Um Estudo Empírico Sobre a Arquitetura e a Propriedade Coletiva de Projetos de Sistemas Operacionais de Tempo Real Livres

Rafael de Carvalho
Departamento de Computação e Sistemas (DECSI) – Universidade Federal de Ouro Preto (UFOP)
CEP 35.931-008 - João Monlevade - MG - Brasil
rafael.c@aluno.ufop.edu.br

Igor Muzetti Pereira
Departamento de Computação e Sistemas (DECSI) – Universidade Federal de Ouro Preto (UFOP)
CEP 35.931-008 - João Monlevade - MG - Brasil
igormuzetti@ufop.edu.br

***Abstract*—With the growth of Internet of Things (IoT) systems, there has been an increase in demand for Real-Time Operating Systems (RTOS). Proportionally, the number of projects focused on developing this type of system has also grown. Some of these projects are produced by Free/Libre Open Source Software (FLOSS) communities. Since production by these communities involves many developers, a common practice includes modularization and collective code ownership. Given this, this study aimed to analyze the source code and understand collective code ownership in relation to modularity and its connection with developers. A sample of nine GitHub projects with over a thousand stars was used. Tools were introduced to perform calculations and analyze whether a developer can be considered a code owner. The algorithm is based on the Simple Matrix and the Degree of Authorship (DOA) method. Notebooks were used to transform the results into information with the help of charts. From these analyses, it was possible to determine the number of contributors and owners in each project. It was found that, on average, the number of owners per module ranged from 15.83% to 46.56%; the number of modules per owner, which ranged from two to twenty-seven, showed that one owner could be responsible for several modules; and finally, the number of owners per module, where it was possible to conclude that, on average, a module may have one or two developers as owners.**

***Keywords*—modularity; collective code ownership; real time system operating.**

***Resumo*—Face ao crescimento de sistemas de internet das coisas (IoT), têm-se observado um aumento na demanda por Sistemas Operacionais de Tempo Real (RTOS). Proporcionalmente, cresceu o número de projetos focados em desenvolver esse tipo de sistema. Entre esses projetos, pode-se encontrar aqueles produzidos por comunidades de software livre (FLOSS). Dado que a produção por essas comunidades envolvem muitos desenvolvedores, uma prática comum abrange a modularização e a propriedade coletiva de código. Diante disso, este estudo objetivou analisar o código-fonte e entender a propriedade coletiva de código quanto à modularidade e a relação dela com os desenvolvedores. É utilizada uma amostra de nove projetos do GitHub com mais de mil estrelas. Foram apresentadas ferramentas para executar cálculos e analisar se o desenvolvedor pode ou não ser considerado proprietário de código. O algoritmo é baseado na Matriz Simples e no método Degree of Authorship (DOA). Já os notebooks transformaram os resultados em informação com auxílio de gráficos. A partir dessas análises foi possível determinar o número de contribuidores e de responsáveis em cada projeto. Verificou-se que em média o número de responsáveis por módulo variou entre 15,83% e 46,56%; a quantidade de módulos por responsável, a qual variou entre dois e vinte e sete, demonstrou que um responsável pode ser proprietário de vários módulos; e por fim a quantidade de responsáveis por módulo, em que foi possível concluir que em média um módulo pode possuir um ou dois desenvolvedores como proprietários.**

***Palavras-chave*—modularidade; propriedade coletiva de código; sistema operacional de tempo real.**

I. Introdução

Sistemas de Internet das Coisas (IoT) são dispositivos de *hardware* e *software* conectados entre si e com a Internet. Eles capturam dados do ambiente, processam esses dados e geram informações aos usuários. Muitos objetos do dia a dia estão se tornando dispositivos conectados, como lâmpadas de LED, máquinas de café, TVs, carros e cidades inteligentes [1].

Alguns sistemas utilizam Sistemas Operacionais de Tempo Real (*Real Time Operating System - RTOS*), o qual oferece suporte a capacidades em tempo real e permite tempos de resposta garantidos a eventos. Ele usa multitarefa preemptiva com baixa latência e eficiência energética [2]. Multitarefa preemptiva se trata de um modelo em que pode ser implementado um conjunto de tarefas em um processador específico, de forma que cada tarefa 'i' chega infinitamente, com cada chegada separada da anterior por pelo menos 'Ti' de tempo [3]. Baixa latência se refere ao tempo mínimo de atraso entre a entrada de um comando e a resposta do sistema. Já eficiência energética consiste em um sistema ter o mínimo de consumo de energia quando submetido a várias cargas de tráfego [4].

Dessa forma, a característica principal de um RTOS é o nível de consistência em relação ao tempo necessário para aceitar e completar a tarefa de um aplicativo [2]. Para implementar essas funcionalidades, a construção desse *software* se torna algo complexo em razão da sua estrutura e por envolver o uso de compiladores, aplicativos e bibliotecas de suporte para *drivers*, protocolos de comunicação e muito mais [1].

Nesse contexto, no cenário atual, nota-se um crescente aumento na demanda por esses sistemas em diversas áreas [5]. Em decorrência disso, há diversos projetos *Free/Libre and Open Source Software* (FLOSS) sendo desenvolvidos, cujo propósito é a criação de sistemas RTOS. A prática de desenvolver sistemas através de projetos FLOSS corresponde a comunidades baseadas na Internet de desenvolvedores que colaboram voluntariamente para criar *software* para eles próprios ou para suas organizações [6].

Dado que esses projetos envolvem vários desenvolvedores, uma prática comum é organizá-los em módulos para reduzir o tempo de desenvolvimento, facilitar o gerenciamento e a sua evolução. Essa abordagem permite que grupos separados possam trabalhar em cada módulo em lugares distintos. Além disso, a modularização possibilita realizar mudanças em um módulo sem afetar os outros. Nesse contexto, um módulo pode ser considerado uma atribuição de responsabilidade [7].

Como todos os membros de uma comunidade de desenvolvedores podem modificar qualquer parte do código, é essencial cultivar um senso de propriedade na comunidade. Isso contribui para aumentar a participação de novos desenvolvedores e daqueles menos ativos [8]. Como preconiza Conway e Spandforfer (1968), existe uma relação direta entre a estrutura dos módulos de um sistema e a estrutura da equipe.

Quanto à representação das estruturas de *software*, geralmente, não é considerado o elemento humano. Em diagramas de classe e de componentes e em histórias de usuário não são inclusas informações sobre as equipes. Diante disso, os modelos de *software* têm dificuldade em demonstrar a relação entre a estrutura de módulos e os desenvolvedores. Mapear essa relação é essencial para entender e ter a capacidade de agir em inúmeros cenários. Por exemplo, quando se encontra uma falha em um módulo, ela deve ser reportada ao contribuidor responsável para que ele possa promover a manutenção do código ou repassar seu conhecimento para que outro colaborador possa fazê-lo [8].

Outro cenário seria a identificação do *Truck Factor* (TF). TF faz uma analogia ao número de pessoas que podem ser atropeladas por um caminhão antes de um sistema enfrentar sérios problemas. Ou seja, essa métrica indica que um desenvolvedor é proprietário de vários módulos, de modo que, caso ele saia, o projeto poderá ser impactado negativamente. Portanto, identificá-lo é fundamental para mostrar se a equipe tem rotatividade e está preparada para mitigar eventual problema [9].

Logo, o presente estudo tem como objetivo analisar o código-fonte de projetos RTOS FLOSS hospedados no GitHub e entender a propriedade coletiva do código. Além disso, busca compreender a modularidade desses *software* e sua relação com os colaboradores. Esse objetivo foi fragmentado em duas Questões de Pesquisa (QPs): QP1. Quantos contribuidores que participaram de cada projeto podem ser considerados responsáveis por cada módulo? QP2. Em projetos de RTOS FLOSS é comum existir módulos com muito ou poucos responsáveis e um contribuidor ser responsável por muitos ou poucos módulos?

Para responder essas questões, foi desenvolvido um algoritmo em Python. Ele se baseia em dois métodos que são apresentados neste artigo. A matriz simples em que linhas representam os módulos e colunas os desenvolvedores [8]. E o método *Degree of Authorship* (DOA) que mede o grau de autoria de cada desenvolvedor [9]. São utilizados dados extraídos via Pydriller, que correspondem ao número de *commits* de um módulo, para identificar os desenvolvedores proprietários. No resultado, uma matriz é exportada gerando um arquivo *Comma Separated Values* (CSV). Em sequência, ela é utilizada em *notebooks* com o intuito de transformar os resultados encontrados em informação e gerar gráficos para facilitar a análise. Essas ferramentas foram usadas em nove projetos FLOSS com mais de mil estrelas no GitHub.

Este artigo está dividido da seguinte maneira. A seção 2 apresenta os trabalhos correlatos. A seção 3 explica sobre a metodologia, indicando as ferramentas que foram utilizadas para alcançar os resultados do estudo. A seção 4 demonstra os resultados obtidos e traz discussões sobre o estudo. As ameaças à validade desse estudo está na seção 5 . Por fim a seção 6 conclui o estudo trazendo as considerações finais.

II. Trabalhos Correlatos

As organizações que projetam sistemas são limitadas a produzir *designs* que são cópias das próprias estruturas de comunicação. Isso significa que a forma como a organização se comunica, pode influenciar diretamente na modularização do sistema que ela cria. Esse fato tem importantes implicações para o gerenciamento dos módulos. Sendo assim, os módulos devem ser organizados de acordo com a necessidade de comunicação, assim como ser flexível e ter uma equipe enxuta para melhor eficiência [10].

A modularização corresponde a uma ferramenta que visa tornar um sistema mais flexível e mais compreensível. Ela também permite a redução do tempo de desenvolvimento [7]. A modularidade fez com que a indústria evoluísse de uma estrutura inicialmente concentrada para outra altamente dispersa. Dessa forma, ela permite que tarefas sejam divididas entre grupos que possam trabalhar de forma independente, e

não precisem fazer parte da mesma empresa. Para otimizar processos de desenvolvimentos de produtos, criou-se o conceito *Dependency Structure Matrix* (DSM). Ele apresenta uma matriz simples que possibilita melhor visualização das tarefas diante dos módulos de um sistema [11].

A propriedade é um aspecto essencial para o desenvolvimento de *software*. Nesta pesquisa, os autores entenderam que medidas de propriedade, como o número de desenvolvedores com pouco conhecimento ou a proporção de propriedade do principal responsável, podem indicar possíveis defeitos antes e após o lançamento. Os pesquisadores classificaram os contribuidores como maiores (propriedade em 5% ou mais) e menores (propriedade abaixo de 5%). Eles demonstraram que binários do Windows com propriedade fraca, ou seja, aqueles em que muitos desenvolvedores contribuem com pequenas quantidades de código, são mais propensos a apresentarem falhas do que binários com propriedade forte. A pesquisa utilizou como amostra o Windows Vista e o Windows 7 [12].

Baseando-se no estudo de Bird et al. (2011), neste estudo, os autores também confirmaram, por meio da implantação de novas métricas, que a propriedade de código se correlaciona com a qualidade do sistema. Foram utilizados nesta amostra os produtos da Microsoft: Office, Office365, Windows e Exchange. Optou-se por analisar os diretórios de código, uma vez que eles agrupam vários arquivos e podem ter propriedades em comum. Essa abordagem ajudou a evitar conjunto de dados desequilibrados. Outro ponto importante foi identificar três tipos distintos de propriedade: individual, em que um desenvolvedor é o principal responsável; coletiva, em que um grupo compartilha a responsabilidade; e não-propriedade, em que há falta dela. Ademais, segundo esse estudo, a falta de propriedade costuma ocorrer em razão da transferência de propriedade entre equipes, atividades de correção de *bugs*, refatoração de código e problemas arquiteturais [13].

Em razão de poucos estudos explorarem a métrica *Truck Factor* (TF), neste trabalho, os autores propuseram uma nova abordagem para estimar os valores de TF. Essa abordagem compreende calcular tais valores se baseando no histórico de commits e utilizando um modelo denominado *Degree-of-Autorship* (DOA). Ela foi aplicada a um conjunto de 133 projetos do GitHub. Posteriormente, foi realizado um *survey* para avaliar a confiabilidade dos resultados. Entre os sistemas analisados, chegou-se ao resultado de que 65% dos sistemas tem um TF maior ou igual a 2 [9]. Vale mencionar que o Método DOA foi proposto com o intuito de identificar o conhecimento dos desenvolvedores acerca de determinadas partes do código, e, consequentemente a sua propriedade [14]

Em razão da métrica TF, é reconhecido que a presença de desenvolvedores, que são os únicos a conhecer certas partes críticas de um sistema, conceituados como heróis, pode aumentar o impacto negativo no projeto, especialmente em momentos que eles o venham a deixar. Contudo, é pouco investigado a relação entre heróis e as tarefas de manutenção. Os autores implementaram uma ferramenta para poder identificá-los. Esse estudo foi realizado em 37 projetos de *software* livre. Os resultados mostraram que a presença desses heróis são comuns em projetos FLOSS, bem como a presença deles pode ser benéfica por reduzir o tempo para implementar solicitações de mudanças [15].

Esta pesquisa se baseou nos mesmos modelos de cálculo utilizados no trabalho de Santos (2018). A principal diferença está nas amostras analisadas. Enquanto o seu trabalho focou em projetos FLOSS escritos em linguagem JAVA, o presente estudo selecionou projetos FLOSS em linguagem C e CPP focados na criação de sistemas RTOS. No trabalho de Santos (2018) e no presente trabalho, foi utilizado o Método DOA para calcular a propriedade de códigos em sistemas distintos. O modelo proporciona uma boa estimativa, bem como respeita o fluxo de experiência de cada desenvolvedor. Isso ocorre porque o número de alterações no repositório de origem está diretamente relacionado ao grau de conhecimento do desenvolvedor sobre determinado arquivo [9].

Esses estudos, quando comparados com o de Avelino (2016), abrangem um número maior de desenvolvedores como parte dos principais contribuidores. Também investigam quais são os módulos que há contribuição de colaboradores e os seus respectivos valores em porcentagem. De forma semelhante ao trabalho de Greiler et al. (2015), optou-se por analisar os diretórios de código. Todavia, o nível de granularidade é definido pelo número de alterações realizadas. Para classificar os contribuidores e identificar o responsável principal, utilizou-se o Método DOA e 80% das mudanças do código. Como o referido método foi criado para selecionar proprietários de módulos, esta se mostrou uma estratégia mais acertada para determinar quem detêm responsabilidade sobre cada arquivo. Entretanto, diferente do estudo de Fritz (2010), não é possível nesses trabalhos identificar quando um desenvolvedor acessou um arquivo. Importante mencionar que foi analisado todo o histórico de commits [8].

Outra importante diferença entre esse estudo e o trabalho feito por Santos (2018) está relacionada com a presença de *notebooks*. Nesse trabalho, eles foram elaborados para facilitar a visualização dos dados gerados pelo algoritmo em arquivo CSV, enquanto naquele foram criadas matrizes de forma manual para melhorar a análise.

III. Metodologia

O desenvolvimento deste trabalho compreende algumas etapas principais. A primeira etapa consiste na seleção de projetos do GitHub e tipos de arquivos a serem analisados. A segunda

etapa envolve a mineração de dados desses projetos com o auxílio da ferramenta Pydriller. Em seguida, executa-se um algoritmo baseado em uma matriz simples e o Método DOA para a construção de tabelas com as propriedades de código de cada projeto. Por fim, utilizam-se *notebooks* para construir gráficos e facilitar o entendimento das questões propostas no presente estudo.

A. Seleção dos Projetos

A seleção se restringe a nove projetos FLOSS focados na construção de sistemas RTOS do GitHub com mais de mil estrelas. Como tais projetos apresentam um grande número de tipos de arquivos por serem complexos e robustos, esta pesquisa se limitou ao escopo de arquivos .C e .CPP. Em razão disso, foram considerados todos diretórios com a presença desses arquivos, inclusive aqueles relacionados a testes. Também foram considerados apenas os *branchs* principais de cada projeto: o *branch main* no Amazon-FreeRTOS, FreeRTOS, RT-Threads e Zephyr; o *branch master* no Contiki, Mbed-OS, NuttX e RIOT; e o *branch develop* no Contiki-NG. Importante frisar que neste estudo cada arquivo é considerado um módulo. A pesquisa dos projetos foi realizada diretamente na plataforma GitHub. Dessa forma, os projetos selecionados foram aws/Amazon-FreeRTOS[1], contiki-os/Contiki[2], contiki-ng/Contiki-NG[3], freertos/FreeRTOS[4], ARMmbed/Mbed-OS[5], apache/NuttX[6], RIOT-OS/RIOT[7], rt-thread/RT-Thread[8] e zephyrproject-rtos/Zephyr[9].

B. Mineração de Softwares com auxílio do Pydriller

Pydriller[10] é um *framework* projetado para auxiliar desenvolvedores na mineração de repositórios de *software*. Ele proporciona APIs intuitivas que permitem a extração de dados essenciais de repositórios Git, como *commits*, desenvolvedores, modificações, *diffs* e código-fonte. Além disso, ele possibilita aos desenvolvedores a manipulação avançada dos dados extraídos, assim como a exportação de maneira rápida para diversos formatos, como arquivos CSV e bancos de dados [16].

O Pydriller foi utilizado nesse trabalho para gerar bases que contêm dados necessários para a criação das tabelas. Tais bases correspondem a arquivos do tipo CSV que são gerados por esse *framework* ao final do processo de mineração de cada repositório. A TABELA I demonstra parte das bases de dados extraídas do projeto, que consiste em dezesseis tipos de modificações. Os nomes dos arquivos e diretórios, bem como os nomes e os e-mails dos autores, foram modificados para melhor visualização e garantir o anonimato dos colaboradores. Perceba que na coluna 'Diretório Antigo', em algumas linhas, não há diretório e nome dos arquivos porque ainda não havia sido criado ou adicionado no GitHub.

TABELA I
TABELA DE COMMITS

Hash do commit	Tipo de Modificação	Diretório Atual	Diretório Antigo	Nome do Autor	E-mail do Autor
8ddf82cf70dc6f951ab477f325dee0efde3ec589	ADD	dir/arquivo1.c		Dev1	dev1@email.com
8ddf82cf70dc6f951ab477f325dee0efde3ec589	ADD	dir/arquivo2.c		Dev1	dev1@email.com
8ddf82cf70dc6f951ab477f325dee0efde3ec589	ADD	dir/arquivo3.c		Dev1	dev1@email.com
8ddf82cf70dc6f951ab477f325dee0efde3ec589	ADD	dir/arquivo4.c		Dev1	dev1@email.com
8ddf82cf70dc6f951ab477f325dee0efde3ec589	ADD	dir/arquivo5.c		Dev1	dev1@email.com
8ddf82cf70dc6f951ab477f325dee0efde3ec589	ADD	dir/arquivo6.c		Dev1	dev1@email.com
8ddf82cf70dc6f951ab477f325dee0efde3ec589	ADD	dir/arquivo7.c		Dev1	dev1@email.com
8ddf82cf70dc6f951ab477f325dee0efde3ec589	ADD	dir/arquivo8.c		Dev1	dev1@email.com
fe95d5c018a7766195667e764cb28eeee4a9bb83	MODIFY	dir/arquivo9.c	dir/arquivo9.c	Dev2	dev2@email.com
fe95d5c018a7766195667e764cb28eeee4a9bb83	MODIFY	dir/arquivo10.c	dir/arquivo10.c	Dev2	dev2@email.com
5b09162bb1812f717e5d3537245c76a7e0fa9c1f	MODIFY	dir/arquivo11.c	dir/arquivo11.c	Dev2	dev2@email.com
df50981dba7da592806485c57bce19b214a2ef28	MODIFY	dir/arquivo12.c	dir/arquivo12.c	Dev2	dev2@email.com
ff00dbecce32aba0433a3c182f5e18727395f85a	MODIFY	dir/arquivo13.c	dir/arquivo13.c	Dev2	dev2@email.com
ff0641fac6c85768241340ccded9b938d1fd51af	MODIFY	dir/arquivo14.c	dir/arquivo14.c	Dev2	dev2@email.com
522b9ad432fd5fbbe069a4cf4319bbab083040b1	MODIFY	dir/arquivo15.c	dir/arquivo15.c	Dev2	dev2@email.com
522b9ad432fd5fbbe069a4cf4319bbab083040b1	MODIFY	dir/arquivo16.c	dir/arquivo16.c	Dev2	dev2@email.com

C. Matrizes Simples e Método DOA

O algoritmo desse trabalho para calcular a propriedade de cada desenvolvedor em relação a cada módulo se baseia tanto em Matrizes Simples quanto no Método DOA. Nessa matriz as linhas são representadas por módulos enquanto as colunas são representadas por desenvolvedores. Cada célula indica a porcentagem de responsabilidade de um autor sobre um módulo. Tal porcentagem reflete a quantidade de *commits* de autoria de um desenvolvedor em relação ao total de *commits* realizados em um determinado arquivo. Isso pode ser demonstrado pela equação 1: [8]:

$$r(d,m) = \frac{\#\{x \in C(m) \mid a(x) = d\}}{\#C(m)} \times 100\% \qquad (1)$$

Nessa equação, r(d,m) se refere à porcentagem de responsabilidade do desenvolvedor d sobre o módulo m; C(m) representa o conjunto de todos os *commits* feitos para o módulo m; x C(m) — a(x) = d conta o número de *commits* no módulo m em que o autor é o desenvolvedor d; enquanto a(x) verifica se o autor do *commit* x é o desenvolvedor d. Vale mencionar que o símbolo significa "contagem de" [8].

Para identificar os desenvolvedores responsáveis pela manutenção de um módulo específico dentro de um projeto de *software* foi adotada a heurística baseada em *commits* (*Commit-Based-Heuristic*). Essa abordagem considera que a responsabilidade de um desenvolvedor é proporcional à quantidade de *commits* que ele realizou em arquivos pertencentes a um determinado módulo. De acordo com essa heurística é importante analisar os arquivos que foram alterados, uma vez que um único *commit* pode afetar vários módulos [17].

[1]https://github.com/aws/amazon-freertos
[2]https://github.com/contiki-os/contiki
[3]https://github.com/contiki-ng/contiki-ng
[4]https://github.com/FreeRTOS/FreeRTOS
[5]https://github.com/ARMmbed/mbed-os
[6]https://github.com/apache/nuttx
[7]https://github.com/RIOT-OS/RIOT
[8]https://github.com/RT-Thread/rt-thread
[9]https://github.com/zephyrproject-rtos/zephyr
[10]https://github.com/ishepard/pydriller

Diante disso, os desenvolvedores são ordenados de forma decrescente pelo número de *commits*. Aqueles principais são os primeiros desenvolvedores cuja soma dos *commits* atinge 80%. Todavia, essa heurística foi adaptada no presente estudo para evitar a presença daqueles com menos de 5% do total de *commits* de um módulo [18].

Para melhor adaptar o algoritmo ao presente trabalho também foi escolhida a métrica DOA para detectar o grau de autoria, a qual é normalizada após o cálculo. Dado um arquivo f com o caminho fp, o grau de autoria de um desenvolvedor d cujo usuário Git foi mapeado para md é dado pela equação 2 [9]:

$$\mathrm{DOA}(md, fp) = 3.293 + 1.098 \times \mathrm{FA}(md, fp) + 0.164 \times \mathrm{DL}(md, fp) - 0.321 \times \ln(1 + \mathrm{AC}(md, fp)) \quad (2)$$

Na equação, o cálculo do DOA necessita de três fatores: (i) primeira autoria (FA), de maneira que se md criou f, FA é 1, caso contrário é 0; (ii) número de entregas (DL), que consiste no número de mudanças em f feitas por md; e (iii) número de aceitações (AC), que compreende o número de mudanças em f feitas por qualquer desenvolvedor, exceto md [9].

O modelo assume FA como sendo o preditor mais forte de autoria. Dados mais recentes relacionados à DL contribuem de modo positivo para a autoria, mas com menos importância. Contudo, mudanças de outros desenvolvedores que se refere à AC diminuem o DOA de alguém, apesar de ser em um ritmo mais lento. Os pesos usados como k = 0,75 e m = 3,293 derivam de experimentos empíricos e parecem fornecer bons resultados [9].

Já os valores 1,098, 0,164 e -0,321 associados às variáveis FA, DL e ln, respectivamente, foram definidos por Fritz (2010) em seu estudo a partir da aplicação de regressão linear múltipla a dados coletados de *logs* de revisão de código-fonte e de *logs* de interação capturados enquanto desenvolvedores trabalhavam.

Depois de definidos os valores de DOA de todos os arquivos alterados por desenvolvedores previamente mapeados, procede-se com a normalização dos resultados. Esse valor normalizado varia entre 0 e 1. É concedido 1 ao desenvolvedor com o DOA absoluto mais alto entre todos aqueles que trabalharam em f. Assim, considera-se um desenvolvedor como autor de um arquivo se seu DOA normalizado for maior que k, e seu DOA absoluto não for inferior ao valor de m [9].

Em síntese, o Método DOA é utilizado para calcular a propriedade e determinar a autoria dos desenvolvedores sobre módulos. Ao realizar tal cálculo o algoritmo faz o mapeamento de propriedade e responsabilidade, gerando um índice de propriedade, que é normalizado e comparado com os parâmetros definidos (k = 0,75 e m = 3,293) para identificar quais autores possuem autoria significativa. Em seguida, a matriz simples é construída, em que as linhas representam módulos e as colunas, desenvolvedores, com cada célula indicando a porcentagem de responsabilidade de um desenvolvedor sobre um módulo, que é calculada a partir da proporção de *commits* atribuíveis a cada um.

D. Algoritmo

O algoritmo desenvolvido para extrair as matrizes foi implementado na linguagem Python. Ele utiliza os dados extraídos pelo *framework* PyDriller. Esses dados compreendem a identificação e a quantidade de *commits* dos desenvolvedores sobre os módulos de cada sistema nos repositórios Git. Foram realizados clones dos repositórios para o ambiente local, facilitando a mineração dos dados, embora o PyDriller permita a mineração diretamente do repositório Git.

Esse algoritmo recebe como parâmetro de entrada o caminho para o repositório em ambiente local. Diretórios com código de terceiros não foram considerados. A saída da execução do algoritmo é um arquivo da matriz em formato *CSV*, que está publicamente disponível para uso e pode ser encontrado em: `https://github.com/rafaeldcx/algorithm-ownership`

E. Notebooks

Notebooks funcionam como cadernos de laboratório, apoiando fluxos de trabalho, código, dados e visualizações que detalham o processo de pesquisa. Eles podem ser lidos por máquinas e humanos, facilitando a interoperabilidade e a comunicação acadêmica. Esses documentos interagem ainda com múltiplos componentes da infraestrutura de bibliotecas digitais, como identificadores digitais, mecanismos de persistência, controle de versão, conjuntos de dados, documentação, *software* e publicações [19].

No presente trabalho, foram utilizados *notebooks* em Python, no formato IPYNB, com o intuito de facilitar a análise de dados. Eles foram criados separadamente para cada projeto. A partir do arquivo CSV gerado pelo algoritmo, são produzidas listas de arquivos associadas a cada responsável. Para os desenvolvedores sem responsabilidade sobre arquivos, é gerada a *string* “Nenhum arquivo encontrado”. Além disso, são criados dois arquivos em formato CSV, que apresentam, respectivamente, o número de responsáveis por módulos e o número de módulos por responsável. A leitura desses arquivos gera dois tipos de gráficos: o *Empirical Cumulative Distribution Function* (ECDF) e o de barras. Vale mencionar que foram usadas as bibliotecas Pandas, Seaborn e Matplotlib. Os *notebooks* estão publicados no GitHub, disponível no link já mencionado.

IV. Resultados e Discussão

Cada projeto gerou um arquivo do tipo CSV, sendo utilizados *notebooks* para melhor visualização dos módulos e responsáveis. Esses arquivos correspondem a matrizes simples que foram geradas para cada projeto. Elas se encontram publicadas no GitHub. A TABELA II exemplifica parte da matriz simples para o projeto Zephyr. Ela demonstra que o desenvolvedor d2 é 100% responsável por vários módulos, enquanto d1, d3, d4 e d6 possuem uma porcentagem baixa de responsabilidade. Isso indica vários casos em que apenas um desenvolvedor possui *commits* para vários módulos. Esse fato pode acarretar um alto risco de manutenibilidade em circunstâncias nas quais esse desenvolvedor responsável por muitos módulos não possa contribuir com o projeto [8].

TABELA II
PARTE DA MATRIZ SIMPLES DO PROJETO ZEPHYR

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
m1	1.47%	86.76%				11.76%						
m2		100%										
m3	12.5%	75%		12.5%								
m4		100%										
m5		100%										
m6		100%										
m7		100%										
m8		100%										
m9		100%										
m10		100%										
m11		100%										
m12		100%										
m13		100%										
m14		100%										
m15		100%										
m16		100%										
m17	7.78%	76.67%	3.33%			7.78%						

Nesse sentido, a presença de responsáveis por muitos módulos pode ser analisada nos gráficos de barras gerados pelo *notebook* com auxílio da biblioteca Matplotlib. Tais gráficos também estão disponíveis no GitHub. Em razão de o projeto Zephyr ser complexo e robusto, com uma grande quantidade de colaboradores e responsáveis, para melhor visualização e análise, seu gráfico se limitou a demonstrar os desenvolvedores responsáveis por mais de 5 arquivos e precisou ser dividido em 3 partes. Ao analisá-lo, pode-se verificar a existência de desenvolvedores definidos pela literatura como ”Heróis”, por serem incansáveis e gerenciarem grandes e críticas porções de um sistema [15]. O gráfico da Figura 1 pode exemplificar o fenômeno do ”herói”. Nota-se que são identificados dois desenvolvedores (2 e 40) que são responsáveis por mais de 3.300 módulos cada um, destoando bastante dos demais. Essa quantidade pode corresponder a uma porcentagem significativa do projeto.

Em relação às questões propostas, foi possível respondê-las após gerar as tabelas, as matrizes de cada sistema e os respectivos notebooks. A TABELA III exibe os resultados de cada projeto analisado. A coluna RespMod (média) se refere à média de responsáveis que há em cada módulo, enquanto ModResp (média) significa a média do número de módulos pertencentes a cada responsável.

TABELA III
COMPARAÇÃO DE PROJETOS

Projetos	Contribuidores	Responsáveis	RespMod (média)	ModResp (média)
FreeRTOS	77	27	1	27
Amazon FreeRTOS	131	61	1	15
Contiki	203	53	2	2
Contiki-NG	281	68	2	3
Riot	371	82	1	4
Apache NuttX	705	122	2	15
Mbed-OS	731	223	2	3
RT-Thread	787	212	2	3
Zephyr	1680	266	2	8

Por meio dos dados apresentados na TABELA III, torna-se possível responder às questões propostas pela pesquisa.

QP1. Quantos contribuidores que participaram de cada projeto podem ser considerados responsáveis por cada módulo?

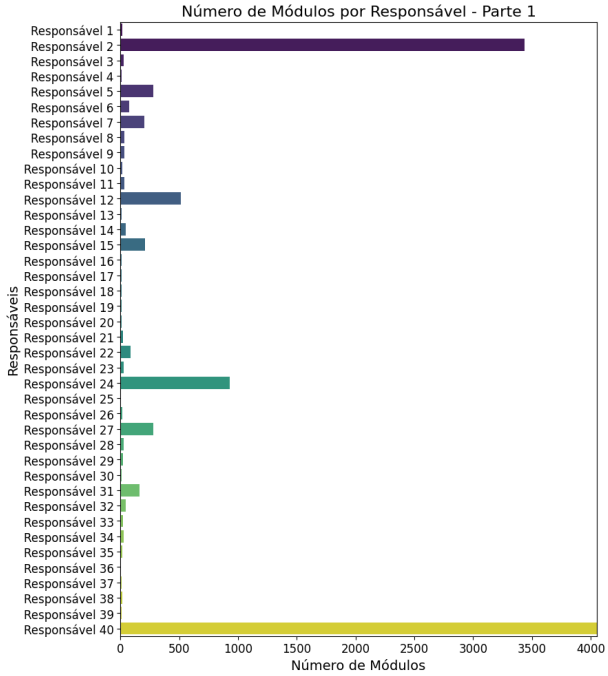

Fig. 1. Projeto Zephyr – Parte 1.

A TABELA III expõe o número de contribuidores que participaram de cada projeto em relação a arquivos em formato .C e .CPP. O projeto com menor número é o FreeRTOS, com 77 colaboradores, enquanto o com maior número corresponde ao Zephyr, com 1.680. Os projetos Mbed-OS, Apache NuttX e RT-Thread possuem uma quantidade relativamente próxima, tendo juntos uma média de 741 desenvolvedores.

Verifica-se que os projetos Contiki e Contiki-NG têm uma quantidade próxima de contribuidores. A explicação pode estar no fato de o segundo ser sucessor do primeiro. O Contiki-NG foi lançado em novembro de 2017 com o objetivo de eliminar algumas limitações do Contiki, facilitando a manutenção e acelerando a sua evolução [20]. O mesmo ocorre com os sistemas FreeRTOS e Amazon FreeRTOS quanto à quantidade de colaboradores. Todavia, diferente do caso anterior, o Amazon FreeRTOS utiliza um *kernel* FreeRTOS com bibliotecas para conectividade, segurança e atualizações *over the air* (OTA) [2].

Para melhor ilustração e análise dos dados da TABELA III, foi gerado o gráfico ECDF em um *notebook* com o auxílio das bibliotecas Seaborn e Matplotlib. Ele representa a proporção do número de contribuidores responsáveis e não responsáveis em relação ao número de módulos, variando entre 0 e 1. Ressalta-se que foram criados gráficos ECDF para cada projeto, contendo apenas seus próprios valores, em seus respectivos *notebooks*. Esses gráficos estão disponíveis no GitHub. Entre os sistemas RTOS analisados, observou-se uma variação de 15,83% a 46,56% de colaboradores responsáveis por módulos.

Verifica-se que 46,56% dos colaboradores do Amazon-FreeRTOS são responsáveis por módulos. Em segundo lugar, vem o FreeRTOS, em que 35,06% são responsáveis. Em sequência, Mbed-OS apresenta a porcentagem de 30,50%. Na contramão desses projetos, Zephyr

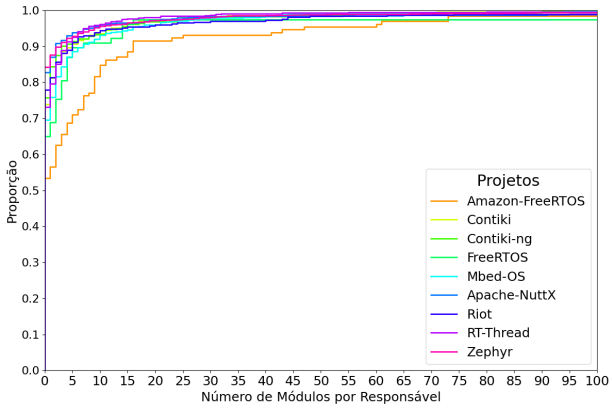

Fig. 2. **Número de módulos por responsável.**

e Apache NuttX possuem os menores percentuais, sendo 15,83% e 17,30%, respectivamente. Contiki e Contiki-NG também possuem valores bastante próximos. Enquanto o primeiro possui um índice de 26,10%, o segundo detém um de 24,19%. Por fim, Riot demonstra um percentual de 22,10%.

É natural que projetos de software livre (FLOSS) tenham mais contribuidores que não sejam responsáveis por algum módulo do que aqueles que são. Aqueles podem dar suporte e reparar partes específicas do sistema, seja por iniciativa própria ou sob indicação de membros ativos. Esse padrão também pode ser explicado por se tratar de uma convenção comum nessas comunidades [8].

Além disso, é de notório conhecimento que comunidades envolvidas com código aberto tendem a depender de alguns poucos desenvolvedores, dos quais são pagos por fornecedores ou empresas de consultoria envolvidos na implementação de determinado sistema [8]. Esse pode ser o caso do Mbed-OS, que é um sistema projetado para microcontroladores ARM Cortex-M pela própria empresa ARM [21]. Outro caso semelhante que pode ser citado é o projeto Zephyr, que é apoiado ativa e financeiramente pela Linux Foundation [22]. Isso pode sugerir que tal situação também ocorra em projetos cujos proprietários são empresas, como o NuttX, que é mantido pela Apache, e o Amazon-FreeRTOS, que, embora tenha como proprietária a AWS, é claramente mantido pela Amazon.

Embora nesta amostra seja averiguado um número maior de colaboradores não responsáveis por nenhum arquivo, quando seus resultados são comparados com outros projetos, em geral, eles apresentam bons valores quanto à propriedade de código. Enquanto na pesquisa conduzida por Santos (2018) o percentual de responsáveis variou entre 0,05% e 14,8%, no presente estudo esses valores variaram entre 15,83% e 46,56%. Sendo assim, isso pode sugerir que os projetos de sistemas RTOS analisados neste estudo podem possuir uma boa propriedade de código, ou seja, a responsabilidade pode ser bem distribuída entre os desenvolvedores.

QP2. Em projetos FLOSS é comum existirem módulos com muitos ou poucos responsáveis, e contribuidores serem responsáveis por muitos ou poucos módulos?

Em média, os projetos variam entre um e dois responsáveis por módulo. Isso indica que, entre desenvolvedores que não são responsáveis, existe pelo menos um ou dois contribuidores que detêm maior conhecimento do código, supervisionam mudanças e lideram a equipe. Essa informação pode ser melhor ilustrada e analisada no gráfico da Figura 3.

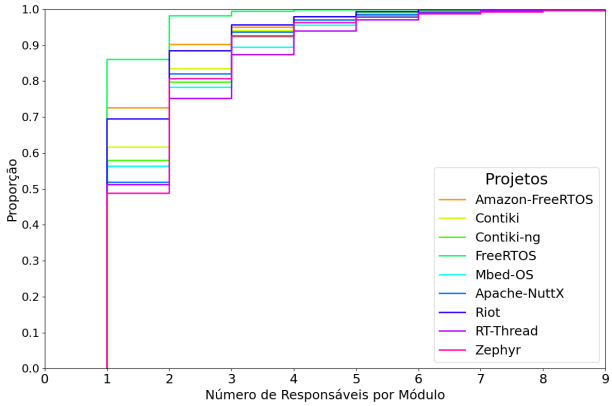

Fig. 3. **Número de responsáveis por módulo.**

Apesar de que possuir muitos responsáveis por módulos possa ser útil em circunstâncias nas quais algum deles precise desistir do projeto, Greiller et al. (2015) afirma que a falta de um proprietário bem definido pode acarretar má manutenção de código, tornando-o mais sujeito a falhas. Ademais, isso também pode levar a uma documentação ausente ou pobre, a um código-fonte ilegível devido à mistura de estilos e inconsistências, e a longos ciclos de correção de *bugs* [15]. Nesse sentido, arquivos de propriedade fraca podem ter, em média, seis vezes mais *bugs* do que aqueles que têm um proprietário forte [13]. Portanto, os sistemas dessa amostra detêm proprietários bem definidos, o que pode indicar que possuem boa manutenção de código, documentação robusta, código-fonte legível e eficiência na correção de *bugs*.

Ao analisar a TABELA III e o gráfico da Figura 2, verifica-se que um desenvolvedor pode ser responsável por vários módulos, uma vez que os valores das médias variaram entre 2 e 27. O projeto FreeRTOS se destacou por possuir, em média, um responsável a cada 27 módulos. Em seguida, destaca-se o Amazon FreeRTOS com um responsável a cada 15 arquivos. Em oposição, os projetos Contiki, Contiki-NG e RT-Thread possuem, em média, dois contribuidores responsáveis por poucos arquivos, variando apenas entre 2 e 3.

Naqueles projetos em que muitos módulos pertencem a um ou poucos responsáveis, pode-se identificar a presença de ”heróis”. Esse fenômeno ocorre quando um pequeno número de desenvolvedores é proprietário de partes críticas e significativas do código que envolve o sistema. É comum a existência deles em projetos FLOSS, podendo trazer tanto vantagens quanto desvantagens [15]. Uma possível desvantagem está associada ao *Truck Factor*, que corresponde a uma métrica cuja função é identificar o quão preparado o projeto está para lidar com a rotatividade de desenvolvedores [9]. Nesses casos, eles podem possuir um *Truck Factor* baixo, ou seja, caso um desenvolvedor definido como ”herói” saia do projeto, este poderá ser impactado negativamente. Em relação à vantagem, pode-se constatar que ”heróis” são mais rápidos para fechar CRs (Requisições de Mudança). Logo, a presença deles pode ser benéfica para esses sistemas, pois reduz o tempo de desenvolvimento [15].

V. Ameaças à Validade

Nesta seção, estão descritas as ameaças à validade deste trabalho. São apresentados tanto motivos que atenuam essas ameaças quanto possíveis ações [23].

Validade Externa. Os resultados obtidos estão restritos a nove projetos FLOSS focados na criação de sistemas RTOS. A princípio, isso pode indicar que tais resultados não se aplicam a projetos fora desse conjunto. Entretanto, esse risco é atenuado pelo fato de que eles estão hospedados no GitHub, o repositório de código mais conhecido atualmente. Em razão de o GitHub ser amplamente utilizado, o fácil acesso a esses projetos pode representar uma amostra mais ampla. Limitar a pesquisa a arquivos em formato C e CPP pode ocasionar uma representação restrita da estrutura e propriedade de código, pois outros arquivos poderiam ser relevantes para a análise. Contudo, isso é atenuado pelo fato de que a maioria dos sistemas RTOS são predominantemente criados nas linguagens C e CPP.

Validade Interna. Os notebooks combinam código, visualizações e texto em um único documento. Segundo Rule (2018), um em cada quatro *notebooks* não possui texto explicativo, contendo apenas visualizações e códigos. Ainda em sua pesquisa, constatou-se que 15 analistas de dados acadêmicos consideram os notebooks como pessoais, exploratórios e desorganizados. Para atenuar esse possível problema, cada etapa e cada código foram explicados de forma organizada, buscando não prejudicar a exploração [24].

Validade de Construção. Todos os dados foram obtidos por meio de mineração e análise do histórico de *commits*. O problema é que muitos *softwares* sofreram mudanças em sua estrutura ao longo dos anos. Com isso, *commits* antigos podem ocultar o trabalho atual no resultado final. Assim, o mais acertado, talvez, seria o algoritmo atribuir pesos diferentes às propriedades relacionadas aos *commits* mais recentes para adequar a propriedade a contribuidores mais recentes, pois desenvolvedores que já saíram do projeto podem ser classificados como responsáveis principais.

Validade de Conclusão. As conclusões do estudo dependem da confiabilidade das métricas utilizadas nesta pesquisa. Todavia, para minimizar essa limitação, foram usadas TF e DOA, amplamente reconhecidas na literatura.

VI. Conclusão

Neste trabalho, foram apresentadas as definições de Matrizes Simples e DOA. A primeira permite mapear de maneira simples e rápida a relação entre módulos e desenvolvedores. Além disso, ela oferece a possibilidade de inserir o elemento humano na estrutura arquitetural de *softwares* [8]. O Método DOA possibilita a identificação da contribuição de cada desenvolvedor em arquivos específicos. Ele considera fatores como a criação, número de *commits* e alterações promovidas por outros colaboradores [9].

Os contribuidores desempenham um papel central na implementação, manutenção e evolução do *software*. Contar com uma visualização clara e objetiva que destaque os principais responsáveis por cada parte do código pode ser valioso para gerenciar e monitorar o progresso e o desenvolvimento de projetos de *software* [8].

Foi desenvolvido um algoritmo, implementado na linguagem Python, capaz de calcular a propriedade de código. Os cálculos são realizados a partir de dados minerados via Pydriller. Nesse contexto, foram extraídos dados de nove repositórios de *software* livres focados no desenvolvimento de sistemas RTOS. Cada um deles possui mais de mil estrelas no GitHub.

Também foram criados notebooks para cada projeto. A partir da matriz gerada pelo algoritmo, eles oferecem informações sobre quais arquivos são associados a cada desenvolvedor, o número de módulos por responsável, o número de responsáveis por módulos e trazem gráficos ECDF e de barras para facilitar as análises. Essas informações foram usadas para responder às questões propostas.

Em relação à QP1, foi possível identificar os desenvolvedores responsáveis por cada módulo. A análise revelou que a maioria dos projetos FLOSS tem um número menor de desenvolvedores que possuem responsabilidade sobre os módulos [8]. Dos sistemas analisados, foi demonstrado que os resultados variaram entre 15,83% e 46,56% quanto ao número de colaboradores considerados responsáveis por algum módulo. Esses valores podem sugerir uma boa distribuição de responsabilidade nesses sistemas quando comparados aos projetos FLOSS analisados por Santos (2018).

Vale ressaltar que, como foram considerados os *masters* de cada projeto, é possível que um colaborador possa ter feito alterações de outros pequenos contribuidores. Entretanto, na esfera de *software* livre, tal contribuidor pode ser tratado como revisor do *master*, aquele que detém autoridade, de modo que possa ser considerado responsável pelas alterações [8].

A QP2 mostrou que, em geral, cada módulo tem um ou dois responsáveis. No entanto, em alguns projetos, havia apenas um pequeno número de desenvolvedores responsáveis por um grande número de módulos. Isso pode indicar o fenômeno do ”herói”, em que certos desenvolvedores concentram grande parte da responsabilidade. Em projetos de *software* livre, tal fenômeno é comum e pode ser essencial para manter o ritmo de desenvolvimento [15]. Mas também representa um risco, pois, caso um desenvolvedor ”herói” saia da equipe, o sistema pode ser descontinuado [9].

Como trabalhos futuros, podem-se incluir mais projetos de sistemas RTOS menos populares e de sistemas FLOSS em geral, com o intuito de realizar uma análise comparativa mais eficiente e fornecer um estudo mais amplo sobre modularidade e propriedade coletiva. Sugere-se investigar o quanto os resultados deste estudo podem ser relevantes para entender desafios de ambientes IoT, como segurança e escalabilidade.

Além disso, para compreender como tais conceitos são percebidos e gerenciados pelos desenvolvedores e realizar uma triangulação dos dados e verificação de hipóteses, fornecendo uma validação cruzada dos resultados obtidos pelo presente estudo, poderá ser enviado um *survey* aos responsáveis identificados, promovendo questionamentos sobre os referidos tópicos.

Agradecimentos

Agradeço primeiramente a minha mãe e família por sempre me apoiarem, ao meu orientador Igor Muzetti pelo apoio, pela paciência, pelo compromisso e pelo conhecimento transmitido nesse período de um ano trabalhando juntos, e por fim aos meus amigos pelo companheirismo nessa caminhada.

Referências

[1] N. Nikolov, O. Nakov, and D. Gotseva, “Operating systems for iot devices,” in *2021 56th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*. IEEE, 2021, pp. 41–44.

[2] N. Nikolov and O. Nakov, “Research of communication between stm32l475 and private cloud realized by using amazon freertos and mqtt,” in *2019 27th National Conference with International Participation (TELECOM)*. IEEE, 2019, pp. 82–85.

[3] C. J. Fidge, "Real-time schedulability tests for preemptive multitasking," *Real-Time Systems*, vol. 14, pp. 61–93, 1998.
[4] G. S. Leh, J. Wu, S. Shukla, M. K. Farrens, and D. Ghosal, "Model-driven joint optimization of power and latency guarantee in data center applications," *SN Computer Science*, vol. 1, pp. 1–14, 2020.
[5] P. Hambarde, R. Varma, and S. Jha, "The survey of real time operating system: Rtos," in *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*. IEEE, 2014, pp. 34–39.
[6] E. v. Hippel and G. v. Krogh, "Open source software and the "private-collective" innovation model: Issues for organization science," *Organization science*, vol. 14, no. 2, pp. 209–223, 2003.
[7] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.
[8] I. M. Pereira, C. S. N. dos Santos, and V. J. P. de Amorim, "Modularidade de código x propriedade coletiva: Um estudo empírico sobre projetos populares de código aberto," in *Anais do XLVII Seminário Integrado de Software e Hardware*. SBC, 2020, pp. 93–103.
[9] G. Avelino, L. Passos, A. Hora, and M. T. Valente, "A novel approach for estimating truck factors," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–10.
[10] M. E. Conway and L. Spandorfer, "A computer system designer's view of large scale integration," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, 1968, pp. 835–845.
[11] C. Y. Baldwin and K. B. Clark, *Design Rules: The Power of Modularity Volume 1*. MIT press, 1999.
[12] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code! examining the effects of ownership on software quality," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 4–14.
[13] M. Greiler, K. Herzig, and J. Czerwonka, "Code ownership and software quality: A replication study," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 2–12.
[14] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 385–394.
[15] F. Ricca and A. Marchetto, "Heroes in floss projects: An explorative study," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 155–159.
[16] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 908–911.
[17] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 3, pp. 309–346, 2002.
[18] J. Coelho, M. T. Valente, L. L. Silva, and A. Hora, "Why we engage in floss: Answers from core developers," in *Proceedings of the 11th international workshop on cooperative and human aspects of software engineering*, 2018, pp. 114–121.
[19] B. M. Randles, I. V. Pasquetto, M. S. Golshan, and C. L. Borgman, "Using the jupyter notebook as a tool for open science: An empirical study," in *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. IEEE, 2017, pp. 1–2.
[20] G. Oikonomou, S. Duquennoy, A. Elsts, J. Eriksson, Y. Tanaka, and N. Tsiftes, "The contiki-ng open source operating system for next generation iot devices," *SoftwareX*, vol. 18, p. 101089, 2022.
[21] C. Bock, M. Marquardt, A. Martens, O. Simanski, and O. Hagendorf, "Smart sensors and actors with bacnet tm and mbed os on cortex-m microcontrollers," in *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*. IEEE, 2019, pp. 937–942.
[22] Y.-k. Lee *et al.*, "Implementation of tls and dtls on zephyr os for iot devices," in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2018, pp. 1292–1294.
[23] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén *et al.*, *Experimentation in software engineering*. Springer, 2012, vol. 236.
[24] A. Rule, A. Tabard, and J. D. Hollan, "Exploration and explanation in computational notebooks," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–12.