Securing Internet of Things (IoT) Data Storage

Savannah Malo

University of New Hampshire, Durham

Follow this and additional works at: https://scholars.unh.edu/honors

Part of the Data Storage Systems Commons, and the Other Computer Engineering Commons

Recommended Citation

Malo, Savannah, "Securing Internet of Things (IoT) Data Storage" (2024). Honors Theses and Capstones. 829.
https://scholars.unh.edu/honors/829

This Senior Honors Thesis is brought to you for free and open access by the Student Scholarship at University of New Hampshire Scholars' Repository. It has been accepted for inclusion in Honors Theses and Capstones by an authorized administrator of University of New Hampshire Scholars' Repository. For more information, please contact Scholarly.Communication@unh.edu.
Securing Internet of Things (IoT) Data Storage

By: Savannah Malo

Advisor: Jason Reeves

Department of Computer Science
University of New Hampshire
# Table of Contents

Abstract .................................................................................................................................................. 2

Introduction ........................................................................................................................................... 3

Background Information ......................................................................................................................... 5
  IoT Device Attacks ................................................................................................................................. 5
  Nature of IoT Devices ............................................................................................................................ 6
  IoT Device Security ............................................................................................................................... 8

Replication System .................................................................................................................................. 10

Testing and Results ................................................................................................................................. 12

Conclusion ............................................................................................................................................... 14

Acknowledgements ................................................................................................................................. 15

References ............................................................................................................................................... 16

Appendices .............................................................................................................................................. 18
  Appendix 1: Python Data Collection Program ..................................................................................... 18
  Appendix 2: Python Encryption Functions ........................................................................................... 19
Abstract

Internet of Things (IoT) devices are commonly known to be susceptible to security attacks, which can lead to the leakage, theft, or erasure of data. Despite similar attack methods used on conventional technologies, IoT devices differ in how they consist of a small amount of hardware, limited networking capability, and utilize NoSQL databases. IoT solutions prefer NoSQL databases since they are compatible for larger datasets, unstructured and time-series data. However, these implementations are less likely to employ critical security features, like authentication, authorization, and encryption. The purpose of this project is to understand why those security measures are not strictly enforced and propose solutions for the IoT industry. To apply this, a motion sensor IoT device with a NoSQL database backend is simulated to test the performance of various security configurations. Enabling a full security database increases read query performance by 202% and a 102.5% increase for write queries. Enabling other security configurations consistently impact performance as well, requiring a balanced alternative. For this configuration, partial security at minimum is recommended for risk mitigation.
Introduction

Internet of Things (IoT) devices are rising in popularity to connect the physical world with the Internet. IoT is a new technology venture since it provides accessibility, affordability, and convenience to a variety of industries. The global prevalence of IoT is already strong, as there are an estimated 17.08 billion devices already in use, a number that is expected to almost double by 2030. Additionally, IoT devices currently stand at 66% of total internet-connected devices and are projected to be at 75% by 2025.

IoT devices collect mass amounts of data, often requiring larger data storage techniques like data lakes. IoT is applied to many different industries, like critical infrastructure, healthcare, defense, and retail and may collect sensitive data like personal information, medical records, business activity, or asset configuration that need to be kept confidential. While there may be regulations in place to enforce protection of sensitive data, IoT devices are still known to be vulnerable to security attacks comparable to non-IoT devices. For example, malware can infect devices and wipe their configurations, rendering them useless from their initially designed intention. Ransomware can lock data files, rendering them inaccessible without a decryption key. Therefore, IoT data is important to protect as it can still be erased, stolen, or leaked.

Current database implementations for IoT devices allow for security measures that follow the recommended practices. This includes encryption, access control, logging, and secure logins. These measures are intended to protect the access of data, its integrity and confidentiality. Regardless of how important these practices are, security on IoT data is not always enforced. This can be for a few reasons, one of them being the potential for drastic overhead on system performance.

In this thesis, we test built-in database security tools to demonstrate the performance impact that can be infeasible to apply in time-sensitive applications. This implementation replicates a smart home device to collect data through a software program that delivers to the database. Four different test cases are used to represent different database configurations. This includes a raw database without security measures as the baseline, a full security database, a database with only encryption mechanisms, and lastly a raw database storing data that is encrypted in software program beforehand.

Then, the performance of read and write queries are monitored on these databases. Results indicate that a full security database doubles the read latency. Isolating encryption measures does not indicate that it is the sole contributor to performance overhead, despite its results still being considerable. Moving encryption to the front end of the system does not shift the latency of performance. These test cases revealed other additional findings that were not initially considered. Read queries caused a much higher latency than write queries, despite conventional consensus being the opposite. Authentication and authorization were not examined; however, the results suggest that they may have considerable impact on performance. Limitations of interpreting the statistical results indicate the need for a larger database implementation.
A new solution is required for the performance efficiency of IoT data storage. Ideally this would be a different form of encryption, such as a more performant algorithm or engine. At a minimum, the IoT industry should use a partial implementation of existing security measures to protect the data at rest. For encryption performance, this could be the use of hashes, or only encrypting particularly sensitive fields of data. Relying solely on a raw database to cope with performance ultimately is weak for security.
Background Information

IoT Device Attacks

IoT devices are commonly compromised by security attacks. Recent trends in attack methods target communications and networking to establish a presence in the system. Once that presence is established, then the attackers infiltrate the inner components of devices, websites, or the backend. The following examples of attacks involve IoT data storage ultimately being compromised.

The Mirai Botnet in 2016 is one of the most infamous IoT attacks, which executed a Distributed Denial of Service (DDoS) on devices across the world. The malware exploits default username and passwords on a commonly used Linux processor, allowing the malware access to alter the configurations on the device. This demonstrates the importance of changing default logins as an access control security measure. Botnets continue to evolve and become more sophisticated, which makes it difficult to stop once they percolate. The Mirai Botnet was also replicated to perform a DDoS attack on Dyn, a domain name service provider, which then had rippling impacts on several websites. Botnets of many varieties are the most common form of attack against IoT devices. With any botnet, compromised devices that store sensitive data may also be leaked. Other recent examples include: FritzFrog P2P Botnet, Echobot, Satori Botnet, and IoTroop Botnet.

Ransomware also occurs. LockerGoga targets IoT devices in industrial and manufacturing companies. The attackers penetrated the internal network, encrypted files on the server, and held them at ransom for the decryption keys. The significance of this is if the decryption key cannot be recovered, the data is lost forever (unless the company has uncompromised backups). This highlights a vulnerability within the accessibility of data storage. A similar example is the Ryuk ransomware.

Silex malware infects IoT devices, corrupting them entirely from any use. Network configurations, firewall rules, and data storage are all erased from the device, rendering it useless after the device restarts. This attack is also conducted on devices using default or easily guessable logins. While data stored in a database server might be safe from this attack, any data stored directly on the device is compromised.

BleedingTooth attacks target vulnerabilities in Bluetooth connections to send a device malware or execute code on the device. Therefore, any data stored on the device could be compromised. Additionally, since Bluetooth communications are entirely unencrypted, that traffic can be sniffed to reveal any sensitive data.
Nature of IoT Devices

IoT devices are generally small, have minimal computing, storage, and networking. Data is sent over the network to be further processed by a front-end application or directly to the backend for storage. Ultimately, the data storage mechanism for IoT devices largely depends on the type of device, how data is retrieved, company preferences, and resources available.

A device that does not require a large transmission of data will use flat files (ex: JSON, CSV) stored directly on the device. This is typically the data collected from the external source, like sensors, and typically transmit one type of data (motion, temperature, etc.). Since these devices do not have a wireless connection, they are unlikely to encounter security issues unless the device is physically compromised, or the user uploads the collected data elsewhere. However, a device that uses a combination of device and external storage can be vulnerable to an attack.

Devices with a large-scale data collection component will use one of two options: direct storage or analyzed storage. Essentially, the data can be stored directly in a database, or sent to an external application, which analyzes or processes the data before storage or during data retrieval. Nonetheless, the underlying databases can be of multiple options: lightweight, NoSQL, time-series and/or cloud storage databases. In this case, data sent from the device is conducted through various communication protocols, for example, HTTP.

Some devices are dependent on timestamps of data collected, meaning data transmission in real time continuous streaming, for example, vehicles and medical technology, while others can be sent in batches. MongoDB is commonly used in the IoT industry for real-time data analytics. MongoDB uses JSON file objects for storage and is best for storing device state. A column-based NoSQL database would be best for grouping retrievals.

Relational (SQL) databases are an uncommon option for IoT since they are not efficient with time-series data. Additionally, as the scale of data increases, so does the lookup time in a SQL database. SQLite functions without a separate server and writes the entire database to disk files. Common IoT applications are simpler, for example, Raspberry Pi devices. SQLite is free and is great for storing key-value pair data. Alternatively, NoSQL databases require less development time, and less data retrieval time. Drawbacks of NoSQL solutions include data retrieval difficulties, data loss, and inefficiencies. Despite these disadvantages, NoSQL databases are the most frequently used for storing IoT data.
In general, NoSQL databases are more difficult to enforce security measures on since there is no enforced structure to the data. This is intended for purposes of scalability, performance, and flexibility, with the drawback of lacking security control. NoSQL databases have various query languages, so security measures would have to be implemented differently for each one. However, the most popular offerings are document-based. One document contains all the information for an object. Groups of documents can then be stored in a collection. This data is unlikely to be encrypted in storage. Additionally, NoSQL databases commonly disable authentication and authorization measures by default. Instead, they must be intentionally turned on and configured.

There are a variety of possible courses of action to address the weaknesses of NoSQL storage. First, hybrid databases are possible. These will allow you to use either relational or NoSQL features as needed for the data. For example, the relational aspects can be used for predictable, smaller amounts of data that require more security. The NoSQL portions can be used for the larger amounts of variable data. Secondly, schema validation ensures that only proper data types are being stored in the database. This is helpful to combat NoSQL injections and ensure data consistency. Finally, NoSQL databases do not generally encrypt the data at rest by default. While encryption features may exist, IoT devices are likely to send data to other devices and/or applications over the network. It makes more sense to have encryption done at the application layer before it gets sent to the database. This can be done with hashes, checksums, or digital signatures.
IoT Device Security

There are several security measures that IoT companies should implement for best practices:

- Data encrypted at rest.
- Data encrypted in transmission (with TLS over SSL preferred).
- The network exposure of the database should be limited to only where needed.
- Role-based access control.
- Logging/auditing of system activity.
- Removal of default logins and replaced with unique and secure ones that may require multi-factor authentication.

Why might IoT companies choose not to enforce security on their data storage? There may be operational as well as technical reasons for this.

**Lack of security awareness** – Companies functioning without a security team are less likely to have response plans and protective measures in place. Hiring a security team, training employees, and awareness programs can improve this situation.

**Low risk data** – No constraints of regulatory compliance enforcing security. Companies may think that since their data has no personally identifiable information (PII), that the data is reasonably safe. However, this does not prevent the devices from necessarily being a target of attack.

**Decreased performance** – Encryption at rest requires more processing power to retrieve the data. Some mechanisms exist to encrypt only sensitive data, while others will encrypt the entire database. Authentication measures add more steps to the data storage and retrieval process. Having a slower performance can impact the user experience, which devalues the perception of the product.

**Increased cost** – Most NoSQL databases used in production are cloud offerings. Because of this, they offer security solutions as a service. This is charged based on number of queries, incoming connections, or the like. Therefore, the larger the data store, the more cost incurred.

For companies using an on-premises database solution, security must be developed themselves, or outsourced to a third-party company, which takes more time and resources to implement. In a competitive business environment, reducing the time to market (TTM) is critical.
Implementing basic database security is not entirely difficult, even for those who are not security trained. Many database solutions use a YAML configuration file for their software. Simply editing this file can enable security mechanisms that are built-in functionalities of the database. For example, in CassandraDB editing the cassandra.yml file from:

\[
\text{client\_encryption\_options: enabled: false}
\]

to

\[
\text{client\_encryption\_options: enabled: true}
\]

allows for client encryption mechanisms to be configured. More advanced databases offerings also include a Graphical User Interface (GUI), where security features can be toggled without the need for handling configuration files.

Since it is relatively easy to implement security features, it appears to be a deliberate choice for IoT companies to disregard them during development. The purpose of further research on this project is to measure the performance hit of enabling security using a replicated system. A 5% threshold is used to measure the acceptable range of performance overhead. This number is a result of a security study conducted within an industrial control setting. While this number is more tightly secure than an IoT context may require, it is a reassurance that this threshold will meet or exceed those demands. Analyzing those results provides a path for a proposed solution that can balance the requirements of security and a well performing product.
Replication System

To simulate an IoT data collection, the Samsung SmartThings home motion sensor device is replicated using an Arduino sensor kit (Figure 1). This can be related to other smart home devices as well, which may use different specifications. For the purposes of this implementation, data collection and storage are the focus. However, these devices typically operate in response to the data collected. For example, the data collected is motion detection and the response would be an alert notification received through a software application.

![Samsung SmartThings vs Replica Sensor](samsung.com) ![CircuitBasics](circuitbasics.com)

(Figure 1: comparison of industry technology and replication device.)

The specifications of the devices are quite similar. The Arduino sensor is slightly smaller in range and dimensions. Both devices extend to the same angle and operate with the same voltage for power (Figure 2). Unfortunately, Samsung has limited documentation provided for the SmartThings devices. Additional specifications like memory, clock speed, networking protocols, and additional hardware are currently unable to be provided. However, Arduino kits are commonly used for mimicking IoT projects and are justifiably fit within the small device with minimal hardware and networking capabilities.

**Sensor Specifications Comparison**

<table>
<thead>
<tr>
<th></th>
<th>Samsung SmartThings</th>
<th>Arduino Sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Range</strong></td>
<td>50 feet</td>
<td>20 feet</td>
</tr>
<tr>
<td><strong>Angle</strong></td>
<td>120 degrees</td>
<td>120 degrees</td>
</tr>
<tr>
<td><strong>Dimensions (LxWxH)</strong></td>
<td>56.6 x 50.2 x 55.7 mm</td>
<td>24.03 x 32.34 x 24.66 mm</td>
</tr>
<tr>
<td><strong>Voltage</strong></td>
<td>3.0-5.0 volts</td>
<td>3.0-5.0 volts</td>
</tr>
</tbody>
</table>

(Figure 2: technical specification used to compare devices used in this project.)
Samsung SmartThings employs Apache CassandraDB for backend data storage. Cassandra is a common choice for an IoT solution as many large companies publicly use it, among those being Blackberry and Bigmate for IoT. Cassandra is open source, NoSQL, and supports horizontal scaling, zero downtime, high performance, and replication. For the purposes of this research system design, CassandraDB is run on a Docker container. The Arduino motion sensor collects data, sends it to a Python program, which then writes queries to the Cassandra database using an established connection (Figure 3).

Two database implementations are used for testing purposes, one with security measures configured and one without. The database without security simply requires a successful connection. The security database ensures authentication and authorization with a user login and permissions granted. Additionally, the data is encrypted, using Cassandra’s built-in DataStax (DSE) engine and the AES 128 ECB algorithm. The latency of queries on both databases is then measured with Cassandra’s built-in NodeTool program.

![Arduino Sensor Data Flow Diagram](image)

(Figure 3: data flow diagram of the replicated system)

The Python code is relatively simple and easy to replicate for other types of Arduino sensor collection (Appendix 1).

Another test case of the project involves encrypting data within the Python program rather than through the DSE engine. Before all variables are passed into the database execution statement, they are processed through the encrypt function (Appendix 2).
Testing and Results

Four test cases are used to measure read and write query latency on the replication system:

1. No security enabled.
2. Authentication, authorization, encryption.
3. Encryption only.
4. Encryption only, conducted in Python before database write.

The NodeTool report on performance is shown in the table below (Figure 4).

<table>
<thead>
<tr>
<th></th>
<th>No Security (baseline)</th>
<th>Security Database</th>
<th>Encryption Only</th>
<th>Python Encryption</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read performance</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(ms/read)</td>
<td>0.319</td>
<td>0.645 (202% higher)</td>
<td>0.2847 (10.8% lower)</td>
<td>0.3493 (109.5% higher)</td>
</tr>
<tr>
<td><strong>Write performance</strong></td>
<td>0.00118</td>
<td>0.00121 (102.5% higher)</td>
<td>0.002 (169.5% higher)</td>
<td>0.000185 (15.7% lower)</td>
</tr>
</tbody>
</table>

(Figure 4: read and write query metrics for the multiple database tables.)

Using the “No Security” database as the baseline for comparison, the results of the test cases are as follows (Figure 5). The previously mentioned 5% threshold is used as the maximum bound for comparing performance.

- Enabling authentication, authorization, and encryption mechanisms have a drastic impact on producing overhead.
- The encryption implementation reports a considerable latency but is slightly lower than the 5% threshold. This inconsistency may be a result of the limitations discussed later (page 13).
- Externalizing the encryption within the Python program does not improve the latency. Again, this is a lower bound result as it does not include the performance transfer to the front end.
It is important to note a few limitations within the results. First, the resulting latencies are only a representation of the database performance. Any additional overhead that may be accounted for by running the Python program is not included. Therefore, the Python Encryption implementation only serves as a lower bound for those results and does not necessarily indicate that shifting encryption to the front end will resolve performance issues.

The encryption only results are reported at a faster latency than the no security database. This result appears counterintuitive to the initial hypothesis, and we believe this to be caused by the sample size of query requests, which requires improvement. More requests handled improves the accuracy of the latency to a refined number. Running additional trials and averaging those results will also improve accuracy. Furthermore, the initial hypothesis is that encryption mechanisms are the sole cause of the high performance overhead found in the security implementation. However, since the encryption only results did not significantly increase the overhead, findings suggest that authentication and authorization may be the cause instead.

Traditionally, it is projected that database writes are more expensive for latency compared to database reads. The opposite is represented in the NodeTool metrics. This is likely due to the varying sample sizes of the read and write queries. Hundreds of write queries were sent to the database in a time-series manner. Meanwhile, only a few dozen read queries were sent to the database in a limited time frame.

Next, this implementation is restricted to one local user experience. The implications of having thousands of users connected will further complicate the database and have a larger overhead on the performance. Additionally, the potential impacts of network latency are not measured. This replica database is hosted in a local container, which does not implement the networking component that IoT companies have.
Conclusion

Results from this research conclude that security configurations consistently impact performance. It appears existing security mechanisms for database solutions result in slower performance that IoT companies do not measure as necessary. While there is no current solution, future work requires a special, non-traditional approach that enables security at a minimum. This could be using a new form of encryption, such as a more performant algorithm or a new encryption engine. Additionally, storing hashes can be done as a minimal measure to secure data integrity. Nonetheless, implementing IoT data security is important, even if there are no regulations that may require it.

Within this specific experiment, there is additional work required to make the results more broadly applicable to the IoT industry.

- Standardizing the sample size of read and write queries so that they are of a comparable number. Additionally, repeated trials are needed for statistical refinement.
  - This, and perhaps other measures can help detect the reason for read query performance being unexpectedly high compared to write query performance.
- A larger, more complex database, hosted in the cloud, with multiple devices connected will better simulate an IoT industrial setting.
  - Within this, include stress and load testing to understand the impacts of performance when there are high rates of database access. This can help in mitigation plans for DoS attacks.
- Trials should be added to analyze the impact that authentication and authorization mechanisms alone have on performance.
  - If those results are minimal, there must be another reason to explain why the encryption only results did not produce as high of an overhead as expected. Within Cassandra specifically, there could be other mechanisms enabled within the encryption engine when security configurations are changed.
- Different IoT devices should be used to experiment with differing data types.
  - More complex data fields are likely to consume more performance to enforce security on. Therefore, understanding how to make that process more efficient will also yield efficiency for simpler implementations.
Acknowledgements

Thank you to Jason Reeves for advising on this project. Your guidance and constructive feedback have been integral to the success of my research.
References


Appendices

Appendix 1: Python Data Collection Program

```python
import datetime
import time
from pyfirmata import util, Arduino, INPUT
from cassandra.cluster import Cluster
from cassandra.auth import PlainTextAuthProvider
import uuid

# Database connection
# Two different containers, one for security (sec), one without (nosec)
# Both have database (keyspace) named 'motion'
# Container is run locally, local IP address used

# authentication mechanism, remove auth provider for no security database
# asterisks are used in lieu of real credentials
auth_provider = PlainTextAuthProvider(username='****', password='****')
cluster = Cluster(['127.0.0.1'], port=9042, auth_provider=auth_provider)
session = cluster.connect()

# Arduino board
board = Arduino('COM3')

# Main loop that will run forever:
loop = util.Iterator(board)
loop.start()

pir = board.get_pin('d:2:i')
pir.mode = INPUT

time.sleep(1)
old_value = pir.read()
device_id = uuid.uuid4()

while True:
    pir_value = pir.read()
    if pir_value:
        # Motion detected, prints on first instance of motion
        if not old_value:
            time1 = datetime.datetime.now()
            statement = 'Motion detected!
            # change database name sec/nosec
            preparedStatement = session.prepare(
                "INSERT INTO motion_sensor_sec (device_id, timestamp, state, value) VALUES (?, ?, ?, ?)"")
            session.execute(preparedStatement, [device_id, time1, pir_value, statement])
            time.sleep(0.5)
    else:
```
# Motion stopped, prints on first instance of motion ending
if old_value:
    timel = datetime.datetime.now()
    statement = 'Motion stopped!'
    # change database name sec/nosec
    preparedStatement = session.prepare(
        '""INSERT INTO motion_sensor_sec (device_id, timestamp, state, value) VALUES (?, ?, ?, ?);""
    
    session.execute(preparedStatement, [device_id, timel, pir_value, statement])
    time.sleep(0.5)
    old_value = pir_value

Appendix 2: Python Encryption Functions

```python
import base64
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

# A's used in lieu of a more complex key
key = 'AAAAAAAAAAAAAAAAAA'

# write to database
def encrypt(raw):
    raw = pad(raw.encode('utf-8'), 16)
    cipher = AES.new(key.encode('utf-8'), AES.MODE_ECB)
    return base64.b64encode(cipher.encrypt(raw))

# read from database
def decrypt(enc):
    enc = base64.b64decode(enc)
    cipher = AES.new(key.encode('utf-8'), AES.MODE_ECB)
    return unpad(cipher.decrypt(enc), 16)
```

# example execution within while loop
# statement = encrypt(statement)