Proceedings

4th International Workshop on Software Engineering Research and Practice for the IoT

SERP4IoT 2022

19 May 2022

Proceedings

4th International Workshop on Software Engineering Research and Practice for the IoT

SERP4IoT 2022

The Association for Computing Machinery
2 Penn Plaza, Suite 701
New York New York 10121-0701

ACM ISBN: 978-1-4503-9332-4

Editorial production by Lisa O'Conner
Cover art production by Annie Jiu

CPS
Conference Publishing Services

IEEE Computer Society
***Conference Publishing Services* (CPS)**
http://www.computer.org/cps

4th International Workshop on Software Engineering Research and Practice for the IoT

SERP4IoT 2022

Table of Contents

Foreword
SERP4IoT 2022

SERP4IoT 2022, the fourth edition of the International Workshop on Software Engineering Research & Practices for the Internet of Things, took place virtually on May 19th, 2022. It was held in conjunction with the 44th ACM/IEEE International Conference on Software Engineering (ICSE 2022).

SERP4IoT begins to be recognised as an annual venue gathering researchers, industrials, and practitioners to share their vision, experience, and opinion on how to address the challenges of, find solutions for, and share experiences with the development, release, and testing of robust software systems for IoT devices.

To date, there is no precise definition of what is software engineering for the IoT, because it encompasses many different aspects of software design, development, evolution, deployment, and operation, with varying and conflicting criteria such as success, longevity, growth, resilience, survival, diversity, sustainability, transparency, privacy, security, etc.

Yet, software engineering is vital for IoT to design systems that are secure, interoperable, modifiable, and scalable. It is crucial to bring good practices for developing projects for IoT, to devise and study the best architectures, to understand and secure communication protocols, and, generally, to overcome the many challenges faced by practitioners and researchers.

The technical program of the workshop includes presentations of 5 papers, contributed by 9 authors from 26 different countries. Each paper was reviewed by at least three Program Committee members. All papers were evaluated according to their relevance to the workshop, soundness, maturity, and quality. They were invited for presentation and inclusion in the workshop proceedings, published by ACM.

Different from a typical conference type workshop, SERP4IoT included several activities to foster participation of the participants in a constructive manner. We had a virtual Beach Ball Buzz game, built affinity maps, and performed card sorting for discussing existing issues and challenges on software engineering research and practices for the IoT. We ended the workshop with an open virtual Fishbowl to allow each attendant to give their opinion in an interactive way.

We would like to thank everyone who made the workshop come to life, in particular the authors, the Program Committee members, and the participants. We were glad to have your virtual presence in the workshop and hope to meet you again at a future SERP4IoT edition!

Yann-Gaël Guéhéneuc
Concordia University, Canada

Shah Rukh Humayoun
San Francisco State University, USA

Rodrigo Morales
Concordia University, Canada

Rubén Saborido
University of Málaga, Spain

Program Committee
SERP4IoT 2022

Darine Ameyed, *ÉTS*
Lotfi Ben Othmane, *Iowa State University*
Francisco Chicano, *University of Malaga*
Ghizlane El Boussaidi, *École de Technologie Supérieure*
Michael Felderer, *University of Innsbruck*
Hugo Sereno Ferreira, *FEUP, University of Porto*
Fehmi Jaafar, *Université du Québec à Chicoutimi*
Foutse Khomh, *Polytechnique Montréal*
Hamid Mcheick, *Université du Québec à Chicoutimi*
Hausi Müller, *University of Victoria*
João Pedro Dias, *FEUP, Universidade do Porto*
Fabio Petrillo, *Université du Québec à Chicoutimi, Canada*
Mónica Pinto, *Universidad de Málaga*
Hironori Washizaki, *Waseda University*

Author Index

"If security is required": Engineering and Security Practices for Machine Learning-based IoT Devices

Nikhil Krishna Gopalakrishna
Purdue University
gopalakn@purdue.edu

Dharun Anandayuvaraj
Purdue University
dananday@purdue.edu

Annan Detti
Purdue University
adetti@purdue.edu

Forrest Lee Bland
Purdue University
fbland@purdue.edu

Sazzadur Rahaman
University of Arizona
sazz@cs.arizona.edu

James C. Davis
Purdue University
davisjam@purdue.edu

ABSTRACT

The latest generation of IoT systems incorporate machine learning (ML) technologies on edge devices. This introduces new engineering challenges to bring ML onto resource-constrained hardware, and complications for ensuring system security and privacy. Existing research prescribes iterative processes for machine learning enabled IoT products to ease development and increase product success. However, these processes mostly focus on existing practices used in other generic software development areas and are not specialized for the purpose of machine learning or IoT devices.

This research seeks to characterize engineering processes and security practices for ML-enabled IoT systems through the lens of the engineering lifecycle. We collected data from practitioners through a survey (N=25) and interviews (N=4). We found that security processes and engineering methods vary by company. Respondents emphasized the engineering cost of security analysis and threat modeling, and trade-offs with business needs. Engineers reduce their security investment if it is not an explicit requirement. The threats of IP theft and reverse engineering were a consistent concern among practitioners when deploying ML for IoT devices. Based on our findings, we recommend further research into understanding engineering cost, compliance, and security trade-offs.

CCS CONCEPTS

• **Computer systems organization** → **Embedded and cyber-physical systems**; • **General and reference** → *Empirical studies*; • **Computing methodologies** → *Machine learning*; • **Security and privacy**;

KEYWORDS

Internet of Things, Machine Learning, Security and Privacy, Cyber-Physical Systems, Embedded Systems, Software Engineering

ACM Reference Format:
Nikhil Krishna Gopalakrishna, Dharun Anandayuvaraj, Annan Detti, Forrest Lee Bland, Sazzadur Rahaman, and James C. Davis. 2022. "If security is required": Engineering and Security Practices for Machine Learning-based IoT Devices. In *4th International Workshop on Software Engineering Research and Practice for the IoT (SERP4IoT'22), May 19, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3528227.3528565

SERP4IoT'22, May 19, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9332-4/22/05.
https://doi.org/10.1145/3528227.3528565

1 INTRODUCTION

The Internet of Things (IoT) paradigm integrates cyber and physical components, connecting devices at the network edge ("Things") to one another and to more powerful resources over the network ("Internet") [15]. There are ~35 billion IoT devices worldwide, projected to double by 2025 [30, 57, 58]. IoT systems can leverage machine learning (ML) [38, 39] to make low-latency intelligent decisions [8, 67]. The resulting intelligent IoT systems could transform many sectors of the economy [42], however, the associated risks are also substantial. To minimize the risks, engineers should adopt ML methods on resource-constrained IoT devices in a secure, privacy-preserving way [16].

Despite the increasing importance of intelligent IoT systems to consumers, industry, and governments, we know relatively little about manufacturers' engineering practices [28, 46, 53]. Concerns about engineering practices are raised by high profile failures, including cyberattacks on waterworks systems leading to poisoned water supply [55], aggressive data collection practices [4, 48] and exploits leading to IoT botnets [1]. Researchers have investigated IoT software defects [46] and security flaws [12, 18, 20–23, 25, 34, 35, 47, 61] from the software perspective using program analysis and failure analysis. Also, researchers have proposed generic models of the secure software development life cycle (SDLC) for the development of ML models and the development of ML-enabled edge devices [28, 53]. However, the challenges of real-world adoption and current industry practices are largely unexplored.

Our goal is therefore to investigate the process of engineering ML-enabled IoT devices in industry. Our general research questions are: *What practices does the industry follow to develop and manage ML-based IoT devices? How is security treated in industry development life cycles?* We investigate these questions in a survey (N=25) and interviews (N=4) with industry practitioners.

Among other findings, our survey respondents and interview subjects emphasized tradeoffs between engineering cost and quality. Market forces reduce the quality and security of IoT products. As one interview subject (P2) said, *"it is a question of if it [better security] will be accepted by the market"*. Larger companies benefit from economies of scale, with in-house ML and security specialists to support IoT products. We also learned that businesses may give up some marketable functionality in order to reduce their risk, *e.g.,* not

storing user data on IoT devices. Across several industry sectors, another common worry is the reverse engineering of proprietary ML models.

2 BACKGROUND

This research is motivated by an industry trend towards computing systems with intelligent components at the network edge, and the associated security and privacy implications. Definitions of an "IoT device" vary [56]; we consider devices with sensors and/or actuators, a network connection, and limited resources in memory, power, and computation [33, 66]. Resource-constrained IoT systems combine sensing and communication capabilities with low cost [50, 70].

Engineering process for IoT: Engineering processes for IoT systems are complex because IoT systems are inherently distributed and resource-constrained, and have physical components alongside virtual ones [68]. Figure 1 depicts a generic engineering lifecycle for ML-based IoT systems, which we used to design our study. This lifecycle combines several existing works [2, 28, 53]. In this model, IoT engineering is a five-step iterative process:

Specification: The purpose of the product is defined, perhaps constraining the hardware and software components.

Design: Decisions are made about system architecture, frameworks are selected, and evaluation techniques are chosen.

Development: Design decisions are implemented using development frameworks. The ML model is optimized by tuning hyperparameters, reducing the computational complexity of the model (eg: deep learning-based models), and manipulating network blocks [26, 40]. The implementation targets a hardware profile but not specific devices, to promote portability.

Deployment: The developed solution is deployed to the target hardware. Deployment-time optimizations such as pruning help fit the model into the IoT device constraints [39]. Optimization strategies are standardized, but the parameters vary based on the available resources of the target hardware [53].

Audit: Here the software components have been deployed to the hardware components, and engineers determine whether the system specification is met. Concerns may be raised about performance goals, fault tolerance [31, 59], or security vulnerabilities. Engineers consider traditional threat models as well as those specific to the use of ML. For example, researchers have proposed attacks involving corrupted training data [69] or reverse engineering a model [49].

Security in IoT: Security is a cross-cutting concern for engineered systems [51]. Security is increasingly incorporated throughout the engineering life cycle (Figure 1) [41]. However, IoT developers find security challenging and complicated [46]. Engineering teams feel responsible for security, but often lack a formal security process [9, 45, 63]. Functionality and deadlines are often prioritized over security [14, 24, 43], and adding security to resource-constrained devices penalizes power consumption, latency, and throughput [11, 60].

Although this engineering process model for ML-based IoT development is a promising start, the research community still lacks insight into industry practices. This knowledge gap hinders our understanding of industry-wide problems and challenges towards building and maintaining secure ecosystems. This study is a step towards filling that gap.

3 RESEARCH QUESTIONS

To understand the processes and challenges of engineering secure ML-enabled IoT systems, we posed five research questions across two themes. The first theme explores ML engineering in a resource-constrained context, with implications for IoT system trustworthiness (*e.g.,* affecting security and privacy). The second theme examines cybersecurity practices for these systems.

Theme 1: Applying machine learning on IoT devices

RQ1: What are the common practices for bringing ML to resource-constrained edge devices? (Process model steps 3a-3d)

RQ2: What are the challenges and consequences developers face due to resource limitations in developing ML software for edge devices? (Steps 3a-3d)

Theme 2: Engineering secure IoT systems

RQ3: How do engineers incorporate security into the IoT engineering process? (Steps 1-5)

RQ4: How do engineers reason about trust in ML-based IoT systems? (Step 4)

RQ5: What other factors affect security practices in IoT engineering? (Steps 1-5)

4 METHODOLOGY

Given our research questions, we chose an exploratory methodology [54] — a mixed quantitative and qualitative approach to explore a phenomenon and develop new research questions. We elicited coarse data with a survey, and detailed insights using interviews.

4.1 Survey

Instrument design: We designed a ∼10-minute, 32-question survey instrument aligned with our research questions. We drew on existing literature for seven demographic questions [10, 13], and developed the other questions using best practices in survey design [29]. The initial set of questions were based on our own industry experience working with ML on IoT devices, and then refined through discussion with practitioners. To test validity and length, we administered the survey to two practitioners and further refined it based on their feedback.

Survey distribution: Given the specialized nature of the engineering security practices under consideration, we distributed the survey widely: on the public platforms Reddit, Hacker News, and TowardsAI; through our personal networks via Facebook and LinkedIn; and on our departmental mailing list. We also asked survey respondents to share the link with their colleagues (snowball sampling [36]). The survey was published in the last week of March 2021 and closed after 5 weeks. We incentivized survey participation with a 1-in-50 chance of winning a $50 gift card.

Analysis method: We analyzed the data using reports generated using the Qualtrics platform. We examined the data from each question, aggregated across all participants. In order to have a uniform scale of results, we have represented all the data in the survey in terms of the percentage of total responses in the diagrams for the purpose of visualization.

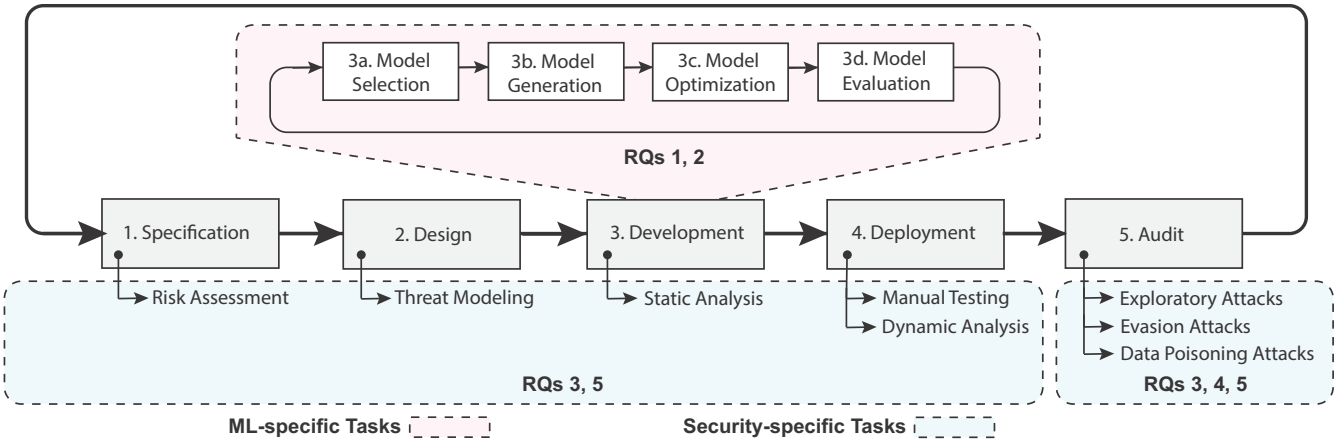

Figure 1: An engineering lifecycle for machine learning-based IoT devices. It combines several models including the SDLC [2, 28, 53].

4.2 Interviews

Protocol design: We designed our interview protocol as an extension of the survey questions. We observed survey responses and developed questions around areas where the survey respondents disagreed or gave unexpected answers. The interview followed a semi-structured interview, with 8 planned questions to permit a 30-40 minute conversation with each subject [27]. To test validity and length, we piloted the interview protocol with one practitioner.

Participant recruitment: We recruited interviewees from the survey respondent pool. Survey respondents had experience in ML and IoT engineering, making them good candidates for a longer interview. Survey respondents could indicate if they were interested in a follow up interview, incentivized with a $25 gift card. We contacted all interested respondents, and interviewed any who replied and completed the interview consent form.

Participant privacy: Audio recordings of interviews were transcribed by a third-party service. We anonymized participant PII (*e.g.*, names of people and companies) before analysis.

4.3 Collected data

Survey: We received a total of **25** survey responses, of which **14** were fully completed. Given the few full responses, we also analyzed the available data from partial responses. The median partial respondent completed 42% of the survey.

Interview: We interviewed 4 experts, with a range of positions and professional experience. The interviews comprised 140 minutes of audio recordings.

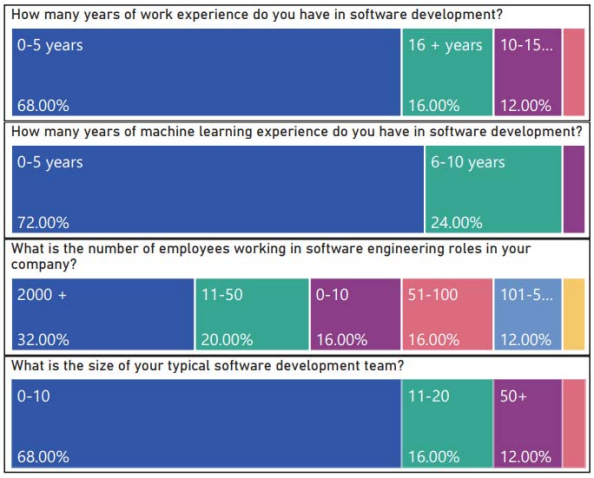

Figure 2: Demographics of survey respondents.

Table 1: Interview Subjects

Identifier	Role (Company type)	Experience
P1	Principal System Architect (HW vendor)	20 years
P2	Senior developer (HW vendor)	20 years
P3	Chief Architect (Start-up)	30 years
P4	ML Engineer (ML services)	3 years

5 RESULTS AND ANALYSIS

We present results corresponding to our RQs. To simplify the presentation, we synthesize survey and interview data for each question.

5.1 Demographics

Survey respondents (Figure 2) hold bachelor's degrees in computer science, software engineering, computer engineering, or electrical engineering; work primarily in the sectors of consumer electronics (27%), IT & telecommunications (22%), automotive (20%), and healthcare & biomedical (15%); and learned about ML techniques from university coursework (41%), self-taught (37%), and from corporate training (20%). They work at a range of company sizes, from under 50 employees (36%) to over 2,000 (32%). They have a range of experience applying ML in software engineering, ~30% more than 5 years and ~70% fewer. At their companies, they reported an almost equal distribution of ML deployment experience: from initial exploration/prototyping stages to "multiple projects" to extensive multi-platform experience (Figure 4).

Interview subjects (Table 1) had a range of job roles, and experience in sectors including manufacturing, consumer electronics, defense, and medical devices.

5.2 Theme 1: Machine Learning for IoT Devices

RQ1: Common ML practices for IoT. *ML modeling:* ML algorithms are one ingredient of next-generation IoT systems. We asked survey respondents and interview subjects where their models come from. Survey respondents rely on academic research, re-using models entirely or tailoring them to their company's needs (Figure 3). Notably, none of the survey respondents indicated that they follow product line development (*i.e.,* , reuse models from one product to the next) for their ML models. P3 characterized the sources used by his start-up:

> *"In the ML world, [if] you don't read a paper every single day, you are in trouble...IEEE papers and...we also look at results that come out of Google, Facebook, Amazon and Microsoft."*

Companies with deeper expertise also develop models internally; P1 said, *"[My company's] research team does a lot of research around machine learning, and we...use the frameworks developed by them."*

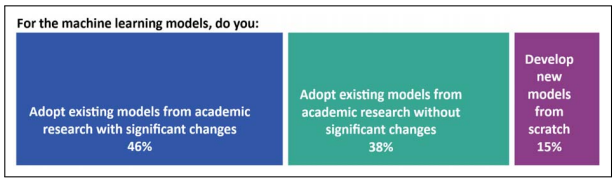

Figure 3: Source of ML models in practice.

ML development: TensorFlow/TF-Lite and PyTorch were the most popular modeling frameworks; Python and C/C++ were the most popular languages. To train and validate models, survey respondents follow standard practices: splitting training and testing data, applying K-fold/cross validation, etc.

Engineering processes: Our data show that the industry movement towards incremental development and agile methodologies [41] includes IoT systems development. Among survey respondents, 48% report using *"Agile"* as their software development process, the most popular response. Our interview subjects concurred. As interviewee P3 said:

> *"We tend to follow the agile flow...2 years ago we [were] mostly waterfall, the old-fashioned way...now...95% of...[our] programs [are agile]."*

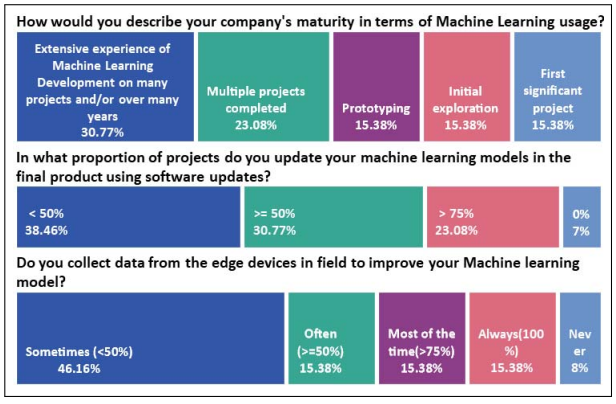

Figure 4: Survey data on ML maturity, software updates, and data collection.

This adoption includes the Continuous Integration/Continuous Deployment (CI/CD) approach. Half of the survey respondents said their teams incorporate ML models into the rest of their IoT systems during CI, 25% said *"Before deployment"*, and only 16% said their integration occurred at software release time. Interviewee P3 said:

> *"At every stage of our Agile flow...[we have a] CI/CD-based validation flow...as part of the weekly sprints trying to meet accuracy, latency and throughput."*

After IoT device deployment, many survey respondents report that they improve the ML models in their products by collecting new data and sending software updates (Figure 4).

RQ2: ML challenges and consequences for IoT. *ML on resource-constrained devices:* IoT engineers work within hardware constraints. Over 90% of survey respondents said they meet constraints by changing the software, not the hardware. To meet their resource constraints, our survey respondents said they use neural network pruning techniques including regularization, second-order methods, and variational dropout. As they do so, survey respondents said they struggle with decreased model performance (38%), memory constraints (23%), and insufficient expertise (23%) (Figure 5). Interviewee P3 went into more detail:

> *"From a technical perspective, one of the biggest problems that we face is the inability of standard tools to be able to squash a model into something that fits with a push of a button."*

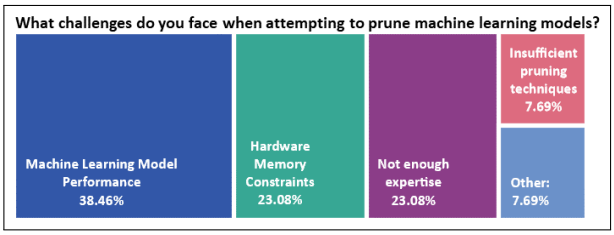

Figure 5: Survey data on ML resource constraints.

Our interview participants went into detail about their strategy for estimating ML model performance: back-of-the-envelope calculations. As P2 said:

> *"I prefer Excel sheet because bringing emulator to a state that you can perform simulation takes time. And also building machine learning algorithms takes time. So it's better [to make a] crude estimate...using Excel sheet...and then simply prepare ML algorithms that simply relies on this crude estimate."*

Working with customers: P4 noted the challenges of ensuring robustness as a customer requirement:

> *"[Clients] give us the validation data set, but not the test data set...Then they used to run the inference at their end on the same device and validate if it works well on the test data set. Even slight changes...distortions...used to give bad accuracy...So if your model should be robust to such kind of things, then you need to have such kind of data in your training data set."*

Edge-Cloud collaboration: Survey respondents described different architectures for data processing. Two-thirds follow a hybrid

strategy, with lighter-weight processing on IoT devices and heavy-weight processing using Fog or Cloud systems. Edge-only processing was the second most common, and Cloud-only processing was rare. When placing computation on Edge devices, engineers reported working around the resource limitations of IoT devices.

5.3 Theme 2: Secure IoT engineering

More than half of respondents have experienced a security vulnerability in their current product. One-third have dealt with 1–3 CVEs (Common Vulnerabilities and Exposures), and one-third with 4 or more. Understanding how they incorporate security and reason about trust in their engineering processes may help reduce CVEs.

RQ3: Incorporating security into IoT engineering. *Security Analysis:* We asked survey respondents to describe the processes their teams follow for security analysis. Code review (42%) and white-box analyses (21%) were the primary ways in which security checks are realized (Figure 6). Survey respondents and interview participants also described conducting security reviews and creating mitigation plans. Interviewee P3 discussed integrating security into the ML development process:

> *"It's not as if every member in the team is...[a security expert, but] they are [generally] aware of the pitfalls and needs. But...[we ensure that] a few experts are always there in the reviews."*

For interviewees working in smaller companies, security analysis was part of every developer's job. In larger organizations, interviewees said that security analysis was done by dedicated security teams. However, developers are still involved and have some familiarity with security analysis methods.

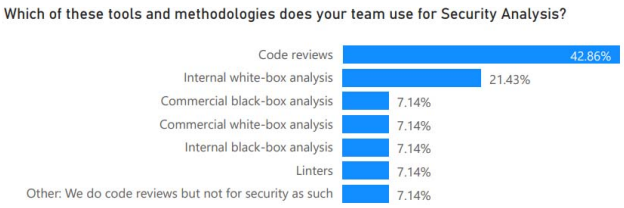

Figure 6: Methods for security analysis. *This question was accidentally single-response, so we suppose the respondents interpreted this as* primary *method.*

Threats and threat models: Our subjects said that security threat analysis was a common part of the development process, but with varying priority depending on the company size and available resources. Our interviewees indicated that the major threat they considered was the loss of intellectual property — reverse engineering of their ML models. Interviewee P1 said the biggest security challenge they face is in-memory re-engineering: *"We try to mimic scenarios that can breach security...We are careful about snoop-out transactions."* At P1's organization the same threat is considered:

> *"One common area where you can snoop things out in hardware is the Memory Management Unit...So if your MMU gets compromised, then you...have physical addresses and you can do whatever you want with it. So, secure hardware design become critical."*

By nature, IoT devices interact with humans and the physical world. This makes privacy a concern for both developers and end-users. Interviewee P2 said privacy was the most difficult aspect of security analysis, and described his company's approach like this:

> *"The privacy, it is the hard problem...it will be really visible to the market...We are trying to not store any private data that could be...used by hacker in any way....we are simply not trying to tackle such cases. And from my previous work...It was always an issue because it is a really hard problem. And it is really easy to...lose your name, lose your brand."*

RQ4: Trust in IoT systems. Our interviewees identified trust in researchers, vendors, data, and tool chains.

As highlighted in Figure 3, survey respondents indicated trust in researchers through the common adoption of research prototypes. Interviewee P3 described how his company's ML model training process is dependent on security features provided by cloud computing platforms:

> *"So in the fully cloud-based solutions we are largely dependent on...the goodness of the cloud. It's almost impossible to see what Azure, AWS, etc., are doing under the hood. So there's a large level of dependence on their security procedures."*

P3 also noted his trust in development tool chains:

> *"We...are not doing a whole lot of analysis on weakness of...tools like TensorFlow. If TensorFlow...has a security hole, there is not much we do about it. ... [We] have wrappers that ensure there is some levels of encryption, unhackability before it...goes on to the eventual edge IoT device. But if you were to question the IDEs and tools chains having security bugs, there is nothing we can do about it."*

P1 pointed out his assumptions of trustworthy data:

> *"We have to ensure that the [training] data...is from a trusted source, otherwise it becomes a nightmare."*

RQ5: Other factors that affect IoT engineering. *Process requirements and regulations:* During our survey we asked participants about restrictions on their engineering processes and products (Table 2). About half comply only with general quality processes (*e.g.,* P3: *"We are an ISO 9001:2015 company. We rigorously follow the ISO standards."*). Other survey respondents comply with governmental safety and security regulations (26%), and with privacy regulations like GDPR and HIPAA (22%). In P1's organization, they prefer to work with metadata instead of data because of HIPAA requirements:

> *"Once you start working with meta-data, then you don't really need...any private information...so, it becomes much easier."*

P1 expanded on the difficulties of regulatory compliance:

> *"For example, anytime I'm working with the medical data, that becomes a very, very tricky situation...[you must] set up proper working environment and...ensure that the data is not leaving your trusted network...not just personal data, but also [its] trends"*

Table 2: Survey data on process requirements and regulations

What regulations do you have to comply with during your software development process?	Proportion
General engineering processes like ISO XXX	44%
Governmental privacy regulations like GDPR / HIPAA / FERPA	26%
Governmental safety / security regulations like the IoT Cybersecurity Improvement Act (requires following NIST guidelines).	22%
Security engineering processes like OWASP SAMM.	7%

Engineering Cost: Several of our subjects pointed out a balance between security and engineering cost. A survey respondent wrote that the most challenging aspect is:

> *"Addressing vulnerabilities properly...within the project budget...[and] supporting cryptographic functionality for encryption, storage, data transmission, and key/certificate management."*

Interviewee P2 observed that user visibility may justify engineering costs:

> *"First of all you must decide if security is required. If you push security to a level that is hard to maintain, and it is adding significant value to the Bill Of Materials cost, then it is a question if it will be accepted by the market. I believe, the argument about security is if it will be visible to the user."*

Interviewee P1 observed that his company invests security resources non-uniformly, with less effort in analyzing software that they release open-source:

> *"When we do an open-source release, we don't worry much about it...any shortcomings we get notified very quickly by the open-source community and we can fix it. Of course, it is not a good thing to release something insecure to open-source which is not adequately tested or verified...We mainly ensure that previous occurrences of security breaches are tested and we make it part of the design process."*

6 DISCUSSION

6.1 Comparison to prior findings

Our findings overlapped with prior knowledge in many aspects. In terms of development tools, our participants followed industry-wide practices such as using ML frameworks like TensorFlow and PyTorch and development toolchains based on the Visual Studio/-Code IDEs. Our participants follow iterative development processes. The use of hybrid Edge-Cloud architectures is widespread. Power, memory constraints, and computational constraints are known to be major challenges within IoT systems. Our participants are aware of security issues such as data poisoning.

The main difference between the research literature and our findings is the discussion of engineering cost. Our participants — perhaps especially those in consumer electronics — reduce security for cheaper production costs. Similarly, there are many interesting methods of emulation, load-balancing, and system validation proposed in the research literature, but most respondents' organizations do not use these methods. Unlike researchers' goals of unbreakable systems, our subjects balance how much security is possible (relative to its engineering cost) and required (relative to market demand). The research literature generally does not consider the engineering cost of proposed techniques. Lastly, the many sources of unverified trust — open-source code, academic research, and development toolchains — was greater than what we understood in the literature.

6.2 Advice for practitioners

Our study revealed a significant gap between how the academic community and industry perceive IoT security. This suggests potential value in cybersecurity workforce development [7]. Outside academia, government guidelines (*e.g.,* from US-NIST [5] and EU-ENISA [3]) describe secure development lifecycles. NIST [6] recommends a thorough study on the customers, users, expected use cases, security risks, and goals during planning, execution, and post-deployment. Our subjects did not describe such a process.

Given the success of automated code analysis methods such as static analysis, black-box and grey-box fuzzing in identifying system vulnerabilities in IT software, we were surprised by practitioners' continued emphasis on code review and white-box analysis in their IoT systems. We recommend practitioners integrate such methods into their product development process [44].

6.3 Future work for researchers

Based on the challenges faced by the practitioners we studied, we suggest three directions for future research.

First, the IoT domain is characterized by tight profit margins and low-cost parts. Many of our research subjects were therefore concerned about the engineering cost of securing IoT devices. It would be helpful for researchers to offer engineering cost-aware security processes suited to the constraints of IoT systems engineering, and practical measurements of this cost. Past research works primarily focus on trade-offs between security and resource costs, such as operation delay and energy [19, 65]. Our work identifies the importance of considering engineering costs, not just the runtime implications. Our work also complements ongoing research to help consumers understand how security affects the cost of commodity IoT devices [32].

Second, practitioners leverage open science and open-source software for their ML modeling and their development toolchains. This accelerates development, but introduces substantial risk. For ML, we recommend that ML researchers carefully document their research prototypes and the limitations of their work, and that they can achieve broader impact by participating in community efforts to develop exemplary ML models (*e.g.,* TorchVision [52] and the TensorFlow Model Garden [62]). Additional studies of how best to reproduce and transfer ML knowledge will be helpful [13, 17, 37]. More broadly, given the reliance of our participants on open-source tools, trustworthy software supply chains will improve the safety and security of IoT systems [64].

Third, the difficulties experienced by practitioners in following the compliance restrictions and regulations identified in Table 2 poses a potential research area. For example, researchers could study the impact of security compliance on security outcomes of IoT applications, and the tradeoff with engineering cost.

7 THREATS TO VALIDITY

Construct validity: Our survey instrument and interview protocol were intended as direct measures of the constructs of interest (*i.e.,* engineering practices), and we used pilot studies as a check.

Internal validity: Our study reports on practices without inferences about cause and effect, so internal validity is not a concern.

External validity: The primary limitation of our study is in its external validity, *i.e.,* generalizability. Our goal was to describe current practices in IoT engineering, focused on machine learning and cybersecurity. As is common with studies of this kind, we used a human-subjects method with a self-report design, which assumes the respondents were truthful. Beyond the trustworthiness of our data, we emphasize that we had relatively few survey responses (N=25). We cannot claim saturation; our results are likely not representative of the entire state of practice. In addition, 40.9% of the survey respondents identified as students for their current position, and their responses might not reflect the practices in the industry. As mitigating factors, our survey reached participants from several industry sectors, and our interview subjects included experts with a long tenure in industry and experience at several companies.

8 CONCLUSION

In this research attempted to broaden the existing understanding of IoT engineering practices related to machine learning and cybersecurity. Through our survey and interviews, we found that the main challenge engineers face when creating an IoT product is balancing among engineering cost, performance, trust, and security. We found that organizations place unverified trust in open-source and academic resources; going so far as to incorporate academic prototypes of ML techniques into their IoT products. Cybersecurity investment varies based on resources, engineering cost and organizational priorities; one organization even explicitly relies on the open-source community to find vulnerabilities in their software. Practitioners have not yet adopted academic research in engineering practices and government recommendations that might address some of their problems. In the future, we recommend that software engineering and cybersecurity researchers incorporate engineering cost considerations into their work, as this was a concern raised by many of our research subjects.

DATA AVAILABILITY

An artifact is available with the data collection instruments, survey data, and interview transcripts. See https://doi.org/10.5281/zenodo.6383066.

RESEARCH ETHICS

The study was approved by our institution's IRB.

REFERENCES

[1] 2016. Hackers Used New Weapons to Disrupt Major Websites Across U.S. https://www.nytimes.com/2016/10/22/business/internet-problems-attack.html. Accessed June 08, 2021.

[2] 2016. *A Primer on Continuous Delivery.* https://feeney.mba/a-primer-on-continuous-delivery.html

[3] 2017. Baseline Security Recommendations for IoT. https://www.enisa.europa.eu/publications/baseline-security-recommendations-for-iot. Accessed June 09, 2021.

[4] 2017. Your Roomba May Be Mapping Your Home, Collecting Data That Could Be Shared. https://www.nytimes.com/2017/07/25/technology/roomba-irobot-data-privacy.html. Accessed June 08, 2021.

[5] 2019. IoT Device Cybersecurity Capability Core Baseline. https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8259A.pdf. Accessed June 09, 2021.

[6] 2020. Foundational Cybersecurity Activitiesfor IoT Device Manufacturers. https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8259.pdf. Accessed June 09, 2021.

[7] 2020. IoT Non-Technical Supporting Capability Core Baseline. https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8259b-draft.pdf. Accessed June 09, 2021.

[8] 2020. System brings deep learning to "internet of things" devices. https://news.mit.edu/2020/iot-deep-learning-1113.

[9] Yasemin Acar, Christian Stransky, Dominik Wermke, Charles Weir, Michelle L. Mazurek, and Sascha Fahl. 2017. Developers Need Support, Too: A Survey of Security Advice for Software Developers. *Proceedings - IEEE Cybersecurity Development Conference, SecDev* (2017). https://doi.org/10.1109/SecDev.2017.17

[10] Deniz Akdur, Vahid Garousi, and Onur Demirörs. 2018. A survey on modeling and model-driven engineering practices in the embedded software industry. *Journal of Systems Architecture* 91 (2018), 62–82. https://doi.org/10.1016/j.sysarc.2018.09.007

[11] Sultan Alharby, Nick Harris, Alex Weddell, and Jeff Reeve. 2018. The Security Trade-offs in Resource Constrained Nodes for IoT Application. *International Journal of Electrical, Electronic and Communication Sciences: 11.0.* 12, 1 (2018), 56–63. https://www.researchgate.net/publication/322747058%0Ahttp://www.waset.org/downloads/15/papers/18ae010177.pdf

[12] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. SoK: Security Evaluation of Home-Based IoT Deployments. In *2019 IEEE Symposium on Security and Privacy, SP.*

[13] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *International Conference on Software Engineering: Software Engineering in Practice.* https://doi.org/10.1109/ICSE-SEIP.2019.00042

[14] Hala Assal and Sonia Chiasson. 2019. "Think secure from the beginning": A survey with software developers. *Conference on Human Factors in Computing Systems - Proceedings* (2019). https://doi.org/10.1145/3290605.3300519

[15] Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A survey. *Computer Networks* (2010). https://doi.org/10.1016/j.comnet.2010.05.010

[16] Saurabh Bagchi, Tarek F. Abdelzaher, Ramesh Govindan, Prashant Shenoy, Akanksha Atrey, Pradipta Ghosh, and Ran Xu. 2020. New Frontiers in IoT: Networking, Systems, Reliability, and Security Challenges. *IEEE Internet of Things Journal* 7, 12 (2020), 11330–11346. https://doi.org/10.1109/JIOT.2020.3007690

[17] Vishnu Banna, Akhil Chinnakotla, and et al. 2021. An Experience Report on Machine Learning Reproducibility: Guidance for Practitioners and TensorFlow Model Garden Contributors. *arXiv* (2021).

[18] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. 2018. If This Then What?: Controlling Flows in IoT Apps. In *Conference on Computer and Communications Security, CCS.*

[19] Chiara Bodei, Stefano Chessa, and Letterio Galletta. 2019. Measuring Security in IoT Communications. *Theoretical Computer Science* 764 (April 2019), 100–124. https://doi.org/10.1016/j.tcs.2018.12.002

[20] Will Brackenbury, Abhimanyu Deora, Jillian Ritchey, Jason Vallee, Weijia He, Guan Wang, Michael L. Littman, and Blase Ur. 2019. How Users Interpret Bugs in Trigger-Action Programming. In *Conference on Human Factors in Computing Systems CHI.*

[21] Z. Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick D. McDaniel. 2019. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *ACM Comput. Surv.* 52, 4 (2019).

[22] Z. Berkay Celik, Patrick D. McDaniel, Gang Tan, Leonardo Babun, and A. Selcuk Uluagac. 2019. Verifying Internet of Things Safety and Security in Physical Spaces. *IEEE Secur. Priv.* 17, 5 (2019).

[23] Z. Berkay Celik, Gang Tan, and Patrick D. McDaniel. 2019. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *26th Annual Network and Distributed System Security Symposium, NDSS.*

[24] Mengsu Chen, Felix Fischer, Na Meng, Xiaoyin Wang, and Jens Grossklags. 2019. How reliable is the crowdsourced knowledge of security implementation?. In *International Conference on Software Engineering (ICSE).*

[25] Long Cheng, Christin Wilson, Song Liao, Jeffrey Young, Daniel Dong, and Hongxin Hu. 2020. Dangerous Skills Got Certified: Measuring the Trustworthiness of Skill Certification in Voice Personal Assistant Platforms. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[26] François Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017* (2017), 1800–1807. https://doi.org/10.1109/CVPR.2017.195 arXiv:1610.02357

[27] Rafael Maiani de Mello and Guilherme Horta Travassos. 2016. Surveys in Software Engineering: Identifying Representative Samples. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '16)*. Association for Computing Machinery, Article 55, 6 pages. https://doi.org/10.1145/2961111.2962632

[28] Joa Pedro Dias and Hugo Sereno Ferreira. 2018. State of the software development life-cycle for the internet-of-things. *arXiv* (2018). arXiv:1811.04159

[29] Paul Dodemaide, Prof. Lynette Joubert, Dr Nicole Hill, and Dr Mark Merolli. 2020. Online Survey Design and Social Media. In *Proceedings of the Australasian Computer Science Week Multiconference (ACSW '20)*. Association for Computing Machinery, Article 36, 8 pages. https://doi.org/10.1145/3373017.3373054

[30] B. Dorsemaine, J. Gaulier, J. Wary, N. Kheir, and P. Urien. 2015. Internet of Things: A Definition Taxonomy. In *2016 9th International Conference on Next Generation Mobile Applications, Services and Technologies*. 72–77. https://doi.org/10.1109/NGMAST.2015.71

[31] El Mahdi El Mhamdi and Rachid Guerraoui. 2017. When Neurons Fail. *Proceedings - 2017 IEEE 31st International Parallel and Distributed Processing Symposium, IPDPS 2017* (2017), 1028–1037. https://doi.org/10.1109/IPDPS.2017.66 arXiv:1706.08884

[32] Pardis Emami-Naeini. 2020. *Informing Privacy and Security Decision Making in an IoT World.* Ph. D. Dissertation. Carnegie Mellon University.

[33] Michael Fagan, Katerina N Megas, Karen Scarfone, and Matthew Smith. 2020. Foundational Cybersecurity Activities for IoT Device Manufacturers. *NIST Interagency/Internal Report 8259* (2020). https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8259.pdf

[34] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security Analysis of Emerging Smart Home Applications. In *IEEE Symposium on Security and Privacy, SP*.

[35] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. 2018. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *25th Annual Network and Distributed System Security Symposium, NDSS*.

[36] Fereshteh Ghaljaie, Mahin Naderifar, and Hamideh Goli. 2017. Snowball Sampling: A Purposeful Method of Sampling in Qualitative Research. *Strides in Development of Medical Education* 14, 3 (2017). https://doi.org/10.5812/sdme.67670 arXiv:http://sdme.kmu.ac.ir/article_90598_3632edfb2e97c38d73c0bdea8753195c.pdf

[37] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *HPCA*. https://doi.org/10.1109/HPCA.2018.00059

[38] Robert M. Hierons. 1999. Machine Learning, by Tom M. Mitchell, McGraw-Hill, 1997 (Book Review). *Softw. Test. Verification Reliab.* 9, 3 (1999), 191–193.

[39] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. *NIPS Deep Learning and Representation Learning Workshop* (2015), 1–9. arXiv:1503.02531 http://arxiv.org/abs/1503.02531

[40] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv* (2017). arXiv:1704.04861

[41] Jez Humble and Gene Kim. 2018. *Accelerate: The science of lean software and devops: Building and scaling high performing technology organizations.* IT Revolution.

[42] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. 2014. Industry 4.0. *Business & information systems engineering* (2014).

[43] Dirk Van Der Linden, Pauline Anthonysamy, Bashar Nuseibeh, Thein Than Tun, Marian Petre, Mark Levine, John Towse, and Awais Rashid. 2020. Schrodinger's security: Opening the box on app developers' security rationale. *Proceedings - International Conference on Software Engineering* (2020). https://doi.org/10.1145/3377811.3380394

[44] MH Lloyd and PJ Reeve. 2009. IEC 61508 and IEC 61511 assessments-some lessons learned. (2009).

[45] Tamara Lopez, Helen Sharp, Thein Tun, Arosha Bandara, Mark Levine, and Bashar Nuseibeh. 2019. Hopefully we are mostly secure': Views on secure code in professional practice. *Proceedings - 2019 IEEE/ACM 12th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2019* (2019). https://doi.org/10.1109/CHASE.2019.00023

[46] Amir Makhshari and Ali Mesbah. 2021. IoT Bugs and Development Challenges. *ICSE 2021* (2021), 460–472. https://doi.org/10.1109/icse43902.2021.00051

[47] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. 2017. An empirical characterization of IFTTT: ecosystem, usage, and performance. In *Proceedings of the 2017 Internet Measurement Conference, IMC*.

[48] Hooman Mohajeri Moghaddam, Gunes Acar, Ben Burgess, Arunesh Mathur, Danny Yuxing Huang, Nick Feamster, Edward W. Felten, Prateek Mittal, and Arvind Narayanan. 2019. Watching You Watch: The Tracking Ecosystem of Over-the-Top TV Streaming Devices. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS*. ACM.

[49] Blaine Nelson, Benjamin I.P. Rubinstein, Ling Huang, Anthony D. Joseph, Steven J. Lee, Satish Rao, and J. D. Tygar. 2012. Query strategies for evading convex-inducing classifiers. *Journal of Machine Learning Research* 13 (2012), 1293–1332. arXiv:1007.0484

[50] Taehyeun Park, Nof Abuzainab, and Walid Saad. 2016. Learning How to Communicate in the Internet of Things: Finite Resources and Heterogeneity. *IEEE Access* 4 (2016). https://doi.org/10.1109/ACCESS.2016.2615643

[51] Roger S Pressman. 2005. *Software engineering: a practitioner's approach.* Palgrave macmillan.

[52] Pytorch. 2017. TORCHVISION - https://pytorch.org/vision/stable/index.html. https://pytorch.org/vision/stable/index.html?highlight=torchvision#module-torchvision

[53] Bin Qian, Jie Su, Zhenyu Wen, Devki Nandan Jha, Yinhao Li, Yu Guan, Deepak Puthal, Philip James, Renyu Yang, Albert Y. Zomaya, Omer Rana, Lizhe Wang, MacIej Koutny, and Rajiv Ranjan. 2020. Orchestrating the Development Lifecycle of Machine Learning-based IoT Applications: A Taxonomy and Survey. *Comput. Surveys* 53 (2020). Issue 4. https://doi.org/10.1145/3398020

[54] Paul Ralph, Sebastian Baltes, Domenico Bianculli, et al. 2020. ACM SIGSOFT Empirical Standards. *CoRR* abs/2010.03525 (2020). arXiv:2010.03525

[55] Frances Robles and Nicole Perlroth. 2021. 'Dangerous Stuff': Hackers Tried to Poison Water Supply of Florida Town. *The New York Times* (Feb. 2021).

[56] Ruben M. Sandoval, Sebastian Canovas-Carrasco, Antonio Javier Garcia-Sanchez, and Joan Garcia-Haro. 2019. A reinforcement learning-based framework for the exploitation of multiple rats in the iot. *IEEE Access* 7 (2019). https://doi.org/10.1109/ACCESS.2019.2938084

[57] F. Schwandt. 2016. Internet of things (IoT) connected devices installed base worldwide from 2015 to 2025 in billions - Statista. (2016). https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide

[58] Eugene Siow, Thanassis Tiropanis, and Wendy Hall. 2018. Analytics for the Internet of Things: A Survey. *ACM Comput. Surv.* 51, 4, Article 74 (July 2018), 36 pages. https://doi.org/10.1145/3204947

[59] Jacob Steinhardt, Pang Wei Koh, and Percy Liang. 2017. Certified defenses for data poisoning attacks. *Advances in Neural Information Processing Systems* 2017-December, i (2017), 3518–3530. arXiv:1706.03691

[60] Manuel Suarez-Albela, Tiago M. Fernandez-Carames, Paula Fraga-Lamas, and Luis Castedo. 2018. A practical performance comparison of ECC and RSA for resource-constrained IoT devices. *2018 Global Internet of Things Summit, GIoTS 2018* (2018). https://doi.org/10.1109/GIOTS.2018.8534575

[61] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *Proceedings of the 26th International Conference on World Wide Web, WWW*.

[62] TensorFLow. 2021. Models datasets - https://github.com/tensorflow/models. https://github.com/tensorflow/models

[63] Tyler W. Thomas, Madiha Tabassum, Bill Chu, and Heather Lipford. 2018. Security during application development: An application security expert perspective. *Conference on Human Factors in Computing Systems - Proceedings* (2018). https://doi.org/10.1145/3173574.3173836

[64] Santiago Torres-Arias. 2020. *In-toto: Practical Software Supply Chain Security.* Ph. D. Dissertation. New York University Tandon School of Engineering.

[65] Wiebke Toussaint and Aaron Yi Ding. 2020. Machine Learning Systems in the IoT: Trustworthiness Trade-offs for Edge Intelligence. In *2020 IEEE Second International Conference on Cognitive Machine Intelligence (CogMI)*. 177–184. https://doi.org/10.1109/CogMI50398.2020.00030

[66] Jeffrey Voas. 2016. Demystifying the Internet of Things. *Computer* 49, 6 (2016), 80–83. https://doi.org/10.1109/MC.2016.162

[67] Fangxin Wang, Miao Zhang, Xiangxiang Wang, Xiaoqiang Ma, and Jiangchuan Liu. 2020. Deep Learning for Edge Computing Applications: A State-of-the-Art Survey. *IEEE Access* 8 (2020). https://doi.org/10.1109/ACCESS.2020.2982411

[68] Elecia White. 2011. *Making Embedded Systems: Design Patterns for Great Software.* O'Reilly Media.

[69] Han Xiao, Huang Xiao, and Claudia Eckert. 2012. Adversarial label flips attack on support vector machines. *Frontiers in Artificial Intelligence and Applications* 242 (2012), 870–875. https://doi.org/10.3233/978-1-61499-098-7-870

[70] Shuochao Yao, Yiran Zhao, Huajie Shao, Sheng Zhong Liu, Dongxin Liu, Lu Su, and Tarek Abdelzaher. 2018. FastDeepIoT: Towards understanding and optimizing neural network execution time on mobile and embedded devices - SenSys 2018 - Proceedings of the 16th Conference on Embedded Networked Sensor Systems. *SenSys2018*. https://doi.org/10.1145/3274783.3274840

Evaluation of IoT Self-healing Mechanisms using Fault-Injection in Message Brokers

Miguel Duarte
up201606298@fe.up.pt
Faculty of Engineering,
University of Porto
Porto, Portugal

João Pedro Dias
jpmdias@fe.up.pt
BUILT CoLAB and
Faculty of Engineering,
University of Porto
Porto, Portugal

Hugo Sereno Ferreira
hugosf@fe.up.pt
INESC TEC and
Faculty of Engineering,
University of Porto
Porto, Portugal

André Restivo
arestivo@fe.up.pt
LIACC and
Faculty of Engineering,
University of Porto
Porto, Portugal

ABSTRACT

The widespread use of Internet-of-Things (IoT) across different application domains leads to an increased concern regarding their dependability, especially as the number of potentially mission-critical systems becomes considerable. Fault-tolerance has been used to reduce the impact of faults in systems, and their adoption in IoT is becoming a necessity. This work focuses on how to exercise fault-tolerance mechanisms by deliberately provoking its malfunction. We start by describing a proof-of-concept fault-injection add-on to a commonly used publish/subscribe broker. We then present several experiments mimicking real-world IoT scenarios, focusing on injecting faults in systems with (and without) active self-healing mechanisms and comparing their behavior to the baseline without faults. We observe evidence that fault-injection can be used to (a) exercise in-place fault-tolerance apparatus, and (b) detect when these mechanisms are not performing nominally, providing insights into enhancing in-place fault-tolerance techniques.

CCS CONCEPTS

• **Computer systems organization** → **Sensors and actuators**; **Reliability**; Distributed architectures; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

IoT, fault-injection, self-healing, dependability, fault-tolerance, middleware

ACM Reference Format:
Miguel Duarte, João Pedro Dias, Hugo Sereno Ferreira, and André Restivo. 2022. Evaluation of IoT Self-healing Mechanisms using Fault-Injection in Message Brokers. In *4th International Workshop on Software Engineering Research and Practice for the IoT (SERP4IoT'22), May 19, 2022, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3528227.3528567

SERP4IoT'22, May 19, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9332-4/22/05…$15.00
https://doi.org/10.1145/3528227.3528567

1 INTRODUCTION

Internet-of-Things (IoT) is being largely adopted, being ubiquitous across application domains, connecting the physical and virtual realms to provide services that improve the quality-of-life and business processes [43]. IoT devices are typically constrained in both computational power and energy, highly distributed (both logically and geographically), and heterogeneous (*e.g.*, different manufacturers and competing standards). The nature of these systems drives the creation of communication protocols that are lightweight and thus compatible with the computational and energy constraints of these systems. Among those protocols, MQTT has been largely adopted as a lightweight TCP-based Machine-to-Machine (M2M) and IoT connectivity protocol [24]. MQTT leverages a publish/-subscribe pattern, in which a middleware broker guarantees the delivery of messages from publisher entities (*i.e.*, entities that publish messages in a given topic) to one or more subscriber entities (*i.e.*, entities that are subscribed to a given topic) [24].

IoT dissemination required end-users to program their own systems, thus leading to the proliferation of low-code development platforms such as Node-RED which has a visual programming language to mashup together hardware devices, APIs, and online services [26, 33, 40, 42]. However, most of these low-code development environments lack of verification and validation mechanisms [13], and do not provide fault-tolerance mechanisms or suggest how to improve the dependability of these systems. This is expected, as most IoT systems disregard these concerns or achieve redundancy only by device/service replication (with additional costs and management complexity) [10, 28, 36]. This is a mostly direct result of the complexity of creating and using fault-tolerance mechanisms in IoT systems, as pointed by Javed *et al.* , *"building a fault-tolerant system for IoT is a complex task, mainly because of the extremely large variety of edge devices, data computing technologies, networks, and other resources that may be involved in the development process"* [28].

The use of self-healing[22] — the ability of a system to automatically detect, diagnose and repair system defects at both hardware and software levels — to attain fault-tolerance on IoT systems has been suggested by several authors [1, 4, 20, 35, 37, 38]. In previous work [14] a set of patterns to achieve fault-tolerance in IoT systems by adding self-healing mechanisms was introduced, along with a reference implementation in Node-RED. This reference implementation, so-called SHEN[1], consists of a set of self-healing add-on *nodes* to the Node-RED visual programming language that can be

[1]SHEN is available on GitHub: `node-red-contrib-self-healing`, https://github.com/jpdias/node-red-contrib-self-healing.

used to improve the visual *flows* with error detection and system health recovery/maintenance mechanisms [15–17].

One way of ensuring that a self-healing/fault-tolerance mechanisms work as intended, is to actually exercise them. In the research field of fault-tolerance, fault-injection has been used as a technique to deliberately cause errors and failures in systems by introducing faults and then observing how it behaves and recovers from them; a now common practice in Chaos Engineering [8].

For the purposes of this work, we assume that the IoT system under study uses MQTT as the communication substrate, thus requiring a message broker to manage all the combinations, and that there is no redundant broker units. Thus, we can instrument an MQTT broker to inject faults in the messages as they are exchanged in the broker. And, as stated in the *fault model* introduced by Esposito [21], we are able to introduce several types of faults at the network level (more specifically, at the message level) namely: (1) *omission*, (2) *corruption*, (3) *reordering*, (4) *duplication*, and (5) *delay*. With an instrumented broker capable of injecting faults on-demand in a running IoT system with self-healing mechanisms, we can: (a) exercise the in-place fault-tolerance mechanisms, and (b) know when these mechanisms are not working correctly, thus finding improvement targets. We proceed to collect empirical evidence that supports these claims by defining two experimental scenarios and a total of six experiments.

The main contributions of this paper are twofold: (1) an instrumentable MQTT broker that allows fault-injection in IoT systems either using pre-defined operators or user-defined ones, and (2) a validation of the self-healing extensions for Node-RED [15, 16].

The remaining of the paper is structured as follows: Section 2 presents some related work and Section 3 introduces fault-injector proof-of-concept, and Section 4 details the experimental setup. Section 5 presents the carried experiments and the obtained results. Finally, Section 6 presents an overall discussion of the experiments and results, Section 7 discusses some of the threats to the validity of the experiments carried in this work, and Section 8 gives some closing remarks and points to future work.

2 RELATED WORK

Although most literature regarding IoT and fault-injection focuses on hardware faults via physical interaction with devices [25, 29, 46] rather than interfering with the application layer logic (including communication channels and middleware components), the usage of software techniques to exercise in-place fault-tolerance mechanisms can be traced to as early as 1993 [5]. More recently, particularly in cloud computing, Chaos Engineering became an umbrella term to techniques that inject, observe and collect information concerning the impact of faults on a running system [8, 11, 41].

Looker and Xu [30] presented an approach to carry fault-injection in the Open Grid Services Architecture (OGSA) middleware, which ensures the exchange of messages across services (*e.g.*, cloud). Jacques-Silva *et al.* [27] suggested the use of partial fault-tolerance (PFT) techniques to improve the dependability of stream processing applications, given that faults in computational nodes or stream operators can provoke massive data losses due to the high data exchange rates of these systems. Esposito [21] surveyed research between 1995 and 2013 on fault-injection in the context of communication software; they found most of them focused on faults at the message-level and other networking failures (38.8%), with the remaining on process crashes (12.24%), application-level faults (18.4%), memory corruptions (8.2%), and others (22.36%). They also introduce a fault-injection approach for publish/subscribe systems that encompass most of the faults presented in previous works. Yoneyama *et al.* [45] performed model-based fault-injection on MQTT, by mimicking unstable network environments via simulated network errors (*e.g.*, connection lost) and delays. Han *et al.* [44] introduced TRAK as a message-agnostic testing tool for injecting delays and provoke higher packet loss in Apache Kafka [23], thus asserting its QoS capabilities. Zaiter *et al.* [47] presented a distributed fault-tolerance approach for e-health systems based on the exchange of messages between LACs (Local Controlling Agents), reducing the reliance on one global agent controller (GAC). Briland *et al.* [9] explores faults where third-parties inject fabricated data and expect it to modify the system's behaviour over time; they propose a Domain-Specific Language (DSL) to generate altered data that can then be injected into the system to observe its behaviour.

In summary, previous software-based fault-injection literature mostly explores faults at the communication/protocol level [6], with few tackling domain-specific behaviors (*e.g.*, modifying sensor readings). Most also rely on fault-injection agents as new system's components, with a single work preferring to modify the middleware [44]). This limits their usage in IoT due to the computational constraints of most entities. Finally, very few works use fault-injection to evaluate the behaviour of in-place fault-tolerance mechanisms [27, 44, 45, 47]. This paper improves on existing work by (1) creating faults by semantically changing messages passed between different parts of the system, (2) providing a DSL comprised of reactive operators[2], (3) modifying a common middleware to target any MQTT-based system, and (4) designed to support in-place evaluation of fault-tolerance mechanisms.

3 INSTRUMENTED MQTT BROKER

To execute the experiments, we modified an commonly used open-source MQTT broker[3] to inject faults into IoT systems. We choose fault-injection at the message broker level due to the fact the broker is a point of convergence of most IoT systems, mostly independent of the message publish/subscribe complexity, thus reducing the impact of the heterogeneity of the IoT system in the fault-injection strategies. The modifications were done to the broker allowed to use it as a proxy to intercept and modify messages before being published to a specific topic. The modified broker is available on GitHub[4]; more details on its inner workings can be found in [19].

Each fault-injection rule consists of a `topic` (where the rule will be applied), and an array of `operators` each one transforming the incoming message and passing it to the next one (as in a *pipes and filters* architecture). Each rule can also have a `startAfter` and `stopAfter` fields that define the number of messages before the faults start and stop being injected. The implemented operators in

[2] Inspired by the ReactiveX operators [31], http://reactivex.io
[3] AEDES MQTT broker, https://github.com/moscajs/aedes
[4] *Instrumentable-Aedes*, https://github.com/SIGNEXT/instrumentable-aedes

this proof-of-concept are the following: (1) `map`, which takes a function as an argument (*e.g.*, multiply by two) which is then applied to the message, (2) `randomDelay` that delays the publication of a message, (3) `buffer` which captures all the messages until a time interval and/or number of messages is reached, publishing them all at once, and (4) `randomDrop`, which randomly drops some of the messages according to a specified probability. While all operators could be implemented through the `map` one, others are provided for ease the configuration of fault-injection, being the `map` operator useful for creating additional faults and transformations on-demand.

These operators were chosen to be implemented since they can be used to mimic some common faults in sensors and IoT systems as a whole [17, 32], faults which can result from the malfunctioning of a given device (*e.g.*, out-of-spec sensor readings can be mimic with a `map` that multiplies the readings by a random number), network issues (*e.g.*, message loss with `randomDrop` and lag with `randomDelay`), or a combination of these factors (and others). These concrete operators were also selected since they can be used to validate the existent Node-RED self-healing *nodes*. Given the aforementioned operators, we can inject all the faults presented in the *fault model* of Esposito [21].

4 PRELIMINARIES

The possible combinations of the system with and without self-healing or fault-injection result in four variations of the system under test (SUT). We called these **BL** (baseline), self-healing (**SH**), fault-injection (**FI**), and self-healing with fault-injection (**FI×SH**).

If the fault-injection and self-healing mechanisms are working correctly we expect that (1) the behavior of **SH** approximates **BL**, as no fault-injection is performed in either system and self-healing mechanisms should have a low impact in a nominal system; (2) the behavior of **FI** is very different from **BL**, since the base system, without self-healing components, should not be able to recover from injected faults, provided the fault is enough to deviate it from nominal operation; and (3) the behavior of **SH** is similar to that of **FI×SH**, showing that the self-healing mechanisms are able to bring a system with injected faults back into nominal behavior.

These assumptions are expected to hold since the self-healing mechanism for each one of the scenarios was selected in accordance with the type of sensing data, sensing data cadence, sensor maximum, and minimum reading values (*i.e.*, device specifications). Other aspects, including network latency and packet loss, are expected to have no impact since the experiments were run in a single machine. Further, the injected faults are directly related to the type and frequency of sensing data being exchanged; thus, it is expected that such errors would emerge in a real-world system.

The experiments were done on a standard Linux laptop with Node-RED version 1.3.2, and the modified AEDES MQTT broker was run with NodeJS version 14.15.5. A replication package for these experiments is available on Zenodo [18].

5 EXPERIMENTS AND RESULTS

Two test scenarios (**S1** and **S2**) were devised and several experiments were done for each scenario. Each one of these used a separate fault-injection configuration so that different operations were applied to the same dataset. For each experiment, messages were relayed to both the baseline (**BL**) and the corresponding self-healing (**SH**) flow simultaneously to ensure that both systems received the same input and that their outputs (*i.e.*, alarm level) could be directly compared. This also ensured that the fault-injection operators, which rely on the MQTT broker's random number generation, do not produce different results when compared to the baseline.

To remain as close as possible to a real-world system, we used a real-world dataset[5] as sensing data. The dataset used contained NO_x(GT) readings from a device *"located on the field in a significantly polluted area, at road level, within an Italian city"* [12]. The sensing data was *replayed* — *i.e.*, each data point was *emitted* using a Python script with a reduced time interval between readings (in the dataset one data point was collected per hour) — reducing the time required for each experiment to run, thus allowing to increase the number of experiments and refinements. Since the dataset does not provide the concrete specification of the sensor's reading range of values, we considered a widely available sensor[6] as a reference, which has a hardware range of 0-1 ppm (0-1000 *ppb*) and a minimum detection limit of 0.005 ppm (5 *ppb*). Thus, only values ranging from 5 to 1000 *ppb* should be considered valid readings. We considered each reading should trigger an alarm according to three concern levels[7]: *off* (0), *warn* (1), and *danger* (2). A threshold of 53 *ppb* was considered as a *warn*ing value, and 212 *ppb* as a *danger* value.

Each experiment replays a total of 120 messages from the used dataset, and faults are injected to messages 10 thru 110 — this allows the system to gain stability before entering a degradation state and can also resume normality after fault-injection stops. In order to collect insights on the behavior of the SUT, all the messages flowing in the different MQTT topics are monitored and logged during the course of the experiment. All the experiments use the same **BL**, with three NO_x sensors used to trigger an alarm according to the defined thresholds. The corresponding **BL** system, implemented as a Node-RED flow, parses the reading received from the sensors and sends the respective alarm level as output, filtered by a report-by-exception node to ensure that values are only emitted when the current alarm level changes. The Node-RED *flows* with self-healing capabilities used in the following scenarios use a subset of the SHEN *nodes*, as presented in previous work [16].

5.1 Sensor Readings Issues (S1)

Here (**S1**) we performed four experiments, each with different types of fault-injection operators applied to the sensor messages passing through the MQTT broker (*i.e.*, simulating sensor malfunctions).

A Node-RED flow, with self-healing mechanisms, was developed to deal with these issues (**SH**). This system expands upon **BL** by introducing self-healing capabilities via `SHEN` nodes. It filters extraneous messages that are outside the expected operating range, compensates for missing values after a certain timeout, joins messages so that they are considered in groups of 3, considers the majority of values with a minimum consensus of 2 (with a 25%

[5]Dataset available at https://archive.ics.uci.edu/ml/datasets/Air+Quality and http://archive.ics.uci.edu/ml/machine-learning-databases/00360/.
[6]https://www.aeroqual.com/product/nitrogen-dioxide-sensor-0-1ppm.
[7]While the *EPA National Ambient Air Quality Standards list 0.053ppm as the avg. 24-hour limit for NO2 in outdoor air*[2], for validation purposes we will consider these limits for each data point and not the 24-hour average value.

difference margin), and compensates for readings for which there is no majority with a mean of the previous readings, besides the basic functionality implemented by **BL**. The `join` and `compensate` *nodes* are configured with a timeout of 6 seconds to have a margin of 1 second in relation to the readings' periodicity (5 seconds).

5.1.1 Experiment S1E1. In this experiment no fault injection was performed, with only the baseline system (**BL**) and the system with added self-healing mechanisms (**SH**) being considered. This allows us to compare the behavior of **BL** with **SH** in normal operation, and creates a base of comparison for the remaining experiments (*i.e.*, experiments with fault-injection). This also provides us a behavioral profile of **BL** when compared with **SH**, giving us insights on how self-healing mechanisms' operate with no added entropy.

We expect the SUT to remain stable during this experiment, outputting the expected alarm levels for the sensor readings' thresholds. Despite the expected similarity in behaviour, it is expected that **SH**'s alarm level output will be more stable than that of **BL**. This is due to the latter not implementing any type of consensus or majority voting and instead simply using the received values directly as a stream.

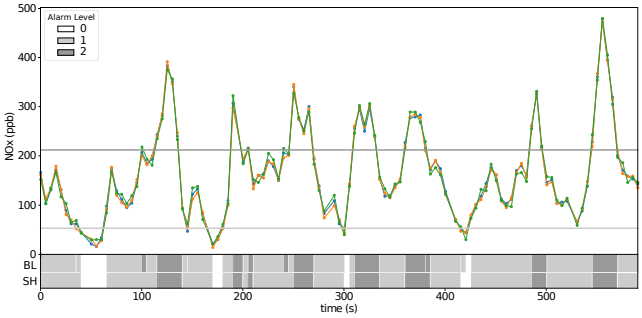

Figure 1: NO_x concentration and alarm status for S1E1.

Fig. 1 shows the experiment results for **BL** and **SH**. Despite the alarm output (represented as shaded areas near the horizontal axis) being very similar for both experiment outputs, stability is higher for **SH**. This can be observed by the lack of fast alarm state changes for borderline values for **SH**, which occur several times for **BL** (*e.g.*, around 35 to 40 seconds into the experiment or around 415 to 420 seconds). The cause of this is likely to be **BL**'s lack of consensus mechanism, given that this system instead simply considers the most recent reading in order to determine the alarm state. When the three sensors report in quick succession, if the values of their readings are near the alarm level thresholds, fluctuations in the alarm level are expected.

The output similarity is confirmed by the alarm level overlap percentage between these two outputs, which is 97.3%. This is also a good sanity check to confirm that the addition of self-healing capabilities to the base system does not significantly change the alarm output status, which means that comparisons for **SH** between **S1E1**, and further experiments, will be meaningful in validating self-healing recovery of injected faults.

Table 1 supports the previous claim — that **SH** provides an improvement in system stability in comparison to **BL** — due to the lower number of alarm state transitions. Additionally, this experiment presents evidence that both **BL** and **SH** correctly implement the expected core functionality (triggering the different alarm levels for different sensor reading thresholds), given that the alarm level at a given point in time corresponds to the sensor readings' distribution along the thresholds (represented in the mentioned figures by the horizontal lines).

Table 1: Count of alarm level state transitions for S1E1-4.

	S1E1		S1E2		S1E3		S1E4	
	BL	**SH**	**BL**	**SH**	**BL**	**SH**	**BL**	**SH**
Off (0)	8	4	10	5	8	4	8	4
Warn (1)	20	13	68	14	26	13	20	13
Danger (2)	11	8	70	8	17	8	11	8
Total	39	25	148	27	51	25	39	25

5.1.2 Experiment S1E2. Considering the baseline (**BL**) and the self-healing (**SH**) systems (which S1E1 shows to be similar in normal operation), we proceed to inject faults on both, obtaining systems **FI** (corresponding to the injection of faults in **BL**) and **FI×SH** (corresponding to the injection of faults in **SH**).

The fault being injected corresponds to an erroneous *Sensor 3*'s reading. As a result, this sensor's readings are altered to be stuck at the upper operating bound (1000 *ppb*). This experiment simulates a fault in which a sensor malfunctions by continuously emitting readings in its top operating bound.

We expect that this fault-injection will provoke an inconsistent output in **FI**, especially if the third sensor is frequently the last to emit its reading, even if only by a slight delay. Due to relying on a majority of at least two values to decide on the alarm level to emit, we expect that the self-healing mechanism will be able to deal with the faults injected in **FI×SH**.

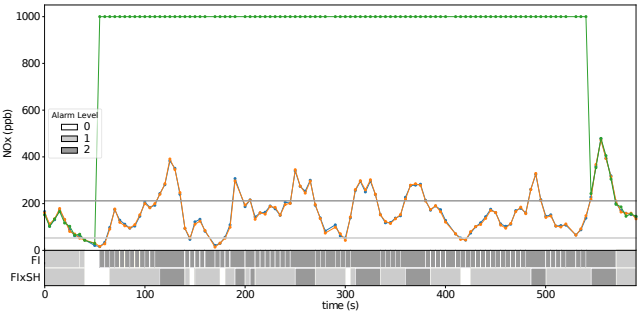

Figure 2: NO_x concentration and alarm status for S1E2.

Fig. 2 shows the experiment results for **FI** and **FI×SH**. The faults injected (**FI**) disrupt the normal function of the system, resulting in constant alternation between alarm states, spending most of the experiment's time in the highest alarm level. Meanwhile, **FI×SH** successfully recovers from the injected faults, having a near-perfect performance in comparison to this system's output for **S1E1**.

These statements are supported by the overlap in alarm levels between **FI×SH** and **SH**, with a near-perfect overlap of 98.1%, and

FI and **BL**, with a much lower overlap percentage of 40.0%. Table 1 also illustrate these conclusions, in which **FI** is much more unstable in comparison to **BL**, being the total number of state transitions for this experiment 148, while there were only 39 state transitions for the base experiment. Furthermore, the number of state transitions with self-healing has increased only marginally, going from 25 in the base experiment (**SH**) to 27 in this experiment (**FI×SH**).

Therefore, **S1E2** demonstrates that the original system cannot handle sensor *stuck-at* issues since the behavior of **FI** is considerably affected, which validates that the performed fault-injection was meaningful enough to disturb the system's regular operation. On the other hand, **FI×SH** can recover from the injected faults, having a remarkably similar behavior to **SH**. This happens since the *stuck-at* fault only affects one sensor, and with the usage of the `replication-voter` node, this reading will be discarded and the other two sensors' values will be considered instead, resulting in a system that operates similarly to the **SH**.

5.1.3 Experiment S1E3. In this experiment, we injected faults in **BL** and **SH** to obtain **FI** and **FI×SH** by multiplying 40% of the readings done by *Sensor 3* by a random factor in the range $[0.2, 2.2]$, simulating *spikes* in sensor readings. The factor is randomized for each *spike* occurrence[8].

We expect that **FI** may output incorrect alarm values (in comparison to **BL**), especially when the altered values switch between alarm level thresholds. On the other hand, **FI×SH** should be able to handle the *spikes* since that, even if one of the three sensors outputs a value considerably different from the others, it will be discarded and the other two sensors' values will be considered instead — due to the usage of the `replication-voter` node.

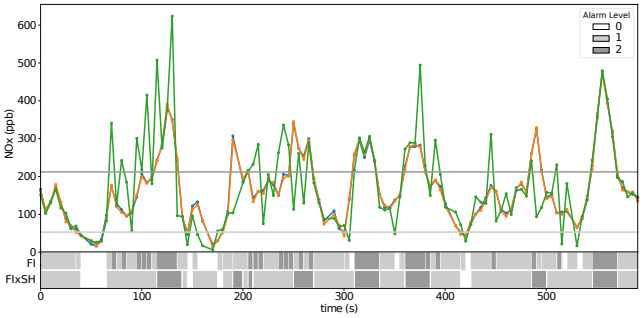

Figure 3: NO_x concentration and alarm status for S1E3.

Fig. 3 shows the experiment results for **FI** and **FI×SH**. **FI** has had a good performance in the presence of the *spikes* (both when increasing and decreasing the read value), but there were still several situations in which the sensor reading *spike* caused the output alarm level to differ from the expected value in **BL**. **FI×SH** has held up to the defined expectations, handling almost all the injected faults and operating similarly to **SH**.

These statements are supported by the overlap in alarm levels between **FI×SH** and **SH**, with a near-perfect overlap of 97.4%, and **FI** and **BL**, with a lower overlap percentage of 76.3%, showcasing the disruption provoked by the injected *spikes*.

[8]This fault is common for sensing devices when they are running out of battery [32].

Table 1 supports the role of the self-healing mechanisms. Despite the difference not being as remarkable as that of the overlap percentages, it is of note to mention that **FI×SH** has the exact same number of alarm level state transitions of **SH** while the number of state transitions has increased for **FI** when compared with **BL**.

Despite this experiment not causing a variation as significant as that of the behavior of **FI** in **S1E2**, we were still able to observe a mismatch between the behavior of this system between the base case and this experiment, even if to a lesser extent. This shows that for the system under study, the *spikes* faults are less concerning when compared with the *stuck-at* ones injected in **S1E2**. Nevertheless, due to the decline in the overlap percentage for **FI** in comparison with **BL**, we can conclude that the faults injected were significant enough to affect the system's correct functioning.

Since **FI×SH** behaved similarly to **SH**, we can confirm that for this experiment the presence of self-healing capabilities are beneficial for the system's correct operation, thus improving its resilience.

5.1.4 Experiment S1E4. In this experiment, we injected faults in *Sensor 3* so that it has a 20% chance of losing messages[9]. The system does not receive any of the lost messages, as these are suppressed before leaving the message broker.

We expect that **FI** may report erroneous alarm values (compared to the base experiment, **S1E1**), especially when the missing values are in proximity to the alarm thresholds. **FI×SH** should be able to handle the injected faults by compensating the missing values by replaying the last message in the expected time interval.

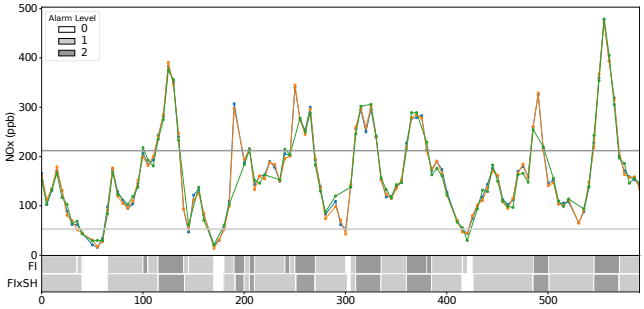

Figure 4: NO_x concentration and alarm status for S1E4.

Fig. 4 shows the experiment results for **FI** and **FI×SH**. **FI** is capable of handling the loss of some readings, thus the alarm output is quite similar to **BL**. **FI×SH** is also able to handle the loss of readings, similarly having almost the same behavior as **SH**.

The similarity in outputs when compared with **S1E1** is a direct result of the low probability of losing one reading. Also, since the values for different sensors in the original dataset are close to one another, even by increasing the probability of values being suppressed from one sensor would result in **FI** outputting the expected alarm values for most of the experiment's duration.

These statements are supported by the overlap in alarm levels between **FI×SH** and **SH**, 98.7%, and **FI** and **BL**, 99.8%, showing that **FI** has a similar behavior to **BL**, and that **FI×SH** has a similar

[9]This fault may occur when a sensor is disconnected, has an intermittent power supply, or the network is unstable [32].

behaviour to **SH**. The previous observations are also supported by the results in Table 1, which shows the number of state transitions for both systems to be identical to those that occurred in **S1E1**.

This experiment did not cause a significant enough deviation from the base experiment's behavior for **FI**, which is corroborated by the high overlap percentage with **BL** of 98.4%, as well as the fact that the number and type of alarm state transitions are identical to those of **S1E1** (*cf.* Table 1). Thus, we conclude that the used dataset may not be the best candidate for this type of fault injection. A higher deviation in operation could possibly be observed if the fault-injection was done to more than one sensor at a time and with a higher probability of losing a message.

5.2 Timing Issues (S2)

In this scenario (**S2**), the Node-RED flow self-healing mechanisms used in **S1** was enriched with *nodes* that detect and mitigate issues with timings (*e.g.*, readings frequency issues) by introducing `debounce` nodes which can filter out extraneous messages based on the expected timing of the system's regular messages. The `join` and `compensate` *nodes* are configured with a timeout of 6 seconds to have a margin of 1 second in relation to the readings' periodicity (5 seconds). A total of two experiments were conducted for **S2**.

5.2.1 Experiment S2E1. No faults are injected for this experiment. Similarly to **S1E1**, the purpose of this experiment is to confirm that the system's base functionality is correctly implemented for both **BL** and **SH**, as well as to provide a base experimental output with which to compare the behavior of the systems in following experiments. As with **S1E1**, we expected that the systems under observation remain stable during this experiment since there are no injected faults and that **SH**'s alarm level output will be more stable than that of **BL**. The results were similar to those of **S1E1** with a similarity of 97.4%.

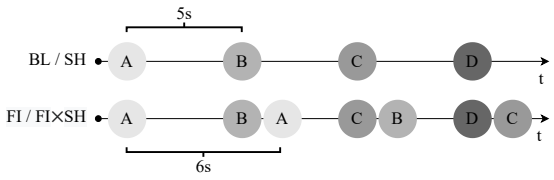

Figure 5: Marble diagram of messages for S2E2. The top diagram depicts the regular flow of messages, while the bottom diagram shows the messages after the fault injection.

5.2.2 Experiment S2E2. In this experiment, to introduce additional noise into the system, each message for *Sensor 3* is repeated after 6 seconds, as depicted in Fig. 5. Since the periodicity of the system's readings in regular circumstances is of 5 seconds, the repeated message will be outputted in close proximity to the next reading.

We expect **FI** to have an output that is less stable than it was for **BL** due to the injected faults; this may be problematic for **FI** since it does not have any concept of message timing. On the other hand, **FI×SH** should be able to cope with the injected faults since the `debounce` *node* will filter out the additional messages that come out of the expected frequency, thus behaving similarly to **SH**.

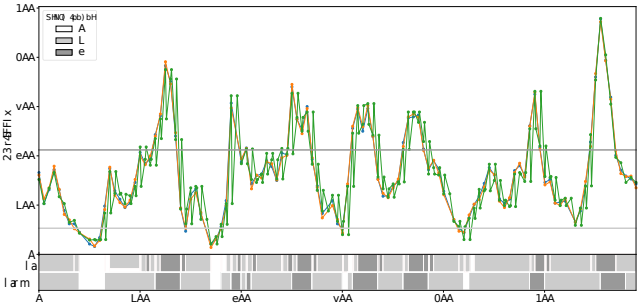

Figure 6: NO_x concentration and alarm status for S2E2.

Table 2: Count of alarm level state transitions for S2E1-2.

	S2E1		S2E2	
	FI	**FI×SH**	**FI**	**FI×SH**
Off (0)	8	4	12	4
Warn (1)	20	13	36	13
Danger (2)	11	8	25	8
Total	39	25	73	25

FI (Fig. 6) performed significantly better than expected, despite the issues with messages near the alarm level thresholds, *i.e.*, repeating the previous message would cause the system to output the previous alarm level once again until it received the following message and went back to the expected state. This is caused by the fact that the periodicity of the sensor readings messages is quite high (*i.e.*, number of sensor readings per unit of time) and, whenever a fault is injected, it is not *in effect* for a long duration. This can be checked by observing the number of alarm state transactions and their duration — even with more state transactions, the time that the alarm stays at that given alert level is short — resetting to a normal state when a new reading appears.

As expected, **FI×SH** was able to cope with the injected faults (Fig. 6), having a near-perfect behavior in comparison to **SH**. These statements are supported by the overlap in alarm levels between **FI×SH** and **SH**, with a near-perfect overlap of 95.7%, and **FI** and **BL**, with a slightly lower overlap percentage of 83.4%, showcasing the disruption provoked by the injected *spikes*. Despite the high overlap percentages for **FI** with **BL**, Table 2 shows that for **S2E2**, **FI** has output nearly two times the amount of alarm level state transitions in comparison to **S2E1**.

S2E2 shows that despite the introduction of faults in **FI** the difference shown by the overlap percentage to **BL** is minimal. Despite this, **FI×SH** cope better with the injected faults, operating closer to **SH**. **FI** also performs worse than **FI×SH** when taking into account the number of alarm level state transitions (*cf.* Table 2). And, while it is not possible to conclude if this experiment caused enough deviation from **BL** to **FI** by taking only in consideration the overlap percentage, the difference in the count of alarm level state transition for **FI** in comparison to **BL** provides evidence that the injected faults have caused issues on the baseline system which the self-healing system is able to sustain.

6 DISCUSSION

The fault-injection experiments **S1E1**, **S1E2**, **S1E3**, **S2E1**, and **S2E2** allowed us to observe that: (1) the self-healing systems (**SH**) do not deviate too much in behavior from the baseline system (**BL**); (2) the faults injected are consequential since there is a deviation on the baseline system in comparison to the base experiment when no fault is being injected; and (3) when the faults injected are consequential, the self-healing systems were able to recover from it, conforming with the normal service, and thus confirming that the self-healing mechanisms were being exercised and performing as expected.

More concretely, by analyzing the Table 1 and Table 2, we can see the impact that fault-injection has in a system without any fault-tolerance mechanisms versus a system with self-healing capabilities. The number of times that the alarm changes state between its three alert levels is considerably higher in all experiments, with a clear impact in the experiments **S1E2**, **S1E3**, and **S2E2**, where the number of transitions was more than two times higher than the expected number of transitions. While the overlap percentages of the different experiments do not provide enough evidence to draw conclusions for most experiments, we can see that **S1E2** performs considerably better when self-healing mechanisms are present.

Additionally, **S1E4** allowed us to verify that it is paramount for the behavior of **BL** to be *noticeably* different when faults are being injected (**FI**) in comparison to the regular operating circumstances. This factor made it so that we considered this experiment inconclusive due to the low entropy caused in the system. Nevertheless, it shows that it is necessary to find this stark difference in expected versus observed output for the baseline system to be sure if the self-healing components are doing any work at all since a naïve system would already be able to *recover* from most injected faults.

It is also noticeable that for the created scenarios, due to the used dataset, some types of fault-injection did not result in much instability. This is due to the fact that all three sensors output readings with values in close to each other. As such, if one or even two sensors fail, it is likely that a naïve system (*e.g.*, **BL**) will still perform as expected, outputting the correct alarm levels for most cases even in the presence of faults (**FI**). This is an indicator that further validation should be performed with other types of datasets and systems, as well as different types of faults.

7 THREATS TO VALIDITY

The experiments presented in this work were carried out using a real-world dataset *replayed* (*i.e.*, simulated) using a Python script. While all the messages were replayed, even if they contained erroneous or invalid readings, we cannot recreate any errors that could exist with the data if any pre-processing was done to the dataset before being put public available. This could be mitigated by using additional datasets of different IoT systems and carrying some extra, even if smaller, experiments in a real testbed. Further, while running the experiments in a simulated environment eases reproducibility, it has the downside of not covering sporadic faults that could occur in the physical testbed, *e.g.*, electromagnetic interference's and network disruptions.

An inadequate selection of the faults injected into the system also poses a threat to the validity of the experiments carried in this work since they have been hand-picked with prior knowledge of the fault-injection and self-healing capabilities. This can result in a bias in the selection, favoring issues that we know about and having more confidence that the proposed solution will handle correctly, instead of the ones that are mostly like to occur. While we attempt to mitigate this by using a real-world dataset, we have the bias of picking the faults injected and the self-healing logic.

Injecting faults at middleware (*i.e.*, message broker) also limits the range of possibilities. While we attempted to replicate some common sensor fault types [32], other faults, including ones at the network level, were not covered by this study. Faults and implementation quirks of the underlying infrastructure — *i.e.*, Python message emitter script, modified MQTT broker, Node-RED, and the self-healing extensions — might also have influenced the outcome of the carried experiments, so they are a confounding variable. We believe this has been mitigated by careful analysis of the expected and actual results, though it is something of concern.

8 CONCLUSIONS

Ensuring the dependability of software systems has been the goal of most fault-tolerance research in the past years [7]. In IoT, ensuring security, reliability, and compliance is becoming a paramount concern due to the recent increase in mission-critical contexts. IoT fault-tolerance is considerably challenging due to several aspects, including (1) high heterogeneity of devices, (2) interaction and limitations of systems deployed in a *physical* environment, (3) field fragmentation, ranging from the high number of communication protocols to the different and competing standards, and (4) intrinsic dependability on hardware that might simply fail [3].

Fault-injection becomes paramount to ensure that fault-tolerance mechanisms perform as they are expected when required. By instrumenting an MQTT broker, we enable the injection of faults at the message-passing level that allows to observe how well components deal with such faults. This provides a means to assert if the self-healing mechanisms configured in the system are sufficient to deal with them. The carried experiments showcase that the self-healing extensions do, indeed, work as expected, with the injected faults causing little to no impact on the delivery of normal service.

As future improvements to the instrumented MQTT broker, we consider the following: (1) simplify the fault-injection configuration by supporting more native language constructs (*e.g.*, arrow functions) and other configuration abstractions (*e.g.*, leverage visual notations), (2) support wildcard topics as per the MQTT specification, and (3) enable switching configuration at run-time instead of having to specify the configuration file when starting the broker. Regarding the experimental stage, it would be interesting to (1) expand the scenarios with more experiments, including more extensive fault-injection pipelines; (2) replicated the experiments using different datasets; (3) extend the usage of self-healing mechanisms, especially the ones implemented as part of SHEN [16]; and (4) explore decentralized IoT systems and orchestrators [34, 39].

ACKNOWLEDGMENTS

This work was partially funded by the Portuguese Foundation for Science and Technology (FCT), *ref.* SFRH/BD/144612/2019.

REFERENCES

[1] Mehmet S. Aktas and Merve Astekin. 2019. Provenance aware run-time verification of things for self-healing Internet of Things applications. *Concurrency Computation* 31, 3 (2019), 1–9.

[2] Ammar A. Al-Sultan, Ghufran F. Jumaah, and Faris H. Al-Ani. 2019. Evaluation of the Dispersion of Nitrogen Dioxide and Carbon Monoxide in the Indoor Café – Case Study. *Journal of Ecological Engineering* 20, 4 (2019), 256–261.

[3] M. Aly, F. Khomh, Y. Guéhéneuc, H. Washizaki, and S. Yacout. 2019. Is Fragmentation a Threat to the Success of the Internet of Things? *IEEE Internet of Things Journal* 6, 1 (Feb. 2019), 472–487.

[4] Rafael Angarita. 2015. Responsible objects: Towards self-healing internet of things applications. *Proceedings - IEEE International Conference on Autonomic Computing, ICAC 2015* (2015), 307–312.

[5] J. Arlat, A. Costes, Y. Crouzet, J.C. Laprie, and D. Powell. 1993. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Trans. Comput.* 42, 8 (1993), 913–923.

[6] Cyrille Valentin Artho, Armin Biere, Masami Hagiya, Eric Platon, Martina Seidl, Yoshinori Tanabe, and Mitsuharu Yamamoto. 2013. Modbat: A Model-Based API Tester for Event-Driven Systems. In *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings (Lecture Notes in Computer Science, Vol. 8244)*, Valeria Bertacco and Axel Legay (Eds.). Springer, 112–128.

[7] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. 2001. Fundamental Concepts of Dependability. *Technical Report Seriesuniversity of Newcastle Upon Tyne Computing Science* 1145, 010028 (2001), 7–12.

[8] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. 2016. Chaos Engineering. *IEEE Software* 33, 3 (2016), 35–41.

[9] Mathieu Briland. 2021. A Language for Modelling False Data Injection Attacks in Internet of Things. (2021), 1–8. http://www.flowbird.group

[10] Bin Cheng, Gurkan Solmaz, Flavio Cirillo, Erno Kovacs, Kazuyuki Terasawa, and Atsushi Kitazawa. 2017. FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities. *IEEE Internet of Things Journal* 4662, c (2017).

[11] D. Cotroneo, L. De Simone, P. Liguori, and R. Natella. 2020. Fault Injection Analytics: A Novel Approach to Discover Failure Modes in Cloud-Computing Systems. *IEEE Transactions on Dependable and Secure Computing* (2020).

[12] S. De Vito, E. Massera, M. Piga, L. Martinotto, and G. Di Francia. 2008. On field calibration of an electronic nose for benzene estimation in an urban pollution monitoring scenario. *Sensors and Actuators B: Chemical* 129, 2 (2008), 750–757.

[13] João Pedro Dias, Flávio Couto, Ana C.R. Paiva, and Hugo Sereno Ferreira. 2018. A Brief Overview of Existing Tools for Testing the Internet-of-Things. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 104–109. https://doi.org/10.1109/ICSTW.2018.00035

[14] Joao Pedro Dias, Hugo Sereno Ferreira, and Tiago Boldt Sousa. 2019. Testing and Deployment Patterns for the Internet-of-Things. In *Proceedings of the 24th European Conference on Pattern Languages of Programs* (Irsee, Germany) *(EuroPLop '19)*. ACM, New York, NY, USA, Article 16, 8 pages.

[15] João Pedro Dias, Bruno Lima, João Pascoal Faria, André Restivo, and Hugo Sereno Ferreira. 2020. Visual Self-Healing Modelling for Reliable Internet-of-Things Systems. In *Proceedings of the 20th International Conference on Computational Science (ICCS)*. Springer, 27–36.

[16] Joao Pedro Dias, André Restivo, and Hugo Sereno Ferreira. 2021. Empowering Visual Internet-of-Things Mashups with Self-Healing Capabilities. In *2021 IEEE/ACM 3rd International Workshop on Software Engineering Research Practices for the Internet of Things (SERP4IoT)*.

[17] Joao Pedro Dias, Tiago Boldt Sousa, Andre Restivo, and Hugo Sereno Ferreira. 2020. A Pattern-Language for Self-Healing Internet-of-Things Systems. In *Proceedings of the European Conference on Pattern Languages of Programs 2020* (Virtual Event, Germany) *(EuroPLoP '20)*. ACM, New York, NY, USA, Article 25, 17 pages.

[18] Miguel Duarte, João Pedro Dias, Hugo Sereno Ferreira, and André Restivo. 2021. Dataset of Fault-Injection experiments in a publisher/subscriber IoT system. https://doi.org/10.5281/zenodo.5148566

[19] Miguel Pereira Duarte. 2021. *MQTT Chaos Engineering for Self-Healing IoT Systems*. Master's thesis. Faculty of Engineering, University of Porto.

[20] Bahadir Dundar, Merve Astekin, and Mehmet S. Aktas. 2016. A Big Data Processing Framework for Self-Healing Internet of Things Applications. In *2016 12th International Conference on Semantics, Knowledge and Grids (SKG)*. 62–68.

[21] Christian Esposito. 2013. *Evaluating Fault-Tolerance of Publish/Subscribe Services*. Springer Milan, Milano, 115–130.

[22] Alan G. Ganek and Thomas A. Corbi. 2003. The dawning of the autonomic computing era. *IBM Systems Journal* 42, 1 (2003), 5–18.

[23] Nishant Garg. 2013. *Apache kafka*. Packt Publishing Birmingham.

[24] Gaston C Hillar. 2017. *MQTT Essentials-A lightweight IoT protocol*. Packt Publishing Ltd.

[25] A. Höller, A. Krieg, T. Rauter, J. Iber, and C. Kreiner. 2015. QEMU-Based Fault Injection for a System-Level Analysis of Software Countermeasures Against Fault Attacks. In *2015 Euromicro Conference on Digital System Design*. IEEE, 530–533.

[26] Felicien Ihirwe, Davide Di Ruscio, Silvia Mazzini, Pierluigi Pierini, and Alfonso Pierantonio. 2020. Low-Code Engineering for Internet of Things: A State of Research. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (Virtual Event, Canada) *(MODELS '20)*. ACM, New York, NY, USA, Article 74, 8 pages.

[27] Gabriela Jacques-Silva, Bugra Gedik, Henrique Andrade, Kun-Lung Wu, and Ravishankar K. Iyer. 2011. Fault Injection-Based Assessment of Partial Fault Tolerance in Stream Processing Applications. In *Proceedings of the 5th ACM International Conference on Distributed Event-Based System* (New York, New York, USA) *(DEBS '11)*. ACM, New York, NY, USA, 231–242.

[28] A. Javed, K. Heljanko, A. Buda, and K. Främling. 2018. CEFIoT: A fault-tolerant IoT architecture for edge and cloud. In *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*. 813–818.

[29] Zahra Kazemi, David Hely, Mahdi Fazeli, and Vincent Beroulle. 2020. A Review on Evaluation and Configuration of Fault Injection Attack Instruments to Design Attack Resistant MCU-Based IoT Applications. *Electronics* 9, 7 (2020), 1153. https://www.mdpi.com/2079-9292/9/7/1153

[30] N. Looker and Jie Xu. 2003. Assessing the dependability of OGSA middleware by fault injection. In *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.* 293–302.

[31] Andrea Maglie. 2016. Reactivex and rxjava. In *Reactive Java Programming*. Springer, 1–9.

[32] Kevin Ni, Nithya Ramanathan, Mohamed Nabil Hajj Chehade, Laura Balzano, Sheela Nair, Sadaf Zahedi, Eddie Kohler, Greg Pottie, Mark Hansen, and Mani Srivastava. 2009. Sensor Network Data Fault Types. *ACM Trans. Sen. Netw.* 5, 3, Article 25 (June 2009), 29 pages.

[33] OpenJS Foundation. 2019. Node-RED, Flow-based programming for the Internet of Things. https://nodered.org/ Last Access 2019.

[34] Duarte Pinto, João Pedro Dias, and Hugo Sereno Ferreira. 2018. Dynamic allocation of serverless functions in IoT environments. In *2018 IEEE 16th international conference on embedded and ubiquitous computing (EUC)*. IEEE, 1–8.

[35] Antonio Ramadas, Gil Domingues, Joao Pedro Dias, Ademar Aguiar, and Hugo Sereno Ferreira. 2017. Patterns for things that fail. In *Proceedings of the 24th Conference on Pattern Languages of Programs*. 1–10.

[36] Partha Pratim Ray. 2016. A survey of IoT cloud platforms. *Future Computing and Informatics Journal* 1, 1-2 (2016), 35–46.

[37] Jan Seeger, Arne Bröring, and Georg Carle. 2020. Optimally Self-Healing IoT Choreographies. *ACM Transactions on Internet Technology* 20, 3, Article 27 (July 2020), 20 pages.

[38] Ronny Seiger, Stefan Herrmann, and Uwe Abmann. 2017. Self-Healing for Distributed Workflows in the Internet of Things. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 72–79.

[39] Margarida Silva, João Dias, André Restivo, and Hugo Ferreira. 2020. Visually-defined real-time orchestration of iot systems. In *MobiQuitous 2020-17th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. 225–235.

[40] Margarida Silva, João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. 2021. A review on visual programming for distributed computation in iot. In *International Conference on Computational Science*. Springer, 443–457.

[41] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. 2018. Engineering Software for the Cloud: External Monitoring and Failure Injection. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs* (Irsee, Germany) *(EuroPLoP '18)*. ACM, New York, NY, USA.

[42] Diogo Torres, João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. 2020. Real-time feedback in node-red for iot development: An empirical study. In *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, 1–8.

[43] Roy Want, Bill N. Schilit, and Scott Jenson. 2017. Enabling the Internet of Things. *Enabling the Internet of Things* (2017), 1–45.

[44] Han Wu, Zhihao Shang, and Katinka Wolter. 2019. TRAK: A Testing Tool for Studying the Reliability of Data Delivery in Apache Kafka. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*.

[45] Jun Yoneyama, Cyrille Artho, Yoshinori Tanabe, and Masami Hagiya. 2019. Model-based Network Fault Injection for IoT Protocols. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2019, Heraklion, Crete, Greece, May 4-5, 2019*, Ernesto Damiani, George Spanoudakis, and Leszek A. Maciaszek (Eds.). INSTICC, SciTePress, 201–209.

[46] David Zooker Zabib, Maoz Vizentovski, Alexander Fish, Osnat Keren, and Yoav Weizman. 2017. Vulnerability of secured IoT memory against localized back side laser fault injection. In *2017 Seventh International Conference on Emerging Security Technologies (EST)*. IEEE, 7–11.

[47] Meriem Zaiter and Salima Hacini. 2020. A Distributed Fault Tolerance Mechanism for an IoT Healthcare system. In *2020 21st International Arab Conference on Information Technology (ACIT)*. 1–6.

Vulnerability Classification of Consumer-based IoT Software

Bara' Nazzal, Atheer Abu Zaid, Manar H. Alalfi, Altaz Valani
Email:{bara.nazzal,aabuzaid,manar.alalfi}@ryerson.ca,avalani@securitycompass.com
Department of Computer Science, Ryerson University, Security Compass
Toronto, ON, Canada

ABSTRACT

This paper surveys and categorizes potential software vulnerabilities in consumer-based IoT applications. We look at the currently available reported vulnerabilities in the SmartThings platform as well as potential vulnerabilities that face IoT platforms in general. We provide a multi-step categorization that applies available guidance as well as connecting it to frameworks such as OWASP and MITRE ATT&CK to classify the vulnerabilities depending on their platform, layer, nature, class as well as the suggested mitigation.

1 INTRODUCTION

One new and emerging class of software is the one that operates/-manages Internet-of-Things (IoT) devices. Whether it is home applications, self-driving cars or every day utilities, more devices are being embedded with computing capabilities and connection to the internet. This can provide much convenience to the users but also poses risks in the field of security and privacy. The global IoT market is expected to grow US 2488 billion by the end of 2022. With the current market competition, the rush to deploy products often results in a lack of security considerations during the design phase, according to Threatpost. Currently, there is no standard body of IoT security regulations for manufacturers to adhere to; consequently, every business must decide independently what controls to employ, without a common security baseline. It is important for researchers and security analysts to continually be up to date with reported and potential security risks a platform could have. It is also important to provide proper categorization for the vulnerabilities and provide scenarios of how these vulnerabilities can be exploited and appropriately mitigated. To tackle this issue we initially limit ourselves to consumer IoT platforms and propose the following research questions:

- *RQ1.* What are the detected vulnerabilities in a consumer IoT platform?
- *RQ2.* What are the potential vulnerabilities that can affect a consumer IoT platform?
- *RQ3.* How can we categorize such vulnerabilities and how do they fit in already established vulnerability categorizing frameworks?

To answer these questions, we develop a classification of software vulnerabilities for the SmartThings IoT platform as an initial step toward supporting regulatory and compliance standards. Our analysis aims to understand the main security components of this platform and the specific security features and standards required to develop a safe and secure IoT SmartHome app.

Second, we study the literature and the vulnerability incidents forums in order to build a catalogue of security vulnerabilities for IoT apps. The catalogue provides a classification scheme that details in each case the deviation from standards, such as NIST 800-53 [20], that could lead to such software security vulnerabilities.

Third, we demonstrate each identified class of vulnerability in small examples to better understand the impact of such vulnerabilities if the system is compromised.

While previous research provides different frameworks for general vulnerability categorization in IoT systems [18] [5][4] [8], in this paper, we apply it specifically to the SmartThings and map those vulnerabilities to OWASP's Top 10 and MITRE ATT&CK. We propose a multi-stepped categorization of SmartThings vulnerabilities based on the platform's component, nature of the attack and proposed mitigation. We applied the concepts provided by Neshenko et al. [18] [5] and tailored it specifically to the platform as well as linking it with scenarios and mitigation based on the MITRE ATT&CK matrix.

We start by giving the necessary background information in Section 2. We survey the available literature regarding vulnerabilities and categorization in Section 3. We introduce a multi-step categorization system and apply it to classify the vulnerabilities found in the platform in Section 4. We provide a detailed look into the vulnerabilities and show their prevalence in real apps in Section 5. Finally, we provide discussion and conclusion in Section 6.

2 BACKGROUND

There are many IoT smart home vendors and platforms. And while they differ on the degree of granularity and how they deal with permissions, generally, they share the key concepts. Such platforms include: Samsung's SmartThings [24], Apple's Homekit [3], openHab[22], Vera Control's vera[29], Google's Weave/Brillio [12] and Open Connectivity Foundation's AllJoyn [21].

In this paper we study and analyse one of the popular smart home IoT platforms, Smasung's SmartThings. We chose to focus on SmartThings because it is a relatively a mature platform with a decent user base and the availability of an apps dataset. Furthermore, it shares important principles with the other platforms, so it can be representative of the other platforms.

The SmartThings platform has three components: The Hub, the apps and the cloud back-end. Each component has its own risks and the relationship between the three can also propose some vulnerabilities that can be exploited. In SmartThings, apps are written in

SERP4IoT'22, May 19, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9332-4/22/05.
https://doi.org/10.1145/3528227.3528566

Groovy and are developed in a sandbox environment, which limits some of the functionality. The general structure of SmartThings apps can be divided into three sections: the definitions which contains information such as the app's name, the preferences which contains the information that the app requires from a user, and the events that happen after the initialization of the app. In the following subsections we're going to present some of the vulnerabilities that can happen on the application's side.

SmartThings offers a vulnerability disclosure platform [26] where users can report security vulnerabilities they find. This can show us the scope and target of the reported vulnerabilities which we use as basis for categorization. In their website they divide targets in scope to include the SmartThings Hub, the Mobile Application for iOS, the SmartThings Mobile Application for Android, the SmartThings Rest APIs and API Testing HTTP.

The description specifies the scope of the vulnerabilities to be OWASP's web and mobile Top 10 issues [23]. Mainly, non-self cross-site scripting issues, dealing with injection, authorization and authentication, remote exploitation, sensitive information leakage, bugs, REST APIs, malicious file uploads and issues with third party libraries. The Open Web Application Security Project (OWASP) is a non-profit organization that provides multiple projects that are helpful in the field of web security. One of their projects is the OWASP Top Ten, which is a list of the most common vulnerabilities that are encountered. We can use OWASP's Top Ten IoT vulnerabilities in this report as bases for our categorization. In Section 4.1 we also use the vulnerabilities to show how a categorization framework can be applied.

Another useful knowledge base is the Common Vulnerabilities and Exposures (CVE) which is a system that provides a reference for publicly known security vulnerabilities [4]. We can use CVE to see real life documented examples of the vulnerabilities and attacks and try to organize and analyze them. We can look for platform specific vulnerabilities or use vulnerabilities from other platforms as bases to derive potential vulnerabilities on the platform under analysis.

Finally, we look at MITRE ATT&CK [16], which provides a knowledge base about common attacks tactics and techniques an adversary might use to infiltrate or compromise a system. It also documents possible mitigation techniques used for each attack technique. We can use ATT&CK to build attack vectors and scenarios as well as using the potential mitigation as bases for categorization. The ATT&CK matrix provide the following list of techniques an attacker goes through to exploit a mobile or an enterprises system as well as suggested mitigation for them: Initial Access, Execution, Persistence, Privilege Escalation, Defense Evasion, Credential Access, Discovery, Lateral Movement, Collection, Command and Control, Exfiltration, Impact, Network Effects, Remote Service Effects.

3 LITERATURE REVIEW

IoT is relatively a new field, so security research in the field is still emerging. Research specific to SmartThings such as the one proposed by Fernandes et al. [10] provide the first effort investigating the programming framework aspects of SmartApps. It attempts to detect vulnerabilities in smart home applications and study what those vulnerabilities entail. The scope of their research does not deal with security problems inherent in Groovy or the OS and the key issues they try to analyze were: the over-privilege aspect of SmartThings IoT apps, whether sensitive data is protected, the safety of third party integration, external input sensitization, and the unsafe use of access-control APIs.

Other research such as those developed by Celik et al. [6], Schmeidl et al. [25], Naeem et al. [17] focus on specific vulnerabilities and issues in the platform, specifically, the data leakage through tainted flows. While Wang et al. [30] looked at abuse prevention, providing useful audit logs in SmartThings Apps.

Multiple research teams investigated vulnerabilities concerned with privilege escalation (over-privilege). Fernandes *et al.* [10] exposed over-privilege vulnerabilities in the SmartThings platform and how they map to design flaws in the system. Proof-of-concept attacks were performed to strengthen the research results. FlowFence [11] was developed to control the access to resources using data flow control methods. ContexIot [13] is an approach that automatically infers context in IoT apps in order to provide an informed permissions control system. Zhang *et al.* developed HoMonit [31], an over-privilege detection system for the SmartThings platform. HoMonit leverages side-channel analysis to detect over-privilege by comparing the source code to the app UI. Abu Zaid et al. developed ChYP [1], a static analysis tool that detects different over-privilege scenarios in SmartThings apps.

Tian *et al.* presented SmartAuth [27], a system that patches over-privilege vulnerabilities in SmartThings apps. SmartAuth generates new authorization screens based on the app's descriptions and source code. Kafle *et al.* [14] demonstrate how to perform privilege escalation in Google's Nest platform and advice that smart-home platforms be secure by design. Jia *et al.* [13] propose an access control for the IoT that is based on context, is fine-grained and provides run-time access control.

Luo *et al.* [15] developed ALTA, a privacy leakage analysis tool. ALTA acquires private information from apps by tracking running apps and extracting fingerprints that are then compared with fingerprints from statically analysing the apps code and descriptions.

There were some interest in validating safety and security properties for IoT apps, for instance, Alhanahnah developed IoTCom *et al.* [2], a tool that verifies safety-security properties by analysing interactions between SmartThings apps. Celik et al. developed SOTERIA [7], an approach that uses SAINT taint models [6] for checking general-and app-specific safety-security properties in vertical SmartThings IoT apps. Nguyen *et al.* developed IoTSan [19], a model checking system that detects safety violations in IoT systems.

In 2018, multiple vulnerabilities were discovered on the SmartThings platform by Claudio Bozzato of Cisco Talos [9]. Notable attack chains include Post- and pre-auth remote execution as well as remote information leakage. Some of the attack vectors show that with an OAuth bearer token one can talk to the remote SmartThings servers as an authenticated user. It can use a SmartApp which makes unknown hardware communicate with the hub and instruct the hub to make network connection and app communicates with local host bound services. This enables remote code execution and information leakage without authentication. Furthermore they can impersonate the remote server and talk to a process in the hub and in turn allows exploiting any bugs. A full listing of vulnerabilities is

[2]

Table 1: Previous Literature and Report Categorization

Vulnerabilities in Literature	Impact	Vulnerability Class	Platform Component	Refrences
Overprivlige	Confidentiality Integrity Availability	Access Control	App Side	Fernandes et al. [10, 11] Zhang *et al.* [31] Tian *et al.* [27] Kafle *et al.* [14] Jia *et al.* [13]
Sensitive Information Leakage	Confidentiality	Weak Prog. Practices	App Side	Celik et al. [6] Schmeidl et al. [25] Naeem et al. [17] Luo *et al.* [15]
Violating Safety Properties	Availability	Weak Prog. Practices	App Side	Celik et al. [7] Alhanahnah *et al.* [2] Nguyen *et al.* [19]
Inadequate Logging	Integrity	Audit Mechanism	App Side	Wang et al. [30]
Hub Firmware Vulnerabilities	Confidentiality Integrity	Authentication Patch Management	Hub Side	Talos [9]

provided by Cisco Talos [9] and a detailed discussion of the vulnerabilities is provided in Section 5.4. There has been some research in the categorizing and surveying IoT vulnerabilities. Elias Bou-Harb and Nataliia Neshenko [18] [5] divided the vulnerabilities based on different scopes. Either on layers, security impact, the attacks, countermeasures, and the situational awareness capabilities.

The layers would classify the vulnerabilities on whether they are device-based, network-based or software-based. The security impact classification would deal on whether the vulnerabilities affect confidentiality, integrity, availability or accountability. The attack classification deals with how the vulnerabilities could be exploited with attacks against confidentiality, data integrity or availability. The countermeasures classification can sort the vulnerabilities based on the techniques that identify the IoT vulnerabilities, such as using access controls, software assurance or security protocols. Finally, the situation awareness capabilities deals with the techniques that tries to gather information about the vulnerabilities, such as vulnerability assessment, honeypots, network discovery and intrusion detection. Furthermore in their survey they provide nine classes of vulnerabilities that IoT security research give. The classes are the following: Physical Security, Energy Harvesting, Authentication, Encryption, Open Ports, Access Control, Patch Management, Weak Programming Practices, and Auditing Mechanism.

Chen et al. [8] tackles the issue of automation in the classification process. They note that previous approaches often provide overlap and requires manual checking. They propose using machine learning techniques such as support vector machine (SVM) algorithms. The dataset used was focused on software side vulnerabilities.

Blinowski and Piotrowski[4] similarly provide IoT vulnerabilities classification by using machine learning techniques to categorize vulnerabilities, but tried to cover vulnerabilities beyond the software and includes the impacted equipment. The researchers used a set based on the CVE database and give seven classes are based on the applications of the IoT devices, which are: Home and SOHO devices; SCADA, Industrial systems and non-home IoT devices; Enterprise and service provider hardware; Mobile phones and tablets; PCs and laptops; and Other.

These classes focus on the targeted platform and are more relevant when studying general IoT vulnerabilities. With a specific platform like SmartThings, we know that we are dealing with a smart home platform, making the other classifications less relevant. In this paper we combine the findings and approaches proposed by previous research and apply them specifically to the SmartThings platform and map them to OWASP's top 10 vulnerabilities and MITRE ATT&CK.

Table 2: OWASP IoT Top 10 Vulnerabilities Categorization

OWASP Top 10	Layer	Impact	Vulnerability Class
Weak, Guessable, or Hardcoded Password	Device Software	Confidentiality Integrity Availability	Authentication
Insecure Network Services Authorization	Network	Integrity Availability	Encryption Open Ports Access Control
Insecure Ecosystem Interfaces	Network	Confidentiality	Access Control Encryption
Lack of Secure Update Mechanism	Software	Availability	Improper Patch Management
Use of Insecure or Outdated Component	Software	Confidentiality Integrity	Weak Programming Practices
Insufficient Privacy Protection	Software Network	Confidentiality	Weak Programming Encryption Access Control
Insecure Data Transfer and Storage	Network Device	Confidentiality	Encryption Open Ports
Lack of Device Management	Software Device	Integrity Availability	Access Control Patch Management
Insecure Default Settings	Software	Confidentiality Integrity	Open Ports Weak Programming
Lack of Physical Hardening	Device	Integrity Availability	Physical Security Energy Harvesting

4 POTENTIAL VULNERABILITIES

Previous research tackled different issues in SmartThings apps and gives us a list of general and app-specific properties that can affect the security. We can also examine a real case example of the hub firmware having potentially exploitable bugs in the TALOS report. We can use these findings along with OWASP's Top 10 IoT vulnerabilities as examples for potential vulnerabilities in the platform.

4.1 Mapping and Categorizing Potential Vulnerabilities

To map these vulnerabilities, we base our framework on Neshenko et al's work [5] [18] which provided an extensive framework for categorizing IoT vulnerabilities in general. To make the framework more specific to the SmartThings platform, we propose adding categorization based on the affected component. This is consistent with the SmartThings vulnerability disclosure program scope, where the categorization is based on the components affected in the SmartThings platform; the hub, the iOS application, the Andriond application, the APIs, and the graph console. This can complement the layers class suggested by Neshenko et al. while also providing more detail specific to SmartThings.

We also propose extending the framework to include detailed mitigation for the vulnerabilities. This can be using ATT&CK matrix which can provide more information regarding the attacks and countermeasures. By providing scenarios of each attack vector, we can construct what areas are targeted from the system and what potential countermeasures there are.

One challenge is that vulnerabilities usually can affect multiple aspects at once. This can make it difficult to classify each attack to just one category. We suggest using platform specific categorization,

Table 3: Vulnerabilities Mapping MITRE'S ATT&CK

SmartThings Vulnerability	ATT&CK Tactic	ATT&CK Technique	Mittigation
Over privilege	TA0029 – Privilege Escalation	T1405-Exploit Trusted Execution Environment (TEE) Vulnerability T1540-Code Injection T1404 - Exploit OS Vulnerability	M1005-Application Vetting M1001 - Security Updates M1006-Use Recent OS Version
Data Leakage via the Event System	TA0031-Credential Access TA0035-Collection	T1517 Access Notifications T1413 Access Sensitive Data in Device Logs T1409 Access Stored Application Data T1416 Android Intent Hijacking T1414 Capture Clipboard Data T1412 Capture SMS Messages T1405 Exploit TEE Vulnerability T1417 Input Capture T1411 Input Prompt T1579 Keychain T1410 Network Traffic Capture or Redirection T1415 URL Scheme Hijacking	M1013-Application Developer Guidance M1012-Enterprise Policy M1005-Application Vetting M1001-Security Updates M1006-Use Recent OS Version
Insecurity in Third-party Integration Unsafe use of groovy dynamic method invocation	T1389-Identify vulnerabilities in third-party software libraries		
Exposing the app through URLs or website calls API-access control	TA0037 - Command and Control	T1481 - Web Service	Cannot be easily mitigated

Table 4: Mobile Mitigation in MITRE'S ATT&CK

ID	Name	Classes of Mitigation Strategies
M1013	Application Developer Guidance	Software Assurance
M1005	Application Vetting	Software Assurance
M1002	Attestation	Access Controls
M1007	Caution with Device Administrator Access	Access Controls
M1010	Deploy Compromised Device Detection Method	Security Protocols
M1009	Encrypt Network Traffic	Security Protocols
M1012	Enterprise Policy	Security Protocols
M1014	Interconnection Filtering	Access Controls
M1003	Lock Bootloader	Access Controls
M1001	Security Updates	Software Assurance
M1004	System Partition Integrity	Security Protocols
M1006	Use Recent OS Version	Security Protocols
M1011	User Guidance	Software Assurance

and in the case of the SmartThings platform, we can use a hierarchy of classes starting with what component and the layer of the platform that is affected. Then, a second classification would be based on the nature of the attack and its impact, whether it targets confidentiality, integrity or availability. A finer-grained categorization can be added for different types of vulnerabilities as well if needed. Finally we can tag the vulnerability based on what countermeasures are suggested. Figure 1 shows the suggested classification steps. This approach can be used for currently known vulnerabilities as well as potential future ones by going through the steps described in Figure 1, the vulnerability can classified. To demonstrate this, we use the vulnerability gathered from the literature, we can categorize them and their effects. We provide Table 1 where we look into how they correspond with the classes provided and we further examine fine grain vulnerability categorization by comparing OWASP's Top 10 IoT vulnerabilities and the nine classes provided by the paper and how they correspond to them. We can use these as extra generic vulnerabilities for testing and a more detailed categorization as seen in Table 2. We provide a summary of the mapping in the following tables. Table 3 shows the vulnerabilities and how they map to ATT&CK scenarios.

4.2 Categorizing Potential Mitigation

In our literature review we saw multiple tools developed for detecting and reporting vulnerabilities. The next step is to determine a way to mitigate and remedy the detected vulnerabilities and a useful approach is to link the categorized vulnerabilities with a categorize mitigation.

Bou-Harb and Neshenko [5] provide potential categorizing of IoT vulnerabilities based on their remediation. They provide three main classes for mitigation strategies: Access Controls, Software Assurance, Security Protocols.

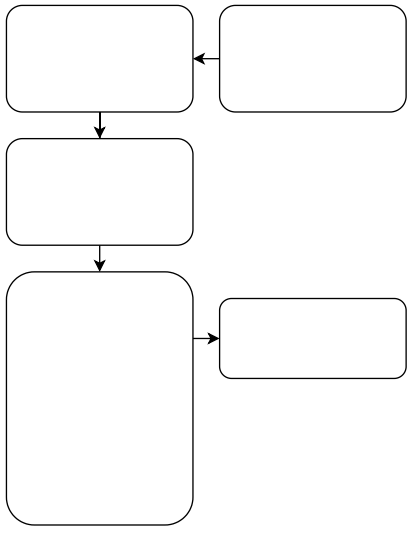

Figure 1: Vulnerability Classification Steps

4

Table 5: Enterprise Mitigation in MITRE'S ATT&CK

M1015	Active Directory Configuration	Classification
M1049	Antivirus/Antimalware	Software Assurance
M1013	Application Developer Guidance	Software Assurance
M1048	Application Isolation and Sandboxing	Security Protocols
M1047	Audit	Software Assurance
M1040	Behavior Prevention on Endpoint	Access Controls
M1046	Boot Integrity	Access Controls
M1045	Code Signing	Access Controls
M1043	Credential Access Protection	Access Controls
M1053	Data Backup	Security Protocols
M1042	Disable or Remove Feature or Program	Security Protocols
M1055	Do Not Mitigate	Security Protocols
M1041	Encrypt Sensitive Information	Security Protocols
M1039	Environment Variable Permissions	Access Controls
M1038	Execution Prevention	Access Controls
M1050	Exploit Protection	Security Protocols
M1037	Filter Network Traffic	Security Protocols
M1035	Limit Access to Resource Over Network	Access Controls
M1034	Limit Hardware Installation	Access Controls
M1033	Limit Software Installation	Access Controls
M1032	Multi-factor Authentication	Access Controls
M1031	Network Intrusion Prevention	Access Controls
M1030	Network Segmentation	Security Protocols
M1028	Operating System Configuration	Software Assurance
M1027	Password Policies	Access Controls
M1056	Pre-compromise	Software Assurance
M1026	Privileged Account Management	Access Controls
M1025	Privileged Process Integrity	Security Protocols
M1029	Remote Data Storage	Software Assurance
M1022	Restrict File and Directory Permissions	Access Controls
M1044	Restrict Library Loading	Access Controls
M1024	Restrict Registry Permissions	Access Controls
M1021	Restrict Web-Based Content	Access Controls
M1054	Software Configuration	Security Protocols
M1020	SSL/TLS Inspection	Software Assurance
M1019	Threat Intelligence Program	Software Assurance
M1051	Update Software	Security Protocols
M1052	User Account Control	Access Controls
M1018	User Account Management	Access Controls
M1017	User Training	Security Protocols
M1016	Vulnerability Scanning	Software Assurance

The MITRE ATT&CK also provides a list of mitigation's, provided in Tables 4 and 5. Each row provides a general mitigation that can be used and under each mitigation class them there are subcategories of techniques that would be used with each venerability. We add a column for classifying these mitigation techniques based on the three main classes mentioned before. The papers discussed that provided tools and techniques for detecting vulnerabilities fall under the software assurance categorization. Table 6 shows the contribution of each paper and report in terms of mitigation. We use the MITRE ATT&CK to show what type of contribution it offers as well as classifying them based on the three main categories provided before. In Table 7 we try to associate the top 10 vulnerabilities with their main mitigation techniques. It should be noted that these vulnerabilities are general and can happen at different levels of the attack, which then requires a combination of multiple mitigation techniques. In the following tables we showcase the main ones.

Table 6: Categorizing the Reviewed Papers and Reports Based on the Mitigation

Reference	ATT&CK Mitigation Classification	Classes of Mitigation Strategies
Fernandes et al. [10]	Application Vetting Application Developer Guidance	Software Assurance
Tian et al. [28]	Application Vetting User Guidance	Software Assurance
Celik et al. [6]	Application Vetting	Software Assurance
Celik et al. [7]	Application Vetting	Software Assurance
Wang et al. [30]	Application Developer Guidance Application Vetting	Security Assurance
Talos. [9]	Security Updates Use Recent OS Version	Software Assurance Security Protocols
Kafle et al. [14]	Application Developer Guidance Application Vetting	Security Assurance
Jia et al. [13]	Caution with Device Admin Access	Access Controls
Luo et al. [15]	Application Vetting	Software Assurance
Alhanahnah. [2]	Application Vetting	Software Assurance
Nguyenet al. [19]	Application Vetting	Software Assurance

Table 7: OWASP IoT Top 10 Vulnerabilities Mitigation

OWASP Top 10	ATT&CK Mitigation
Weak, Guessable, or Hardcoded Password	Application Developer Guidance User Guidance, Application Vetting
Insecure Network Services Authorization	Caution with Device Administrator Access
Insecure Ecosystem Interfaces	Encrypt Network Traffic Interconnection Filtering
Lack of Secure Update Mechanism	Security Updates Use Recent OS Version
Use of Insecure or Outdated Component	Application Developer Guidance Security Updates Use Recent OS Version
Insufficient Privacy Protection	Encrypt Network Traffic User Guidance
Insecure Data Transfer and Storage	Encrypt Network Traffic
Lack of Device Management	Application Developer Guidance Application Vetting
Insecure Default Settings	Application Developer Guidance
Lack of Physical Hardening	Limit Hardware Installation

5 DETAILED LOOK INTO SOME OF THE VULNERABILITIES

5.1 Over-privilege

Listing 5.1 shows an example of a non compliant code with which proposes an overprivilege vulnerability. The description of the app did not include anything about the lock device.

Listing 5.1: Example of Non-Compliant Code with Overprivilege Vulnerability

```
preferences {
   section ("Alert me when the alarm goes off") {
      input "theAlarm", "capability.alarm",
      input "theLock", "capability.lock"}}
```

While listing 5.2 shows an example of how the application can be compliant and avoids the vulnerability. The application installed by the user will display the description "Alert me when the alarm goes off". It indicates the use of the Alarm device, which is what should be requested by the app and not another type of device

Listing 5.2: Example of Compliant Code- Overprivilege

```
preferences {
   section ("Alert me when the alarm goes off") {
      input "theAlarm", "capability.alarm"}}
```

5

Looking at this issue, we can see that it falls under the application side and proposes I2 and I8 risks of OWASP. We can see It is used in Privilege Escalation through techniques such as TEE exploits, code injection and OS vulnerability exploits. Mitigation suggested for such vulnerability includes application vetting, security updates and using recent OS versions.

It is also important that access to resources is validated and described by the software. Listing 5.3 shows an example of a compliant code from an alarm app.

Listing 5.3: Example of Compliant Code for Validating Access

```
preferences {
   section ("Alert me when the alarm goes off") {
      input "theAlarm", "capability.alarm"} }
...
def statusChanged(evt) {
   thaAlarm.siren() }
```

The value of the alarm is checked whether it is "siren" is compliant with requesting access to the alarm device. While Listing 5.4 shows a non-compliant code where actuating the "unlock" command is not compliant with the permission requested which is the alarm device and not the lock.

Listing 5.4: Example of Non-compliant Code for Validating Access

```
preferences {
   section ("Alert me when the alarm goes off") {
      input "theAlarm", "capability.alarm"}}
...
def statusChanged(evt) {
   thaAlarm.unlock()
   // assuming the device id theAlarm is an id for a physical
        device with alarm and lock capabilities}
```

5.2 Tainted Dataflows and Potential Sensitive Information Leakage

Another issue that happens on the application side is the issue of tainted data flows. We define tainted data flows as the movement of potential sensitive data within the app from variables the containing the sensitive information, called sources, to functions and areas of the code that can leak them, called sinks. The risk happens if the app is run through an insecure channel, where the data can be compromised. This can happen either by an attacker intentionally writing a malicious program or through carelessness of the programmer. Furthermore, programs might not provide adequate description and conceal their functionality where the user's private information might be leaked.

An example of a non-compliant code is given at List 5.5.

Listing 5.5: Example of Non-compliant Code with a Tainted Dataflow

```
preferences {
   section("Tainted input") {
      input "sensitiveData", "number", required: true, title: "
           Input?" }}
def installed() {
   def message = "This contains $sensitiveData"
   sendSms(message)}
```

In this code it shows that some sensitive data is taken as user input, it is then used through the program, where it is used in a variable *message* which then gets sent through SMS. This shows a tainted flow and a pattern where sensitive data can leak. An example of a compliant code would avoid tainting the flow by either avoiding sinks as shows in Listing 5.6 or by sanitizing the flow at one point, as shown in Listing 5.7.

Listing 5.6: Example of Compliant Code with a Dataflow- No sink used

```
preferences {
   section("Tainted input") {
      input "sensitiveData", "number", required: true, title: "
           Input?"}}
def installed() {
   def message = "This contains $sensitiveData"
   //sendSms(message) Sink was commented}
```

Listing 5.7: Example of Compliant Code with a Sanitized Dataflow

```
preferences {
   section("Tainted input") {
      input "sensitiveData", "number", required: true, title: "
           Input?" }}
def installed() {
   def message = "This contains $sensitiveData"
   message = sanitize();
   sendSms(message)}
```

5.3 Violating Security and Safety Properties

Celik et al. [7] gathered general and app- specific properties that should be followed to maintain the expected safe and secure behavior of an IoT app, labeled *S1-S6*. In this section we will overview them and provide examples of their applications.

At the app level, property *S1* assesses that a device attribute is not changed to conflicting values by the event handler. For instance, the expected behavior of a smoke detector is to turn on when smoke is detected and turn off when it is not. It would be a violation of this property if the devices turns off the alarm whenever it turns on. The following listing, based on examples in the IoTBench Suite [7], shows a compliant app that behaves as expected.

Listing 5.8: Example of a Compliant App

```
def initialize() {
 subscribe(smoke, "smoke", smokeHandler)
}
def smokeHandler(evt) {
 if(evt.value == "detected") {
  alarm.on()
  mySwitch?.on()
 }else if(evt.value == "clear") {
  alarm.off() }}
```

A non compliant version of this code would assign a negating attribute to the code which would break the functionality of the app. App vetting tools can be used to assure compliance of the code to such properties by generating a state model of the app and assuring that no conflicting attributes are assigned on the same conditional path.

Listing 5.9: Example of an Non-compliant App to S1 Prop.

```
def initialize() {
 subscribe(smoke, "smoke", smokeHandler)}
def smokeHandler(evt) {
 if(evt.value == "detected") {
  alarm.on()
      alarm.off()
 }else if(evt.value == "clear") {
  alarm.off()}}
```

6

According to property *S2*, an event handler should not assign the same value to an attribute multiple times on the same execution path. A non compliant example is given in Listing 5.10.

Listing 5.10: Example of an Non-compliant App to S2 Prop.

```
def initialize() {
 subscribe(smoke, "smoke", smokeHandler)}
def smokeHandler(evt) {
 if(evt.value == "detected") {
    alarm.on()
    \\action should not repeat on the same event
      alarm.on()
 }else if(evt.value == "clear") {
  alarm.off()}}
```

For property *S3* the event handler should not change an attribute to the same value on a complementing events. Listing 5.11 illustrates a non-compliant code.

Listing 5.11: Example of an Non-compliant App to the S3 Property

```
def initialize() {
 subscribe(smoke, "smoke", smokeHandler)}
def smokeHandler(evt) {
 if(evt.value == "detected") {
  alarm.on()
 }else if(evt.value == "clear") {
      alarm.on()}}
```

Property *S4* states that event handlers that are non-complement should not assign conflicting values to an attribute. Listing 5.12 shows a non compliant example where two event handlers, the smoke handler turns on the alarm when smoke is detected and a timer event handler disables it at midnight, which results into contradicting values in the possible case of smoke being detected at midnight.

Listing 5.12: Example of an Non-compliant App to S4 Prop.

```
def initialize() {
 subscribe(smoke, "smoke", smokeHandler)
   subscribe(time, "time", timerHandler)}
def smokeHandler(evt) {
  if(evt.value == "detected") {
    alarm.on()
  }else if(evt.value == "clear") {
     alarm.off()}}
def timerHandler(evt) {
  if(evt.value == "midnight") {
     alarm.off() }
  else {
     alarm.on()}}
```

Listing 5.13: Example of an Non-compliant App to S5 Prop.

```
def initialize() {
   //incorrect subscription
 subscribe(smoke, "smoke", wrongHandler)}
def smokeHandler(evt) {
 if(evt.value == "detected") {
    alarm.on()}
 else if(evt.value == "clear") {
      alarm.off()}}
```

Property *S5* states that events must be subscribed to an event handler with the proper logic to handle it. A non compliant code for example might not properly subscribed the events to the handler. Listing 5.13 illustrates this case where the app doesn't subscribe the smoke event to the handler that has the logic to handle it.

5.4 Firmware Vulnerabilities

In the previous sections, we've looked into the TALOS report [9] as an example of publicized vulnerabilities that existed in the SmartThings platform. The vulnerabilities were mainly discovered in the Hub firmware, nonetheless, they could be used to construct attack vectors that would compromise the whole system and allow the attacker to fully exploit it.

Listing 5.14: Example of ffmpeg Code Execution

```
machineA -- $echo talos | nc -l -p 3333
machineB -- $ffmpeg -f data -i tcp://@<machineA>:3333 -map 0 -
      codec copy -y -f data /tmp/test
```

5.4.1 Code Execution. For example, if we look into the reported Real Time Streaming Protocol (RTSP) configuration and how its inadequate handling of the URL fields can be used for command injections by sending HTTP requests. This happens because the *video-core* process in the hub which uses *ffmpeg* for handling the video has the capability to execute shell commands.

An example provided by the report in the following listing shows how one can call pass a string such as 'talos' to ffmpeg which would write it into a */tmp/test* directory.

For this vulnerability, one scenario, if an attacker has an OAuth token, which would allow him to configure the camera, can send a change password request, which would return a message containing a url with arbitrary code to the hub processes including the *video-core* process. Another scenario is if the attacker manages to mimic the backend SmartThings servers, then they'll be able to send such requests to the hub. Finally, one can make custom apps which can establish a localhost connection with the hub and send such HTTP requests to the process.

To solve such vulnerability it is important to neutralize the argument delimiters. Messages containing URLs should not be sent directly to a process that can execute them. Furthermore, for the different scenarios, application vetting techniques can be used to assure that devoloped apps are not exploiting such vulnerability. And since this is a firmware level vulnerability, it is important to update the system and assure that the devices are using latest version.

5.4.2 Overflow. Another example is how video-core can mishandle JSON payloads which could be used to achieve buffer overflows. If a POST request is made to a path "/samsungWifiScan", json-c library is used to manage JSON objects and send them as parameters to be handled by other functions which put them on a buffer. The vulnerability happens because the length of JSON fields are not checked when copied to the buffer stack. This can be exploited to overflow the buffer. This buffer overflow can also happen by accessing other functionalities in video-core dependant on the json-c library such as requests to the "/cameras" and "/credentials" paths. Furthermore, the mishandling of JSON payloads can be used to make SQL injection attacks where the database is also accessed. This happens because the JSON values are not sanitized, and when they are called for example to update the credentials database, a SQL injection command can be sent through its values.

Similarly, a buffer overflow could be achieved due to video-core's handling of the smart camera callbacks. This can be done by sending a command to video-core to check on a smart camera using a callback. The camera's response is then passed as a parameter to function which generates a POST request using it. Because the response is not checked, this can be used to overflow the buffer. Another buffer overflow could happen in how it handles data, such

as livestreams' data, clips information or cloud server connection settings, that is stored in the SQLite database. A field could be overridden and copied to the buffer, and since they have no restrictions, this can lead to a buffer overflow. An additional overflow example could happen by handling cookies which are unrestricted in length.

Attack vectors include that if somebody impersonates the servers, he can send HTTP requests to the cores without them being modified and achieve the overflow or the SQL injection attack. Another is that a custom SmartThings app to send the HTTP request. A combination of checking the JSON value's or the camera's response length and sanitizing it for possible injection attacks on the firmware level and software vetting can be used for such vulnerability.

5.4.3 Other vulnerabilities. Other examples of vulnerabilities reported include video-core's handling of pipelined HTTP requests, which can be lead to the overriding and interfering with other requests. An integer underflow could also happen where the hubCore handles corrupted files leading to infinite loop. hubCore also sends crash reports through an insecure connection which can lead to information disclosure. Finally, the remote servers were also reported to be vulnerable due to mishandling JSON messages or parameters in the sync operations that can be triggered through video-core.

6 CONCLUSIONS

While the field of IoT is getting popular, the research is still scarce and the lack of standardization poses a challenge when it comes to the security of the platforms. Analyzing IoT is harder than other platforms due to the complexity and the multi dimensional nature of the system. While previous research provides different frameworks for general vulnerability categorization in IoT systems, in this paper, we apply it specifically to the SmartThings and mapped those vulnerabilities to OWASP's Top 10 and MITRE ATT&CK. We propose a multi-stepped categorization of SmartThings vulnerabilities based on the platform's component, nature of the attack and proposed mitigation. We applied the concepts provided by Neshenko et al. [18] [5] and tailored it specifically to the platform as well as linking it with scenarios and mitigations based on the MITRE ATT&CK matrix.

We saw good success in categorizing the vulnerabilities but some challenges still remain nonetheless, such as the issue of automating the process and standardization. There is also some overlap and intersection when trying to categorize the vulnerabilities. We anticipate that our analysis will help us in encoding the identified classes of vulnerabilities in an abstract form that can form the basis of an automated framework for the identification of such vulnerabilities in existing SmartHome apps. The vulnerability classification can be used as a set of anti-patterns that can be used to educate developers on coding practices that they should avoid, otherwise, the entire IoT ecosystem will be exposed to costly security breaches that not only may cause business loss but also may hinder user privacy and safety.

REFERENCES

[1] Atheer Abu Zaid, Manar H. Alalfi, and Ali Miri. 2019. Automated Identification of Over-Privileged SmartThings Apps. In *(ICSME)*. 247–251.

[2] Mohannad Alhanahnah. 2019. Advanced Security Analysis for Emergent Software Platforms. (2019).

[3] Apple Last accessed: May 31, 2021. Apple HomeKit. https://www.apple.com/ios/home/

[4] Grzegorz J. Blinowski and Pawel Piotrowski. 2020. CVE based classification of vulnerable IoT systems. *CoRR* abs/2006.16640 (2020). arXiv:2006.16640

[5] Elias Bou-Harb and Nataliia Neshenko. 2020. Taxonomy of IoT Vulnerabilities. *Cyber Threat Intelligence for the Internet of Things* (2020), pp.7–58.

[6] Z. Berkay Celik, Leonardo Babun, Amit K. Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. 2018. Sensitive Information Tracking in Commodity IoT. *(USENIX Security 18)* (2018), pp.1687–1704.

[7] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. In *(USENIX ATC)*.

[8] Haibo Chen, Dalin Zhang, Jinfu Chen, Wei Lin, Dengzhou Shi, and Zian Zhao. 2020. An Automatic Vulnerability Classification System for IoT Softwares. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 1525–1529.

[9] Cisco Talos Intelligence Last accessed: May 31, 2021. Vulnerability Spotlight: Multiple Vulnerabilities in Samsung SmartThings Hub. https://blog.talosintelligence.com/2018/07/samsung-smartthings-vulns.html

[10] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. May 2016. Security Analysis of Emerging Smart Home Applications. *IEEE Symp. Secure. Privacy pp. 636-654* (May 2016).

[11] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *25th USENIX*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 531–548.

[12] Google Last accessed: May 31, 2021. Google's Weave/Brillo. https://developers.google.com/weave/

[13] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Zhuoqing Morley Mao, Atul Prakash, and SJ Unviersity. 2017. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms.. In *NDSS*, Vol. 2. 2–2.

[14] Kaushal Kafle, Kevin Moran, Sunil Manandhar, Adwait Nadkarni, and Denys Poshyvanyk. 2019. A Study of Data Store-Based Home Automation. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*. Association for Computing Machinery, New York, NY, USA, 73–84.

[15] Yuan Luo, Long Cheng, Hongxin Hu, Guojun Peng, and Danfeng Yao. 2021. Context-Rich Privacy Leakage Analysis Through Inferring Apps in Smart Home IoT. *IEEE Internet of Things Journal* 8, 4 (2021), 2736–2750.

[16] MITRE ATT&CK Last accessed: May 31, 2021. MITRE ATT&CK. https://attack.mitre.org/

[17] Hajra Naeem and Manar H. Alalfi. 2020. Identifying Vulnerable IoT Applications using Deep Learning. In *(SANER)*. 582–586.

[18] Nataliia Neshenko, Elias Bou-Harb, Jorge Crichigno, Georges Kaddoum, and Nasir Ghani. 2019. Demystifying IoT Security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-scale IoT Exploitations. *IEEE Communications Surveys & Tutorials* (2019).

[19] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V. Krishnamurthy, Edward J. M. Colbert, and Patrick McDaniel. 2018. IotSan: Fortifying the Safety of IoT Systems. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (Heraklion, Greece) *(CoNEXT '18)*. 191–203.

[20] NIST SP 800-53 Last accessed: May 31, 2021. NIST. https://csrc.nist.gov/publications/detail/sp/800-53/rev-5/final

[21] Open Connectivity Foundation Last accessed: May 31, 2021. Open Connectivity Foundation's AllJoyn. https://openconnectivity.org/

[22] openHab Last accessed: May 31, 2021. openHab. https://www.openhab.org

[23] OWASP Last accessed: May 31, 2021. OWASP Top 10 Internet of Things 2018. https://owasp.org/www-pdf-archive/OWASP-IoT-Top-10-2018-final.pdf

[24] Samsung Last accessed: May 31, 2021. Samsung SmartThings. https://www.smartthings.com

[25] Florian Schmeidl, Bara' Nazzal, , and Manar Alalfi. May 2019. Security Analysis for SmartThings IoT Applications. *(MOBILESoft)* (May 2019).

[26] SmartThingsVulnerabilityDisclosure Last accessed: May 2, 2021. SmartThings Vulnerability Disclosure. https://bugcrowd.com/smartthings

[27] Yuan Tian, Nan Zhang, Yue-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. 2017. SmartAuth: User-Centered Authorization for the Internet of Things. In *26th USENIX*. 361–378.

[28] Yuan Tian, Nan Zhang, Yue-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. 2017. SmartAuth: User-Centered Authorization for the Internet of Things. *USENIX Security Symposium* (2017), pp.361–378.

[29] Vera Control Last accessed: May 31, 2021. Vera Control's Vera3. https://getvera.com/

[30] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. 2018. Fear and-Logging in the Internet of Things. *(NDSS)* (2018).

[31] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. 2018. Homonit: Monitoring smart home apps from encrypted traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1074–1088.

Building blocks for IoT testing - a benchmark of IoT apps and a functional testing framework

Rareș Cristea
rares.cristea@unibuc.ro
University of Bucharest
București, Romania

Mihail Feraru
mihail.feraru@s.unibuc.ro
University of Bucharest
București, Romania

Ciprian Paduraru
ciprian.paduraru@fmi.unibuc.ro
University of Bucharest
București, Romania

ABSTRACT

IoT security is a topic that offers numerous opportunities for improvement and development. In this paper, we first present a set of open-source mock IoT applications along with the necessary infrastructure specifically designed to emulate a real IoT system. With our app set, users can add their own applications, automation rules, and communication flows with little technical effort, and test different scenarios to reproduce bugs that are not specific to the use of a single device. Second, we describe a functional testing framework for the IoT that is inspired by behavior-driven development (BDD), a testing methodology that serves as a proof-of-concept for how the application set can be used in different test scenarios. The application set and the functional testing framework are independent of each other. Our goal is to help IoT developers and testers find new testing techniques and benchmarking them in a reproducible, comparable, and less biased environment. We believe that they form the basis for a better understanding of how to test systems composed of heterogeneous devices to find issues and vulnerabilities that arise mainly from their interaction and data persistence management.

ACM Reference Format:
Rareș Cristea, Mihail Feraru, and Ciprian Paduraru. 2022. Building blocks for IoT testing - a benchmark of IoT apps and a functional testing framework. In *4th International Workshop on Software Engineering Research and Practice for the IoT (SERP4IoT'22), May 19, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3528227.3528568

1 INTRODUCTION

The Internet of Things (IoT) is an ever-growing field today. Examples such as smart cities connecting transportation networks, intelligent car systems, and smart home systems are more present in our lives than ever. Sensors, actuators, end-user applications, gateways, and servers are interconnected through various communication protocols and methods without us even realising it. It is common for an IoT system to be composed of devices from different vendors, resulting in a unique combination of technologies that interact in unpredictable ways. However, as one would expect, the rapid development and deployment cycles lead to major challenges, including security and privacy issues. Problems can occur at multiple levels, from single isolated applications to networked application streams exposed to various risks, such as Distributed Denial-of-Service (DDoS) [1] , identity management and user security issues [20], etc. After reviewing the literature and finding that there is a lack of common ground for evaluating and researching testing approaches, we add the following contributions in this paper from the authors' knowledge:

- The first open-source set of IoT-specific software applications that serves as a benchmark for the community to test different testing methods and compare them efficiently by providing a reproducible environment, a less biased comparison of results, and a simpler process for evaluating the effectiveness of a particular tool or technique. In addition to this suite of applications, we also provide tools to automate the development process and communication infrastructure so that users can focus on the methods and algorithms rather than the technical work. We hope that facilitating and standardizing these processes will increase interest in and accelerate the advancement of testing methods in the IoT space.
- The first open source framework that serves as a basic method for functional testing in the IoT domain. We consider that the proposed system can be easily extended by the user, with the possibility of adding their own set of applications and customized communication flows between them. We also provide the tools needed to automate issue detection. In our framework, the concept of *issue* can be understood as a template. Examples include exploit detection in the security domain, performance issues, or classic source code bugs. The user of the framework can be either a software developer or a quality assurance professional (tester).

As a side story, the applications in our application set were initially developed as part of an undergraduate course in Software Engineering at the University of Bucharest. The goal of the course is to teach students the general practices of software engineering in the field of IoT. The students have to define a free-form software development project, evaluating the topics presented in the lectures of the course: Application Analysis, Architecture, Security, Agile Practices, etc. In the 2020-21 edition of the course, students were required to create a software project that met the criteria of vision for our application set. As the literature suggests, providing students with a real-world use case for their work projects motivates them and leads to a better software development process and final product [15]. Since the applications are developed by multiple independent teams, they can better represent the IoT system scenario: Devices from different vendors are integrated into complex systems that interact in unpredictable ways and present new testing challenges. In the following Section, we review the existing literature on testing

SERP4IoT'22, May 19, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9332-4/22/05…$15.00
https://doi.org/10.1145/3528227.3528568

heterogeneous IoT systems, current challenges, proposed solutions, and arguments that support the topic of our paper. A technical description of our initial set of applications, test infrastructure, and user guidance on extending the existing set with new or different applications is presented in Section 3. Then, Section 4 describes the testing framework we propose for functional testing in the IoT domain. Evaluation of our approaches is presented in Section 5, along with a discussion of their strengths and weaknesses. Finally, in Section 6 Conclusions and future work are presented.

2 RELATED WORK

In [19], the authors noted that new methods of security assessment in IoT could benefit from the existence of a set of applications on which to test new tools. The goal is to provide a benchmark that can be used to detect known and new bugs and compare the performance of the detection with other similar tools. A similar tool is FuzzBench [16], which is "on its way to becoming a standard benchmarking platform for fuzzers." Fuzzbench uses the OSS-Fuzz applications as targets against which it compares the fuzzing tool under test. Due to the heterogeneity of the IoT landscape, to the authors' knowledge, there is no repository of IoT applications (open source or not).

IoT systems involve interactions between multiple communication protocols, devices, and software from different vendors. We accept that such systems are highly heterogeneous, a feature that increases testing complexity. In their meta-analysis, [10] concludes that the majority of industry testers of heterogeneous systems prefer manual exploratory testing, which is highly dependent on the tester's skills and can hardly be evaluated objectively.

The lack of methodology and automated tools for testing heterogeneous IoT systems is reinforced by [7]. Their analysis of available tools and platforms for IoT testing shows that most of these tools are tied to a specific vendor and support only a limited number of devices or protocols. In addition, almost all of the tools use a simulated environment rather than an emulated or real-world environment, making testing easier to set up and less expensive, but reducing the accuracy of the process. Other tools are only available on remote test runners, which impacts data protection and the ability to test certain custom conditions.

The need for a common set of applications that serve as benchmarks and playgrounds for testing techniques is emerging in other areas as well. For example, [9] highlights that almost every paper uses a different set of web applications in its analysis. This affects the reliability of comparisons between approaches because the starting point is different. [9] concludes that "having a set of subject systems that every-one can use for rigorous controlled experimentation is needed" containing not just applications, but a complete architecture of software, test runners, and faulty data. FIRMCORN [11] has focused on streamlining the testing process by using a mix of emulating the firmware and using the real device. However, the devices used are only routers and webcams, and the testing methodology used only one device at a time, rather than a more realistic scenario - as part of a larger IoT system.

An approach to functional testing similar to our own has been explored by [4] in collaboration with an industry partner. Their framework, PatrIoT, aims to provide a foundation for building integration test environments for IoT systems that allow real hardware to be mixed with simulated devices. To benchmark their approach, they have developed a set of pilot applications that are not readily reproducible, highlighting the need for a common set of benchmarks for researchers.

Private companies are also interested in exploring the challenges of heterogeneous, interconnected devices. A study published by TrendMicro, [12], assesses the security risks of two smart homes they call "complex IoT environments." They show how networking multiple IoT devices and orchestrating them through an automation server can create a plethora of security risks, vulnerabilities or malfunctions.

3 APPLICATION SET, INFRASTRUCTURE FOR DEPLOYMENT AND TESTING

In our work, we first propose a set of applications that can be used to test and benchmark different testing methods in the IoT field community. In this Section, we briefly introduce these applications, the backend component implemented to facilitate deployment and testing, and an overview for users on how to set up their own test and application suite. We consider that the implemented infrastructure can be reused for the user's own suite with minimal effort.

3.1 Description of the current application set

The application set at the moment consists of five interconnected software applications that mimic the behaviour of smart devices, which we have connected through various examples of automation workflows. Our motivation for the smart home subset of systems comes from observing the rise of proprietary products such as Google Home Assistant, Amazon Alexa, or Facebook Portal, as well as open source solutions such as openHAB or Home Assistant. Also from our previous study, [19], we concluded that smart home IoT use cases could be a main target for a first set of applications [1].

Below is a brief description of each application:

(1) *SmartFlower* - A *smart pot* application designed to help the user manage the resources of a plant.
(2) *WindWow* - A smart home application designed to facilitate the automatic and distant use of a window in an indoor space.
(3) *Smarteeth* - An application running a *smart toothbrush* that helps users track information about the health of their mouth and use appropriate toothbrushing programs.
(4) *SmartKettle* - An *smart kettle* Application that speeds up and automates the preparation of drinks.
(5) *SmartTV* - An *smart TV* application where users can have individual profiles and, based on their behavior, a recommendation system is used to recommend movies or TV shows.

[1]The repository containing the applications and the Hub can be accesed at this link https://github.com/unibuc-cs/IoT-application-set.

3.2 Communication and deployment infrastructure

Although currently the five applications are focused on smart home systems, the technologies developed and the overall deployment infrastructure are independent of the type of applications. Along this application set, we have built the necessary infrastructure to allow users and the community to customize and scale the existing set, as well as deployment scripts to easily evaluate different testing methods and algorithms.

The infrastructure built is independent of the programming languages in which the applications are developed. The current corpus includes applications written in C/C++, as we have found this to be the most commonly used programming language for open source IoT projects [6]. Further planned work will extend the set to include other applications written in other languages, such as Python. Each of the current applications in our set models a smart home device. At the communication level, they all needed to implement an HTTP-based layer that allows sending and receiving messages between applications and from external users. HTTP was chosen as the communication protocol for some of the following reasons:

(1) It is widely used and a very popular choice for a variety of applications due to its simplicity and synchronous request-response pattern.
(2) Many existing open source tools have already been developed for it and provide good support and compatibility.
(3) The OpenAPI platform [2] provides a rigorous specification standard for device communication over HTTP.

We chose to use the OpenAPI specification to describe the communication interface for all of our applications, creating a common interface between them. Our experiments have shown that encapsulating exposed functionalities under a common and strict specification simplifies the testing process, even if they are not designed to be cohesive.

At the core of a number of IoT systems, such as in the smart home domain, is the communication network-style infrastructure on top of orchestration software deployed in a physical location, usually known as a *Hub Application*. In addition to developing the use cases in our application set, we also developed such a hub application that is responsible for defining and executing automation tasks between the software and the devices. We chose to write it in Python as this allows for rapid prototyping. Using OpenAPI Generator [3] we automatically generated the client code necessary to interact with the smart applications according to the HTTP specifications.

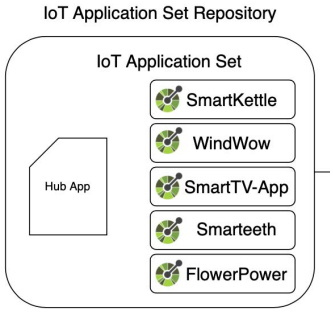

Figure 1: The figure describes the components of our application set and test infrastructure, i.e., the hub app and our initial set of IoT applications, each with an OpenAPI specification.

One drawback of the HTTP protocol, while widely used in IoT, is that it is not the most appropriate protocol in some use cases. A concrete example is applications that rely on intensive processing of asynchronous events and are deployed on devices with limited computational resources and capabilities [21], [17]. To address this issue, we plan to extend the testing framework to include MQTT communication, which is described in its own terms as "The Standard for IoT Messaging" [4]. We will leverage a similar initiative to OpenAPI, namely AsyncAPI [5], which aims to provide the same level of standardisation for event-driven systems typically built around publisher-subscriber protocols like MQTT.

3.3 User Guide to Testing, Adding New Applications, and Setting Up the Testing Infrastructure

When adding new applications to the corpus, the user can define new automation tasks in the hub app. A good practise for the automation tasks is to cover as many features of the new application as possible. Then our test framework can identify and cover these tasks and run in-depth tests on them. The current application repository is shown in Figure 1. During a test process, each application is deployed in a *Docker* container on the same network so that they can discover each other. Deployment configuration is automatically managed using the $docker - compose$ tool. The only requirements for adding a new application to the repository are:

(1) The app must contain an OpenAPI specification file.
(2) The app must be compatible with deployment inside a Docker container or at least added to a Docker network. [3]

The test framework can be set up by using the mentioned docker-compose configuration from the repository. For the interested user, we describe what happens behind the scenes in the backend. For example, the following steps are automatically executed by our system:

(1) Each *smartdevice* is deployed in a Docker container.

[2] OpenAPI specification can be consulted here: https://spec.openapis.org/oas/latest.html.
[3] OpenAPI Generator can be consulted here: https://github.com/OpenAPITools/openapi-generator.
[4] MQTT specification can be consulted here: https://mqtt.org/mqtt-specification/.
[5] AsyncAPI specification can be consulted here: https://www.asyncapi.com/docs/specifications/v2.0.0.

(2) The managing communication API between the applications is automatically built using their provided OpenAPI specification. In this step, each application becomes a *client* in the test infrastructure.
(3) The created clients are added to the Docker container of the hub application.
(4) The hub application starts executing automation rules and test cases based on the received events.

The resulting system architecture after the setup process is shown in Figure 2.

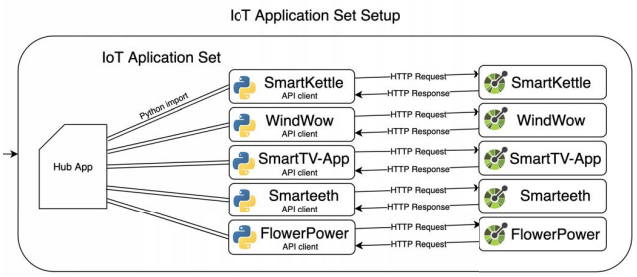

Figure 2: Application set architecture after the system is set up. API clients are generated, the hub application imports the clients and uses them to send HTTP requests to the applications.

3.4 Automation rules and bugs injection methodology

As we mentioned earlier, IoT systems consist of heterogeneous devices that interact with each other in complex patterns. To illustrate and analyse how the system can be tested and what challenges need to be overcome in different scenarios, we created a set of functional test cases on this infrastructure to evaluate the correctness of the communicating applications. We also added artificial bugs of several classes into the applications to provide a baseline for evaluating test methods and algorithms.

In our case study, we mimicked a smart home system where the communication flows between the participating applications are described by the commonly known concept of *automation rules*. The automation rules are stored in the hub application and can be triggered by one of the conditions listed below:

(1) A direct action manually requested by the user.
(2) An event generated and registered by a device.
(3) A scheduled event set by the user or application logic.

Based on our current application set, we have defined the following rules as a concrete example of possible automation processes:

(1) If the temperature of $WindWow$ is T degrees, then set the RPM of $SmartKettle$ to T.
(2) If the temperature of $FlowerPower$ is above 30 degrees, reduce the luminosity of $WindWow$ to half.
(3) If the luminosity of $WindWow$ is less than L, turn on the solar lamp of $FlowerPower$.
(4) The brightness of the $SmartTV$ is set relative to the luminosity provided by $WindWow$.
(5) The lower the temperature of the $WindWow$, the higher the temperature of the $SmartKettle$.

4 TESTING FRAMEWORK AND BASELINE METHODS

Unit testing the individual parts of a software system and ensuring that each part works correctly does not guarantee that the entire system will work correctly. By integrating multiple devices into complex logical flows and considering the persistence mechanisms behind the IoT software, it is expected that subtle interoperability and integration bugs will occur. Integration testing under the assumption of data persistence introduces new challenges, which we can divide into two perspectives:

- Testing whole multi-part systems is slower and additional technical work must be done to efficiently reproduce the reported issues.
- The issues are related to invalid states and business logic constraint violations rather than individual machine crashes, making them harder to detect. For example, in an industrial robotics pipeline, all the individual devices might be working correctly, but the orchestration logic might be causing malfunctions that damage the system, such as when two robot arms collide. We have seen an even more difficult situation with smart home systems, for example, where the end user can set their own automation rules. The end user then becomes the *programmer* of the system, but is generally unaware of the technical aspects of the system.

In [5], a survey was conducted with industry IoT solution providers and concluded that interoperability and integration are important concerns for developers. Given the above arguments and the lack of open source software addressing this issue, we believe that providing a functional method for testing integration processes is essential to advancing the current state of the art in IoT.

The implementation of our test framework suite is independent of the applications under test. In our case, we used our application suite, which consists of the applications defined in Section 3. Users of the framework can (and should) replace our applications with their own.

4.1 Functional Testing Framework

Functional testing was defined by [13] as an approach in which the design of a program is viewed as an integrated collection of functions. Although this testing method is probably one of the best known for testing an application, the unique nature of an IoT system is that the collection of functions is not part of the same software program and the success of a test case generally depends on functions from different interconnected devices. Functional testing may be the preferred testing approach for conformance testing. Our work aims to continue and improve [14] by extending conformance testing not only to a single IoT device, but to the entire IoT system. We also want to involve as many stakeholders involved in the development as possible [18]. Therefore, we chose to define test scenarios using Behave, a Python library for Behavior-Driven Development (BDD) that uses tests written in natural language [2].

BDD is a methodology that supports collaboration between business, development, and testing stakeholders using a common specification and approach to describe functionality. We found the principles of this methodology suitable for writing our test cases, as it communicates in a clear way the state of the system, the trigger conditions for applying various actions (e.g., the automation rules in our case), and the expected results after applying these actions. Overall, our experiments have resulted in facilitating the definition of test cases, interpretation, and discovery of problems in applications. Listing 1 describes several test cases for one of the above automation rules. To further understand the presented example, we briefly describe the structure of a test listing. The keyword *given* is used to set the state of the test. Triggers can be specified with the clause *When*, while expected results are marked with *Then*. All these clauses can be composed hierarchically using logical operators. In the given example, a table of values is also specified. Running the test means that the framework behind BDD takes each line of parameters and replaces them in the test specification. Thus, a test written as in Listing 1 is more like a template specification with parameters that are later defined by a requirements table, i.e., another functional test specified by each row in the table.

```
Feature: TV auto-brightness
  Scenario Outline: TV brightness is set based on window
     luminosity level
    Given TV brightness is set to <X>, window luminosity
     to <Y> and base luminosity to <Z>
      When automation rules are triggered for TV
      Then TV brightness is set to max(10 - <Y>/10 + <Z>,
     <Z>) <= 10
     Examples:
      | X  | Y  | Z  |
      | 3  | 20 | 1  |
      | 4  | 70 | 0  |
      | 1  | 0  | 10 |
```

Listing 1: An example of a template test and multiple functional test cases for the brightness automation rule 4 presented in 3.4.

The expected test suite in the IoT domain could involve either individual components or complex interaction flows and data exchange between multiple applications. In our environment, we allow users to easily specify their own test case data for scenarios related to their applications and flows under test using BDD methodology and syntax. In our current version of the framework, the following steps must be followed to create a new test:

(1) An existing automation task must be identified or a new one created in the hub application. In general, most of the interest in IoT is in testing flows that connect different endpoints of different applications. This is an important confirmation of the need for an orchestration component such as the Hub application.
(2) Add a new test using BDD syntax similar to the test shown in Listing 1. Adding a test is a two-step process. First, you need to specify the description of the test and optionally a table of individual test cases (as in our example). Then, you need to write a source code script that implements the additional operations defined in the test. Such code could, for example, define data structures for data persistence between test cases, communication flows, or new orchestration code on the existing applications.

The architectural connections between the applications, the hub application, the test case templates, and the concrete scenarios (i.e., the concrete table containing the parameters for testing the test case templates) are shown in Figures 3 and 4. An example of an application flow is shown in Figure 5. Two of the rules we designed in the hub app can interact (one triggers the other). When rule 2 is activated, it checks (by calling one of the apps) to get the temperature. When the hub receives this data, it stores it so that it can be used by another rule. According to rule 2, if the temperature provided by *FlowerPower* is above 30 degrees, then *WindWow* luminosity is reduced by half. The luminosity is also a value that is tracked by the hub. It also stores it and is immediately used by rule 3. When the luminosity is below 30, the plant lamp is turned on by *FlowerPower*.

5 EVALUATION

To provide a basis for problem discovery, we have introduced intentional problems in the smart applications and automation rules. At this point, it is important to note that the applications already contained a significant number of real issues identified by our framework. In what follows, we categorise the issues based on the level of interaction at which they occur, rather than the technical or logical cause. This decision illustrates our focus on identifying problems that arise from the integration and interoperability of different applications, rather than individual isolated software components for which there are a plethora of solutions in the literature. We consider that there are three categories of such issues:

(1) Application-level issues - malfunctions that result in invalid responses or crashing of a single, individual application.
(2) Rule-level issues - violations of the expected behaviour of a single automation rule that connects one or more applications.
(3) Persistence-level issues - violations of the expected behaviour after applying multiple rules in succession. These are usually fixed by the persistence management components of the applications.

As mentioned in the study by [22], an important class of issues are related to the triggering of unexpected actions in the flow of automated, interconnected applications. These bugs are the ones we consider at the *Rule level* and *Persistence level* we define, and are considered the most difficult to detect because they are generally a specific type of concurrency bug. The work in [22] analyzes the root cause of this category of issues and concludes that the main causes are:

(1) race conditions between rules.
(2) events missing or received in an unexpected order.
(3) duplicate or conflicting actions.

In Table 1 we have listed examples of real or artificially introduced issues in our application, categorized by the above levels. They are designed to cover all scenarios described by [22], linking application malfunctions and crashes to higher logic bugs.

Our functional testing approach is able to formulate faulty test cases for all these scenarios (see the source code repository for

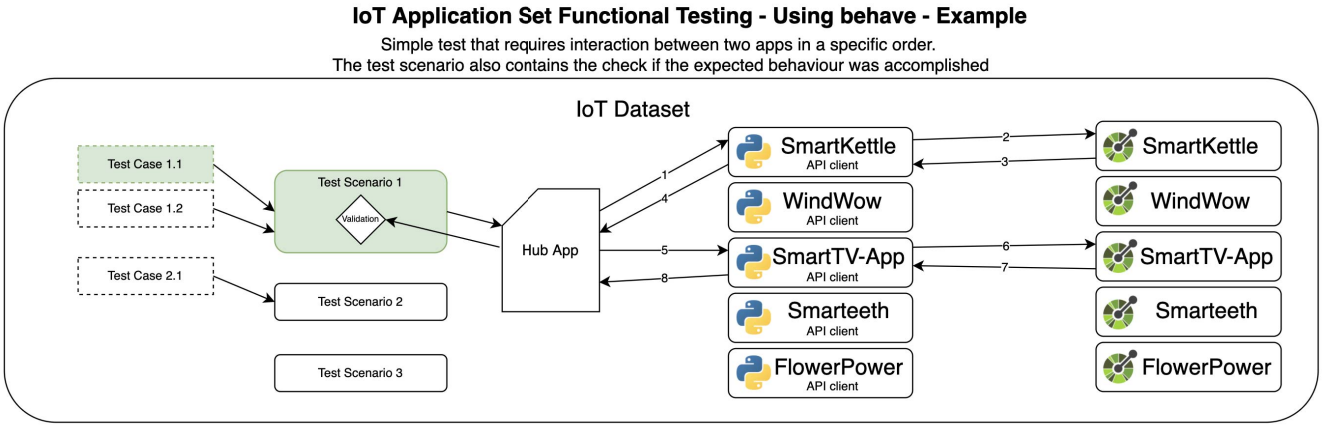

Figure 3: Application set architecture after setting up the system and defining the test cases. Upon execution, the Test Scenario, using the selected Test Case is run on the hub app. The hub app manages the requests and responses, and after the test is complete, compares the result value to the expected value of the test case.

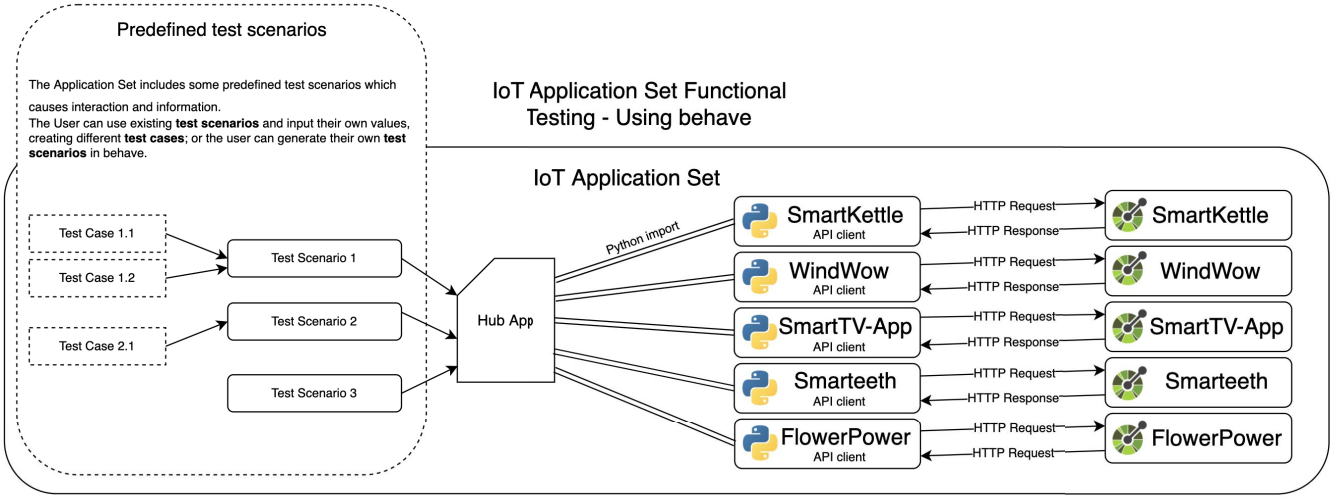

Figure 4: A comprehensive description of the Functional Testing framework. The figure describes how the framework is applied on the application set: from describing the Test Cases and Test Scenarios, to the communication with the applications.

examples of tests). It can guarantee that the system works correctly in a handful of scenarios and provide a basis for regression testing to ensure that future changes do not break the system. The application set contains 10 test scenarios that trigger various automation tasks in the hub application. Each automation requires an interaction between at least two applications, with one application generating data that is passed to another application to use.

On the other hand, the exploratory component of our approach is manual, as is typical of functional testing methods in general. Therefore, the detection of bugs is limited to the skills and time of the human tester or developer writing the functional tests. In general, concurrency bugs have a low reproducibility rate due to the order of the occurring events. Thus, we consider this a limitation from our existing framework and consider that more efforts must be spent in future work to address the the problem of detecting and reproducing concurrency issues.

6 CONCLUSION AND FUTURE WORK

In this work, we have first presented a small set of applications along with the necessary infrastructure to deploy and evaluate different methods and algorithms in the field of IoT testing. Then, we presented a framework independent of our application set that allows users to perform various functional tests for single or a set of interconnected application streams. We hope that the community

Examples of issues discovered	Application(s)	Rule(s)
Flowerpower: does not check for optional key existence in JSON object on PUT /settings	FlowerPower	-
TV brightness should be set to a maximum of 10, but the value is not validated by the app.	SmartTV	Rule 4
Rule 2 will reduce the window's luminosity if the temperature is over 30 degrees, then Rule 3 will unnecessarily turn on the lamp because the luminosity is too low.	FlowerPower, WindWow	Rule 2, Rule 3
Windwow crashes when trying to set luminosity to 25 and curtains are closed on GET /settings/{settingName}/{settingValue} (artificial bug)	WindWow	-
SmartKettle's temperature decreases for WindWow's temperatures under 0 degrees celsius instead of increasing	SmartKettle, WindWow	Rule 5
Smarteeth: "localhost" set as the hostname of the listening server thus refusing outside connections	SmartTeeth	
In FlowerPower: activateSolarLamp does not change luminosity	FlowerPower	

Table 1: Table presenting some examples of issues that have been discovered in our application set. Each issue discovered was triggered by one or multiple applications communicating. Some of the issues were discovered as part of an automatizing rule defined in our Hub Application.

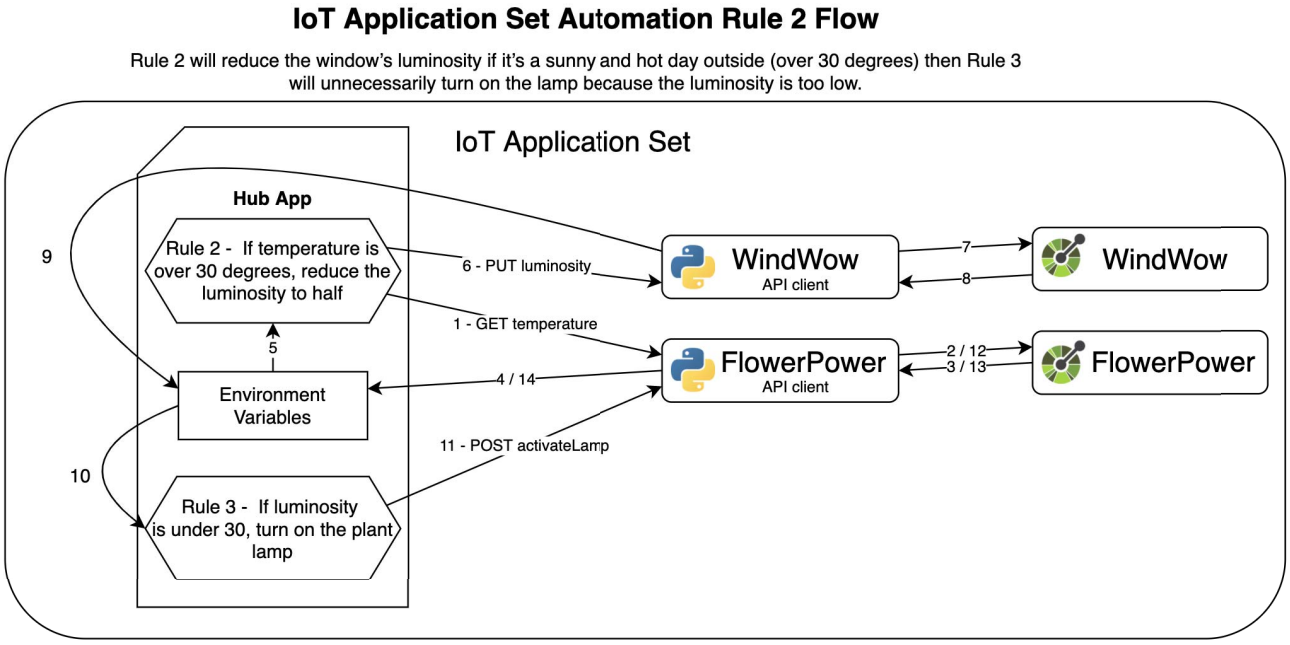

Figure 5: Flow of a multi-rule level automation (i.e., persistence management issues testing). *1,2,3,4* - The request response is sent from the Hub by rule 2, and the response is saved in the Hub's Environment Variables. *5* - Depending on the response value, Rule 2 can be continued, therefore *6,7,8,9* makes the request to WindWow, and stores luminosity in Hub's Environment Variables. If luminosity is under 30, according to Rule 3, the plant lamp is activated in steps *11,12,13,14*

developing or testing IoT applications will benefit from our foundational open-source software to experiment more quickly with new methods and increase the stability, security, and performance of applications in this domain.

The threats to the validity of our approach can be summarised in two points: First, creating a set of mock applications may not simulate the full range of common vulnerabilities in real applications. Adding some real applications to the application set could mitigate this threat. Second, a more comprehensive approach would also mean running the applications on real hardware rather than virtual machines. While this is beyond the scope of this article, it would increase the attack space and could lead to more interesting results. One of our work-in-progress plans, is to extend the application set to a variety of IoT software and communication protocols such as MQTT or ZigBee [8]. We also plan to integrate Hub software applications used by the general public, such as openHAB or Home Assistant. We could then provide a larger collection

of challenges. The current work can also be continued by experimenting with other types of testing beyond the functional testing method proposed in this paper. The practice of testing systems using fuzz testing is on the rise, and various fuzzers, both white-box and black-box based methods, can be added to expand the scope of the test suite of our current baseline methods. Fuzzing has a large exploratory aspect that contrasts with the limited ability of functional testing to examine application states in depth. Running tests on emulated or real-world devices could create new vulnerabilities for the deployed software or hardware of the system under test. Work is currently underway to extend our application to typical embedded devices on the market. Since we have not insisted on a concrete methodology for comparing research results using our application set, further work could be to develop a rigorous set of procedures for benchmarking test techniques and tools.

7 ACKNOWLEDGEMENTS

We thank the final year Informatics undergraduate students, 2020 Promotion of the Computer Science and Mathematics Faculty of the University of Bucharest that created the applications that are now part of the applications set.

This research was supported by the European Regional Development Fund, Competitiveness Operational Program 2014-2020 through project IDBC (code SMIS 2014+: 121512).

REFERENCES

[1] Yahya Al-Hadhrami and Farookh Khadeer Hussain. 2021. DDoS attacks in IoT networks: a comprehensive systematic literature review. *World Wide Web* 24, 3 (01 May 2021), 971–1001. https://doi.org/10.1007/s11280-020-00855-2

[2] Richard Jones Benno Rice and Jens Engel Revision. 2022. Behave. https://behave.readthedocs.io/en/stable/index.html. Accessed: 2022-01-10.

[3] Carl Boettiger. 2015. An Introduction to Docker for Reproducible Research. *SIGOPS Oper. Syst. Rev.* 49, 1 (jan 2015), 71–79. https://doi.org/10.1145/2723872.2723882

[4] Miroslav Bures, Bestoun S. Ahmed, Vaclav Rechtberger, Matej Klima, Michal Trnka, Miroslav Jaros, Xavier Bellekens, Dani Almog, and Pavel Herout. 2021. PatrIoT: IoT Automated Interoperability and Integration Testing Framework. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE. https://doi.org/10.1109/icst49551.2021.00059

[5] Miroslav Bures, Matej Klima, Vaclav Rechtberger, Xavier Bellekens, Christos Tachtatzis, Robert Atkinson, and Bestoun S. Ahmed. 2020. Interoperability and Integration Testing Methods for IoT Systems: A Systematic Mapping Study. In *Software Engineering and Formal Methods*. Springer International Publishing, 93–112. https://doi.org/10.1007/978-3-030-58768-0_6

[6] Fulvio Corno, Luigi De Russis, and Juan Pablo Sáenz. 2020. How is Open Source Software Development Different in Popular IoT Projects? *IEEE Access* 8 (2020), 28337–28348. https://doi.org/10.1109/ACCESS.2020.2972364

[7] Joao Pedro Dias, Flavio Couto, Ana C.R. Paiva, and Hugo Sereno Ferreira. 2018. A Brief Overview of Existing Tools for Testing the Internet-of-Things. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. https://doi.org/10.1109/icstw.2018.00035

[8] Sinem Coleri Ergen. 2004. ZigBee/IEEE 802.15. 4 Summary. *UC Berkeley, September* 10, 17 (2004), 11.

[9] Vahid Garousi, Ali Mesbah, Aysu Betin-Can, and Shabnam Mirshokraie. 2013. A systematic mapping study of web application testing. *Information and Software Technology* 55, 8 (Aug. 2013), 1374–1396. https://doi.org/10.1016/j.infsof.2013.02.006

[10] Ahmad Nauman Ghazi, Kai Petersen, and Jürgen Börstler. 2015. Heterogeneous Systems Testing Techniques: An Exploratory Survey. In *Lecture Notes in Business Information Processing*. Springer International Publishing, 67–85. https://doi.org/10.1007/978-3-319-13251-8_5

[11] Zhijie Gui, Hui Shu, Fei Kang, and Xiaobing Xiong. 2020. FIRMCORN: Vulnerability-Oriented Fuzzing of IoT Firmware via Optimized Virtual Execution. *IEEE Access* 8 (2020), 29826–29841. https://doi.org/10.1109/ACCESS.2020.2973043

[12] Stephen Hilt, Numaan Huq, Martin Rösler, and Akira Urano. 2017. Cybersecurity Risks in Complex IoT Environments: Threats to Smart Homes, Buildings and Other Structures. (2017). Accessed: 2022-01-10.

[13] William E Howden. 1980. Functional program testing. *IEEE Transactions on Software Engineering* 2 (1980), 162–169.

[14] Hiun Kim, Abbas Ahmad, Jaeyoung Hwang, Hamza Baqa, Franck Le Gall, Miguel Angel Reina Ortega, and JaeSeung Song. 2018. IoT-TaaS: Towards a Prospective IoT Testing Framework. *IEEE Access* 6 (2018), 15480–15493. https://doi.org/10.1109/ACCESS.2018.2802489

[15] Chang Liu. 2005. Enriching software engineering courses with service-learning projects and the open-source approach. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 613–614. https://doi.org/10.1109/ICSE.2005.1553612

[16] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1393–1403. https://doi.org/10.1145/3468264.3473932

[17] Nitin Naik. 2017. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In *2017 IEEE international systems engineering symposium (ISSE)*. IEEE, 1–7.

[18] Anton Okolnychyi and Konrad Fögen. 2016. A study of tools for behavior-driven development. *Full-scale Software Engineering/Current Trends in Release Engineering* (2016), 7.

[19] Ciprian Păduraru, Rareș Cristea, and Eduard Stăniloiu. 2021. RiverIoT - a framework proposal for fuzzing IoT applications. *International Conference on Software Engineering ICSE 2021, Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT)* (2021), 52–58.

[20] Kazi Masum Sadique, Rahim Rahmani, and Paul Johannesson. 2020. IMSC-EIoTD: Identity Management and Secure Communication for Edge IoT Devices. *Sensors (Basel, Switzerland)* 20, 22 (16 Nov 2020), 6546. https://doi.org/10.3390/s20226546 33207820[pmid].

[21] Tetsuya Yokotani and Yuya Sasaki. 2016. Comparison with HTTP and MQTT on required network resources for IoT. In *2016 international conference on control, electronics, renewable energy and communications (ICCEREC)*. IEEE, 1–6.

[22] Wei Zhou, Chen Cao, Dongdong Huo, Kai Cheng, Lan Zhang, Le Guan, Tao Liu, Yan Jia, Yaowen Zheng, Yuqing Zhang, Limin Sun, Yazhe Wang, and Peng Liu. 2021. Reviewing IoT Security via Logic Bugs in IoT Platforms and Systems. *IEEE Internet of Things Journal* 8, 14 (July 2021), 11621–11639. https://doi.org/10.1109/jiot.2021.3059457

Software Engineering Approaches for TinyML based IoT Embedded Vision: A Systematic Literature Review

Shashank Bangalore Lakshman
Boise State University
Boise, ID, USA
shashankbangalor@u.boisestate.edu

Nasir U. Eisty
Boise State University
Boise, ID, USA
nasireisty@boisestate.edu

ABSTRACT

Internet of Things (IoT) has catapulted human ability to control our environments through ubiquitous sensing, communication, computation, and actuation. Over the past few years, IoT has joined forces with Machine Learning (ML) to embed deep intelligence at the far edge. TinyML (Tiny Machine Learning) has enabled the deployment of ML models for embedded vision on extremely lean edge hardware, bringing the power of IoT and ML together. However, TinyML powered embedded vision applications are still in a nascent stage, and they are just starting to scale to widespread real-world IoT deployment. To harness the true potential of IoT and ML, it is necessary to provide product developers with robust, easy-to-use software engineering (SE) frameworks and best practices that are customized for the unique challenges faced in TinyML engineering. Through this systematic literature review, we aggregated the key challenges reported by TinyML developers and identified state-of-art SE approaches in large-scale Computer Vision, Machine Learning, and Embedded Systems that can help address key challenges in TinyML based IoT embedded vision. In summary, our study draws synergies between SE expertise that embedded systems developers and ML developers have independently developed to help address the unique challenges in the engineering of TinyML based IoT embedded vision.

CCS CONCEPTS

• **Software and its engineering** → **Software development methods**.

KEYWORDS

Software Engineering; IoT; TinyML; Embedded Vision; Systematic Literature Review

ACM Reference Format:
Shashank Bangalore Lakshman and Nasir U. Eisty. 2022. Software Engineering Approaches for TinyML based IoT Embedded Vision: A Systematic Literature Review. In *4th International Workshop on Software Engineering Research and Practice for the IoT (SERP4IoT'22), May 19, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3528227.3528569

SERP4IoT'22, May 19, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9332-4/22/05…$15.00
https://doi.org/10.1145/3528227.3528569

1 INTRODUCTION

The desire for a better lifestyle and the growing abundance of challenges faced on the planet have necessitated technological breakthroughs. 'Edge Intelligence' aims to harness the combined power of IoT and ML to solve unique problems at scale, right at the place where the data gets generated [28]. Millions of ubiquitous smart embedded vision systems (cameras with local computing power and network connectivity) are deployed and will be deployed at the edge in various domestic, industrial, and commercial applications (some examples are illustrated in Figure 1). IoT sensor networks collect gigabytes of data every minute, and their computational systems must process the data streams in real-time to provide valuable, actionable insights to people, and other systems [28]. IoT embedded vision systems are being deployed at the far edge for valuable gains in speed, power efficiency, cost efficiency, privacy, and autonomy [28]. Many previously intractable problems that previously required human experts and sophisticated hardware-software systems for decision-making are now starting to be automated by deploying start-of-art ML models on lean embedded IoT.

SE for ML has evolved over the last decade [12], aimed at fielding a plethora of challenges faced by ML developers to harness the power of Deep Neural Networks (DNN). Similarly, SE for IoT embedded systems has also evolved over the past decade into a successful research domain [22]. Because TinyML combines ML for computer vision (CV) with IoT embedded systems, engineering, deployment, and maintenance of TinyML applications, requires a well-defined and customized SE approach to address the unique challenges in the domain. Without a guiding set of SE approaches for TinyML based IoT embedded vision, product development lifecycle can involve high costs, low productivity, and weaknesses in the field [12]. The need for low latency, low power, low cost, and robustness in these applications requires SE processes to be aware of systems architecture, algorithms, and challenges in real-world deployments. Our research summarizes state-of-art SE approaches applicable to solve the unique challenges in TinyML engineering and proposes a streamlined SE workflow to simplify the development, deployment, and maintenance of IoT embedded vision applications, in addition to suggestions for developer tools.

Figure 1: Emerging embedded vision IoT applications

2 BACKGROUND

It is projected that a total of 2.5 billion edge AI devices will ship with a TinyML chipset in 2030 [25]. Early Proof-of-Concept (POC) applications using TinyML based IoT embedded vision have showcased huge potential to enable smart manufacturing, smart home, smart city, smart devices, infotainment, autonomy, and smart agriculture applications on extremely lean IoT systems. However, this approach is yet to scale into massively deployed products and solutions in the real world, as these systems are challenged by portability, robustness, reliability, development costs, and skill gaps [30].

The rapid growth in CV applications powered by Deep Learning (DL) approaches is due to DNN-based approaches extracting features automatically from training data, without the need for heuristic rules [16]. Typical CV applications are driven by Convolutional Neural Networks (CNN) as visual information has rich spatial information. Trained models also must be trained with the appropriate quantity and quality of data based on an understanding of input data expected in real-world deployments. As DL became popular over the decade, there was an uncontrolled growth in the size of DL models. This growth resulted in steep demand for both computing power and memory availability in training as well as inference systems [31].

Migrating such monstrous DL models into leaner IoT embedded systems for inferencing tasks is a unique challenge that requires a systematic overhaul towards hardware-aware engineering [31]. Typical advanced MCUs are those that are capable of running highly-optimized, compressed DNN models by leveraging only a few 100 kBs of memory, computing speed in the order of MIPS (Million Instructions Per Second) or GIPS (Giga Instructions Per Second) at a power consumption of <1W [8]. 'Model reduction', 'parameter quantization', 'knowledge distillation', and 'Neural architecture search' are the key tools for overcoming machine learning challenges on the edge [13] [30] [31].

IoT embedded vision applications offer the advantage of low latency, real-time actionable insights, lower cost of data transmission, higher reliability under economy of scale. TinyML can be broadly defined as ML approaches capable of performing on-device analytics for a spectrum of sensing modalities at "mW" (or below) power range, on lean and battery-operated devices. A few examples of off-the-shelf IoT embedded vision hardware capable of running TinyML applications are showcased in Figure 2. A comparison of their key technical specifications is presented in Table 1. If the TinyML engineering process can be made more efficient, then the lifetime cost of applications can be driven further low by the massive market potential for IoT embedded vision applications.

ML has gained massive popularity with application developers due to the availability of hardware-agnostic open-source ML frameworks such as TensorFlow and PyTorch. Similarly, TinyML has embraced a framework approach to simplify developer experience and amplify productivity. TinyML has multiple frameworks such as TensorFlow Lite Micro (TFLM), uTensor, STM32Cube.AI, and Embedded Learning Library (ELL). However, our research only focused on TFLM framework due to its popularity and being independent on the hardware.

TFLM is an open-source ML inference framework (developed by Google and Harvard University) for running deep-learning models on MCU based embedded systems with very lean computing and memory infrastructure. TFLM addresses the resource constraints faced with running DL models on embedded systems with 32-bit MCUs and a few 100kB of memory, without hardware-specific customizations [8]. In TFLM, the trained model is interpreted by the application code by giving it pre-processed input data and then performing post-processing of model output and output handling [32].

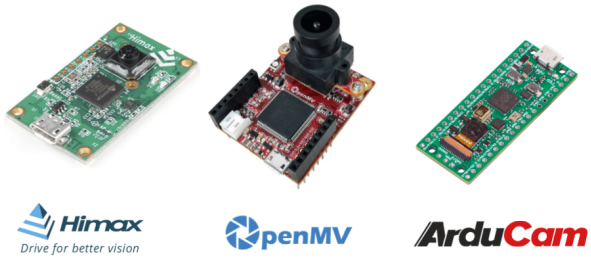

Figure 2: Example IoT embedded vision devices [7]

A large part of modern CV research and development has been driven by approaches unconstrained by hardware infrastructure, resulting in very large DL models. To democratize CV on the edge [8], it is necessary to take a hardware-aware approach to ML engineering, so that inference can be performed on lean IoT systems while maintaining compatibility to multiple MCU platforms.

Lack of SE expertise in both IoT embedded vision systems and ML engineering creates an open gap in SE approaches required for TinyML powered IoT embedded vision applications. A summary of key challenges involved in TinyML engineering is listed in Table 2. In addition, the challenges are grouped under key milestones in the TinyML engineering lifecycle.

3 RESEARCH METHODOLOGY

The previous section described the numerous challenges faced by TinyML application developers. Our systematic literature survey addresses two research questions to summarize SE best practices reported in the literature and visualizes a streamlined workflow for TinyML engineering. TinyML launched only in 2018 and hence, there is a lack of published research into software engineering approaches and best practices. Our research performed a systematic literature review of related prior art in TinyML, Machine Learning, Computer Vision, and Embedded Systems, with the overall goals described here:

- Summarization of SE best practices applicable to TinyML engineering for computer vision applications.
- Visualization of a SE workflow that can enable quick and easy-to-manage TinyML engineering for developing and deploying CV applications on edge.
- Suggestion of ideas for new developer support features in production tools (IDE, packages, test tools, etc.) to accelerate market adoption of TinyML for CV applications.

Table 1: Comparison of hardware specifications of off-the-shelf IoT embedded vision hardware [7]

IoT device	HIMAX WE-I Plus EVB	OPENMV Cam H7 R2	ARDUCAM Pico4ML
Compute	WE-I Plus ASIC (HX6537-A), ARC 32-bit EM9D DSP with FPU @400MHz	ARM 32-bit Cortex-M7 CPU w/ Double Precision FPU @480MHz	RP2040 Dual-core Arm Cortex-M0+ processor @133MHz
Memory	2MB flash and 2MB SRAM	2MB flash and 1MB SRAM	2MB flash and 256kB SRAM
Vision sensor	Himax HM0360 AoS TM ultra-low power VGA CCM, 1/6", 640×480 pixels @60FPS	MT9M114: CMOS Image Sensor, 1/6", 640×480 pixels @40FPS	Himax HM01B0 CCM, 1/11", 320 x 240 pixels @60FPS
Device cost	$65	$65	$50

3.1 Research Questions (RQs)

We identified two Research Questions (RQs) that span SE challenges and approaches in large ML domain as well as IoT.

- **RQ1: Which widely used SE practices from large production AI/ML engineering are applicable to TinyML engineering?**
  - Need: To identify SE practices used in generic large AI/ML systems as well as CV-centric large AI/ML systems from a thorough literature survey of state-of-art while determining their applicability to TinyML engineering.
  - RQ1 aggregates SE solutions already prevalent in practice while ensuring their applicability to challenges imposed on the model size by TinyML constraints.
- **RQ2: Which widely used SE practices from traditional embedded systems engineering are applicable to TinyML engineering?**
  - Need: To identify SE practices used in embedded systems engineering from a thorough literature survey of state-of-art while determining their applicability to TinyML engineering.
  - RQ2 aggregates SE solutions that are already prevalent in practice while ensuring their applicability to challenges imposed by TinyML vision applications.

3.2 Research protocol

TinyML and TFLM were introduced in 2018, and the published scientific literature is limited. Only a handful of papers pertain to embedded vision applications. Additionally, the TinyML literature is mostly gray literature, tutorials, workshops, talks, and blogs. To address challenges in TinyML engineering reported in the literature, the systematic literature survey focused on aggregating solutions from a broad spectrum of information sources.

We gathered 43 information sources from arXiv, IEEE, ACM Digital Library, Springer Link, ML conference tutorials, and workshops using sample search queries described in Table 3. Then, we studied the abstract and conclusions from the gathered literature to refine the source pool to 20. Finally, we forward and backward snowballed on Google Scholar, Google Search, and related Blogs to derive 13 additional sources. In total, we have 33 information sources as the primary pool for this study.

4 RESULTS AND DISCUSSION

The first section ties various findings into the research questions defined at the start of the project. The second section ties the SE approaches and best practices to derive a SE workflow for TinyML CV application developers. In the interest of brevity, only the most compelling results relevant to IoT embedded vision systems are presented. Challenges and solutions in TinyML engineering are presented in Table 4 and 5, with classification based on steps involved in TinyML workflow process. Answers from RQ1 and RQ2 were evaluated for applicability to TinyML engineering of edge CV applications through thorough study.

4.1 RQ1 findings

Established SE practices in large-scale AI/ML and CV engineering are applicable broadly to solve many of the challenges in IoT embedded vision engineering. However, there is a significant need for hardware-aware Neural Architecture Search (NAS) approaches, hardware-aware co-optimization approaches, reference datasets for IoT embedded vision applications, and standardized benchmarking techniques and metrics.

4.2 RQ2 findings

It is to be noted that established SE practices from embedded/IoT engineering largely apply to IoT embedded vision systems. However, there is a significant need for portable libraries for pre-processing and post-processing functions that are key components of the ML inference architecture. SE practices need tools capable of design space search and automatic version control for deployment-ready TinyML models.

4.3 SE workflow for TinyML engineering

Based on the results of the systematic literature survey, we see the need to define a software engineering workflow specifically to cater to TinyML engineering. Inspired by CRISP-DM, Microsoft ML workflow, and TFLM workflows, we present a modified workflow in Figure 3. The workflow can be adopted by CV application developers alongside results summarized by this systematic literature survey. The proposed SE workflow divides the engineering process into steps that are suggested to be performed 'in situ' aka 'training environment with CPU and GPU' and to be performed 'in vivo' aka 'lean edge hardware performing inference in real-world deployment'.

It is beneficial to break down the business requirements (customer requirements) into three inter-related components: model requirements, data requirements, and system requirements. This approach helps delegate ownership of delivering individual components to different individuals or teams. It also enables autonomy

Table 2: Key challenges in TinyML based IoT embedded vision engineering

CHALLENGE	DESCRIPTION
Requirements specifications and design	System design choices: No clear decision framework or benchmark that can guide TinyML based IoT embedded vision application developers with regard to hardware/model design choices to be made for a given application/market/customer requirement [21]. Compiler choices: Embracing sophisticated compilers can help optimize for specific MCU targets. However, this affects portability, and hence, it challenges large-scale deployment under availability constraints. Need to enable platform specific optimizations without need for specialized compiler [8]. Ensuring the security of embedded devices is a challenge for IoT applications [18] [26]. Requirements in the field may evolve, and the system needs to grow its intelligence to meet the goals [12]. IoT embedded vision devices need to be "less chatty" to reduce the power consumed by communications [18]. IoT embedded devices have a high degree of heterogeneity [5]. Although hand coding and code generation can leverage specific optimizations at the cost of flexibility [5]. On-device training is challenging in small memory footprint devices [6]
Data and model search	Lack of curated datasets derived from IoT embedded vision sensors for different applications [5]. Lack of standard tools to compare performances between different algorithms against different MCU systems [5].
Model development and validation	Challenging to assess the impact of different levels of quantization and precision as well as model peak memory requirements [5]. Due to the wide spectrum of MCU designs, it is beneficial to take a framework with generic compiler features for design portability. However, this generic compiler might not be able to fully utilize unique hardware features offered in some families of MCUs [8]. TinyML workflow is not hardware-aware due to limitations in generic frameworks [8]. Naive joint optimization techniques such as Neural Architecture Search (NAS), Quantization and Pruning have cross-interactions which may result in sub-optimal results [27]. Lack of emulation tools for TinyML engineering increases the time and effort spent on model development [11].
Application development and deployment	Application portability across different devices and different vendors is a challenge in TinyML engineering [24]. Power profiling is complex since data paths and pre-processing steps can vary significantly between devices [5]. TinyML powered applications are lean, and they lack tools for measurement and visualization of data. This imposes additional challenges to debuggability during model development and deployment due to memory constraints [5]. TinyML capable devices have drastically different power consumption, which makes it harder to maintain accuracy consistently across the devices [5]. Real-world conditions can rapidly evolve due to changes in the sensing environment or the aging of vision sensors. TinyML improves a significant challenge to online learning due to the low amount of compute available [26]. Since edge CV applications may be used to drive time-critical decisions, the reliability of TinyML systems needs to be high. Robustness needs to be built into the TinyML models to enable safe fail-over mechanisms in the case of ambiguous environments [26]. Models deployed on TinyML vision systems must be capable of being upgraded without significant intervention or system downtime.
SE methodologies	Not much is known about efficient, best practices for TinyML application IoT embedded vision product development, deployment and maintenance. Agile development practices for TinyML engineering are not well established. TinyML CV application requires expertise in embedded systems, computer vision, and machine learning. Since TinyML is still in a nascent stage, there is a lack of skilled engineers who have sufficient expertise to address challenges in TinyML engineering for CV apps [24]

Table 3: Protocol summary of systematic literature review

Research Questions	RQ1: Which widely used SE practices from large production AI/ML engineering are applicable to TinyML engineering? RQ2: Which widely used SE practices from traditional embedded systems engineering are applicable to TinyML engineering?
Sample Search strings	"TinyML" AND "Computer Vision" "Software Engineering" AND "Edge Computer Vision" "Software Engineering" AND "Machine Learning" "Software Engineering" AND "Embedded Systems"
Search strategy	DB search: arXiv, IEEE, ACM Digital Library, Springer Link Backward and forward snowballing using Google Scholar Manual search using TinyML.org and Google Search
Inclusion Criteria	The paper is written in English Grey-literature is accepted since TinyML is quite recent
Exclusion Criteria	The paper is related to software engineering applied to traditional computing TinyML powered by non-TFLM (TensorFlow Lite Micro) frameworks
Study type	Primary and Secondary studies

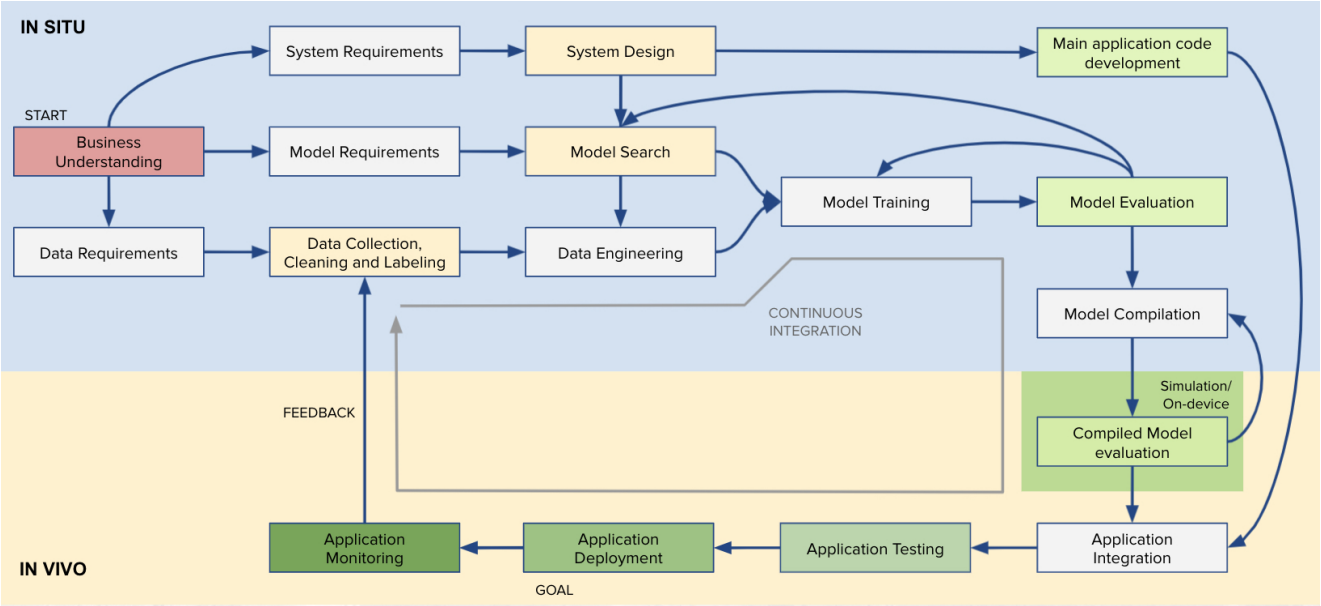

Figure 3: SE workflow for TinyML engineering for CV applications

for those teams to understand requirements to apply their expertise in system design, model search, and setting up pipelines for data collection, cleaning, and labeling. Once the system design is generated, it will provide additional inputs to constrain model search and data engineering. Given that the application must run on a lean MCU resource, system design provides the boundary conditions on computing-memory complexity of algorithms used in data engineering (pre-processing and post-processing) and machine learning.

Main application development can be carried out concurrently with model development. Concurrent development allows embedded developers to develop and debug heuristic pipelines to perform sensing and actuation based on expected outcomes from the ML model. They can also debug this code using calibration functions to emulate outputs of the final ML model. The model development process can continue after model search and data engineering steps. Using best practices in model training, a suitable model can be derived and evaluated against a validation data set. Feedback from the evaluation step can lead model developers into model search or model training to reconfigure the model or its hyper-parameters.

Once the model validation is satisfactory, the model can proceed to the compilation steps, which are automatically taken care of by the TFLM framework. Hardware-specific libraries can be used during this process to take advantage of custom hardware features available in target MCUs. After the model is compiled successfully, compiled model evaluation can be carried out in a simulator in the

Table 4: SE challenges and solutions in TinyML engineering for edge CV apps

WORKFLOW STEP	CHALLENGE	PROPOSED SOLUTION(S)	ORIGIN
Business (Customer) Understanding	TinyML engineering requires expertise in AI, ML, CV, and Embedded Systems.	Decoupling teams which create systems, data, and model data is helpful to carry out work concurrently [1].	RQ1
	Customer needs are changing dynamically.	In order to streamline the TinyML engineering process, customer needs must be version-controlled to pursue incremental improvement on tail features requested by the customer [12].	RQ1
System Requirements	Deployment in real-world environments is challenging.	Define common implicit assumptions related to visual data (illumination, shadows, etc).	RQ1
	Mismatch in hardware systems and portability of models across a family of hardware is required.	Need to manage the combinatorial explosion of hardware devices using strict control of device types and chipsets used in the TinyML systems [9].	RQ2
	Performance degradation in a real-world scenario is a major challenge to TinyML CV systems on the edge.	Design decisions must be driven by environmental challenges that the TinyML CV application is likely to experience. Data augmentation of the training data set is necessary [9].	RQ1
Model Requirements	Determination of models applicable to TinyML projects is not well documented.	TinyML systems need to select target MCU hardware for their projects based on requirements of model size, inference speed, and power limitations necessary for the application [5].	RQ2
	Determining useful models for lean edge is not an easy task.	Constrained Neural Architectural Search (NAS) can be used to narrow down model architectures relevant to the system design choices [10] [23].	RQ1
Data Requirements	Data sets available can have very limited data.	Test-driven development can be used to ensure test data is not used during training as well as to identify ways to test corner cases based on system specs [12].	RQ2
System Design	Model development tools are tedious to maintain for different target hardware systems.	Setting up TinyML TFLM framework infrastructure correctly at start will help iterate faster [8].	RQ1
	Model development involved high complexity when catering to an array of applications.	Model management and version control are very important for streamlining TinyML development efforts [9].	RQ1
	Vision data related to videos often contain redundant information temporally, and it leads to additional unnecessary computation on the lean edge hardware.	Event-based approaches to visual information processing can be applied to reduce the burden imposed by redundant computations [29].	RQ1
Data Engineering	Data derived from different sensors and sources can have various file formats.	No direct access to data, only programmatic interfaces (8-bit RGB image instead of image.jpg) [9].	RQ1
	Real-world conditions might not be represented in distributions found on data sets.	Data augmentation can be used to simulate real-world conditions (scaling, noise, shadows, color, lighting conditions, etc.).	RQ1
Model Training	Hyperparameter search process can experience issues in reproducibility.	Usage of deterministic randomness (using exposed random seeds) is helpful for iteration with hyperparameter tuning [9].	RQ1
	Training TinyML CV models with a performance driven perspective is hard.	Building training pipeline by adding verification-based counter-examples will be helpful [9] [17].	RQ1
	TinyML application developers might lack deep expertise in ML and DL.	TinyML application developers must leverage AutoML tools to automate the training process, especially when a large amount of training and validation data is available.	RQ1

Table 5: SE challenges and solutions in TinyML engineering (continued)

WORKFLOW STEP	CHALLENGE	PROPOSED SOLUTION(S)	ORIGIN
Model Training (contd.)	Web automation of TinyML engineering.	EdgeImpulse and Qeexo among other startups offer AutoML capabilities that are delivered over web, directly into edge CV applications [2].	RQ1
Model Evaluation	TinyML models need to be robust to take care of environmental anomalies and noise.	Test-driven development must rely on measuring local and global adversarial robustness [17].	RQ1
	Lack of standardized TinyML performance metrics in reported literature.	MLPerfTiny benchmark has been published by MLCommons to provide the first industry standard benchmark suite for ultra low power ML systems [3].	RQ1
Model Compilation	Performance cannot be measured until an application is deployed on the edge device system.	Compilers must be capable of providing access to performance metrics for MCUs as they are typically single-threaded [8].	RQ1
	Compiler settings need to be optimized for hardware system targeted by the system design choices.	Hardware aware compiler is helpful to take desirable tradeoffs automatically without the need for understanding of MCU architectures [15] [33].	RQ1
Compiled Model Evaluation	It is hard to debug compiled model performance on the edge device due to compute constraints.	Device simulators and emulators must be used in TinyML development. Device simulators can help contrast performance of raw models and various flavors of compiled TinyML models [9].	RQ2
Application Design	Design tradeoffs are hard to optimize until the model is optimized.	Automated tools can aid the search for tradeoffs between performance, accuracy, and power [4].	RQ1
Application Evaluation	Application issues are harder to debug since the system involves heuristic code, as well as trained TinyML model.	Review of bug taxonomy for IoT applications, can help avoid critical issues in application code [19] [14].	RQ2
	Testing TinyML models can be derailed due to bias in tests.	Dedicated QA teams can help with fair debug with developer bias [9].	RQ2
Application Deployment	Real-world deployment of TinyML is hard even after successful validation of models in training environment.	Reproducibility is driven when algorithms, parameters, and data/labels are 'uniquely identifiable' and 'retrievable' in real-world. [9].	RQ1
	Fixing bugs in embedded application deployment is hard.	Over-the-air (OTA) updates must be available to seamlessly deliver bug fixes and upgrades for both firmware and TinyML model [20].	RQ2
Application Monitoring	Performance improvements in the field is challenging due to limitations on edge hardware.	Continuous Integration (CI) of Labeling and Retraining using feedback from monitoring can help drive application performance with new data distributions are encountered in the field [9].	RQ1
	It is hard to measure the performance of the TinyML CV application in the field.	Device-level monitoring tools can be used to improve debuggability in the field [9].	RQ2

training environment or on-device inference environment based on the availability. The advantage of executing this step in a simulator environment has been described earlier in the results section. If there is a significant drop in model performance after compilation, the feedback must be relayed back to the model compilation step so that necessary actions can be taken.

Once the compiled model delivers performance comparable to the original ML model, the model can be delivered to the application integration team, which will integrate the model into the application code, followed by application testing with test data set and deployment in the field. Monitoring application performance in the real-world situations can provide valuable feedback, which can be used for Continuous Integration (CI) and guide the augmentation or addition of suitable new data from real-world distributions.

4.4 Future IoT embedded vision developer support tools

We are proposing certain key features for next-generation IoT embedded vision developer support tools. These features enable a better developer experience, reduce bugs, improve debuggability, and productize scale IoT embedded vision applications.

(1) Embedded IoT developer tools must integrate TinyML frameworks such as TFLM, through an interactive user-interface.
(2) SE workflow must be integrated into the toolchain to guide developers through the application design, development, and deployment process.
(3) Allow developers to import MCU models into the tool and visualize the MCU architecture. Allow comparison of different

MCU choices to guide design choices based on application performance targets (FPS, TOPS, Precision support, etc).
(4) Allow emulation of devices to test the compatibility of compiled models prior to deployment on physical hardware. This creates an opportunity to visualize application and model operation without significant constraints on measurability.
(5) Automatically capture device capabilities and set constraints into the design space to guide the TinyML development process.
(6) Allow integration of custom libraries into the toolchain to extend generic compiler capabilities to help developers leverage unique hardware features on the target MCU.
(7) Support multiple device management for IoT embedded systems including the capability to send firmware, code, and model updates over-the-air (OTA).
(8) Allow configuration and interaction with a large number of IoT embedded vision devices through RESTful API interfaces.

5 THREATS TO VALIDITY

As TinyML is evolving at a fast pace, framework designs are bound to evolve rapidly and new SE approaches might be necessary. Hence, the study lacks the perspective that may be obtained through surveys and interviews of TinyML CV application developers in the industry. Amount of TinyML literature is limited since TinyML and TFLM concepts were introduced very recently (2018). It is hard to quantify existing TinyML SE practices from only a handful of peer-review publications.

As the field evolves rapidly, additional challenges and best practices will come to light which can challenge some of the conclusions drawn by our study. Novel model architectures, model search approaches, TinyML development techniques may significantly alter the proposed SE workflow and developer tool features.

6 CONCLUSION

There is significant scope for more profound research into SE approaches for production-scale engineering of IoT embedded vision applications. Hardware-aware generic TinyML compiler tool features will be required to extract full performance out of lean edge devices. At the same time, application portability continues to challenge fixed-form compilers that focus on narrow optimizations for highly specific hardware systems.

The proposed SE workflow is useful for concurrent work by model developers, application code developers, and data engineers, with high potential to evolve with TinyML research. This workflow also accelerates the TinyML engineering process by delegating work to teams with the necessary expertise. The integration of the CI approach into the workflow provides a robust approach for refining the TinyML product in the field without requiring expensive recalls or repairs.

Further experimental research on the proposed SE best practices and workflow will help validate and refine the SE approaches for production-scale engineering TinyML CV applications.

REFERENCES

[1] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st ICSE: SE in Practice (SEIP)*, pages 291–300, 2019.
[2] Arm. TinyML brings AI to smallest arm devices, Aug 2021.
[3] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau, U. Thakker, A. Torrini, P. Warden, J. Cordaro, G. D. Guglielmo, J. Duarte, S. Gibellini, V. Parekh, H. Tran, N. Tran, N. Wenxu, and X. Xuesong. MLPerf Tiny Benchmark, 2021.
[4] C. Banbury, C. Zhou, I. Fedorov, R. M. Navarro, U. Thakker, D. Gope, V. J. Reddi, M. Mattina, and P. N. Whatmough. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers, 2021.
[5] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, D. Patterson, D. Pau, J. sun Seo, J. Sieracki, U. Thakker, M. Verhelst, and P. Yadav. Benchmarking TinyML Systems: Challenges and Direction, 2021.
[6] H. Cai, C. Gan, L. Zhu, and S. Han. Tiny transfer learning: Towards memory-efficient on-device learning. *CoRR*, abs/2007.11622, 2020.
[7] H. Chaudhary. Comparison of ai vision boards with onboard camera: We-i plus evb vs openmv cam h7 vs pico4ml, Jul 2021.
[8] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden. Tensorflow lite micro: Embedded machine learning on TinyML systems, 2021.
[9] D. Doria, I. Ernst, and B. Kadlec. *CVPR18: Tutorial: Software Engineering in Computer Vision Systems.* ComputerVisionFoundation, Jun 2018.
[10] I. Fedorov, R. P. Adams, M. Mattina, and P. N. Whatmough. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers, 2019.
[11] M. Gielda. Running tf lite on microcontrollers without hardware in renode, Feb 2022.
[12] G. Giray. A software engineering perspective on engineering machine learning systems: State of the art and challenges. *Journal of Systems and Software*, 180:111031, 2021.
[13] A. Goel, C. Tung, Y. Lu, and G. K. Thiruvathukal. A survey of methods for low-power deep learning and computer vision. *CoRR*, abs/2003.11066, 2020.
[14] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan. A comprehensive study on deep learning bug characteristics, 2019.
[15] L. Lai, N. Suda, and V. Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus, 2018.
[16] Y. Lecun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.
[17] A. Lomuscio. *TinyML talks: Verification of ML-based AI systems and its applicability in edge ML.* tinyML, Oct 2021.
[18] M. Loukides. TinyML: The challenges and opportunities of low-power ML applications, Oct 2019.
[19] A. Makhshari and A. Mesbah. Iot bugs and development challenges. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 460–472, 2021.
[20] V. Mehta. Challenges of edge AI inference, Jul 2021.
[21] S. Mukherjee, Y. Verma, A. Gopani, and A. Choudhary. What are the challenges of establishing a TinyML ecosystem, Mar 2020.
[22] R. Oshana and M. Kraeling. *Software engineering for embedded systems: Methods, practical techniques, and applications.* Newnes, 2019.
[23] F. Paissan, A. Ancilotto, and E. Farella. PhiNets: a scalable backbone for low-power ai at the edge, 2021.
[24] V. J. Reddi, B. Plancher, S. Kennedy, L. Moroney, P. Warden, A. Agarwal, C. Banbury, M. Banzi, M. Bennett, B. Brown, S. Chitlangia, R. Ghosal, S. Grafman, R. Jaeger, S. Krishnan, M. Lam, D. Leiker, C. Mann, M. Mazumder, D. Pajak, D. Ramaprasad, J. E. Smith, M. Stewart, and D. Tingley. Widening access to applied machine learning with TinyML, 2021.
[25] A. Research. Global shipments of TinyML devices to reach 2.5 billion by 2030.
[26] M. Shafique, M. Naseer, T. Theocharides, C. Kyrkou, O. Mutlu, L. Orosa, and J. Choi. Robust machine learning systems: Challenges, current trends, perspectives, and the road ahead. *IEEE Design Test*, 37(2):30–57, 2020.
[27] M. Shafique, T. Theocharides, V. Reddy, and B. Murmann. TinyML: Current progress, research challenges, and future roadmap. In *2021 58th ACM/IEEE Design Automation Conference, DAC 2021*, Proceedings - Design Automation Conference, pages 1303–1306. Institute of Electrical and Electronics Engineers Inc., Dec.
[28] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.
[29] A. Sironi. *TinyML talks: Machine learning for event cameras.* tinyML, Oct 2021.
[30] S. Soro. TinyML for ubiquitous edge AI, 2021.
[31] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang. Hardware for machine learning: Challenges and opportunities. *2017 IEEE Custom Integrated Circuits Conference (CICC)*, Apr 2017.
[32] P. Warden and D. Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and ultra-low power microcontrollers.* OReilly Media Inc., 2020.
[33] D. Xu, T. Li, Y. Li, X. Su, S. Tarkoma, T. Jiang, J. Crowcroft, and P. Hui. Edge intelligence: Architectures, challenges, and applications, 2020.