UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

O ANALIZĂ COMPARATIVĂ A METODELOR DE TESTARE PENTRU SISTEMELE INTERNET-OF-THINGS

Absolvent

Mihail Feraru

Coordonator științific

Prof. dr. habil. Alin Ștefănescu

București, iunie 2022

Rezumat

Sistemele IoT sunt eterogene, distribuite și complexe datorită interconectării unui număr mare de componente hardware și software. În lucrarea de față, am argumentat că aceste caracteristici cresc dificultatea testării aspectelor funcționale și nefuncționale, testarea reprezentând un subiect de interes atât pentru cercetători, cât și pentru dezvoltatorii industriali.

În urma analizei literaturii de specialitate, am constatat lipsa unor repere obiective de comparare și analiză a tehnicilor de testare, fiecare publicație utilizând propriile aplicații, dispozitive și metrici pentru evaluare. Astfel, reproducerea și compararea rezultatelor este dificilă.

Pentru a depăși acest obstacol, am construit o suită open-source de aplicații, care să modeleze cât mai bine caracteristicile unui sistem IoT real, suita mimând o rețea a unei locuințe inteligente. Pentru a asigura caracterul eterogen, dar și comunicarea complexă, aplicațiile au fost construite de multiple entități independente (echipe de studenți, autorul prezentei lucrări, dezvoltatori independenți) și apoi integrate în fluxuri de automatizare care implică angrenarea mai multor aplicații. Suita conține o serie de defecte (bugs) naturale sau injectate artificial, din clase variate. Am prezentat clasificarea acestora atât raportat la sisteme de categorisire standard, precum Common Weakness Enumeration (CWE), dar și folosind o ierarhie proprie bazată pe nivelul de interconectare la care se manifestă defectul.

Utilizând suita construită, am efectuat experimente demonstrative, folosind diferite tehnici de testare, cum ar fi testare funcțională manuală, analiză statică a codului sursă, *fuzzing* și verificare formală. În urma experimentelor, am tras concluzia că realizarea suitei de aplicații este relevantă pentru analiza și îmbunătățirea practicilor de testare existente și că este necesară încurajarea creării de medii de test publice.

O parte din rezultatele prezentate în această lucrare au fost deja publicate în articolul realizat de Rareș Cristea, Mihail Feraru și Ciprian Păduraru (2022), „Building blocks for IoT testing - a benchmark of IoT apps and a functional testing framework”, în *4th International Workshop on Software Engineering Research & Practices for the Internet of Things*.

Îi mulțumesc domnului profesor Alin Ștefănescu pentru sprijinul fără de care realizarea acestei lucrări nu ar fi fost posibilă. Le mulțumesc, de asemenea, domnului profesor Ciprian Păduraru și doctorandului Rareș Cristea pentru sfaturile oferite și colaborarea din timpul activității mele de cercetare.

Abstract

IoT systems are heterogeneous, distributed, and complex due to a large number of interconnected software and hardware components in their composition. In this paper, we argue that given these characteristics, functional and non-functional testing of IoT systems is considerably more difficult, testing being of high interest for researchers and practitioners.

Analyzing the related work, we found a lack of objective benchmarks for comparing testing techniques, because, usually, each published paper describes a custom set of applications, devices, and evaluation metrics. Thus, reproducing and comparing results is difficult.

To overcome the presented challenges, we built an open-source set of applications, which should mimic the characteristics of a real IoT system, the set being a simulation of a smart home. To assure that we have a heterogeneous system in which complex communication is involved, the applications were built by multiple independent parties (students, the author of this paper, and other developers) and then integrated using automation flows. The application set contains multiple, real and injected, bugs of different types. We offer a custom classification based on the degree of interconnection.

Using the described set, we ran a series of experiments using testing techniques such as manual functional testing, static analysis, fuzzing, and formal verification. We concluded that the set of applications we built is relevant for advancing the state-of-the-art of IoT testing techniques.

Part of this research was already published in the paper written by Rareș Cristea, Mihail Feraru, and Ciprian Păduraru (2022), „Building blocks for IoT testing - a benchmark of IoT apps and a functional testing framework," in *4th International Workshop on Software Engineering Research & Practices for the Internet of Things*.

I would like to thank Professor Alin Ștefănescu for their support, without which this work would not have been possible. I also thank Professor Ciprian Păduraru and PhD student Rareș Cristea for their advice and collaboration during my research.

Cuprins

Capitolul 1

Introducere

Internet of Things este unul dintre subiectele de cel mai mare interes în sfera tehnologiei, alături de inteligența artificială, tehnologia blockchain și realitatea virtuală. Definiția exactă a acestui termen variază semnificativ atât în lucrările științifice, cât și în presă sau publicațiile companiilor din domeniul tehnologiei informației sau domenii adiacente. Prima dată, utilizat de Kevin Ashton într-o prezentare ținută pentru Procter & Gamble, în anul 1999 (fapt amintit în articolul aceluiași autor, anume „That 'Internet of Things' Thing"), acesta se referea la dispozitivele care cu ajutorul senzorilor dau capacitatea computerelor de a "vedea", "auzi" și "simți" mediul înconjurător. Astăzi, atunci când vorbim de IoT înglobăm o gamă largă de concepte și dispozitive: rețele wireless, electrocasnice inteligente, automatizări rezidențiale sau industriale, vehicule autonome, toate se pot încadra sub eticheta IoT.

Probabil cea mai răspândită și populară aplicare a sistemelor IoT sunt locuințele inteligente. Datorită interesului crescut pentru eficientizarea consumului de energie, reducerea emisiilor de carbon, dar și al beneficiilor promise pentru calitatea vieții, locuințele și orașele inteligente interconectate, folosind internetul au devenit un vis tehnologic atât al cetățenilor, cât și al companiilor sau guvernelor. De la iluminare cu senzori de mișcare până la asistenți virtuali care ne învață preferințele legate de muzică sau temperatură, suntem tot mai înconjurați de *lucruri* (*things*) inteligente, interconectate, ce procesează cantități enorme de date despre mediul nostru, dar și despre noi.

Deși este o nișă relativ tânără, rețelele de dispozitive inteligente își fac loc în tot mai multe industrii și domenii de activitate cu o lungă istorie. În fabricile moderne, se utilizează rețele complexe de senzori, roboți și dispozitive de coordonare pentru a facilita linii de producție. În agricultură, atât monitorizarea, cât și îngrijirea culturilor se poate realiza folosind senzori și drone conectate la internet. În sectorul public, există inițiative de digitalizare a infrastructurii și comunicării dintre instituții și cetățeni, scopul final fiind crearea de orașe cu adevărat inteligente.

Pentru a asigura reușita implementării acestor noi tehnologii, considerăm că existența unor metodologii, tehnici și unelte de testare adaptate la complexitatea și provocările

specifice sistemelor IoT este absolut necesară. Astfel, prezenta lucrare își propune să facă cunoscute cititorului principalele provocări întâlnite în testarea sistemelor IoT, ilustrând avantajele și dezavantajele mai multor metodologii întâlnite în literatura de specialitate sau industrie, prin experimente practice. Aspectul de care vom fi cel mai preocupați este testarea securității.

În acest capitol, va fi expusă, detaliat, motivația pentru alegerea temei lucrării și vom argumenta pe scurt relevanța acesteia, apoi vom continua prin parcurgerea unui scurt istoric al domeniului tratat, iar în final, va fi prezentată structura capitolelor ce urmează.

1.1 Motivația pentru temă

M-am alăturat echipei de cercetare a proiectului *Security Assessment of Smart Home Interconnected Applications (SASHA)* [1], realizat de Universitatea din București în colaborare cu Universitatea Politehnică din București, motivat fiind de provocarea de a explora securitatea sistemelor informatice într-o arie relativ tânără a tehnologiei, în care lipsesc practicile consacrate. Am avut, astfel, oportunitatea de a înțelege provocările și obstacolele întâlnite în dezvoltarea aplicațiilor IoT, în particular a celor pentru locuințe inteligente, de a explora și analiza soluțiile și practicile existente, iar mai apoi am putut contribui la realizarea unui articol de cercetare, alături de domnul profesor Ciprian Păduraru și doctorandul Rareș Cristea, anume **Building blocks for IoT testing - a benchmark of IoT apps and a functional testing framework** prezentat la **4th International Workshop on Software Engineering Research & Practices for the Internet of Things (SERP4IoT 2022)**, articol care țintește să îmbunătățească metodele de evaluare și comparare al tehnicilor de testare din domeniul Internet of Things, făcând primii pași spre construirea unui set de aplicații IoT destinate *benchmarking*-ului.

Pentru a fructifica experiența acumulată în timpul activității mele de cercetare, am decis să realizez o lucrare despre contribuția în cadrul proiectului SASHA și pentru realizarea articolului științific menționat anterior. Am dorit, de asemenea, să validez relevanța setului de aplicații propus în articol prin realizarea de experimente de comparare a câtorva tehnici de testare regăsite în literatură și industrie, o atenție sporită fiind acordată tehnicilor de testare a securității.

1.2 Scurt istoric al Internet of Things

Conceptul de dispozitive interconectate există încă din timpul apariției telegrafului, însă primele inițiative cu adevărat moderne le întâlnim abia în anii 1980 - 1990. Prima discuție cu urmări practice despre dispozitive inteligente conectate într-o rețea apare în

[1] Mai multe informații despre proiect pot fi găsite la adresa https://github.com/unibuc-cs/river/wiki/sasha.

1982, când studenții de la Carnegie Mellon University au folosit un tonomat Coca-Cola modificat și la *ARPANET* (precursorul internetului modern) pentru a raporta automat inventarul și încasările. În decursul a puțini ani, lucrări precum Weiser (1999), dar și alte publicații de presă sau academice au conturat viziunea despre dispozitivele interconectate și interacțiunea lor cu viețile noastre. Raji (1994) descrie în abstractul lucrării sale rețelele inteligente astfel (tradus din engleză):

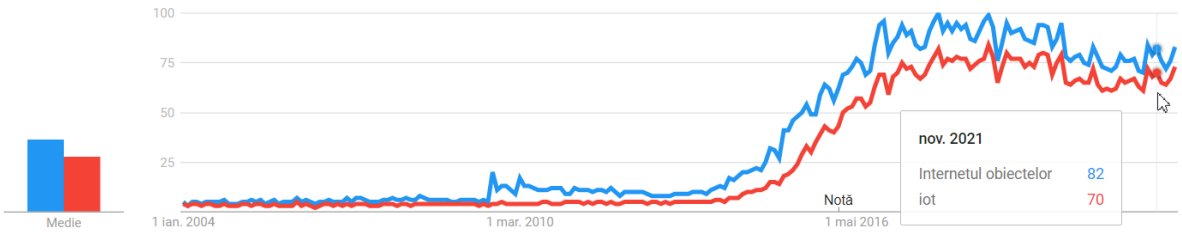
"Rețelele de control transportă pachete mici de date la un număr mare de noduri, astfel integrând și automatizând totul de la aparate casnice până la întregi fabrici. [...]"

În anii următori, numeroase companii precum Microsoft, Novell sau LG au dezvoltat produse și soluții formate din mai multe dispozitive interconectate. Termenul de Internet of Things devine consacrat abia în 1999, așa cum am menționat anterior, datorită lui Kevin Ashton însă va mai dura încă zece ani până la o creștere considerabilă a popularității.

Jackie Fenn (2011) include Internet of Things pe lista tehnologiilor *on the rise*, iar conform *Size of the global Internet of Things (IoT) market from 2009 to 2019* (f.d.) piața valora 300 de miliarde de dolari. Astăzi, valorează de aproape șase ori mai mult, 1.700 de miliarde de dolari, iar conform Google Trends este într-o explozivă creștere a popularității după anul 2015.

Astăzi, Internet of Things este unul din cele mai populare subiecte atât în presă, în publicații precum Forbes sau The Economist, cât și în mediul academic, fiind publicate din 2010 peste 400 de articole doar despre asigurarea calității în IoT conform Ahmed et al. (2019).

Figura 1.1: Statistici Google Trends pentru Internet of Things din 2004 până în prezent



1.3 Structura lucrării

Capitolul curent a oferit o perspectivă de ansamblu asupra motivației, scopului și temei lucrării. În cele ce urmează, capitolul 2 va prezenta o serie de noțiuni fundamentale pentru înțelegerea subiectelor tratate, oferind definiții și explicații pentru diverse concepte întâlnite în domeniul IoT sau al testării sistemelor informatice. Va fi analizată, apoi, literatura relevantă temei în vederea conturării obiectivelor lucrării. Capitolul 3 conține o descriere atât teoretică, cât și tehnică a sistemelor IoT, oferind definiții riguroase și

exemple de protocoale, arhitecturi și dispozitive utilizate în practică. În capitolul 4 găsim motivația pentru realizarea unui set de aplicații de evaluare pentru tehnicile de testare și descrierea implementării tehnice a unui astfel de *benchmark*. Capitolul 5 continuă prin evaluarea câtorva tehnici de testare pe setul de aplicații propus anterior. Va fi analizată o serie diversă de tehnici cum ar fi testarea funcțională și de integrare, testarea exploratorie folosind algoritmi de *fuzzing*, analiză statică pentru descoperirea de vulnerabilități în codul sursă, iar în final câteva elemente de analiză cu modelare formală. Concluziile și subiectele ce rămân deschise pentru o cercetare viitoare vor fi expuse în capitolul 6. La finalul lucrării, se vor regăsi glosarul termenilor și abrevierilor, bibliografia și anexele conținând cod sursă sau diagrame.

Capitolul 2

Preliminarii

În acest capitol, vor fi prezentate noțiuni introductive despre natura sistemelor Internet of Things (IoT), o prezentare sumară a caracteristicilor acestora și a contextului în care sunt utilizate, dar și noțiuni generale despre testare, tipurile de testare și câteva particularități pentru domeniul de interes. Vom continua prin analiza câtorva studii cantitative sistematice despre testarea IoT pentru o mai bună înțelegere a contextului, provocărilor și soluțiilor existente. În final, vom stabili obiectivele lucrării și cum vor fi acestea realizate.

2.1 Noțiuni elementare

Despre IoT

În analiza extensivă a literaturii IoT realizată de Nord, Koohang și Paliszkiewicz (2019), autorii au decis să utilizeze o definiție propusă de Ford Insights Insight (2017) (tradus din engleză):

"[...] Internet of Things (IoT) este definit ca interconectarea computerelor și dispozitivelor prin internet, dând posibilitatea de a genera date ce pot servi analizei și creării de noi operațiuni. [...]"

Alte lucrări, precum cea realizată de I. Lee și K. Lee (2015) și Huang et al. (2015), preferă definiții mai largi, considerând că orice dispozitiv fizic conectat la internet este parte a IoT. Pentru o viziune mai clară, vom considera că **IoT** este reprezentat de totalitatea dispozitivelor și computerelor conectate la internet, care colectează sau procesează date, ori interacționează cu mediul înconjurător. Astfel, acest tip de sisteme au caracter *eterogen*, pot fi *distribuite* pe spații geografice întinse, utilizează tehnologii de comunicare *wireless* și cresc rapid în complexitate o dată cu extinderea, din cauza *interconectării* unui număr mare de părți.

Un mod important de a privi sistemele IoT este să le vedem a fi Event-Driven Architectures (EDAs). Un **EDA** este un sistem construit în jurul producerii și consumului

de *evenimente*. Un eveniment reprezintă orice schimbare de stare a sistemului ca întreg. Acestea pot fi generate atât de senzori care monitorizează mediul, cât și de interacțiunea dintre om și computer sau simpla trecere a timpului. Transmiterea lor se poate face centralizat cu ajutorul unui distribuitor (*message broker*) sau descentralizat în model Peer-to-Peer (P2P). Consumatorii evenimentelor pot genera noi evenimente în urma procesării.

Definim informal un sistem *eterogen*, fiind un sistem în care părțile sale componente au în general natură diferită, concret în cazul pe care îl tratăm, eterogenitatea este dată de varietatea dispozitivelor *hardware*, tehnologiilor *software* și de interconectarea unui număr mare de astfel de componente în medii impredictibile. Astfel, putem trage concluzia că un sistem IoT este un EDA cu caracter eterogen în care software-ul este strâns legat de hardware, spre deosebire de computerele de uz general.

Despre testare

Testarea funcțională a fost introdusă conceptual de Howden (1980), acesta propunând tratarea programelor ca o colecție integrată de funcții. Practicile și viziunile asupra metodelor de testare a aspectelor funcționale s-au rafinat continuu de-a lungul anilor, majoritatea eforturilor fiind îndreptate spre testarea software în medii izolate. În general, testarea funcțională presupune testarea comportamentului unui program în diferite scenarii, ignorând detaliile de implementare, acesta fiind tratat ca un *black box*. În contextul IoT, Cortés et al. (2019) observă că majoritatea eforturilor de testare se duc spre aspectele funcționale, însă acestea reprezintă doar o singură piesă din multitudinea de proprietăți ce asigură calitatea și buna funcționare a dispozitivelor.

În cazul testării aspectelor funcționale, putem împărți testarea pe mai multe nivele, acestea fiind în general considerate: testarea unitară, adică testarea izolată a unei singure funcționalități, testarea de integrare, adică testarea funcționării corecte, atunci când mai multe părți ale sistemului sunt implicate în realizarea unei funcționalități și, în final, testarea de sistem, care evident se referă la testarea întregului sistem, acesta putând să fie compus din mai multe componente *software* și *hardware*.

Testarea aspectelor nefuncționale se concentrează asupra unei game largi de probleme cum ar fi securitatea, conectivitatea, rezistența la stres și multe altele ce pot influența în mod indirect funcționarea unui sistem.

Un interes aparte în prezenta lucrare va fi acordat testării securității, deoarece este unul din subiectele de cel mai mare interes în rândul utlizatorilor și al producătorilor, așa cum constată Ahmed et al. (2019) și I. Lee și K. Lee (2015). Compromiterea securității IoT poate duce la nefuncționarea corectă, expunerea de date confidențiale sau chiar punerea în pericol de vieți omenești în cazul infrastructurilor critice.

În contextul IoT, testarea prezintă particularități aparte, deoarece spre deosebire de

testarea în arii clasice ale dezvoltării *software*, unde nivelul *hardware* poate fi considerat sigur și suficient testat, aici avem de a face cu o legătură strânsă între *software* și *hardware*, ambele fiind insuficient testate și cu capabilități reduse.

2.2 Analiza literaturii existente

Pentru realizarea lucrării, am parcurs articolele științifice din cardul conferințelor și jurnalelor recunoscute, dar și publicații private și independente din industrie. În continuare, vom contura o imagine de ansamblu a metodologiilor, tehnicilor, practicilor și uneltelor utilizate pentru testarea sistemelor IoT, răspunzând la o serie de întrebări pe baza articolelor analizate:

Q1. Care sunt aspectele sistemelor IoT pentru care există cel mai mare interes din punct de vedere al testării, atât pentru utilizatori, cât și pentru producători?

Q2. Care sunt metodologiile și uneltele de testare curente și cum acoperă acestea nevoile constatate anterior?

Q3. Care sunt principalele obstacole întâmpinate de cercetători și dezvoltatori?

Q4. Cum putem compara obiectiv diferite metodologii și eficiența acestora în a doborî obstacolele descoperite?

Motivația pentru **Q1** este nevoia de a afla care sunt ariile în care utilizatorii și producătorii își doresc o îmbunătățire a situației curente, astfel creăm posibilitatea de valoare adăugată pentru industrie, cât și pentru mediul academic. **Q2** urmărește să stabilească starea curentă a tehnologiilor și metodologiilor, iar **Q3** să găsească principalele lipsuri ale stării curente, astfel ne putem îndrepta cercetarea spre nevoile concrete ale utilizatorilor, producătorilor și cercetătorilor. Răspunzând la **Q4**, vom înțelege care sunt pașii necesari pentru a avansa în domeniul testării IoT și cum putem înlesni acest proces și pentru alți cercetători.

Q1. Care sunt aspectele sistemelor IoT pentru care există cel mai mare interes din punct de vedere al testării, atât pentru utilizatori, cât și pentru producători?

O analiză sistematică a 478 de articole publicate în perioada 2009 - 2017 realizată de Ahmed et al. (2019), propune o taxonomie cuprinzătoare pentru cercetarea din domeniul IoT, clasele cele mai generale fiind: asigurarea calității (*en. Quality Assurance*), performanța dispozitivelor, confidențialitate și încredere, securitate și testare. Efortul de cercetare este îndreptat cu precădere spre calitatea și securitatea sistemelor IoT, aspect

tratat în 68 de lucrări. Arii similare cu un număr semnificativ de lucrări sunt *design*-ul protocoalelor și arhitecturilor sigure cu 165 de lucrări și testarea securității cu 51 de lucrări. Privind aceste cifre, putem observa o atenție sporită oferită securității, aceasta fiind prezentă în multiple categorii. Consider justificat acest interes, deoarece impactul digitalizării mediului înconjurător aduce toate riscurile asociate sistemelor informatice în viața cotidiană, dar și în infrastructuri critice. I. Lee și K. Lee (2015) susțin această opinie, considerând că numărul în creștere al dispozitivelor interconectate poate crea reacții în lanț dezastruoase în cazul unei breșe de securitate. Aceștia accentuează nevoia ca *business*-urile să ofere interes sporit și să depună efort pentru asigurarea securității și confidențialității sistemelor produse.

Observăm că securitatea este tema comună a multor lucrări, așa că o vom considera alături de funcționarea corectă a sistemelor, problema principală în IoT.

Q2. Care sunt metodologiile de testare curente și cum acoperă acestea nevoile constatate anterior?

Deși metodologiile de testare ale sistemelor IoT sunt variate și în număr mare, acestea nu au suport empiric foarte solid. Pentru metodele formale de analiză și testare cum ar fi *model-based testing* (testarea bazată pe modelare formală) sau *runtime verification* (verificarea în timpul execuției), Ahmed et al. (2019) constată că există un număr restrâns de experimente care să ateste eficiența și un număr și mai restrâns de inițiative de adoptare în industrie. Consider că una din posibilele cauze este dificultatea de a construi un set de proprietăți matematice suficient de cuprinzătoare pentru a aduce valoare suficientă, dar și efortul mare necesar pentru a implementa aceste metode de analiză. Alte metode formale de testare utilizează metode de analiză statistică a comportamentului dispozitivelor și sistemelor.

Protocoalele de comunicație reprezintă un subiect de interes pentru testare. Avem de a face atât cu verificare formală, testare aleatorie sau testarea conformității cu specificațiile. Spre deosebire de dezvoltarea software generală unde protocoalele sunt considerate a fi testate exhaustiv deja, în dezvoltarea sistemelor IoT acestea încă reprezintă un teritoriu explorat insuficient.

Interesați de testarea interoperabilității și integrării sistemelor, Bures et al. (2020) observă o serie de publicații care se concentrează pe testarea combinatorială, *path-based testing*, dar și tehnici de testare individuală clasice, care combinate pot reprezenta o soluție pentru testarea de integrare.

Cortés et al. (2019) constată că testarea de performanță, testarea funcțională și de utilizabilitate reprezintă cele mai întâlnite abordări din literatura evaluată, cu apariții în 27%, respectiv 12% și 14% din articolele analizate. De asemenea, aceștia observă o ambiguitate în ceea ce privește utilizarea termenilor de ”testare funcțională” și ”testare

de sistem" față de testarea software generală unde sunt utilizați cu mult mai multă precizie. Autorii atribuie această ambiguitate naturii eterogene și distribuite a sistemelor, propunând că noi tehnici trebuie explorate.

În sfera practică a testării, Dias et al. (2018) realizează o listă a uneltelor software și platformelor utilizate în industrie. Întâlnim unelte orientate pe testarea individuală a dispozitivelor, respectiv a codului care se execută pe sistemele *embeded*, dar și pentru testarea rețelelor. De asemenea, întâlnim soluții de testare pentru toate nivelele de la testare unitară la testare de acceptanță, însă niciuna nu oferă o acoperire integrală și nici universală, acestea fiind adesea legate de o platformă sau tehnologie anume.

Q3. Care sunt principalele obstacole întâmpinate de cercetători și dezvoltatori?

Un aspect semnalat de toate lucrările analizate este caracterul eterogen al sistemelor IoT, acestea fiind compuse din dispozitive și software produse de multiplii terți, o gamă largă de protocoale de comunicație și configurații posibile. I. Lee și K. Lee (2015) expun provocările principale ale domeniului din perspectiva dezvoltatorilor, securitatea, confidențialitatea și haosul fiind principalele provocări. Prin haos, autorii se referă la plenitudinea de standarde, protocoale, comunicații complexe și dispozitive puțin testate, ce sunt utilizate în practică în momentul de față. Acest caracter *haotic* prezintă un risc crescut de a genera evenimente negative în infrastructuri critice cum ar fi cele medicale sau industriale. O altă perspectivă practică este prezentată de Dias et al. (2018), care concluzionează că există o lipsă de unelte software destinate pentru testarea sistemelor eterogene și distribuite în mod automat, majoritatea soluțiilor fiind specializate pentru un anume set de tehnologii sau un singur manufacturier. În plus, lipsesc soluții de testare pentru caractere nefuncționale ale sistemelor cum ar fi securitatea, confidențialitatea sau *management*-ul actualizărilor software.

Atât Cortés et al. (2019), cât și Ahmed et al. (2019) semnalează necesitatea, dar și dificultatea creării de noi tehnici și metodologii pentru testarea sistemelor eterogene și distribuite, deoarece cele pentru sistemele tradiționale nu sunt eficiente sau aplicabile pentru industria IoT. Încă o provocare este reprezentată de compromisul dintre securitate și optimizarea costurilor de producție a dispozitivelor, *hardware*-ul mai ieftin nu dispune de capabilități de securitate foarte avansate, producătorii fiind astfel puși în dificultate. Spre deosebire de sistemele clasice, unde considerăm protocoalele și *hardware*-ul suficient testate, în dezvoltarea produselor IoT, aceste nivele nu sunt încă suficient acoperite.

Q4. Cum putem compara obiectiv diferite metodologii și eficiența acestora în a doborî obstacolele descoperite?

Deși articolele analizate conțin o viziune cuprinzătoare asupra tehnicilor, metodologiilor și soluțiilor de testare, niciunul din acestea nu menționează metode de evaluare și comparare obiectivă. Luând în considerare lipsa de date empirice pentru validarea diferitelor metode de testare, putem deduce că există o necesitate pentru stabilirea unui cadru de comparare și evaluare obiectivă pus la dispoziție cercetătorilor. Fără acest cadru, progresul spre doborârea provocărilor impuse de natura domeniului nu poate fi cuantificat cu ușurință. Acest lucru lasă loc aprecierilor calitative, care sunt deseori subiective și au o contribuție mai mică la cunoașterea științifică decât cele cantitative.

Observăm că și în alte arii există necesitatea metodelor obiective de evaluare și comparare, de exemplu, în testarea aplicațiilor *web*, Garousi et al. (2013) constată că fiecare articol publicat utilizează un alt set de aplicații pentru evaluarea tehnicii de testare propuse, astfel compararea obiectivă a articolelor devine dificilă. Alt exemplu este reprezentat de Brockman et al. (2016) în sfera *reinforcement learning*, autorii propunând un mediu unitar cu multiple probleme de reper (*en. benchmark*) pentru evaluarea și compararea algoritmilor propuși de cercetători.

2.3 Obiective

În urma analizei făcute asupra literaturii existente, am identificat o serie de caracteristici definitorii pentru sistemele Internet of Things, acestea sunt eterogen, distribuite, prezintă interacțiuni complexe, produc și procesează cantități mari de date și sunt integrate în medii fizice. Toate acestea aduc o serie unică de provocări pentru dezvoltarea și testarea atât a *software*--ului, cât și a *hardware*-ului. Aceste provocări sunt dificile și puțin explorate în comparație cu provocările din ariile clasice de dezvoltare *software*, precum dezvoltarea aplicațiilor web.

Pentru a contribui la cunoștințele din domeniul testării sistemelor IoT, lucrarea de față își propune să realizeze următoarele obiective:

- să stabilească un cadru teoretic și tehnologic pentru sistemele IoT, familiarizând cititorul cu caracteristicile, metodele de proiectare, dezvoltare și testare ale acestor sisteme, precum și obstacolele și provocările existente (în capitolul 3);
- să prezinte un set de aplicații IoT construit pentru a servi la compararea metodelor de testare și să argumenteze relevanța acestuia (în capitolul 4);
- să exemplifice utilizarea setului de aplicații pentru evaluarea diferitelor tehnici de testare întâlnite în literatură sau practică, să discute avantajele și dezavantajele lor și să constate eficiența lor (în capitolul 5).

Capitolul 3

Modelarea sistemelor Internet of Things

Pentru o analiză cât mai riguroasă este nevoie de o înțelegere profundă a sistemelor testate. În acest capitol, vom oferi o imagine de ansamblu asupra rețelelor IoT, ilustrând caracteristicile acestora cu exemple reale din diferite industrii. Pe baza exemplelor expuse, vom discuta despre caracteristicile arhitecturale ale rețelelor, diferitele părți componente, topologiile comune și care este parcursul fluxurilor de date.

Arhitecturile specifice IoT impun anumite constrângeri metodelor de comunicație, așa că vom explora care sunt cele mai comune protocoale de comunicație utilizate, care sunt particularitățile lor și care este motivația din spatele utilizării acestora. Nu ne vom rezuma doar la protocoalele dintr-un singur nivel al modelului Open Systems Interconnection (OSI) [1], ci vom urmări o privire de ansamblu.

Canalele de comunicație au nevoie de părți comunicante, așa că vom continua prin a analiza dispozitivele și caracteristicile acestora utilizate în sistemele IoT, un punct important în această discuție fiind despre limitările hardware impuse de eficiența costurilor și cum acest aspect afectează capabilitățile de securitate. Deoarece software-ul este strâns legat de hardware în acest domeniu, discuția va fi extinsă și asupra interacțiunii dintre cele două, în special despre modul în care dezvoltarea sau testarea uneia dintre componente o afectează pe cealaltă.

În finalul capitolului, vom trece din planul tehnic și tehnologic în cel teoretic și vom modela formal și riguros rețelele IoT, folosindu-ne de teoria grafurilor și teoria automatelor finite. Vor fi descrise atât topologiile și fluxurile de date fizice, cât și fluxurile la nivel logic. Ne propunem să obținem o viziune clară asupra proprietăților acestor sisteme, pentru a putea exprima mai ușor aspectele testate și pentru a deschide calea spre metode de testare formală, cum ar fi cele bazate pe rezolvarea de Satisfiability Modulo Theories (SMTs) [2].

[1] OSI este un model conceptual folosit pentru descrierea părților componente ale unui sistem de telecomunicații.

[2] Problemele de satisfiabilitate sunt probleme de algebră booleană, care urmăresc a determina dacă o expresie se poate satisface sau nu. Mai multe detalii despre SMTs vor fi oferite în capitolul 5.

3.1 Arhitectură

Așa cum ne spun Khodadadi, Dastjerdi și Buyya (2016), componentele fundamentale și arhicunoscute ale sistemelor IoT sunt: dispozitivele cu capabilități senzoriale, apelarea de servicii de la distanță, comunicarea peste rețea și procesarea de evenimente într-un context specific. Aceste părți componente, puse împreună, creează sisteme cu caracter puternic eterogen, așa cum am menționat și anterior. Prin urmare, asigurarea interconectivității și interoperabilității devine un obiectiv greu de atins în lipsa unor procedee corespunzătoare de abstractizare. Aceste procedee de abstractizare se regăsesc în modelele arhitecturale propuse pentru sistemele IoT, modele pe care le vom analiza în paragrafele următoare, dar nu înainte de a arunca o scurtă privire asupra câtorva exemple practice.

Pentru a înțelege mai bine nevoia de abstractizare, să analizăm câteva exemple din lumea reală. Ne putem imagina o locuință inteligentă, în care avem instalați multipli senzori pentru temperatură, luminozitate, mișcare etc. Aceștia transmit date către un computer central, care le agregă și le pune la dispoziție utilizatorilor locuinței. De asemenea, utilizatorii își pot stabili pe baza datelor colectate o serie de automatizări, cum ar fi mișcarea draperiilor, reglarea temperaturii, încuierea ușii, etc. Toate facilitățile pot fi acționate de la distanță prin intermediul internetului. În plus, locuința este conectată la sistemul inteligent al municipalității pentru a comunica informații despre consumul de energie, apă sau alte resurse. În cadrul sistemului municipal, scenariul se repetă. Regăsim o serie de senzori, centre de comandă și dispozitive de acțiune. Este evident cât de rapid crește complexitatea, chiar și într-un caz relativ restrâns.

Un alt exemplu poate fi reprezentat de o linie de producție, industrială complet automatizată. Multiplii senzori colectează date despre funcționarea aparaturii. Unul sau mai multe centre de comandă interpretează datele pentru a orchestra brațele robotice din cadrul liniei de producție. Angajații respectivei fabrici pot interacționa cu sistemele acesteia pentru monitorizare și control manual atunci când este nevoie. Toate datele colectate sunt trimise către *cloud*-ul companiei pentru analiza și optimizarea proceselor de producție. Acest gen de sisteme reprezintă o nișă specifică, anume Industrial Internet of Things (IIOT) și putem privi o ilustrare a unei astfel de rețele în figura 3.1.

În continuare, ne vom concentra atenția asupra a trei modele arhitecturale populare în IoT, care oferă o abstractizare suficient de robustă pentru cazurile prezentate mai sus. Vor urma, astfel, în ordinea complexității, arhitectura *3-layer*, *5-layer* și Service Oriented Architecture.

3-layer

Wu et al. (2010) spune că în ciuda lipsei unei definiții unificate pentru IoT, arhitectura *3-layer* este larg acceptată și cunoscută. Acest fapt este bine fundamentat, deoarece găsim referințe ale acestui model arhitectural în multiple publicații la care ne vom re-

Figura 3.1: Ilustrare a unei simple rețele IIoT

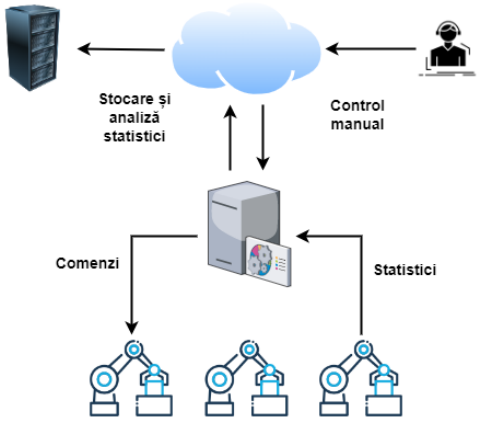
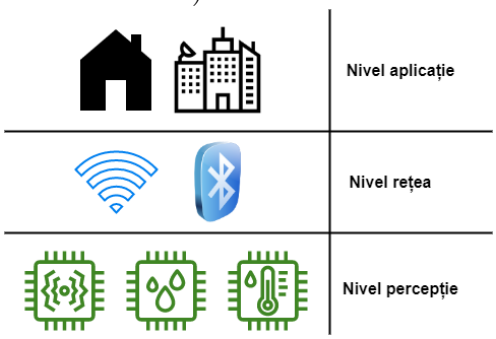


Figura 3.2: Arhitectura *3-layer* (ilustrație adaptată după figura 1 din articolul „Internet of Things (IoT) Security: Current Status, Challenges and Countermeasures")



feri în paragrafele următoare. Mai jos sunt enumerate nivelurile componente ale acestei arhitecturi, iar în figura 3.2 regăsim și o reprezentare grafică.

1. Nivelul de percepție (*perception layer*)
2. Nivelul rețelei (*network layer*)
3. Nivelul aplicației (*application layer*)

Nivelul de percepție conține senzori, cititoare de RFID sau coduri de bare, sisteme Global Positioning System (GPS) sau camere de filmat. Acesta facilitează colectarea de date despre mediul înconjurător al sistemului IoT. Identificăm cu ușurință acest nivel în exemplul nostru anterior, senzorii termici sau de mișcare, cât și dispozitivele de măsurat consumul de energie sau apă fac parte din acesta. Acest nivel poate fi considerat a fi colecția de *things* din IoT.

La nivelul rețelei, întâlnim infrastructura care pune la dispoziție comunicarea dintre *things*. Această infrastructură este formată din routere, *switch*-uri, centre de procesare și management, etc. De exemplu, într-o locuință inteligentă rețeaua *wireless* și centrul de control fac parte din nivelul de rețea.

Procesarea datelor și executarea fluxurilor specifice se petrece la nivelul aplicației. Acest nivel reprezintă totalitatea componentelor software și hardware, care facilitează serviciile dorite de utilizator. De exemplu, software-ul de automatizare, prezent pe centrul de comandă dintr-o locuință inteligentă, face parte din acest nivel.

Lin et al. (2017) constată că, deși această arhitectură este fundamentală și simplă, funcționalitățile și operațiunile desfășurate de sistem la nivelul rețelei și aplicației sunt diverse și complexe. În opinia autorilor, este nevoie de dezvoltarea unui nivel de serviciu (en. *service layer*) care să medieze interacțiunea dintre rețea și aplicație. Acest nivel adițional ar facilita o mai mare flexibilitate sistemului.

5-layer

Arhitectura *3-layer* nu conține metode de management suficient de bune și nu ia în considerare nevoia de a modela domeniile de *business* în perspectiva lui Wu et al. (2010). Pentru a combate aceste dificultăți, autorii propun o arhitectura cu cinci nivele, după cum urmează:

1. Nivelul de percepție (*perception layer*);
2. Nivelul de transport (*transport layer*);
3. Nivelul de procesare (*processing layer*);
4. Nivelul aplicației (*application layer*);
5. Nivelul *business* (*business layer*).

Primele două nivele corespund cu cele din arhitectura *3-layer*, îndeplinind aceleași funcții. În continuare, nivelul de procesare preia o parte din atribuțiile nivelului aplicației din arhitectura anterioară. Din acest nivel, fac parte bazele de date, *cloud*-ul și toate celelalte aparaturi de procesare de date pentru a facilita utilizarea acestora la nivelul aplicației. În viziunea autorilor, această separare dintre procesare și utilizare în scopuri aplicate este necesară pentru flexibilitatea și scalabilitatea IoT.

În final, avem de a face cu nivelul de *business*, care ar trebui să reprezinte un nivel de management peste nivelul aplicației, fiind preocupat de modelul de *business* și dezvoltarea pe termen lung. Din păcate, autorii nu pun la dispoziție o descriere detaliată, aplicarea acestui tip de arhitectură rămânând la latitudinea dezvoltatorului. O imagine mai clară asupra nivelului de *business* ne-o oferă Khan et al. (2012), specificând că acesta conține modelele de *business*, grafice și organigrame bazate pe datele obținute din nivelul de aplicație. Scopul acestui nivel este să producă strategii și decizii de *business*, deci este un nivel de interacțiune între om și mașină.

Service Oriented Architecture

O arhitectură orientată pe servicii, în orginal Service Oriented Architecture (SOA), propune împărțirea componentelor și funcționalităților unui sistem în unități mici și independente, numite servicii. Această arhitectură nu impune un număr fix de nivele, însă trebuie să existe cel puțin un nivel care facilitează două activități fundamentale: descoperirea serviciilor (*service discovery*) și invocarea acestora (*service calling*).

Khodadadi, Dastjerdi și Buyya (2016) susțin că acest tip de arhitectură asigură interoperabilitatea dispozitivelor eterogene, această proprietate fiind esențială pentru sistemele IoT. De asemenea, oferă și un exemplu de SOA, adăugând un nivel de servicii în clasica

arhitectură *3-layer*. Funcționarea independentă a serviciilor asigură că sistemul va continua să funcționeze în general corect, chiar dacă un număr mic de servicii a încetat să funcționeze. Acest aspect este foarte important pentru fiabilitatea sistemelor. Același tip de arhitectură este descris și de Lin et al. (2017) într-un meta-studiu.

O altă perspectivă asupra unui posibil tip de SOA vine din partea lui Tan și N. Wang (2010), care propun o arhitectură *5-layer* modificată. Pe lângă nivelul rețelei, aceștia introduc un nivel de coordonare al serviciilor, care are rolul de a unifica comunicarea, asigurând interoperabilitatea. Din păcate, nu sunt oferite detalii suplimentare despre implementarea acestui tip de arhitectură.

3.2 Protocoale de comunicație

În discuția anterioară despre arhitectura sistemelor IoT am observat că un rol important este jucat de mecanismele de comunicare. Aceste sisteme sunt nevoite să coordoneze un număr mare de dispozitive și să transmită cantități enorme de date într-un mod fiabil, rapid și rezilient. De aceea sunt utilizate o gamă largă de protocoale de comunicație, fiecare cu avantajele și dezavantajele sale, în încercarea de a depăși provocările impuse de natura distribuită și eterogenă a sistemelor. În rândurile următoare, vor fi analizate o serie de protocoale utilizate în mod uzual. Vom trata protocoalele în ordine crescătoare, conform situării lor în modelul OSI. O ilustrație a acestui model se găsește în anexa B, figura B.1.

Deoarece majoritatea comunicațiilor se desfășoară în mod *wireless*, un protocol foarte popular pentru nivelul fizic și *data link* este *IEEE 802.15.4*. Așa cum ne spun Lin et al. (2017), *IEEE 802.15.4* este un protocol pentru comunicații cu consum scăzut de energie și rată mică de transfer. Datorită versatilității lui, acest protocol reprezintă baza pentru alte protocoale de nivel mai înalt precum *ZigBee*, *6LoWPAN* sau *WirelessHART*, despre care vom vorbi în rândurile următoare.

Construit peste *IEEE 802.15.4*, *6LoWPAN* acoperă și nivelul de rețea OSI, folosind adresare *IPv6*, astfel poate să ofere suport pentru rețele cu cost și consum de energie scăzute pentru un număr foarte mare de dispozitive IoT.

ZigBee acoperă aproape toate nivelurile OSI, și anume: nivelul fizic, nivelul *data link*, nivelul de rețea, de transport și cel al aplicației. Un avantaj foarte important al acestui protocol este posibilitatea de a crea topologii de rețea variate. Fiecare dispozitiv (nod) al rețelei poate juca rol de coordonator, router sau aplicație. Coordonatorul este punctul central al rețelei și atribuie rolurile router, acestea fiind responsabile pentru transmiterea pachetelor spre dispozitivele potrivite. Câteva exemple de topologii *ZigBee* se pot observa în anexa B, figura B.2. O alternativă a acestui protocol este *Z-Wave*, un protocol mult mai simplu, destinat rețelelor de mici dimensiuni (de până la 232 de noduri), oferind caracteristici similare în privința consumului de energie și ratelor de transfer de date.

Autointitulat protocolul principal pentru IoT, Message Queue Telemetry Transport (MQTT) este un protocol situat la nivelurile de sesiune și aplicație al OSI, destinat transmiterii de mesaje folosind modelul *producător-consumator* (en. *publisher-subscriber*). Acesta oferă relizientă și scalabilitate, fiind în același timp un protocol "ușor" din punct de vedere al resurselor computaționale necesare. De asemenea, este potrivit pentru sistemele de tip EDA, categorie în care sistemele IoT se încadrează, după cum am discutat în capitolul al doilea. O alternativă pentru acest protocol este The Advanced Message Queuing Protocol (AMQP), însă acesta este mult mai complex și consumă, în general, mai multe resurse computaționale.

Probabil cel mai comun și cunoscut protocol de comunicație folosit în internet este Hypertext Transfer Protocol (HTTP). Acesta este utilizat și în cadrul sistemelor IoT, însă din cauza complexității acestuia și a faptului că este mult prea detaliat în specificarea cererilor, acesta nu este alegerea cea mai populară. În schimb, protocolul Constrained Application Protocol (CoAP) este o versiune simplificată și optimizată pentru IoT a protocolului HTTP.

3.3 Caracteristicile dispozitivelor hardware

Am menționat anterior că spre deosebire de alte subdomenii ale tehnologiei informației, în IoT, software-ul este în strânsă legătură cu hardware-ul, uneori fiind inseparabile, deoarece nu există abstractizări hardware comune, utilizate de mai mulți producători. Vom menționa pe scurt care sunt principalele caracteristici ale dispozitivelor pe care le întâlnim în sistemele din piață:

- **putere computațională mică** - senzorii, dispozitivele de control etc. au, în general, resurse computaționale foarte reduse, adică procesoare cu frecvențe de ordinul megahertzilor și memorii de ordinul *kilobytes*-ilor;
- **lipsa capabilităților avansate de securitate** - facilități standard pe arhitecturi de procesoare pentru computere de uz personal, precum memoria neexecutabilă, unitatea de management a memoriei, etc., sunt inexistente pe foarte multe dispozitive IoT;
- **lipsa de transparență** - codul sursă al programelor care rulează pe aceste dispozitive nu este disponibil, iar în multe cazuri, nu pot fi extrase programele în format binar (*firmware*-ul nu este disponibil pentru public).

3.4 Model teoretic

În încercarea de a crea un cadru obiectiv pentru analiza sistemelor IoT din punct de vedere al proprietăților și potențialelor defecte din acestea, vom introduce câteva noțiuni teoretice pentru descrierea acestor sisteme. În secțiunea anterioară, am discutat despre comunicarea fizică a dispozitivelor, diferitele topologii pe care le-am putea întâlni și protocoalele utilizate. Vom extinde discuția spre fluxurile logice prezente în aceste rețele, deoarece, deși topologiile de tip stea sunt des întâlnite, nodul central servește ca un intermediar pentru fluxuri logice între un număr mare de dispozitive, acestea neputând fi reduse la o interacțiune nod central - dispozitiv.

Vom folosi ca fundație specificația propusă de Păduraru, Cristea și Stăniloiu (2021), deoarece propune o descriere formală a rețelelor IoT, folosind teoria grafurilor pentru a ușura automatizarea proceselor de testare, fără a fi nevoie să se specifice detalii tehnice, precum protocoalele sau software-ul utilizat. Autorii propun să vizualizăm o aplicație IoT ca o serie de perechi *input-output*, care pot fi descrise de un graf orientat $G = (V, E)$ în modul următor:

1. orice vârf $v \in V$ reprezintă un dispozitiv fizic, pe care se execută unul sau mai multe procese software;

2. o muchie orientată $e \in E$ de la un vârf v_1 la un vârf v_2 descrie un flux de date de la *output*-ul lui v_1 spre *input-ul* lui v_2;

3. orice vârf $v \in V$ este consumatorul datelor pentru un set de producători:

$$V_{in}(v) = \left\{ v_{in} \;\middle|\; \exists v \in V \wedge (v_{in}, v) \in E \right\}$$

4. orice vârf $v \in V$ este producător pentru un set de consumatori:

$$V_{out}(v) = \left\{ v_{out} \;\middle|\; \exists v \in V \wedge (v, v_{out}) \in E \right\}$$

5. orice vârf $v \in V$ este caracterizat de un *buffer* de *input* și unul de *output*, aceștia fiind descriși de relația între producător și consumator, astfel încât:

$$\text{Buffer}_{in}(v) = \bigcup_{v_{in} \in V_{in}(v)} \text{Buffer}_{out}(v_{in})$$
$$\text{Buffer}_{out}(v) = \bigcup_{v_{out} \in V_{out}(v)} \text{Buffer}_{in}(v_{out})$$

6. graful G este dinamic, vârfuri și muchii putând fi adăugate sau eliminate în timpul funcționării rețelei.

Un aspect important, discutat de autori, este că muchiile rețelei trebuie privite dintr-o perspectivă probabilistă, anume ele pot lipsi în timpul funcționării sistemului. De exemplu, un senzor ar putea fi deconectat, fără energie sau să comunice date la intervale foarte lungi de timp. O viziune mai clară asupra acestui aspect o putem avea privind figura 3.3. În această figură, fluxurile de date "curg" de la senzori spre nodurile de *output* prin intermediul punctului central, *hub*-ul rețelei. Observăm că unele muchii sau vârfuri din graf sunt obligatorii pentru funcționarea sistemului, eliminarea lor făcând imposibilă o execuție coerentă. Acest set de vârfuri și muchii obligatorii sunt notate V_r și E_r și determină graful funcțional minimal G_r. (ie. $r \rightarrow \textit{required}$)

Dacă considerăm orice graf $G \in G_{total}$ o stare posibilă a rețelei (ie. G conține toate muchiile și vârfurile obligatorii), observăm că avem de a face cu un automat finit. Putem descrie stările acestui automat, considerând $S = \left\{ G \;\middle|\; G \in G_{total} \wedge G_r \subseteq G \right\}$ și operațiile posibile de modificare ale grafului cărora le vom asocia următorul alfabet:

$$E_e^{\oplus} := \text{adăugarea muchiei } e \quad E_e^{\ominus} := \text{ștergerea muchiei } e$$
$$V_v^{\oplus} := \text{adăugarea vârfului } v \quad V_v^{\ominus} := \text{ștergerea vârfului } v$$

$$\Sigma = \left\{ E_e^{\oplus} \;\middle|\; e \in E_{total} \right\} \bigcup \left\{ E_e^{\ominus} \;\middle|\; e \in E_{total} \right\} \bigcup$$
$$\bigcup \left\{ V_v^{\oplus} \;\middle|\; v \in V_{total} \right\} \bigcup \left\{ V_v^{\ominus} \;\middle|\; v \in V_{total} \right\}$$

Folosind alfabetul definit mai sus, avem următoarea funcție de tranziție:

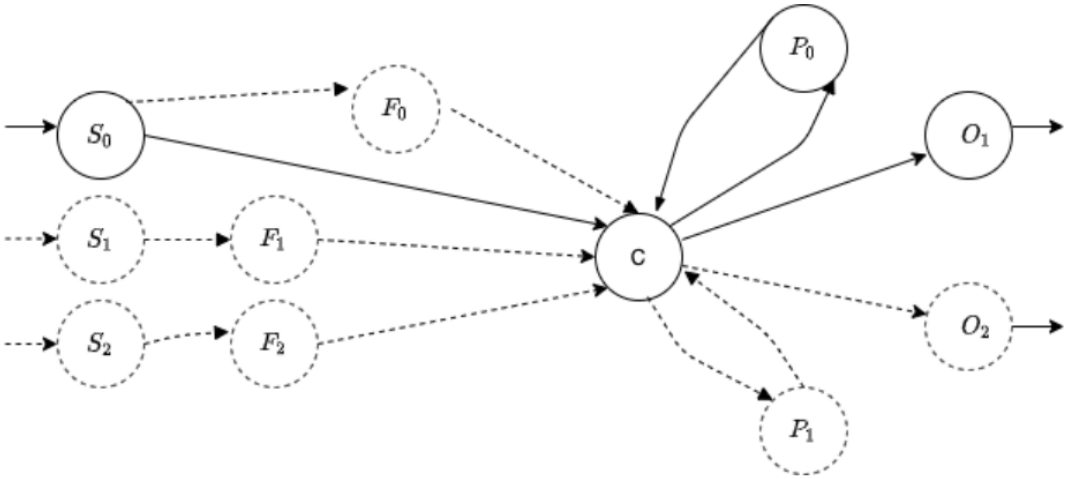
$$\delta(G = (V, E), E_e^{\oplus}) = G' = (V, E \cup \{e\}) \quad \delta(G = (V, E), E_e^{\ominus}) = G' = (V, E \backslash \{e\})$$
$$\delta(G = (V, E), V_v^{\oplus}) = G' = (V \cup \{v\}, E) \quad \delta(G = (V, E), V_v^{\ominus}) = G' = (V \backslash \{v\}, E)$$

Observăm că numărul de stări posibile ale acestui automat crește exponențial, mai exact proporțional cu numărul de submulțimi de vârfuri și muchii valide (totuși, număr mai mic decât $2^{|V|+|E|}$, deoarece nu toate combinațiile de vârfuri și muchii sunt valide).

O descriere mult mai precisă a sistemului ar putea fi reprezentată de utilizarea lanțurilor Markov, deoarece muchiile opționale au probabilități asociate, însă această abordare depășește scopul lucrării de față.

Descrierea propusă de Păduraru, Cristea și Stăniloiu (2021) se concentrează pe fluxurile fizice de date dintre dispozitive, însă un aspect important al acestor tipuri de sisteme este reprezentat de fluxurile logice de date, așa cum am văzut în discuția anterioară despre arhitecturi și protocoale, datele colectate de un senzor sau generate de un utilizator pot influența multiple părți ale sistemului, de la simplii actuatori până la sisteme complexe de analiză și decizie.

Figura 3.3: Ilustrarea grafului orientat asociat unei rețele IoT, unde toate S_i sunt senzori (*input*), toate F_i sunt noduri de "filtrare", P_i sunt noduri de procesare, iar O_i noduri de ieșire (*output*). În centrul rețelei se află nodul central, denumit și *hub*. Vârfurile și muchiile punctate reprezintă părți ale rețelei care pot lipsi. (Ilustrație preluată din articolul „RiverIoT - a Framework Proposal for Fuzzing IoT Applications" al lui Păduraru, Cristea și Stăniloiu (2021))



Într-o manieră similară celei de mai sus, putem descrie graful de fluxuri logice G_L care conține aceleași vârfuri, însă nu și aceleași muchii. Deoarece abordarea este foarte similară, nu vom repeta notațiile propuse mai sus, dar vom analiza un scurt exemplu ilustrativ. Să presupunem că în figura 3.3 există o dependență logică între vârfurile S_2, P_1 și O_2. Atunci, în graful G vor fi prezente muchiile: S_2F_2, F_2C, CP_1, P_1C și CO_2, deoarece datele nu pot urma alt drum fizic fără intermediarii F_2 și C. Pe de altă parte, în graful G_L vor apărea doar muchiile determinate de drumul $S_2P_1O_2$, deoarece doar acestea iau parte la procesarea logică a fluxului de date.

Concluzionăm că am atins primul obiectiv al lucrării, anume stabilirea unui cadru tehnologic, dar și teoretic pentru sistemele IoT. Vom continua în capitolul următor cu descrierea modului în care am construit o suită de aplicații IoT, care să ilustreze cât mai bine caracteristicile tratate. Această suită va fi utilizată în finalul lucrării pentru evaluarea câtorva tehnici de testare.

Capitolul 4

Construirea unei suite de aplicații pentru evaluarea tehnicilor de testare

După cum am argumentat în capitolele precedente, sistemele IoT sunt complexe, eterogene și distribuite, caracteristici care aduc o serie de dificultăți greu de depășit în testare. Am oferit o argumentare detaliată asupra acestor aspecte în secțiunea **2.2.Q3**. Pentru a avansa în ce privește starea curentă a tehnicilor de testare, avem nevoie de metrici [1] obiective, care ar ușura munca cercetătorilor de a reproduce rezultate și de a le compara. În urma analizei literaturii din secțiunea **2.2.Q4**, am constatat imposibilitatea stabilirii unor astfel de metrici, deoarece majoritatea publicațiilor utilizează o suită proprie de aplicații pentru desfășurarea experimentelor. Astfel, a compara numărul de defecte descoperite, timpul sau alți parametri este irelevant, deoarece mediul variază.

Regăsim aceeași concluzie enunțată de Păduraru, Cristea și Stăniloiu (2021), în articolul lor, care propune o specificație formală pentru descrierea sistemelor IoT. Pentru evaluarea riguroasă a specificației propuse este nevoie de construirea unei suite de aplicații cât mai apropiate de realitate. Această inițiativă este materializată și prezentată de lucrarea lui Cristea, Feraru și Păduraru (2022). Aceasta va fi prezentată în paginile ce urmează, evidențiind contribuțiile proprii.

Un aspect important a suitei de aplicații construit este faptul că este *open-source*. Într-o industrie în care majoritatea producătorilor nu pun la dispoziție decât sursele deja compilate ale *firmware*-ului, iar în multe cazuri acestea se află pe dispozitive ce nu permit accesarea lui, existența *software*-ului *open-source* este crucială pentru a crea un mediu transparent pentru experimente.

O observație importantă este că suita de aplicații este realizată în mare măsură imitând aplicațiile reale dintr-o locuință inteligentă, deoarece este unul din cele mai întâlnite scenarii de utilizare a tehnologiilor IoT, și pentru că oferă un grad de complexitate și interconectare suficient de mic încât să fie realizat fără eforturi majore, dar și suficient de

[1] În acest context, „metrici” se referă la metode de măsurare a rezultatelor unor experimente.

mare, încât să mimeze comportamentul unui sistem real, aducând același tip de provocări.

În prezentul capitol, vom discuta despre modul de construire al suitei de aplicații și particularitățile fiecărei aplicații în parte, prezentând funcționalitatea și tehnologiile utilizate pentru realizarea ei. Apoi, ne vom concentra asupra infrastructurii de interconectare a aplicațiilor și vom discuta despre protocoalele utilizate și motivația alegerii lor. Pentru a ilustra o situație cât mai apropiată de realitate, aplicațiile sunt angrenate în fluxuri de automatizări orchestrate de o unitate centrală, analiza acestor fluxuri fiind de asemenea prezentă în acest capitol. După înțelegerea structurală a setului de aplicații, vom vedea cum îl putem folosi pentru compararea și analizarea tehnicilor de testare și cum poate fi acesta util pentru cercetători sau profesioniști. În final, ne vom concentra asupra limitărilor și posibilelor îmbunătățiri ce ar putea fi aduse în viitor setului de aplicații.

4.1 Descrierea aplicațiilor

Pentru a ilustra cât mai bine o situație reală, aplicațiile utilizate au surse multiple, o parte fiind dezvoltate de echipe independente de studenți, o altă parte au fost scrise de echipa de cercetare SASHA, iar altele au fost obținute din surse publice. Au fost aduse modificări pentru a facilita integrarea, dar și pentru a introduce defecte (*en. bugs*) artificiale pe lângă cele deja existente. Această diversitate se regăsește și în sectorul comercial, acolo unde dispozitive și software de la producători diferiți sunt utilizate în medii complexe și condiții impredictibile. Considerăm că această abordare de construire a aplicațiilor este suficient de robustă și suficient de apropiată de realitate.

În paragrafele următoare, vom analiza caracteristicile fiecărei aplicații [2], caracteristici precum modul de funcționare, tehnologiile utilizate pentru dezvoltare, dependența și interacțiunea față de alte aplicații sau relevanța pentru evaluarea tehnicilor de testare. Toate aplicațiile simulează procese des întâlnite în locuințele inteligente.

FlowerPower

Descriere: Un *ghiveci inteligent* care expune prin intermediului unui API supravegherea și îngrijirea plantelor. De asemenea, poate notifica utilizatorul în legătură cu anumite schimbări legate de starea plantelor.

Tehnologii utilizate: *C++* și *GNU Makefiles* pentru dezvoltare, împreună cu bibliotecile *Pistache* pentru servirea de cereri Hypertext Transfer Protocol (HTTP), *RapidJson* pentru lucrul cu date în format JavaScript Object Notation (JSON) și *Mosquitto* pentru comunicarea cu un distribuitor de mesaje prin protocolul Message Queue Telemetry Transport (MQTT).

[2] Codul sursă al aplicațiilor, infrastructura și documentația acestora pot fi găsite la https://github.com/unibuc-cs/IoT-application-set.

În context: Interacționează cu aplicația WindWow în legătură cu aspecte precum temperatura sau luminozitatea.

Sursă: Echipă de studenți înscriși la cursul de Ingineria Programării, anul 2020.

WindWow

Descriere: O fereastră inteligentă ce permite monitorizarea temperaturii și luminozității din cameră, luminozitatea fiind reglabilă prin intermediul draperiilor acționate de utilizator de la distanță.

Tehnologii utilizate: *C++* și *CMake* pentru dezvoltare, împreună cu bibliotecile *Pistache*, *NLohmann-JSON* pentru lucrul cu date în format JSON și *Mosquitto*.

În context: Senzorii de luminozitate și temperatură colectează date relevante pentru o multitudine de alte dispozitive din rețea, cum ar fi SmartFlower și SmartTV, care își pot calibra funcționarea în funcție de datele primite.

Sursă: Echipă de studenți înscriși la cursul de Ingineria Programării, anul 2020.

Smarteeth

Descriere: O periuță electrică ce permite colectarea de statistici de sănătate și multiple programe de utilizare.

Tehnologii utilizate: *C++* și *CMake* pentru dezvoltare, *Paho* pentru lucrul cu date în format JSON și *Mosquitto*.

În context: Dispozitivul este relativ izolat, interacționează minimal, doar cu SmartKettle.

Sursă: Echipă de studenți înscriși la cursul de Ingineria Programării, anul 2020.

SmartKettle

Descriere: Fierbător inteligent dotat cu senzori pentru temperatură și vâscozitate, poate fi programat pentru a prepara băuturi în mod recurent.

Tehnologii utilizate: *C++*, *Bash scripting*, *Makefiles*, *NLohmann-JSON* și *Pistache*.

În context: Preia date legate de temperatură de la WindWow pentru a determina temperatura de preparare a băuturilor.

Sursă: Echipă de studenți înscriși la cursul de Ingineria Programării, anul 2020.

SmartTV

Descriere: TV inteligent, programabil, dotat cu sistem de recomandări personalizabil pentru fiecare utilizator.

Tehnologii utilizate: *C++*, *CMake*, *NLohmann-JSON* și *Pistache*.

În context: Își reglează luminozitatea conform datelor primite de la senzorii aplicației WindWow.

Sursă: Echipă de studenți înscriși la cursul de Ingineria Programării, anul 2020.

SeraSmart

Descriere: Seră inteligentă pentru îngrijirea plantelor asistată de computer, dotată cu senzori similari cu cei ai aplicației FlowerPower.

Tehnologii utilizate: *Python 3.7*, *Flask* pentru *server*-ul HTTP, *PyYAML*, *Mosquitto* și *SQLite*.

Sursă: Echipă de studenți înscriși la cursul de Ingineria Programării, anul 2021.

SoundSystem

Descriere: Music player programabil, cu acces la internet și spațiu de stocare propriu.

Tehnologii utilizate: *Golang* pentru dezvoltare, *Echo Library* pentru *server*-ul HTTP și *Paho*.

În context: Preia comenzi directe de la utilizatori și informează aplicația Hub despre starea sa curentă.

Sursă: Autorul prezentei lucrări

PhilipsHue Simulator

Descriere: O aplicație formată din mai multe componente: leduri, becuri, un TV și *hub*-uri de sincronizare. Simulează jocul de lumini al unui sistem PhiliphsHue.

Tehnologii utilizate: *Python*, *Rust* și *Cargo* pentru dezvoltare, *warp* pentru *server*-ul HTTP.

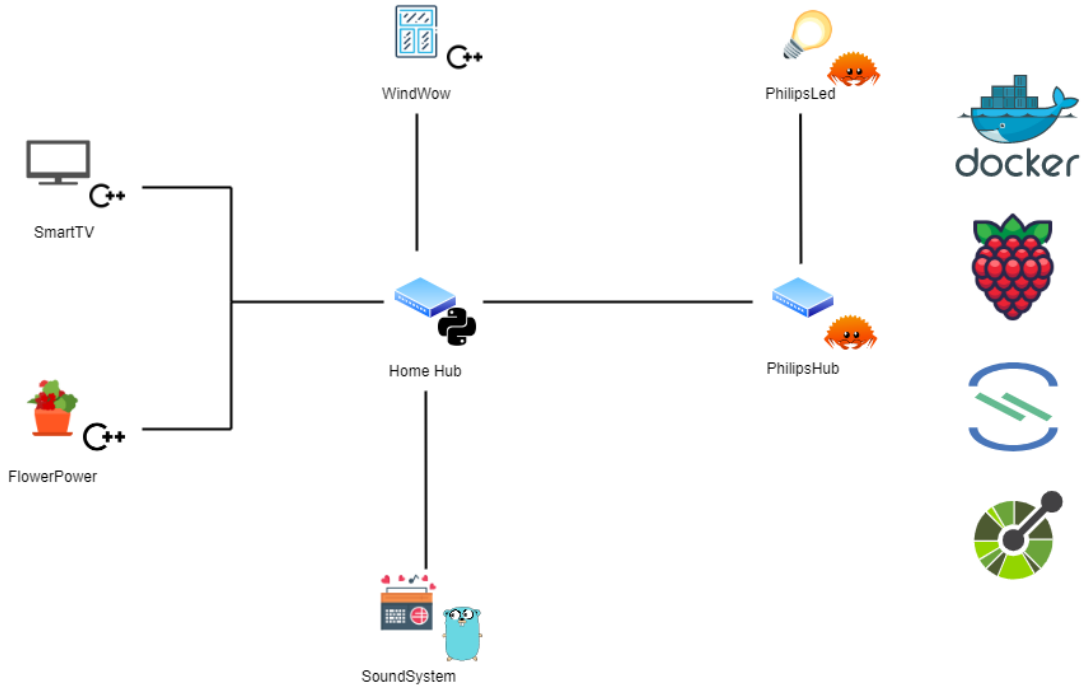
În context: TV-ul comunică date despre videoul afișat spre HDMIBox, acesta procesându-le și trimițându-le mai departe la PhiliphsHub. *Hub*-ul va sincroniza celelalte dispozitive luminoase (leduri, becuri etc.), astfel încât jocul de lumini este în armonie cu videoul.

Sursă: Autorul prezentei lucrări

Observăm ca aplicațiile prezentate sunt variate atât din punct de vedere al tehnologiilor utilizate (*C++*, *Python*, *Go*, *Rust*), cât și din punct de vedere al funcționării lor, acestea acoperind o gamă largă de utilizări posibile în cadrul unei locuințe inteligente.

În secțiunea următoare vom prezenta modul în care aceste aplicații au fost integrate într-o suită coerentă cu fluxuri de intercomunicare complexe.

Figura 4.1: Infrastructura parțială a suitei de aplicații. În centrul rețelei se află aplicația Hub care orchestrează comunicațiile dintre celelalte aplicații. Lateral sunt menționate tehnologiile utilizate pentru *deployment* (Docker, RaspberryPi) și specificații (OpenAPI, AsyncApi).



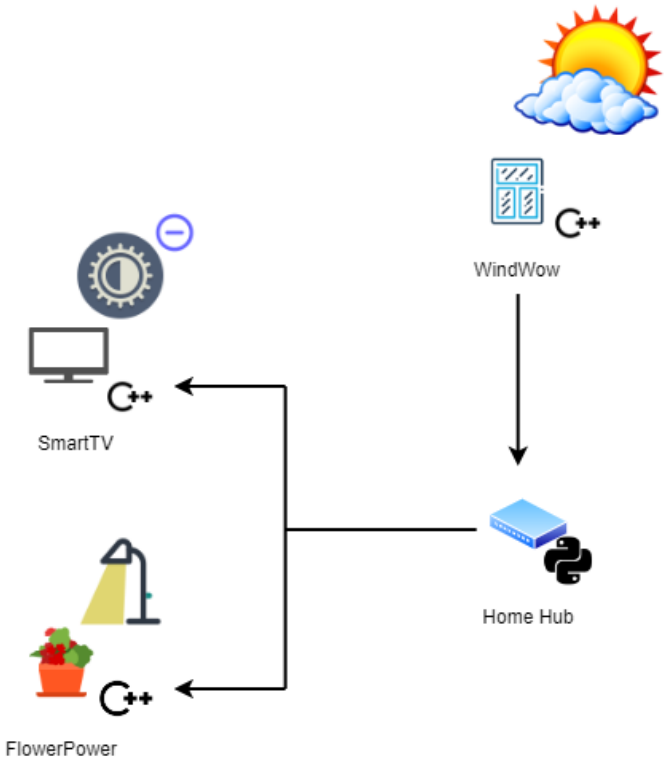
4.2 Infrastructura

Așa cum am menționat anterior, dispozitivele dintr-un sistem IoT sunt interconectate și fac parte din fluxuri de complexități variate. Pentru a ne păstra cât mai aproape de un sistem real, aplicațiile mai sus menționate au fost puse într-o rețea comună și orchestrate folosind o aplicație centrală, în general cunoscută sub numele de *hub*. Rețeaua poate fi desfășurată (en. *deployed* - *deployment*) atât într-un mediu virtualizat, folosind tehnologiile *Docker* și *docker-compose*, dar și într-un mediu cu hardware fizic folosind o serie de *script*-uri de automatizare, dar și câteva operații manuale. În experimentele desfășurate, am folosit atât mediul virtualizat, cât și un *deployment* pe arhitectura *x86* [3] și pe *ARMv7l*, [4] folosind un *Raspberry Pi 2 Model B*. În figura 4.1, putem observa câteva din aplicațiile descrise mai sus și topologia rețelei care le pune împreună.

Punctul central al rețelei este reprezentat de aplicația *hub*. Aceasta este responsabilă cu monitorizarea și orchestrarea tuturor dispozitivelor din rețea și permite implementarea fluxurilor de automatizare. Un astfel de flux poate fi observat în figura 4.2, unde avem prezentată sincronizarea luminozității mai multor dispozitive pe baza datelor colectate de aplicația WindWow.

[3] Arhitectură comună pentru computerele de uz personal, folosită de brand-uri precum Intel și AMD.
[4] Arhitectură utilizată pentru dispozitive mici și portabile cum ar fi telefoanele mobile, întâlnită des și în IoT.

Figura 4.2: Ilustrarea unui flux de automatizare. Aplicația WindWow comunică aplicației Hub o luminozitate scăzută, așa că aplicația Hub va ajusta luminozitatea aplicației SmartTV și va porni lampa asociată aplicației FlowerPower.



Hub-ul este o aplicație proprie dezvoltată cu limbajul *Python* și permite integrarea a două protocoale de comunicație în momentul de față: HTTP și MQTT. Am ales să nu folosim o aplicație *hub* comercială cum ar fi *OpenHAB* sau *Home Assistant*, deoarece avantajele aduse de o soluție personalizată din punct de vedere al timpului de integrare și al simplității au fost superioare.

Pentru a asigura omogenitatea comunicării într-un sistem cu componente eterogene, toate aplicațiile au specificații *OpenAPI* [5] pentru descrierea rutelor HTTP expuse și specificații *AsyncAPI* [6] pentru descrierea formatului mesajelor transmise prin intermediul protocolului MQTT. Folosind *OpenAPI Generator* [7], specificațiile *OpenAPI* pot fi transformate în cod de interacțiune cu aplicațiile. Codul generat a fost utilizat în aplicația de *hub*, pentru a oferi acces uniform la aplicații.

Față de protocolul HTTP, protocolul MQTT operează în manieră *publisher/subscribe*,

[5]Specificație bazată pe limbajul YAML care poate descrie modul de autentificare, verbele și rutele unui API bazat pe protocolul HTTP. Mai multe detalii pot fi găsite la adresa https://swagger.io/specification/.

[6]Specificație construită peste OpenAPI, aceasta fiind orientată spre descrierea de sisteme EDA. Mai multe informații pot fi găsite la adresa https://www.asyncapi.com/.

[7]Utilitar software care transformă specificații OpenAPI în cod sursă în diferite limbaje de programare. Utilitarul poate fi găsit la https://openapi-generator.tech/.

nu *request/reqsponse*, astfel devine necesară existența unui distribuitor de mesaje în rețea. Pentru această sarcină am utilizat *Eclipse Mosquitto*, deoarece este un produs *open-source* ușor de instalat și folosit atât în mediu virtualizat, cât și nevirtualizat.

Întreaga configurație este stocată în câteva fișiere în format, YAML Ain't Markup Language™ (YAML) sau JSON, aceasta specificând numele și calea aplicațiilor, porturile de rețea expuse și alți parametri specifici. De asemenea, conține informații despre aplicația *hub* și serviciile externe necesare (cum ar fi distribuitorul de mesaje). Configurația este citită de utilitarele construite pentru a facilita cele mai comune operații: compilarea aplicațiilor, rularea aplicațiilor pe mediul local, virtualizat sau nu, executarea unor teste, repornirea sau oprirea aplicațiilor, instalarea dependințelor software, etc.

Docker este o soluție software pentru izolarea aplicațiilor în containere. Față de mașinile virtuale clasice care fie utilizează facilitățile de virtualizare ale unității centrale de procesare, fie emulează în întregime o unitate centrală, *Docker* pune la dispoziție un strat de abstractizare între aplicații și sistemul de operare, astfel încât acestea se execută într-un mediu izolat denumit container, fără acces la restul sistemului. De asemenea, configurarea dependințelor și mediului din container în care va fi executată aplicația sunt specificate în mod textual, în așa numitele *Dockerfiles*. Această abordare aduce o serie de avantaje printre care se numără o mai bună securitate pentru un cost mic de performanță și o bună reproductibilitate a mediilor. Pentru a orchestra mai multe containere, folosim *docker-compose* care poate fi configurat în mod textual. Acesta permite și configurarea de topologii, medii de stocare și interacțiunea cu sistemul gazdă.

Utilizarea suitei de aplicații în containere de tip *Docker* este convenientă, deoarece consumă puține resurse și nu necesită pregătiri suplimentare, însă pentru a simula un scenariu mai apropiat de realitate, aplicațiile pot fi puse pe dispozitive hardware fizice cum ar fi *Raspberry Pi*. Am pus la dispoziție scripturile necesare pentru compilarea aplicațiilor și instalarea dependințelor pentru un sistem *Raspbian ARMv7l*. Aplicațiile împreună cu *hub*-ul și distribuitorul de mesaje au fost desfășurate pe mai multe *Raspberry Pi's* conectate prin *wi-fi*.

4.3 Extinderea suitei de aplicații

Deoarece există un număr foarte mare de posibile tipuri de sisteme IoT și utilizări ale acestora, suita de aplicații nu acoperă decât un mic număr din total. Astfel, încurajăm extinderea și îmbogățirea suitei cu aplicații și fluxuri de automatizare proprii atât din partea cercetătorilor, cât și a profesioniștilor. Pașii pentru includerea unei noi aplicații sunt simplii și puțini la număr, astfel asigurând un proces cât mai puțin anevoios. În rândurile ce urmează, vom parcurge ce caracteristici trebuie sa aibă o nouă aplicație și ce pași trebuie urmați pentru includerea ei.

Caracteristici necesare:

1. Aplicația trebuie să comunice folosind cel puțin unul din protocoalele HTTP sau MQTT.

2. Sunt necesare specificații *OpenAPI* și/sau *AsyncAPI*, în funcție de protocoalele folosite.

3. Codul trebuie să poată fi compilat pentru arhitectura *x86* sau să poată fi cel puțin executat într-un emulator.

Integrarea în suită [8]:

1. Crearea configurațiilor necesare pentru generarea de clienți de comunicare (momentan doar pentru HTTP).

2. Punerea la dispoziție a unui *script* de instalare al dependințelor aplicației.

3. Punerea la dispoziție a unui *script* de compilare a aplicației, pentru arhitecturile țintă.

4. Punerea la dispoziție a unui *script* pentru executarea aplicației pe un port de rețea arbitrar.

5. Crearea unei configurații în format *Dockerfile* și adăugarea unei intrări în configurația generală a suitei.

6. Copierea codului sursă a noii aplicații în directorul corespunzător.

Deși recomandăm integrarea aplicațiilor *open-source*, în cazul în care sursa nu este disponibilă, se pot omite pașii de compilare și copiere a sursei aplicației. Este necesar să ne asigurăm că binarele puse la dispoziție pot fi executate pe arhitectura *x86*, fie direct, fie prin intermediul unui emulator.

4.4 Defecte, evaluare și limitări

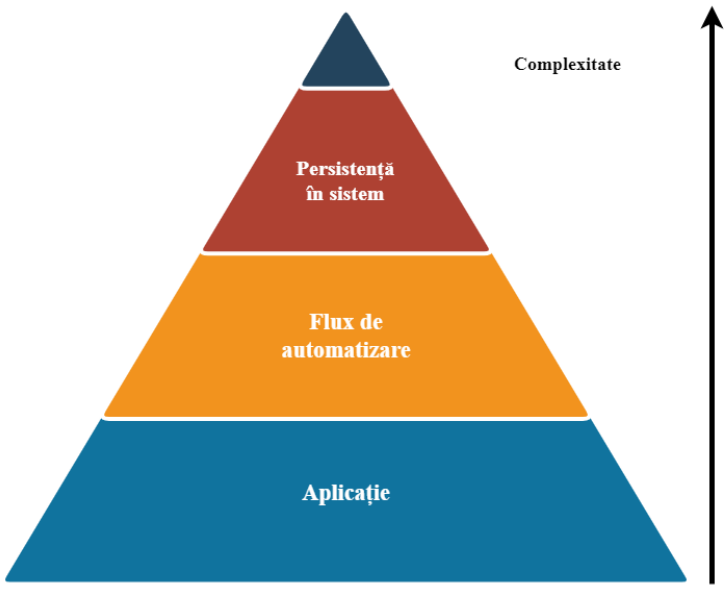
Scopul suitei de aplicații este compararea metodelor de testare, așa cum am menționat anterior. Pentru a oferi un reper relevant, este necesar ca aplicațiile să conțină defecte realiste. Cum aceste aplicații au fost dezvoltate de multiple echipe de studenți, conțin în mod natural o serie de defecte, defecte pe care le-am identificat în urma testării funcționale a suitei, dar despre acest proces vom discuta în capitolul următor. Pe lângă defectele existente, am considerat necesară îmbogățirea suitei cu o gamă mai largă de defecte. Acestea au fost introduse în codul aplicațiilor sau în regulile de automatizare din aplicația *hub*, unele defecte fiind o combinație din ambele.

[8] Tutorialul complet poate fi găsit la adresa https://github.com/unibuc-cs/IoT-application-set/blob/master/HOW_TO_ADD_AN_APP.md.

Deoarece una din caracteristicile principale ale IoT este numărul mare de părți comunicante, am ales să împărțim defectele descoperite și injectate în trei categorii bazate pe nivelul de interacțiune dintre părțile sistemului. Aceste categorii sunt următoarele:

1. Defecte la nivel de aplicație - acestea afectează doar aplicații individuale, efectele comune fiind funcționarea incorectă sau oprirea funcționării în totalitate.
2. Defecte la nivel de flux de automatizare - caracterizate de rezultatul incorect la finalul executării unui singur flux de automatizare ce conține multiple aplicații.
3. Defecte la nivel de persistență în sistem - atunci când execuția a mai multor fluxuri de automatizare care nu sunt disjuncte duce la funcționarea incorectă a întregului sistem.

Figura 4.3: Cele trei niveluri de defecte reprezentate în ordinea complexității acestora.



În general, defectele din prima categorie au fost cel mai des abordate în literatură, majoritatea soluțiilor existente concentrându-se pe acestea. Pe de altă parte, defectele din celelalte categorii au un grad de dificultate de detectare mult mai ridicat, deoarece sunt cauzate, în general, de concurență și sunt în strânsă legătură cu domeniul specific în care este utilizată aplicația. Să luăm un exemplu pentru a ne face o imagine mai clară, un defect din categoria întâi poate fi reprezentat de un acces invalid de memorie care cauzează un răspuns întârziat, un răspuns cu date aleatorii sau chiar nefuncționarea completă a aplicației. Metricile de detectare sunt clare și independente de logica concretă a aplicației. Un defect din categoria a treia este mult mai subtil și dificil de detectat. Spre exemplu, să considerăm că un senzor de lumină inteligent determină acționarea draperiilor în funcție de nivelul de luminozitate înregistrat. Utilizatorul comandă manual draperiile,

în timp ce acestea sunt în curs de acționare și cauzează blocarea mecanismului. În acest caz, detectarea defectului nu mai este la fel de evidentă, iar producerea acestuia necesită suprapunerea mai multor evenimente independente.

În studiul realizat de Zhou et al. (2021), se menționează o clasă de defecte cauzate de acțiuni impredictibile în cadrul fluxurilor de automatizări. Această clasă de defecte este ceea ce noi considerăm a fi defectele de nivel doi și trei în categorisirea proprie. Conform autorilor studiului, cauzele acestei clase de defecte sunt următoarele:

1. *race condition* între diferite fluxuri;
2. evenimente lipsă sau procesate într-o ordine imprevizibilă;
3. acțiuni duplicate sau în conflict.

În rândurile de mai jos, vom analiza câteva exemple de defecte existente și introduse din toate cele trei categorii numite [9]. De asemenea, vom prezenta liniile de program responsabile pentru producerea defectului, alături de modul în care acesta poate fi rezolvat. Secțiunile de cod vor fi reprezentate folosind formatul *diff* [10]. În cadrul suitei de aplicații, toate defectele sunt reparate, însă acestea pot fi activate selectiv folosind capabilitățile de *patching* ale utilitarului *git*.

În aplicația WindWow, am introdus un defect artificial care cauzează un acces al unei adrese invalide de memorie. Acesta este declanșat atunci când draperiile sunt trase și încercăm să setăm o luminozitate mai mare de 25.

```
--- a/apps/windwow/Window.cpp
+++ b/apps/windwow/Window.cpp
@@ -115,10 +115,9 @@ class Window {
                        if(settings[i].name == "luminosity") {
                            if(settings[i].value > 25.0) {
                                if(changed_state[3] % 2 == 0) {
-                                   // BUG TO UNPATCH
                                    // START FAKE BUG
-                                   // volatile int *p = nullptr;
-                                   // p[50] = 0xdeadbeef;
+                                   volatile int *p = nullptr;
+                                   p[50] = 0xdeadbeef;
                                    // END FAKE BUG
```

Listing 4.1: Accesarea adresei nule va cauza oprirea aplicației WindWow

[9] O listă completă a defectelor și codului aferent acestora poate fi găsită la https://github.com/unibuc-cs/IoT-application-set/tree/master/bug_unpatches.

[10] Format introdus de utilitarul POSIX *diff* care permite vizualizarea diferențelor între fișiere text prin prefixarea liniilor cu simboluri care reprezintă adăugarea, ștergerea sau modificarea. Colorarea liniilor aduce un plus vizibilității.

Un defect găsit în aplicația SmartPot este lipsa validărilor asupra datelor primite de la utilizatori. Aplicația folosește formatul JSON pentru primirea cererilor. Datele din aceste cereri sunt reprezentate în memorie prin intermediul unui dicționar (tabel de dispersie, *hash-map*). Aplicația nu validează existența cheilor accesate atunci când primește o cerere de actualizare a setărilor, astfel cauzând nefuncționarea aplicației.

```
--- a/apps/flowerpower/src/SmartPotEndpoint.cpp
+++ b/apps/flowerpower/src/SmartPotEndpoint.cpp
@@ -269,8 +269,7 @@ namespace pot
         double sensorMax = document["max"].GetDouble();

         // Valoarea o updatam in MQTT.
-        // BUG TO UNPATCH don't check for HasMember
-        if ((!document.HasMember("nutrientType")) or
-            document["nutrientType"].IsNull())
+        if (document["nutrientType"].IsNull())
         {
             Sensor aux = smartPot->GetSensor(sensorNameMap[
    sensorTypeID]);
             aux.SetMinValue(sensorMin);
```

Listing 4.2: Lipsa validării cheilor JSON în aplicația SmartPot

Unul din defectele introduse la nivel de flux de automatizare este sanitizarea [11] insuficientă a datelor transmise spre SmartTV. Cum am văzut în secțiunea precedentă, există un flux de automatizare care reglează luminozitatea SmartTV-ului, în funcție de datele colectate de la aplicația WindWow. SmartTV-ul acceptă valori de luminozitate în intervalul $(1, 10)$, însă aplicația *hub* poate trimite valori în afara acestuia, cauzând eșecul cererii de actualizare.

```
--- a/hub/app.py
+++ b/hub/app.py
@@ -181,8 +181,6 @@ def rule4(env: Environment):
             10 - env.data["windwow"]["luminosity"]/10 +
    brightness_base,
             brightness_base
         )
-    # // BUG TO UNPATCH forget to call min
-    env.data["smarttv"]["brightness"] = min(
-       env.data["smarttv"]["brightness"],
-       10)
     env.clients["smarttv"].set_brightness_level_post(
```

[11] Neologism, tradus din engleză, *to sanitize* - *sanitization*. În context, se referă la curățarea datelor posibil malițioase. O traducere inexactă, dar ad litteram este *sanitization* = *sanitație*.

```
            int(env.data["smarttv"]["brightness"])
        )
```

Listing 4.3: Valorile luminozității transmise spre SmartTV nu sunt doar în intervalul (1, 10)

Intersectarea dintre două fluxuri de automatizare legate de WindWow și FlowerPower cauzează un defect din categoria a treia, anume defecte la nivel de persistență. Așa cum am văzut în cazul unei temperaturi prea ridicate, luminozitatea va fi redusă prin acționarea draperiilor, însă acest eveniment poate declanșa aprinderea lămpii FlowerPower în mod nenecesar, astfel încălzind excesiv planta.

```
--- a/hub/app.py
+++ b/hub/app.py
@@ -165,8 +165,7 @@ def rule3(env: Environment):
     luminosity_sensor_id = 3
     threshold = env.settings["plant_lamp_window_treshold"]

-    # // BUG TO UNPATCH don't check window temperature
-    if env.data["windwow"]["luminosity"] < threshold and
-        env.data["windwow"]["temperature"] < 30:
+    if env.data["windwow"]["luminosity"] < threshold:
         env.clients["flowerpower"].activate_solar_lamp_get()
```

Listing 4.4: Lipsa verificării temperaturii cauzează un conflict între două fluxuri de automatizare

Pentru a evalua o tehnică sau o unealtă de testare folosind suita de aplicații, putem activa selectiv defectele sau tipurile de defecte asupra cărora suntem interesați. În urma aplicării metodologiei ce se dorește a fi evaluată, putem colecta o serie de metrici ce vor servi ca mijloc de comparare cu alte metodologii. Câteva posibile astfel de metrici sunt timpul și numărul de defecte descoperite, ratele de detecție per categoria de defecte sau acoperirea codului sau a stărilor posibile ale sistemului. Rămâne la latitudinea cercetătorilor să decidă care sunt cele mai potrivite metrici pentru evaluarea metodologiei în cauză și să interpreteze rezultatele. Oferirea unei metodologii de alegere a experimentelor de evaluare este în afara scopului acestei lucrări.

Deși defectele introduse sunt comparabile cu cele existente în realitate și pot oferi un punct de plecare pentru analiza tehnicilor de testare, trebuie să luăm în considerare și limitările acestei suite de aplicații. Un punct slab este reprezentat de lipsa software-ului găsit pe dispozitive reale, aplicațiile curente fiind simulări ale unor procese reale. De asemenea, acestea sunt proiectate să funcționeze în principal doar pe sisteme a căror arhitectură suportă un sistem de operare compatibil cu Portable Operating System Interface (POSIX) sau similar, pe când în realitate multe dispozitive nu au un sistem de operare. În

plus, suita conține un număr mic de protocoale de comunicație (doar HTTP și MQTT). Toate aceste limitări vor fi adresate în viitoarele activități de cercetare.

Acum, după ce am construit o suită de aplicații care să fie similară unui sistem dintr-o locuință inteligentă cu fluxuri de automatizări și comunicare între dispozitive prin protocoale specifice, vom continua în capitolul următor prin aplicarea câtorva metodologii și unelte de testare. Vom folosi defectele prezentate pentru a compara aceste metodologii și pentru a putea analiza avantajele și dezavantajele acestora în raport cu metricile de evaluare alese.

Capitolul 5

Evaluarea unor tehnici de testare

Pentru a dovedi relevanța suitei de aplicații construită anterior, vom desfășura o serie de experimente folosind câteva tehnici și tehnologii de testare. Acestea fac parte din următoarele categorii: testare funcțională, testare exploratorie, analiză statică și verificare formală.

În cadrul analizei, ne vom concentra asupra următoarelor criterii de evaluare:

- tehnica este automată sau manuală (sau parțial automată);
- complexitatea pregătirii mediului de testare;
- categoriile de defecte (*bug*-uri) care pot fi detectate;
- numărul de defecte descoperite (raportat la timpul de execuție dacă este cazul);
- adaptarea la provocările specifice IoT (eterogenitate, sisteme distribuite);
- limitări și puncte slabe.

5.1 Testarea funcțională

Am explicat în capitolul al doilea, secțiunea 2.1.2, ce este testarea funcțională și care sunt caracteristicile acesteia, așa că nu vom relua explicarea noțiunii. În secțiunea curentă, vom vorbi despre o serie de tehnici de testare funcțională, pe care le-am utilizat atât pentru a valida funcționarea corectă a suitei de aplicații, cât și pentru a oferi dovada reproductibilității defectelor existente.

Metodologia Behaviour-Driven Development

Behaviour-Driven Development (BDD) este o metodologie de dezvoltare succesoare a Test-Driven Development (TDD), care se preocupă de crearea de specificații clare și automatizabile pentru testarea sistemului țintă. Pentru a atinge acest scop, BDD propune o

terminologie standardizată pentru proiectarea cazurilor de test pentru a ușura formularea lor atunci când mai multe părți sunt implicate (*stakeholders*), cum ar fi dezvoltatori, clienți sau personalul de vânzări. Vom prezenta, pe scurt, această metodologie bazându-ne pe studiul realizat de Solis și X. Wang (2011) despre caracteristicile BDD.

Conform studiului, această metodologie pune accent pe folosirea unui limbaj comun, standardizat și independent de domeniul de aplicare pentru descrierea scenariilor de test. Scenariile sunt apoi formulate ca text în limbaj natural, cât mai simplu posibil și orientate spre descrierea de comportamente (en. *behaviour*). De exemplu, formatul cel mai des întâlnit pentru formularea scenariilor de test poate fi observat mai jos în fragmetul 5.1.

```
Scenario X: Titlul scenariului
Given: Contextul în care va fi efectuat testul
When: Este observat un anume eveniment
(ex. click-ul unui utilizator pe un buton anume)
Then: Rezultatul dorit
```

Listing 5.1: Structură comună a unui scenariu de test folosind BDD

Această standardizare oferă posibilitatea transpunerii directe a scenariilor în teste automate pentru a putea valida funcționalitățile sistemului.

În articolul realizat de Cristea, Feraru și Păduraru (2022), am abordat testarea funcțională a suitei de aplicații propuse, folosind concepte inspirate din metodologia BDD. Deoarece testarea unitară nu poate asigura funcționarea corectă a unui sistem atunci când mai multe componente sunt integrate împreună, testele funcționale au avut în vedere angrenarea mai multor aplicații din suită și verificarea fluxurilor de date, care circulă între ele. De asemenea, această metodă servește ca exemplu de bază pentru utilizarea și modul de funcționare al suitei de aplicații construite.

Pentru descrierea și automatizarea testelor am utilizat biblioteca Python Behave, [1] care permite formularea scenariilor de test în limbaj natural și le asociază cu testele automate scrise în limbajul Python. Deși scenariile sunt descrise în limbaj natural, ele trebuie să respecte sintaxa impusă de standardul Gherkin [2], care asigură structurarea corectă a scenariilor de test. În cardul articolului, am prezentat un exemplu de scenariu de test pe care îl vom reproduce mai jos în fragmentul cu numărul 5.2. Acest scenariu este declanșat de regulile de automatizare din aplicația *hub* și vizează setarea corespunzătoare luminozității *smart TV*-ului în funcție de datele colectate de senzorul de luminozitate al ferestrei. Testul va fi executat de *framework*-ul Behave, pentru toate valorile din tabelul de exemple.

```
Feature: TV auto-brightness
  Scenario Outline: TV brightness is set based on window luminosity level
```

[1] Documentația și codul sursă pot fi găsite la adresa https://behave.readthedocs.io/en/stable/.
[2] Mai multe informații și specificația completă a limbajului pot fi găsite la adresa https://cucumber.io/docs/gherkin/.

```
    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 5.2: Exemplu de scenariu de test BDD pentru aplicațiile din suită

Pentru a adăuga un nou test în cadrul suitei de aplicații este nevoie de formularea lui în limbaj natural, urmând constrângerile impuse de sintaxa Gherkin, iar apoi realizarea testului automat în limbajul Python. Un exemplu, de test se poate regăsi în fragmentul cu numărul 5.3. Pasul *given* creează contextul propice testării, anume setează senzorii la valorile dorite. Pasul *when* declanșează regulile de automatizare din interiorul aplicației *hub*, iar apoi pasul *then* va verifica dacă valoarea luminozității colectată de la *smart TV* corespunde cu rezultatul așteptat.

```
from behave import *
import app

@given('TV brightness is set to {X}, window luminosity to {Y} and base
    luminosity to {Z}')
def step_impl(context, X, Y, Z):
    app.env.clients["smarttv"].set_brightness_level_post(int(X))

    app.env.clients["windwow"]\
        .settings_setting_name_setting_value_post(
            "luminosity",
            int(Y),
        )

    app.env.settings["tv_base_brightness"] = int(Z)

@when('automation rules are triggered for TV')
def step_impl(context):
    app.env.run_simple()

@then('TV brightness is set to max(10 - {Y}/10 + {Z}, {Z}) <= 10')
def step_impl(context, Y, Z):
    Y = float(Y)
    Z = float(Z)
    assert app.env.data["smarttv"]["brightness"] == int(
        min(max(10 - Y/10 + Z, Z), 10)
```

)

Listing 5.3: Exemplu de test automat folosind biblioteca Behave

În urma aplicării acestei tehnici pe suita de aplicații, am remarcat următoarele:

- tehnica este manuală, deci limitată de experiența și timpul investit de cel care testează;
- toate defectele existente în cardul suitei de aplicații pot fi detectate și reproduse, însă succesul detectării este dependent de factorul uman;
- configurarea mediului de testare nu aduce complexitate suplimentară.

Concluzionăm că acest tip de testare este potrivit atât testării individuale a aplicațiilor, cât și a acestora după integrare, însă această tehnică este mai degrabă aplicabilă în procesul de dezvoltare pentru testarea scenariilor pozitive, decât pentru descoperirea de defecte neobișnuite. Automatizarea generării de scenarii de test variate și reducerea dependenței față de factorul uman pot reprezenta două direcții interesante de cercetare, ce ar ajuta la îmbunătățirea tehnicii.

5.2 Testarea exploratorie folosind *fuzzing*

Testarea exploratorie își propune găsirea de stări marginale ale unui sistem pentru a descoperi un număr cât mai mare de defecte. Acest tip de testare nu se concentrează în general pe scenariile pozitive, ci pe cele negative în care datele de intrare sunt de multe ori invalide parțial sau total.

În paragrafele următoare, ne vom concentra pe o tehnică anume de testare exploratorie și anume *fuzzing*-ul. Așa cum ne explică Manes et al. (2021), *fuzzing*-ul [3] este o tehnică larg răspândită care presupune trimiterea de date de intrare malformate semantic sau sintactic spre un sistem software și monitorizarea reacției acestuia.

În general, un algoritm de *fuzzing* generează în mod aleator mutații pentru un set de date de intrare, apoi trimite datele de intrare la sistemul software țintă, iar în final evaluează starea rezultată și constată dacă au fost detectate defecte. Acești pași se pot desfășura la infinit sau într-un interval de timp stabilit. În fragmentul 5.4 este reprezentată în pseudocod structura generală a algoritmului descris, așa cum ne-o înfățișează Manes et al. (2021).

Input: `Configuration`, t_{limit}
Output: `Bugs`

[3] O discuție amplă despre *fuzzing* poate fi găsită în materialul realizat de Andreas Zeller et al. (2021), *The Fuzzing Book*, Retrieved 2021-10-26 15:30:20+02:00, CISPA Helmholtz Center for Information Security, URL: https://www.fuzzingbook.org/.

```
while t_elapsed < t_limit:
    Configuration ← Schedule(Configuration, t_elapsed, t_limit)
    TestCase ← Generate(Configuration)
    NewBugs, ExecInfo ← Evaluate(Configuration, TestCase)
    Configuration ← Update(Configuration, ExecInfo)
    Bugs ← Bugs ⋃ NewBugs

return Bugs
```

Listing 5.4: Structura generală a unui algoritm de fuzzing

Deși structura generală este simplă, există o gamă foarte largă de algoritmi de *fuzzing*. Distingem trei categorii importante, în funcție de cunoștințele semantice pe care acesta le deține despre interacțiunea cu sistemul:

- *black-box* - nu cunosc semantica datelor de intrare și nici codul sursă al aplicației;
- *grey-box* - dețin informații parțiale despre semantica datelor de intrare și au parte de un *feedback* parțial din partea sistemului țintă;
- *white-box* - au cunoștințe depline despre semantica datelor și sistemul vizat.

Un aspect foarte important al *fuzzer*-elor este modul în care generează noi date de intrare. O abordare foarte populară este utilizarea de algoritmi genetici, astfel alegându-se mostrele cu cel mai mare succes raportat de o anume metrică. Pentru metricile de succes se utilizează, în general, acoperirea totală a codului (sursă sau cod mașină în funcție de caz).

Pentru detectarea defectelor este nevoie de un așa numit *oracol*. Acesta este un mecanism care permite evaluarea stării sistemului țintă și poate raporta anumite tipuri de defecte. De exemplu, sistemul de operare va semnala un acces invalid de memorie, astfel un algoritm de *fuzzing* poate lua la cunoștință că a descoperit un defect.

RESTler

RESTler este un *grey-box stateful fuzzer* [4] specializat în REST APIs create peste protocolul HTTP. Acest *fuzzer* este prezentat în articolul publicat de Atlidakis, Godefroid și Polishchuk (2019). Autorii ne spun că RESTler are două metode prin care își ghidează generarea de date de test:

- prin inferențe bazate pe specificația sistemului țintă, acesta deduce relațiile de tip producător-consumator dintre cereri (de exemplu, „cererea B va fi făcută doar după cererea A, deoarece A creează o resursă x necesară pentru cererea B”);

[4] Spre deosebire de *fuzzer*-ele obișnuite, cele *stateful* rețin informații despre starea sistemului țintă și relațiile dintre mai multe date de intrare separate. De exemplu, un *stateful fuzzer* pentru protocolul HTTP ar trebui să poată înțelege relația dintre o cerere de tip *DELETE* urmată de una de tip *GET*.

- prin analizarea *feedback*-ului primit de la sistemul țintă pentru a determina ordinea corectă a cererilor (de exemplu, „cererea C este refuzată dacă nu este efectuată strict după lanț de cereri A,B").

RESTler detectează defectele folosind un oracol personalizat, compus din mai multe criterii de detectare: codul de stare al răspunsului la cerere (de exemplu, codul 500 reprezintă o eroare), timpul mare de răspuns la cerere (sau lipsa lui totală) și o serie de reguli legate de relațiile dintre cereri și răspunsuri (de exemplu, dacă a fost inițiată o cerere de tip *DELETE /resource/x*, orice cerere de tip *GET /resource/x* ar trebui să întoarcă codul *404 Not Found*, altfel putem considera că am descoperit un defect). Mai multe detalii despre algoritmii utilizați pot fi găsite în articolul menționat anterior, figura 3 conținând și pseudocodul acestora.

Am ales utliziarea *RESTler*, deoarce toate aplicațiile din suită au deja specificații OpenAPI pentru descrierea rutelor HTTP, aceasta fiind specificația utilizată și de *fuzzer*. În urma rulării, am observat că *fuzzer*-ul poate detecta majoritatea defectelor a căror efect este nefuncționarea completă a aplicației sau întoarcerea unui răspuns de tip *500 Internal Server Error*. Deoarece API-urile aplicațiilor sunt relativ mici, RESTler parcurge majoritatea stărilor posibile în doar câteva minute. Pentru o explorare mai cuprinzătoare, ar fi necesară îmbunătățirea specificațiilor și a dicționarelor de valori utilizate de RESTler, însă acești pași sunt în afara scopului lucrării.

În comparație cu tehnica anterioară, componenta exploratorie a *fuzzing*-ului îmbunătățește considerabil raportul dintre efort și numărul de defecte descoperite. Pe de altă parte, *RESTler* nu este specializat pentru sistemele IoT și nu este capabil să testeze mai multe aplicații simultan, decât dacă acestea sunt încapsulate sub un REST API unificat.

5.3 Analiză statică

Tehnicile anterioare evaluate pot fi toate considerate dinamice în natură, deoarece presupun executarea codului aplicațiilor și observarea comportamentului în timpul execuției. O altă abordare care necesită mai puține resurse computaționale și nu este preocupată de executarea aplicațiilor este analiza statică.

Analiza statică presupune mecanisme de recunoaștere și deducție a tiparelor problematice în codul sursă sau codul mașină al aplicațiilor. Așa cum ne spune analiza realizată de Gosain și Sharma (2015), există mai multe tehnici de analiză statică:

- recunoașterea constructelor sintactice problematice;
- reprezentarea stărilor aplicației ca noduri într-un graf și analizarea fluxurilor de date;
- interpretarea abstractă a programelor;

- analiza bazată pe constrângeri și satisfacerea acestora.

Această tehnică de analiză nu depinde de date de intrare concrete pentru aplicație și nici nu necesită analizarea întregului program pentru a putea oferi concluzii pertinente. Pe de altă parte, aceste caracteristici cauzează o rată mare a fals-pozitivelor și o imposibilitate a verificării corectitudinii funcționale. Astfel, este o tehnică utilă pentru detectarea defectelor, însă nu suficient de comprehensivă pentru a fi singura tehnică utilizată.

În următoarele paragrafe, vom prezenta evaluarea a două unelte software pentru analizarea statică a programelor, ambele specializate pe limbajele *C* și *C++*.

cppcheck

cppcheck [5] este un analizator static specializat pentru limbajele C și C++. Acesta urmărește detectarea de greșeli comune în programare și se concentrează, în special, pe acele greșeli care pot cauza comportamente nedefinite sau sunt periculoase. Acest analizator promite o rată mică a fals pozitivelor.

Am utilizat *cppcheck* pentru a analiza toate cele cinci aplicații C++ din suită fără a fi nevoie de configurări suplimentare. Procesul de detectare al potențialelor probleme este în totalitate automat, însă pentru stabilirea sigură a existenței unui defect și reproducerea acestuia este nevoie de intervenția manuală a unui expert.

Utilitarul a reușit să identifice cu succes două dintre defectele prezente în aplicații, acestea fiind foarte comune conform Common Weakness Enumeartion (CWE). În fragmentul 5.5, putem observa mesajele care semnalează dereferențierea unui pointer nul și dubla eliberare a memoriei.

```
windwow/Window.cpp:120:33: warning: Possible null pointer dereference: p [
    nullPointer]
flowerpower/src/SmartPotEndpoint.cpp:436:24: error: Memory pointed to by '
    target_p' is freed twice. [doubleFree]
```

Listing 5.5: Cele două defecte detectate de *cppcheck*

Din păcate, *cppcheck* nu reușește să proceseze întodeauna corect semanticile mai complicate, în fragmentul 5.6 avem un exemplu de semnal fals pozitiv. Variabila *setResponse* este inițializată pe ambele ramuri ale unei structuri $if-else$ care urmează după declararea acesteia, însă *cppcheck* nu interpretează corect acest construct.

```
windwow/WindowEndpoint.cpp:257:21: note: Uninitialized variable:
    setResponse
```

Listing 5.6: Exemplu de fals pozitiv detectat de *cppcheck*

Observăm că *cppcheck* este potrivit doar pentru analizarea unor greșeli comune și relativ simple de programare, acesta fiind exclusiv pentru limbajele C și C++. Analizatorul

[5] Documentația și codul sursă al aplicației pot fi găsite la https://cppcheck.sourceforge.io/.

nu poate fi utilizat pentru testarea interacțiunilor dintre aplicații și nici pentru descoperirea de defecte cauzate de tranziții de stare complexe. Deși este rapid și nu necesită configurări, aria de aplicabilitate a acestei unelte rămâne destul de restrânsă.

weggli

weggli [6] este un utilitar pentru căutări cu înțelegere semantică în interiorul codului sursă al programelor C și C++. Deși nu necesită configurări suplimentare, acest utilitar nu vine cu o serie de reguli de analiză standard, această sarcină rămânând în grija expertului care îl folosește.

Sintaxa folosită pentru efectuarea de căutări este similară cu cea a aplicației *grep*, [7] însă permite și utilizarea de variabile sau constructe sintactice complicate, nu doar căutări *regex*.

În fragmentul 5.7 este o expresie ce va semnala toate zonele de cod sursă unde se inițializează un pointer de tip arbitrar cu valoarea nulă, iar apoi acesta este accesat fără a fi inițializat cu o valoare nenulă. *var* ține locul oricărui identificator și poate fi menționat în continuare, iar caracterul *underscore* poate înlocui orice expresie, observăm cât de ușor putem identifica structuri semantice complexe.

```
weggli --cpp '{
    _* $var = nullptr;
    $var[_];
}' ./apps/
```

Listing 5.7: Interogare weggli pentru a găsi accesări de pointer nul

Un alt exemplu poate fi observat în 5.8, unde am dorit să găsim toate accesările de chei JSON pentru care nu s-a verificat existența.

```
weggli --cpp '{
    Document $document;
    NOT: $document.HasMember(_);
    $document[_]._();
}' ./apps/
```

Listing 5.8: Interogare weggli pentru a găsi accesarea de chei JSON fără verificare

weggli poate detecta majoritatea defectelor prezente în suită care sunt legate de un construct de cod C++. Dezavantajele acestei abordări sunt însă reprezentate de faptul că este nevoie de interacțiune umană pentru a formula interogările, majoritatea interogărilor rezultând în a nu găsi niciun defect, dar și faptul că nu se pot detecta defecte de integrare sau logică complexă.

[6] Codul sursă al aplicației poate fi găsit la https://github.com/googleprojectzero/weggli.

[7] Utilitar foarte popular pe sistemele **nix* pentru a căuta în interiorul documentelor text, folosind expresii regulate (*regex*).

5.4 Verificare formală

Prin verificare formală înțelegem practica de a demonstra sau a infirma corectitudinea funcționării unui sistem luând în calcul un set de proprietăți dorite. Această practică necesită transpunerea în limbaj matematic a algoritmilor și proprietăților sistemului țintă. O abordare foarte populară în verificarea formală este verificarea modelului (en. *model checking*) folosind programe de calculator cunoscute ca Automated Theorem Provers (ATPs), adică programe care pot demonstra automat teoreme. O clasă particulară a acestui tip de programe este reprezentată de cele care rezolvă Satisfiability Modulo Theories (SMTs), pe care le-am menționat în capitolele anterioare. În rândurile următoare, ne vom concentra asupra tehnicii *model checking* și o vom exemplifica folosind logica temporală și limbajul *TLA+*.

Așa cum ne spun Muller-olm et al. (1999), *model checking* este o procedură care determină dacă un model abstract M al unui sistem țintă satisface o formulă logică ϕ, adică $M \models \phi$. Modelul abstract este, în general, o structură similară cu cea a unui automat finit (ie. conține stări și tranziții), iar ϕ este un set de proprietăți exprimate folosind logică temporală.

Vom privi un exemplu practic de modelare a unui sistem simplu pentru o mai bună înțelegere. Să ne reamintim exemplul oferit de Zhou et al. (2021) pentru un defect de tip *race condition*:

"[...] For example, «When motion is detected, turn on the switch» and «Every day at midnight, turn off the switch» will conflict if motion is detected at 12 pm. [...]"

Putem defini starea sistemului luând în considerare trei parametri: starea întrerupătorului becului (oprit/pornit), starea senzorului de mișcare (detectează mișcare sau nu detectează mișcare) și timpul curent. Așa cum ne sugerează Lamport (2005), pentru reprezentarea timpului vom folosi o variabilă numerică, cu valori în $\mathbb{N}^*$. În ecuațiile următoare, vom folosi notații din logica temporală de tip *modal μ-calculus*, care este o extindere a logicii Hennessy-Milner, construită peste o structură Kripke. Pentru o descriere detaliată a structurilor și notațiilor utilizate, poate fi consultat articolul publicat de Muller-olm et al. (1999).

Setul de variabile care descriu starea sistemului poate fi reprezentat astfel:

$$\begin{gathered}
\text{Vars} \triangleq < \text{light_state}, \text{motion_state}, \text{now} > \text{, unde} \\
\text{light_state} \in \text{LightStates} = \{\text{„on”}, \text{„off”}\}, \\
\text{motion_state} \in \text{MotionStates} = \{\text{„detected”}, \text{„not detected”}\}, \\
\text{now} \in \mathbb{N}^*
\end{gathered}$$

Starea inițială a sistemului fiind următoarea:

$$\begin{aligned}\text{Init} \triangleq & \wedge \text{light\_state} = \text{„off”} \\ & \wedge \text{motion\_state} = \text{„not detected”} \\ & \wedge \text{now} = 1\end{aligned}$$

Să definim, acum, pe rând acțiunile care determină tranzițiile stărilor. Vom nota cu x' starea viitoare a variabilei x, iar pentru a menține notația scurtă, $x' = x$ va fi x_u. Cea mai simplă este cea care determină trecerea timpului:

$$\text{clock} \triangleq \wedge \text{now}' = \text{now} + 1 \wedge \text{light\_state}_\text{u} \wedge \text{motion\_state}_\text{u}$$

Apoi, acțiunea indusă de „When motion is detected, turn on the switch”, poate fi descrisă astfel:

$$\text{motion\_sensor} \triangleq \wedge \exists s \in \text{MotionStates} : (\wedge \text{motion\_state}' = s \wedge f_\text{motion}(s)) \wedge \text{now}_\text{u}$$

Unde f_motion este definită astfel:

$$f_\text{motion}(x) = \begin{cases} \text{light\_state}' = \text{„on”}, & \text{dacă } x = \text{„detected”} \\ \text{light\_state}_\text{u}, & \text{altfel} \end{cases}$$

În final, acțiunea care descrie „Every day at midnight, turn off the switch” este următoarea:

$$\text{light\_timer} \triangleq \wedge f_\text{time}(\text{now}) \wedge \text{motion\_state}_\text{u} \wedge \text{now}_\text{u}$$

Unde f_time este definită astfel (NB. nu suntem interesați de definiția concretă a funcției IsMidnight):

$$f_\text{time}(x) = \begin{cases} \text{light\_state}' = \text{„off”}, & \text{dacă IsMidnight}(x) = \mathit{true} \\ \text{light\_state}_\text{u}, & \text{altfel} \end{cases}$$

Putem, acum, să definim acțiunea Next care denotă o tranziție ramificată folosind oricare din acțiunile definite anterior:

$$\text{Next} \triangleq \vee\text{clock} \vee \text{motion\_sensor} \vee \text{light\_timer}$$

Proprietatea principală pe care vrem să o respecte sistemul definit este următoarea: „Dacă este detectată mișcare, becul este aprins.”, enunțată formal:

$$\text{MotionLightValid} \triangleq (\text{motion\_state} = \text{„detected”}) \Rightarrow (\text{light\_state} = \text{„on”})$$

În final, putem defini specificația completă a sistemului:

$$\text{Spec} \triangleq \text{Init} \wedge [\text{Next}]_{\text{Vars} \wedge \text{MotionLightValid}}$$

Se poate observa ușor (fără a utiliza un computer) că dacă avem condițiile următoare:

$$\text{motion\_sensor} = \text{„detected”} \wedge \text{IsMidnight}(\text{now}) = \mathit{true},$$

formula MotionLightValid nu va fi satisfăcută, astfel fiind evidențiat un defect în proiectarea sistemului.

Dacă am considera însă un sistem complex, evaluarea manuală a tuturor stărilor modelului nu este fezabilă. Pentru verificarea automată a modelului, putem folosi limbajele *TLA+* și *PlusCal* proiectate de Laslie Lamport împreună cu software-ul de verificare automată *TLC*.

TLA+ permite modelarea unui sistem folosind o sintaxă apropiată de cea matematică, iar *PlusCal* este un limbaj cu sintaxă imperativă ce permite descrierea algoritmilor, aceștia fiind traduși într-o specificație *TLA+* pentru a fi verificați automat. O descriere amănunțită a limbajului poate fi consultată în articolul lui Leslie Lamport (Mai 1994), „The temporal logic of actions”, în *ACM Transactions on Programming Languages and Systems* 16.3, pp. 872–923, DOI: 10.1145/177492.177726, URL: https://doi.org/10.1145/177492.177726. În anexa C, fragmentul de cod C.1, se găsește specificația completă a sistemului descris anterior.

Aplicând această tehnică de testare pe suita de aplicații construită în cadrul lucrării, putem identifica doar defectele la nivel de persistență în sistem, deoarece este evaluat doar modelul sistemului, nu și implementarea sa. În scenariile simple (cum ar fi cel al unei locuințe inteligente), raportul dintre efort și beneficii nu este avantajos în cazul acestei tehnici de testare, însă putem specula că în cazul proiectării sistemelor complexe la scară mult mai mare (de exemplu un oraș inteligent), verificarea formală poate prevedea încă din timpul proiectării un număr larg de defecte foarte subtile.

Încheiem prezentul capitol cu concluzia că suita de aplicații construită, în ciuda limitărilor sale, s-a dovedit utilă pentru evaluarea tehnicilor de testare abordate. În tabelul B.1 din anexa B, avem un sumar comparativ al caracteristicilor observate pentru fiecare tehnică de testare cu care am desfășurat experimente.

Capitolul 6

Concluzii

Așa cum am stabilit la începutul lucrării, construirea setului de aplicații a fost fundamentată pe modelarea teoretică și tehnologică a sistemelor IoT prezentată cititorului, iar tehnicile de testare evaluate servesc drept exemplu pentru viitori cercetători interesați de îmbunătățirea practicilor de testare. Astfel, constatăm că obiectivele stabilite în capitolul 2 au fost atinse.

Luând în considerare limitările suitei de aplicații construită, întrevedem ca direcții viitoare de cercetare: extinderea setului la aplicații reale, din industrie, utilizarea mai multor protocoale de comunicație specifice, cum ar fi CoAP sau ZigBee și diversificarea defectelor injectate în suită. Vom putea, astfel, să ne apropiem și mai mult de provocările din scenariile reale.

În urma analizei tehnicilor de testare, am observat necesitatea unei metodologii riguroase de utilizare a suitei de aplicații. În cercetările viitoare, ar putea fi conceput un ghid de metodologie. De asemenea, pentru transparentizarea rezultatelor, ar putea fi construită o platformă care să pună la dispoziție măsurători și evaluare automată pentru diferite tipuri de tehnici proiectate de cercetători. O inițiativă similară care ar putea servi drept model este *Google Fuzz Bench* [1].

În plan teoretic, un subiect promițător pentru viitoarele cercetări este verificarea formală. Modelele matematice din literatură care descriu sistemele IoT sunt încă în stadii incipiente, existând o plajă largă de îmbunătățiri ce pot fi aduse.

Conchidem că suita de aplicații Internet of Things construită își îndeplinește țelul, aceasta reprezentând un mediu stabil pentru analizarea, evaluarea și compararea tehnicilor de testare.

[1] Adresa platformei se află la https://google.github.io/fuzzbench/.

Glosar

AMQP The Advanced Message Queuing Protocol. 21

API Application Programmable Interface. 26, 30, 42, 43

ATPs Automated Theorem Provers. 46

BDD Behaviour-Driven Development. 38, 39

CoAP Constrained Application Protocol. 21, 50

CWE Common Weakness Enumeartion. 44

EDA Event-Driven Architecture. 10, 11, 21, 30

EDAs Event-Driven Architectures. 10

GPS Global Positioning System. 18

HTTP Hypertext Transfer Protocol. 21, 26, 28, 30, 32, 37, 42, 43

IIOT Industrial Internet of Things. 17

IoT Internet of Things. 6–8, 10–22, 24, 25, 29, 31, 33, 38, 43, 50

JSON JavaScript Object Notation. 26, 27, 31, 35, 45

MQTT Message Queue Telemetry Transport. 21, 26, 30, 32, 37

OSI Open Systems Interconnection. 16, 20, 21

P2P Peer-to-Peer. 11

POSIX Portable Operating System Interface. 36

QA Quality Assurance. 12

RFID Radio-Frequency Identification. 18

SASHA Security Assessment of Smart Home Interconnected Applications. 7, 26

SMTs Satisfiability Modulo Theories. 16, 46

SOA Service Oriented Architecture. 17, 19, 20

TDD Test-Driven Development. 38

YAML YAML Ain't Markup Language™. 30, 31

Bibliografie

Ahmed, Bestoun S., Miroslav Bures, Karel Frajtak și Tomas Cerny (2019), „Aspects of Quality in Internet of Things (IoT) Solutions: A Systematic Mapping Study", în *IEEE Access* 7, pp. 13758–13780, DOI: 10.1109/access.2019.2893493, URL: https://doi.org/10.1109/access.2019.2893493.

Ashton, Kevin (Iun. 2009), „That 'Internet of Things' Thing", în *RFID Journal*, URL: https://www.rfidjournal.com/that-internet-of-things-thing.

Atlidakis, Vaggelis, Patrice Godefroid și Marina Polishchuk (Mai 2019), „RESTler: Stateful REST API Fuzzing", în *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, DOI: 10.1109/icse.2019.00083, URL: https://doi.org/10.1109/icse.2019.00083.

Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang și Wojciech Zaremba (2016), *OpenAI Gym*, DOI: 10.48550/ARXIV.1606.01540, URL: https://arxiv.org/abs/1606.01540.

Bures, Miroslav, Matej Klima, Vaclav Rechtberger, Xavier Bellekens, Christos Tachtatzis, Robert Atkinson și Bestoun S. Ahmed (2020), „Interoperability and Integration Testing Methods for IoT Systems: A Systematic Mapping Study", în *Software Engineering and Formal Methods*, Springer International Publishing, pp. 93–112, DOI: 10.1007/978-3-030-58768-0_6.

Cortés, Mariela, Raphael Saraiva, Marcia Souza, Patricia Mello și Pamella Soares (2019), „Adoption of Software Testing in Internet of Things", în *Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing - SAST 2019*, ACM Press, DOI: 10.1145/3356317.3356326, URL: https://doi.org/10.1145/3356317.3356326.

Cristea, Rareș, Mihail Feraru și Ciprian Păduraru (2022), „Building blocks for IoT testing - a benchmark of IoT apps and a functional testing framework", în *4th International Workshop on Software Engineering Research & Practices for the Internet of Things*.

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

Garousi, Vahid, Ali Mesbah, Aysu Betin-Can și Shabnam Mirshokraie (Aug. 2013), „A systematic mapping study of web application testing", în *Information and Software Technology* 55.8, pp. 1374–1396, DOI: 10.1016/j.infsof.2013.02.006, URL: https://doi.org/10.1016/j.infsof.2013.02.006.

Gosain, Anjana și Ganga Sharma (2015), „Static Analysis: A Survey of Techniques and Tools", în *Intelligent Computing and Applications*, Springer India, pp. 581–591, DOI: 10.1007/978-81-322-2268-2_59, URL: https://doi.org/10.1007/978-81-322-2268-2_59.

Howden, W.E. (Mar. 1980), „Functional Program Testing", în *IEEE Transactions on Software Engineering* SE-6.2, pp. 162–169, DOI: 10.1109/tse.1980.230467, URL: https://doi.org/10.1109/tse.1980.230467.

Huang, Xin, Paul Craig, Hangyu Lin și Zheng Yan (Mai 2015), „SecIoT: a security framework for the Internet of Things", în *Security and Communication Networks* 9.16, pp. 3083–3094, DOI: 10.1002/sec.1259, URL: https://doi.org/10.1002/sec.1259.

Insight, Forbes (2017), „Internet of Things-From theory to reality", în *Forbes Insight, Jersey City*.

Jackie Fenn, Hung LeHong (2011), *Hype Cycle for Emerging Technologies*, rap. teh., Gartner.

Khan, Rafiullah, Sarmad Ullah Khan, Rifaqat Zaheer și Shahid Khan (Dec. 2012), „Future internet: The internet of things architecture, possible applications and key challenges", în *2012 10th International Conference on Frontiers of Information Technology*, Islamabad, Pakistan: IEEE.

Khodadadi, F., A.V. Dastjerdi și R. Buyya (2016), „Internet of Things: an overview", în *Internet of Things*, Elsevier, pp. 3–27, DOI: 10.1016/b978-0-12-805395-9.00001-0, URL: https://doi.org/10.1016/b978-0-12-805395-9.00001-0.

Lamport, Leslie (2005), „Real-Time Model Checking Is Really Simple", în *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 162–175, DOI: 10.1007/11560548_14, URL: https://doi.org/10.1007/11560548_14.

— (Mai 1994), „The temporal logic of actions", în *ACM Transactions on Programming Languages and Systems* 16.3, pp. 872–923, DOI: 10.1145/177492.177726, URL: https://doi.org/10.1145/177492.177726.

Lee, In și Kyoochun Lee (Iul. 2015), „The Internet of Things (IoT): Applications, investments, and challenges for enterprises", în *Business Horizons* 58.4, pp. 431–440, DOI: 10.1016/j.bushor.2015.03.008, URL: https://doi.org/10.1016/j.bushor.2015.03.008.

Lin, Jie, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang și Wei Zhao (Oct. 2017), „A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications", în *IEEE Internet of Things Journal* 4.5, pp. 1125–1142, DOI: 10.1109/jiot.2017.2683200, URL: https://doi.org/10.1109/jiot.2017.2683200.

Manes, Valentin J.M., HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz și Maverick Woo (Nov. 2021), „The Art, Science, and Engineering of Fuzzing: A Survey", în *IEEE Transactions on Software Engineering* 47.11, pp. 2312–2331, DOI: 10.1109/tse.2019.2946563, URL: https://doi.org/10.1109/tse.2019.2946563.

Muller-olm, Markus, Schmidt, David și Bernhard Steffen (Ian. 1999), „Model-Checking: A Tutorial Introduction", în vol. 1694, pp. 330–354.

Nord, Jeretta Horn, Alex Koohang și Joanna Paliszkiewicz (Nov. 2019), „The Internet of Things: Review and theoretical framework", în *Expert Systems with Applications* 133, pp. 97–108, DOI: 10.1016/j.eswa.2019.05.014, URL: https://doi.org/10.1016/j.eswa.2019.05.014.

Păduraru, Ciprian, Rareș Cristea și Eduard Stăniloiu (Iun. 2021), „RiverIoT - a Framework Proposal for Fuzzing IoT Applications", în *2021 IEEE/ACM 3rd International Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT)*, IEEE, DOI: 10.1109/serp4iot52556.2021.00015, URL: https://doi.org/10.1109/serp4iot52556.2021.00015.

Raji, R.S. (Iun. 1994), „Smart networks for control", în *IEEE Spectrum* 31.6, pp. 49–55, DOI: 10.1109/6.284793, URL: https://doi.org/10.1109/6.284793.

Size of the global Internet of Things (IoT) market from 2009 to 2019 (f.d.), Accesat: 15 Martie 2022, URL: https://www.statista.com/statistics/485136/global-internet-of-things-market-size/.

Solis, Carlos și Xiaofeng Wang (Aug. 2011), „A Study of the Characteristics of Behaviour Driven Development", în *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, IEEE, DOI: 10.1109/seaa.2011.76, URL: https://doi.org/10.1109/seaa.2011.76.

Tan, Lu și Neng Wang (Aug. 2010), „Future internet: The Internet of Things", în *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*, IEEE, DOI: 10.1109/icacte.2010.5579543, URL: https://doi.org/10.1109/icacte.2010.5579543.

Weiser, Mark (Iul. 1999), „The computer for the 21 st century", în *ACM SIGMOBILE Mobile Computing and Communications Review* 3.3, pp. 3–11, DOI: 10.1145/329124.329126, URL: https://doi.org/10.1145/329124.329126.

Wu, Miao, Ting-Jie Lu, Fei-Yang Ling, Jing Sun și Hui-Ying Du (Aug. 2010), „Research on the architecture of Internet of Things", în *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*, IEEE, DOI: 10.1109/icacte.2010.5579493, URL: https://doi.org/10.1109/icacte.2010.5579493.

Yousuf, Tasneem, Rwan Mahmoud, Fadi Aloul și Imran Zualkernan (Dec. 2015), „Internet of Things (IoT) Security: Current Status, Challenges and Countermeasures", în *Inter-*

national Journal for Information Security Research 5, pp. 608–616, DOI: 10.20533/ijisr.2042.4639.2015.0070.

Zeller, Andreas, Rahul Gopinath, Marcel Böhme, Gordon Fraser și Christian Holler (2021), *The Fuzzing Book*, Retrieved 2021-10-26 15:30:20+02:00, CISPA Helmholtz Center for Information Security, URL: https://www.fuzzingbook.org/.

Zhou, Wei, Chen Cao, Dongdong Huo, Kai Cheng, Lan Zhang, Le Guan, Tao Liu, Yan Jia, Yaowen Zheng, Yuqing Zhang, Limin Sun, Yazhe Wang și Peng Liu (Iul. 2021), „Reviewing IoT Security via Logic Bugs in IoT Platforms and Systems", în *IEEE Internet of Things Journal* 8.14, pp. 11621–11639, DOI: 10.1109/jiot.2021.3059457, URL: https://doi.org/10.1109/jiot.2021.3059457.

Anexa A

Lista defectelor din suită

Figura A.1: Lista defectelor din suită (A)

Id	Level	Where	Type	Description
a1_activatesolarlamp_luminosity_not_changed	Application	flowerpower	Real, programming error	activateSolarLamp does not change luminosity
a2_dont_call_abs_on_temperature_diff	Rule	Rule 5, smartkettle, windwow	Real, CWE-1284: Improper Validation of Specified Quantity in Input	SmartKettle's temperature decreases for WindWow's temperatures under 0 degrees celsius instead of increasing
a3_dont_call_min_for_brightness	Rule	Rule 4, smarttv	Real, CWE-1284: Improper Validation of Specified Quantity in Input	TV brightness should be set to a maximum of 10, but the value is not validated by the app
a4_dont_check_temperature	Persistence	Rules 2 and 3, flowerpower, windwow	Real, business logic error	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
a5_json_key_not_checked	Application	flowerpower	Real, CWE-476: NULL Pointer Dereference	Does not check for optional key existence in JSON object on PUT /settings
a6_listen_to_localhost	Application	smarteeth	Real, misconfiguration	Smarteeth: "localhost" set as the hostname of the listening server thus refusing outside connections
a7_windwow_set_setting_crash	Application	windwow	Injected, CWE-824: Access of Uninitialized Pointer	Windwow crashes when trying to set luminosity to 25 and curtains are closed on GET /settings/{settingName} /{settingValue}

Figura A.2: Lista defectelor din suită (B)

Id	Level	Where	Type	Description
b1_invalid_sensor_type	Application	flowerpower	**Real**, CWE-457: Use of Uninitialized Variable	sensorNameMap[sensorTypeID] is accessed without checking for existence
b2_smarteeth_config_nullptr	Application	smarteeth	**Injected**, CWE-476: NULL Pointer Dereference	currentConfig is not checked to be non-null
b3_smartkettle_oob_read_boil_hist	Application	smartkettle	**Real**, CWE-125: Out-of-bounds Read	boilHistory[size - 1] is accessed without checking if size == 0
b4_unchecked_map_access	Application	soundsystem	**Injected**	Song IDs are not validated for existence when subscribing
b5_unchecked_error	Application	soundsystem	**Injected**	getReqBodyInto() call is not checked for errors
b6_settings_race_condition	Application	windwow	**Injected**, CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	no locking in setSettingsJSON() in a threaded envionment
b7_command_injection	Application	windwow	**Injected**, CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	Command injection via MQTT client call
b8_double_free	Application	flowerpower	**Injected**, CWE-415: Double Free	Pointer freed twice in specific conditions

Anexa B

Ilustrații suplimentare

Figura B.1: Modelul conceptual Open Systems Interconnection (imagine prealuată de pe Wikipedia)

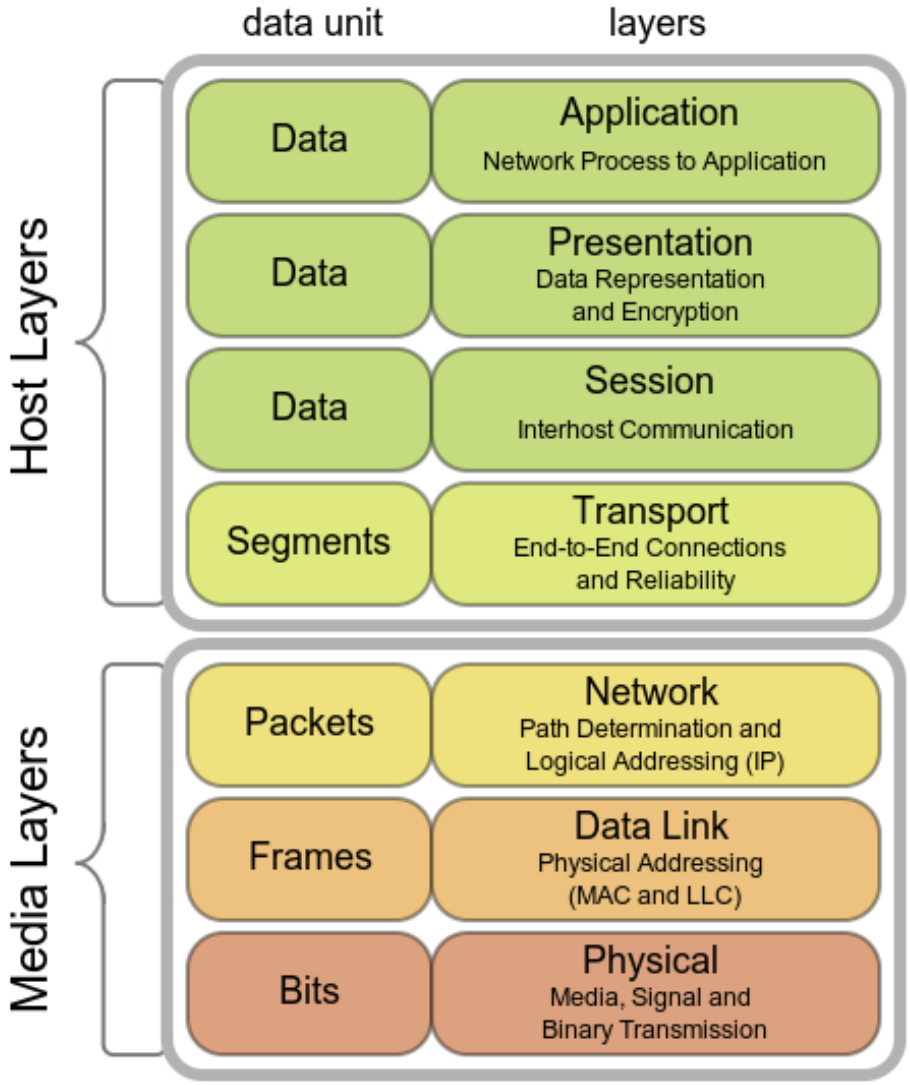
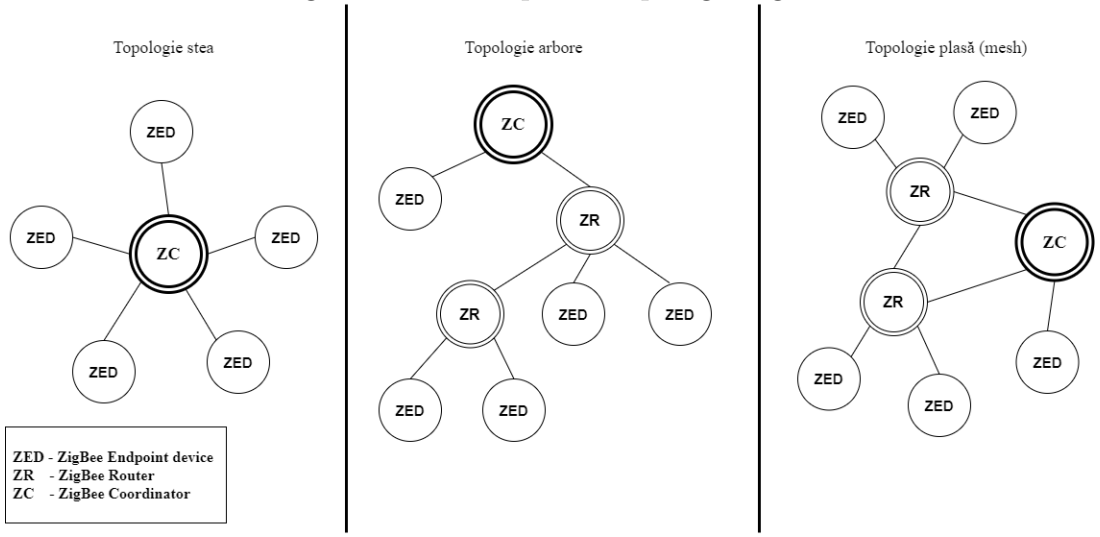


Figura B.2: Exemple de topologii ZigBee



Tehnica de testare	Grad automatizare	Complexitate	Niveluri defecte	Procentaj defecte (cunoscute) detectate (relativ la nivelurile relevante)	Observații
Testare funcțională BDD	Manual	Scăzută	Toate	Nu este relevant	Componentă exploratorie insuficientă, bun pentru scenarii pozitive
Fuzzing cu RESTler	Automat	Medie	Aplicație, flux	TODO: ???	Lipsa înțelegerii stării aplicației, potrivit pentru defecte de logică
Analiză statică cu weggli	Parțial automat	Medie	Aplicație	???	Potrivit pentru defecte de memorie
Analiză statică cu cppcheck	Parțial automat	Scăzută	Aplicație	???	Rată mare de fals-pozitive
Verificare formală cu TLA+	Parțial automat	Ridicată	Persistență	100%	Potrivit pentru defecte de concurență, necesită cunoștințe matematice

Tabela B.1: Sumar comparativ al tehnicilor de testare abordate

Anexa C

Cod sursă suplimentar

```
----- MODULE light_example -----
EXTENDS Naturals, TLC

LightStates == {"on", "off"}
MotionStates == {"detected", "not detected"}
IsMidnight(x) == IF x % 5 = 0 THEN TRUE ELSE FALSE

\(\* --algorithm light\_example
variables
    light\_state = "off",
    motion\_state = "not detected",
    now = 1;

process motion\_sensor = 1
begin
A:
    while TRUE do
        with c\_state \in MotionStates do
            motion\_state := c\_state;

            if c\_state = "detected" then
                light\_state := "on";
            end if;
        end with;
    end while;
end process

process clock = 2
begin
B:
    while TRUE do
        now := now + 1;
    end while;
```

```
end process

process light\_timer = 3
begin
C:
    while TRUE do
        if IsMidnight(now) then
            light\_state := "off";
        end if;
    end while;
end process

end algorithm; *)
\* BEGIN TRANSLATION
VARIABLES light\_state, motion\_state, now

vars == << light\_state, motion\_state, now >>

ProcSet == {1} \cup {2} \cup {3}

Init == \(\* Global variables *)
        /\ light\_state = "off"
        /\ motion\_state = "not detected"
        /\ now = 1

motion\_sensor == /\ \E c\_state \in MotionStates:
                      /\ motion\_state' = c\_state
                      /\ IF c\_state = "detected"
                            THEN /\ light\_state' = "on"
                            ELSE /\ TRUE
                                 /\ UNCHANGED light\_state
                 /\ now' = now

clock == /\ now' = now + 1
         /\ UNCHANGED << light\_state, motion\_state >>

light\_timer == /\ IF IsMidnight(now)
                     THEN /\ light\_state' = "off"
                     ELSE /\ TRUE
                          /\ UNCHANGED light\_state
               /\ UNCHANGED << motion\_state, now >>

Next == motion\_sensor \/ clock \/ light\_timer

Spec == Init /\ [][Next]\_vars

\* END TRANSLATION
```

```
MotionLightValid == (motion\_state = "detected") => (light\_state = "on")

=========
```

Listing C.1: Specificația TLA+ a scenariului descris în capitolul 5, secțiunea despre verificare formală