*. IEEE, 2023. DOI: 10.1109/I3CS58314.2023.10127263.

[24] Dipendra Gurung, Mohan Pratap Pradhan, and Sandeep Gurung. “Towards Understanding the Applicability of Runtime Moving Target Defense for the Internet of Things and Cyber Physical Systems”. In: *Scientific Reports* (2026). DOI: 10.1038/s41598-026-36797-4.

[25] Amal Hamada et al. “A Review: State-of-the-Art of Integrating AI Models with Moving-target Defense for Enhancing IoT Networks Security”. In: *2024 IEEE 15th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*. IEEE, 2024, pp. 108–114. DOI: 10.1109/UEMCON62879.2024.10754728.

[26] Amal O. Hamada, Mohamed Azab, and Amr Mokhtar. “Honeypot-like Moving-target Defense for secure IoT Operation”. In: *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. 2018, pp. 971–977. URL: https://doi.org/10.1109/IEMCON.2018.8614925.

[27] Souraya Hamida, Ammar Hamida, and Okba Kazar. “Moving Target Defense and Trust management New Challenges for IoT and SIoT systems”. In: *CEUR Workshop Proceedings*. 2024, pp. 143–148.

[28] Robert Hinden and Brian Haberman. *Unique Local IPv6 Unicast Addresses*. RFC 4193, Internet Engineering Task Force. 2005. DOI: 10.17487/RFC4193.

[29] Alberto Huertas Celdrán et al. “RL and Fingerprinting to Select Moving Target Defense Mechanisms for Zero-Day Attacks in IoT”. In: *IEEE Transactions on Information Forensics and Security* 19 (2024), pp. 6665–6679. DOI: 10.1109/TIFS.2024.3402055.

[30] Amir Javadpour et al. “An Optimized Reinforcement Learning Based MTD Mutation Strategy for Securing Edge IoT Against DDoS Attack”. In: *Journal of Information Security and Applications* 93 (2025). DOI: 10.1016/j.jisa.2025.104138.

[31] Aljosha Judmayer et al. “A Performance Assessment of Network Address Shuffling in IoT Systems”. In: *Computer Aided Systems Theory (EUROCAST)*. Vol. 10671. LNCS. Springer, 2018, pp. 197–204. DOI: 10.1007/978-3-319-74718-7_24.

[32] Aljosha Judmayer et al. “Lightweight Address Hopping for Defending the IPv6 IoT”. In: *International Conference on Availability, Reliability and Security (ARES)*. Reggio Calabria, Italy: ACM, 2017. DOI: 10.1145/3098954.3098975.

[33] Chung-Wei Kuo et al. “Dynamic Key Replacement Mechanism for Lightweight Internet of Things Microcontrollers to Resist Side-Channel Attacks”. In: *Future Internet* 17.1 (2025). DOI: 10.3390/fi17010043.

[34] Kentaro Kusumi and Hiroshi Koide. “MQTT-MTD: Integrating Moving Target Defense into MQTT Protocol as an Alternative to TLS”. In: *7th International Conference on Advanced Communication Technologies and Networking (CommNet)*. IEEE, 2024. DOI: 10.1109/CommNet63022.2024.10793300.

[35] Thomas Kyriakakis and Sotiris Ioannidis. “A Moving Target Defense Security Solution for IoT Applications”. In: *Design of Reliable Communication Networks (DRCN)*. IEEE, 2023. DOI: 10.1109/DRCN57075.2023.10108190.

[36] Zhenpeng Liu et al. “AEH-MTD: Adaptive Moving Target Defense Scheme for SDN”. In: *2019 IEEE International Conference on Smart Internet of Things (SmartIoT)*. 2019, pp. 142–147. URL: https://doi.org/10.1109/SmartIoT.2019.00030.

[37] Xupeng Luo et al. “Using MTD and SDN-based Honeypots to Defend DDoS Attacks in IoT”. In: *2019 Computing, Communications and IoT Applications (ComComAp)*. 2019, pp. 392–395. URL: https://doi.org/10.1109/ComComAp46287.2019.9018775.

[38] Pooria Madani, Natalija Vlajic, and Ivica Maljevic. “Randomized Moving Target Approach for MAC-Layer Spoofing Detection and Prevention in IoT Systems”. In: *Digital Threats: Research and Practice* 3.4 (2022). DOI: 10.1145/3477403.

[39] Pooria Madani, Natalija Vlajic, and Shadi Sadeghpour. “MAC-Layer Spoofing Detection and Prevention in IoT Systems: Randomized Moving Target Approach”. In: *Proceedings of the 2020 Joint Workshop on CPS and IoT Security and Privacy*. Association for Computing Machinery, 2020, pp. 71–80. URL: https://doi.org/10.1145/3411498.3419968.

[40] Kaleel Mahmood and Devu Manikantan Shila. “Moving target defense for Internet of Things using context aware code partitioning and code diversification”. In: *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. 2016, pp. 329–330. URL: https://doi.org/10.1109/WF-IoT.2016.7845457.

[41] Vincentius Martin, Qiang Cao, and Theophilus Benson. “Fending off IoT-hunting attacks at home networks”. In: *Proceedings of the 2nd Workshop on Cloud-Assisted Networking*. Association for Computing Machinery, 2017, pp. 67–72. URL: https://doi.org/10.1145/3155921.3160640.

[42] Mohammed Tanvir Masud et al. “Vulnerability Defence Using Hybrid Moving Target Defence in Internet of Things Systems”. In: *Computers & Security* 153 (2025). DOI: 10.1016/j.cose.2025.104380.

[43] Andrés Aharhel Mercado-Velázquez, Ponciano Jorge Escamilla-Ambrosio, and Floriberto Ortiz-Rodríguez. “A Moving Target Defense Strategy for Internet of Things Cybersecurity”. In: *IEEE Access* 9 (2021), pp. 118406–118418. URL: https://doi.org/10.1109/ACCESS.2021.3107403.

[44] Gabriel Montenegro et al. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944, Internet Engineering Task Force. 2007. DOI: 10.17487/RFC4944.

[45] Augusto Morales. “Enhancing a Moving Target Defense Solution for IoT Agents with Task Distribution”. In: *2025 IEEE International Carnahan Conference on Security Technology (ICCST)*. 2025, pp. 1–6. DOI: 10.1109/ICCST63435.2025.11295070.

[46] Moteiv Corporation. *Tmote Sky: Ultra Low Power IEEE 802.15.4 Compliant Wireless Sensor Module*. Datasheet. 2006. URL: https://www.snm.ethz.ch/snmwiki/pub/uploads/Projects/tmote_sky_datasheet.pdf.

[47] Renzo E. Navas, Laurent Toutain, and Georgios Z. Papadopoulos. "Moving Target Defense Techniques for the IoT". In: *Intelligent Security Management and Control in the IoT*. 2022, pp. 267–292. URL: https://doi.org/10.1002/9781394156030.ch11.

[48] Renzo E. Navas et al. "IANVS: A Moving Target Defense Framework for a Resilient Internet of Things". In: *IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2020. DOI: 10.1109/ISCC50000.2020.9219728.

[49] Renzo E. Navas et al. "MTD, Where Art Thou? A Systematic Review of Moving Target Defense Techniques for IoT". In: *IEEE Internet of Things Journal* 8.10 (2020), pp. 7818–7832. DOI: 10.1109/JIOT.2020.3040358.

[50] Renzo E. Navas et al. "Physical resilience to insider attacks in IoT networks: Independent cryptographically secure sequences for DSSS anti-jamming". In: *Computer Networks* 187 (2021). DOI: 10.1016/j.comnet.2020.107751.

[51] Van-Tien Nguyen, Renzo E. Navas, and Guillaume Doyen. "Lightweight Security for IoT Systems Leveraging Moving Target Defense and Intrusion Detection". In: *IEEE/IFIP Network Operations and Management Symposium (NOMS)*. Seoul, South Korea: IEEE, 2024. DOI: 10.1109/NOMS59830.2024.10575913.

[52] Xin Niu and Jiazhong Lu. "Moving target defense controller of mobile system based on Openflow sensor security scheme". In: *Computer Communications* 161 (2020), pp. 142–149. DOI: 10.1016/j.comcom.2020.05.004.

[53] Francesca Nizzi et al. "IoT Security via Address Shuffling: The Easy Way". In: *IEEE Internet of Things Journal* 6.2 (2019), pp. 3764–3774. DOI: 10.1109/JIOT.2019.2892003.

[54] OpenAI. *ChatGPT*. Large language model, https://chat.openai.com. 2024.

[55] Arnold Osei, Yaser Al Mtawa, and Talal Halabi. "Mitigating Adversarial Reconnaissance in IoT Anomaly Detection Systems: A Moving Target Defense Approach Based on Reinforcement Learning". In: *EAI Endorsed Transactions on Internet of Things* (2024). DOI: 10.4108/eetiot.6574.

[56] Arnold Brendan Osei et al. "Optimized Moving Target Defense Against DDoS Attacks in IoT Networks: When to Adapt?" In: *GLOBECOM 2022 – 2022 IEEE Global Communications Conference*. 2022, pp. 2782–2787. URL: https://doi.org/10.1109/GLOBECOM48099.2022.10001502.

[57] Matthew J. Page et al. "The PRISMA 2020 statement: an updated guideline for reporting systematic reviews". In: *Systematic Reviews* 10.1 (2021), p. 89. DOI: 10.1186/s13643-021-01626-4.

[58] Zubaida Rehman et al. “Proactive Defense Mechanism: Enhancing IoT Security Through Diversity-Based Moving Target Defense and Cyber Deception”. In: *Computers & Security* 139 (2024). DOI: 10.1016/j.cose.2023.103685.

[59] Melissa L. Rethlefsen et al. “PRISMA-S: an extension to the PRISMA statement for reporting literature searches in systematic reviews”. In: *Systematic Reviews* 10.1 (2021), p. 39. DOI: 10.1186/s13643-020-01542-z.

[60] Nico Saputro et al. “A Review of Moving Target Defense Mechanisms for Internet of Things Applications”. In: *Modeling and Design of Secure Internet of Things*. Ed. by Charles A. Kamhoua et al. John Wiley & Sons, 2020, pp. 563–614. DOI: 10.1002/9781119593386.ch24.

[61] Sang Seo and Dohoon Kim. “IoDM: A Study on a IoT-Based Organizational Deception Modeling with Adaptive General-Sum Game Competition”. In: *Electronics* 11.10 (2022). URL: https://doi.org/10.3390/electronics11101623.

[62] Dilli Prasad Sharma. “Evaluating Moving Target Defense Methods Using Time to Compromise and Security Risk Metrics in IoT Networks”. In: *Electronics* 14.11 (2025), p. 2205. DOI: 10.3390/electronics14112205.

[63] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252, Internet Engineering Task Force. 2014. DOI: 10.17487/RFC7252.

[64] Swati et al. “Securing IIoT Systems Against DDoS Attacks with Adaptive Moving Target Defense Strategies”. In: *Scientific Reports* 15 (2025). DOI: 10.1038/s41598-025-93138-7.

[65] Texas Instruments. *CC2420: 2.4 GHz IEEE 802.15.4 / ZigBee-Ready RF Transceiver*. Datasheet (SWRS041). 2007. URL: https://www.ti.com/lit/ds/symlink/cc2420.pdf.

[66] Matheus Torquato et al. “Exploratory Performance Evaluation of VM Migration as MQTT Moving Target Defense”. In: *2025 21st International Conference on Network and Service Management (CNSM)*. 2025, pp. 1–7. DOI: 10.23919/CNSM67658.2025.11297545.

[67] Andrei Vlădescu and Ion Bica. “Securing IoT Networks through Moving Target Defence”. In: *2025 25th International Conference on Control Systems and Computer Science (CSCS)*. 2025, pp. 552–557. URL: https://doi.org/10.1109/CSCS66924.2025.00087.

[68] Bingdong Wang et al. “Small-Signal-Stability-Guaranteed Moving Target Defense Against Load Redistribution Attack on IoT-Based Smart Grid”. In: *IEEE Internet of Things Journal* 12.11 (2025), pp. 17413–17429. DOI: 10.1109/JIOT.2025.3539653.

[69] Shengling Wang et al. “Moving Target Defense for Internet of Things Based on the Zero-Determinant Theory”. In: *IEEE Internet of Things Journal* 7.1 (2020), pp. 661–672. DOI: 10.1109/JIOT.2019.2943151.

[70] Zesheng Xi et al. “A Dynamic SDN-Based Address Hopping Model for IoT Anonymization”. In: *Computer Modeling in Engineering & Sciences* (2025). DOI: 10.32604/cmes.2025.066822.

[71] Jiajun Yan et al. “Securing Forwarding Layers from Eavesdropping Attacks Using Proactive Approaches”. In: *CMC-Computers Materials & Continua* 79.1 (2024), pp. 563–580. DOI: 10.32604/cmc.2024.048922.

[72] Kimberly Zeitz et al. “Designing a Micro-Moving Target IPv6 Defense for the Internet of Things”. In: *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*. Pittsburgh, Pennsylvania, USA: ACM, 2017. DOI: 10.1145/3054977.3054997.

[73] Jianjun Zheng and Akbar Siami Namin. “A Survey on the Moving Target Defense Strategies: An Architectural Perspective”. In: *Journal of Computer Science and Technology* 34.1 (2019), pp. 207–233. DOI: 10.1007/s11390-019-1906-z.

[74] Tianshuai Zheng et al. “SARM: A network State-Aware Adaptive Routing Mutation method for power IoT”. In: *Computer Networks* 255 (2024). DOI: 10.1016/j.comnet.2024.110889.

[75] Yuyang Zhou et al. “Toward Proactive and Efficient DDoS Mitigation in IIoT Systems: A Moving Target Defense Approach”. In: *IEEE Transactions on Industrial Informatics* 18.4 (2022), pp. 2734–2744. DOI: 10.1109/TII.2021.3090719.

List of Figures

1 IoT device classes and their communication path through the edge gateway. . 10
2 Comparative analysis: Static systems vs. Moving Target Defense. 12
3 Systematic Literature Review Flowchart 17
4 MTD orchestrator system architecture. 33
5 Cooja simulation topology (schematic, not to scale) 34
6 MTD Orchestrator rule-based decision and execution flow. 36
7 IPv6 address shuffling sequence: from attack detection to route re-establishment. 39
8 Threat-model network topology 42
9 Cooja Network panel showing the simulated topology 49
10 Cooja Log Listener output 63
11 Reactive cycle counts per scenario, grouped by selected branch. 70
12 Two universal attacker-side metrics. 72
13 Flood attacker delivery rate, MTD vs no MTD. 77
14 Per-state energy breakdown on a logarithmic y axis ($\log_{10}$ scale) 79
15 Scan attacker-rate sweep: branch composition vs injection rate (single 30-min run per rate.) 83

Listings

1 Sensor address registry 50
2 Proactive IPv6 shuffle callback 51
3 Per-interval timer jitter 52
4 Per-sensor unicast of the CMD_PORT_HOP command 52
5 Reactive trigger with per-type firing policy and cooldown gate 53
6 Threat-type selection in the reactive handler 54
7 Per-sensor silence watchdog 56
8 Pre-shuffle silence drain 56
9 Sliding-window anomaly check 57
10 Per-second rate limiter 58
11 Binary CoAP GET builder 60
12 Forged RPL DIO Control Message Object 61
13 Core of the per-line log accumulator. 67