Uso de ferramentas de análise estática para identificar vulnerabilidades em sistemas operacionais em C/C++ para dispositivos IoT

Gabriel Thiago Henrique Dos Santos[1], **Luciana Andréia Fondazzi Martimiano**[2]

[1]Departamento de Informática – Universidade Estadual de Maringá (UEM)
87020-900 – Maringá – PR – Brazil

{ra107774,lafmartimiano}@uem.br

Abstract. *This paper describes a static code analysis that was carried out using three static analysis tools, RATS, CppCheck and FlawFinder, in operating systems (OS) for IoT device. Six OS were analyzed: RIOT, Contiki, FreeRTOS, AmazonFreeRTOS, Particle and Zephyr. After the analysis, it was possible to list the possible vulnerabilities and erros in such systems and the total number of errors found out in the selected versions of the OSs, as well as the errors every 1K of line of code.*

Resumo. *Este artigo descreve os resultados da análise estática de código fonte realizada com as ferramentas RATS, CppCheck e FlawFinder em sistemas operacionais para dispositivos de Internet das Coisas (IoT). Seis sistemas operacionais foram analisados, a saber: RIOT, Contiki, FreeRTOS, AmazonFreeRTOS, Particle e Zephyr, resultando em uma lista de possíveis vulnerabilidades e erros em tais sistemas. A partir dos resultados das análises, foi possível identificar o número total de erros encontrados nas diferentes versões dos sistemas operacionais, juntamente com os erros a cada 1K de linha de código.*

1. Introdução

Em ambientes de IoT, diversos dispositivos diferentes estão se comunicando, realizando tarefas e gerando cada dia mais informações. Neste contexto, é importante que os sistemas operacionais usados nestes dispositivos (mesmo os mais customizados e menos robustos) sejam seguros e garantam os três principais pilares de segurança, confidencialidade, integridade e disponibilidade.

Todo dispositivos IoT possui um Sistema Operacional (SO) para seu funcionamento. Independentemente do SO, os pilares de segurança devem ser garantidos levando em consideração as restrições de processamento e armazenamento desses dispositivos. Além disso, o SO deve fornecer mecanismos seguros para autenticar as entidades que podem acessar os dispositivos e/ou o ambiente onde eles estão.

Um estratégia para garantir mais segurança em sistemas operacionais de dispositivos IoT é encontrar as possíveis vulnerabilidades com o uso de ferramentas de análise estática de código-fonte. Essas ferramentas possibilitam encontrar vulnerabilidades a partir de uma base já conhecida, sem a necessidade de execução dos sistemas operacionais, tornando o processo menos oneroso em termos de configuração do ambiente de experimentação.

Neste contexto, este artigo descreve o trabalho realizado para identificar vulnerabilidades em diferentes versões de seis sistemas operacionais utilizados em dispositivos IoT. Seis sistemas operacionais, *RIOT*, *Contiki*, *FreeRTOS*, *AmazonFreeRTOS*, *Particle* e *Zephyr*, foram analisados por três ferramentas de análise estática do código fonte, *CppCheck*, *FlawFinder* e *RATS*.

O artigo está organizado da seguinte forma: a Seção 2 descreve os trabalhos relacionados; a Seção 3 descreve o estudo experimental que foi realizado; a Seção 4 apresenta a discussão dos resultados, bem como as ameaças ao estudo; por fim, a Seção 5 apresenta as conclusões e as sugestões de trabalhos futuros.

2. Trabalhos Relacionados

Os estudos realizados por [Pereira and Vieira 2020] apresentam uma categorização das vulnerabilidades encontradas pelas ferramentas *FlawFinder* e *CppCheck* no projeto de código aberto do *Mozilla*. Como resultados, os autores descrevem que a ferramenta *CppCheck* apresentou melhor desempenho com relação à quantidade de falso-positivos quando comparada à *FlawFinder*. Os autores argumentam ainda que houve um alto índice de falso-positivos, algo que pode limitar a análise estática em projetos de grande porte, com manutenções constantes.

Com o objetivo principal de desenvolver uma ferramenta customizável para análise estática de códigos em C/C++ para detecção de vulnerabilidades, [Arusoaie et al. 2017] conduziram um estudo para comparar o desempenho de ferramentas já existentes. Para o estudo, o *Toyota ITC test suite* [Shiraishi et al. 2015], com 638 casos de testes (tipos e subtipos de defeitos), foi utilizado para avaliar e comparar as ferramentas *Clang*, *The System Analyzer*, *Cppcheck*, *Flawfinder*, *Flint++*, *Frama-C*, *Facebook Infer*, *Oclint*, *Sparse*, *Splint* e *Uno*. Considerando a taxa de detecção de vulnerabilidades, o melhor resultado foi da ferramenta *Oclint* (44,13), seguida das ferramentas *Clang* (35,84) e *FrameC* (27,86).

[Al-Boghdady et al. 2021] realizaram um estudo para encontrar vulnerabilidades em diferentes versões sistemas operacionais para dispositivos de IoT, utilizando as ferramentas *Cppcheck*, *Flawfinder* e *RATS*. Uma das conclusões está relacionada ao número de colaboradores do projeto: o aumento de colaboradores auxilia na redução de vulnerabilidades, enquanto que projetos com menor número de colaboradores tendem a reduzir o número de vulnerabilidades mais devagar. O trabalho descrito neste artigo teve como base o trabalho de [Al-Boghdady et al. 2021].

3. Estudo Experimental

Para que um conjunto de informações possa ser considerado conhecimento científico, a validade e a veracidade dessas informações devem ser comprovadas. Como forma de garantir a validade dos resultados, foi elaborado um estudo experimental de acordo com as sugestões do trabalho de [Wohlin et al. 2012].

A seguir estão descritos o ambiente, os sistemas operacionais e as ferramentas para a análise estática utilizados no estudo.

3.1. Ambiente

Como foram realizadas análises estáticas dos códigos-fontes, não houve necessidade de execução/emulação dos sistemas operacionais em máquinas virtuais. Apenas uma máquina com as seguintes configurações foi utilizada: Linux Debian 11, Processador AMD Ruzen 5-3500U, 8 GBytes de RAM e Placa de vídeo AMD Radeon Rx Vega.

Para o estudo, foram selecionados seis sistemas operacionais, tendo como base para a seleção as classificações dos mais utilizados e em ascensão em G2[1] e *SaaSHub*[2]. A Tabela 1 apresenta algumas informações sobre os sistemas selecionados, incluindo a quantidade de linhas de código de cada versão (SLOC - *Source Lines of Code*). Foram utilizadas quatro versões de cada SO, a partir de 2015, totalizando 24 versões.

Os sistemas operacionais utilizados são desenvolvidos nas linguagens C/C++ (RIOT e Particle) ou C (Contiki, FreeRTOS, AmazonFreeRTOS e Zephyr) a partir de uma arquitetura de *microkernel* (RIOT, FreeRTOS e AmazonFreeRTOS) ou monolítica (Contiki, Particle e Zephyr).

Tabela 1. Informações sobre os Sistemas Operacionais utilizados no estudo

SO IoT	Versão	Ano	SLOC
RIOT	R 2015.09	2015	298.439
	R 2017.10	2017	913.716
	R 2020.04	2020	1.823.632
	R 2021.01	2021	1.515.335
Contiki	R 3.0	2015	266.431
	R v4.0	2017	179.994
	R v4.4	2019	219.694
	R v4.6	2020	227.611
FreeRTOS	v. 8.2.3	2015	2.865.725
	v. 10.0.0	2017	3.156.885
	v. 10.2.0	2019	3.133.693
	v. 10.4.1	2020	4.258.598
AmazonFreeRTOS	v. 1.0.0	2017	402.095
	v. 1.4.0	2018	931.824
	v. 201908.00	2019	2.479.517
	v. 202012.00	2020	2.776.034
Particle	v. 1.0.0	2019	806.628
	v. 1.5.0	2020	726.988
	v. 2.0.0	2020	741.357
	v. 3.0.0	2021	757.006
Zephyr	v. 1.8.0	2017	2.298.857
	v. 1.13.0	2018	3.722.381
	v. 2.1.0	2020	595.958
	v. 2.5.0	2021	749.692

[1]https://www.g2.com/
[2]https://www.saashub.com/

As três ferramentas de análise estática descritas a seguir foram escolhidas para o estudo.

- *Cppcheck* (v. 2.5): Realiza detecção de vulnerabilidades, comportamentos indefinidos e construções de códigos considerados perigosos de programas em C/C++ [Marjamäki 2007]. Possui um diferencial de análise sensível ao fluxo, variando de outras ferramentas que são sensíveis ao caminho, identificando *bugs* únicos.
- *Flawfinder* (v. 2.0.18): Mantém uma base de dados integrada de funções C/C++ com vulnerabilidades ou problemas bem conhecidos, como funções com risco de *buffer overflow*, usos de meta caracteres em *shell* e de aquisições a números aleatórios, realizando comparação entre as funções e o código fonte [Wheeler 2001].
- *RATS (Rough Auditing Tool for Security)* (v. 2.4): Desenvolvida pela *Secure Software Solutions*, e mais tarde adquirida pela *Fortify*. Utiliza uma abordagem similar ao *FlawFinder*, com suporte para análises de programas em linguagens como C, C++, Perl, PHP e Python [Solutions 2001].

3.2. Execução dos Experimentos

Cada uma das três ferramentas foi executada tendo como entrada cada uma das 24 versões dos seis sistemas operacionais, totalizando 72 análises.

Para execução dos experimentos, foram utilizados *scripts* para automatizar o processo para cada versão, pois todas as ferramentas são executadas em linha de comando. Para o uso do *CppCheck*, foi utilizado um parâmetro para habilitar todas as verificações de vulnerabilidades (*–enable=all*), seguido da forma padrão de execução. Já com a *FlawFinder* e a *RATS*, que não têm necessidade de argumentos extras, foi utilizada a execução padrão dos *scanners*. Para cada uma das execuções de cada arquivo de código fonte foram geradas as saídas resultantes correspondentes e agrupadas por versão.

Após todas execuções, os resultados de cada arquivo foram filtrados para conter apenas os erros e as vulnerabilidades encontrados pela ferramenta, para que então fosse possível construir as tabelas com os resultados. Os erros por 1K/SLOC também são contabilizados para cada ferramenta e para cada versão dos sistemas.

4. Resultados e Discussão

Nesta seção são apresentados os resultados[3] dos experimentos realizados com as ferramentas nos seis sistemas operacionais e uma discussão dos resultados.

4.1. Resultados para o sistema AmazonFreeRTOS

A Figura 2 apresenta os resultados para o sistema operacional AmazonFreeRTOS.

Nota-se um aumento no total de erros nas novas versões com as ferramentas *CppCheck* e *FlawFinder*. No entanto, considerando a taxa de erros por 1K/SLOC, houve uma diminuição apontada pela *CppCheck* nas três últimas versões. Já com a *RATS*, houve uma redução a partir da segunda versão, ocasionando assim na queda no erro por 1K/SLOC. Em relação ao trabalho de [Al-Boghdady et al. 2021], a quantidade total de erros em

[3]Os resultados completos podem ser acessados em
https://drive.google.com/drive/folders/1Cwgfe2jzaXzk0dJIzGwUkLxOSHOQiW9q?usp=sharing

cada *release* se manteve similar, tendo variância com relação aos erros por 1K/SLOC de cada versão. Isto ocorre porque, apesar de serem selecionadas versões dos mesmos anos, correções e alterações podem ocorrer no código fonte. Além disso, atualizações nas ferramentas de análise de estática também podem influenciar, pois há um aumento da base de erros e vulnerabilidades.

Tabela 2. Total de erros e erros por 1K/SLOC do AmazonFreeRTOS

Ferramenta	Versão do SO	Total de Erros	Erros por 1K/SLOC
CppCheck	v. 1.0.0 (2017)	3335	8,294
	v. 1.4.0 (2018)	7263	7,794
	v. 201908.00 (2019)	15746	6,350
	v. 202012.00 (2020)	13679	4,927
FlawFinder	v. 1.0.0 (2017)	1746	4,342
	v. 1.4.0 (2018)	4209	4,517
	v. 201908.00 (2019)	16757	6,758
	v. 202012.00 (2020)	30569	11,011
RATS	v. 1.0.0 (2017)	315	0,783
	v. 1.4.0 (2018)	977	1,048
	v. 201908.00 (2019)	771	0,311
	v. 202012.00 (2020)	279	0,101

4.2. Resultados para o sistema FreeRTOS

Na Figura 3 são apresentados os resultados para o sistema operacional FreeRTOS.

Em relação ao trabalho de [Al-Boghdady et al. 2021], as taxas de total de erros e de erros a cada 1K/SLOC se mantiveram com a *CppCheck*. Já com a *FlawFinder*, ambos os experimentos tiveram pouca variação de aumento ou redução. Com a *RATS*, houve uma redução na quantidade total de erros, e como consequência dos erros por 1K/SLOC. Esta redução se deveu, principalmente, ao fato de que as versões mais recentes do SO tiveram maior preocupação com o desenvolvimento de um sistema mais seguro.

Tabela 3. Total de erros e erros por 1K/SLOC do FreeRTOS

Ferramenta	Versão do SO	Total de Erros	Erros por 1K/SLOC
CppCheck	v. 8.2.3 (2015)	14087	4,916
	v. 10.0.0 (2017)	15610	4,945
	v. 10.2.0 (2019)	15730	6,350
	v. 10.4.1 (2020)	19854	5,020
FlawFinder	v. 8.2.3 (2015)	14922	5,207
	v. 10.0.0 (2017)	14703	4,657
	v. 10.2.0 (2019)	14607	4,661
	v. 10.4.1 (2020)	13777	3,235
RATS	v. 8.2.3 (2015)	307	0,107
	v. 10.0.0 (2017)	246	0,078
	v. 10.2.0 (2019)	212	0,068
	v. 10.4.1 (2020)	151	0,035

4.3. Resultados para o sistema RIOT

A Figura 4 apresenta os resultados para o sistema operacional RIOT.

Para o sistema RIOT, houve diminuição na quantidade de erros por 1K/SLOC em todas as ferramentas para todas versões, com exceção da *FlawFinder* na versão 2021.01, apesar de ter havido aumento na quantidade total de erros em algumas versões. Este fato é positivo, pois o número de linhas de código aumentou entre a primeira e a terceira versão em seis vezes, aproximadamente (saltando de 298.439 mil para 1.823.632 milhão de linhas). Apesar da quantidade total de erros se manter alta, a boa orquestração à cada 1K/SLOC mostra o foco constante de melhora no processo de desenvolvimento por parte dos responsáveis. Em relação ao trabalho de [Al-Boghdady et al. 2021], ainda que duas das versões selecionadas em cada ano tenha tido alterações, as taxa de erros se mantiveram equivalentes.

Tabela 4. Total de erros e erros por 1K/SLOC do RIOT

Ferramenta	Versão do SO	Total de Erros	Erros por 1K/SLOC
CppCheck	R 2015.09	1202	4,028
	R 2017.10	1761	1,927
	R 2020.04	2214	1,214
	R 2021.01	2667	1,760
FlawFinder	R 2015.09	957	3,207
	R 2017.10	1897	2,076
	R 2020.04	2908	1,595
	R 2021.01	3300	2,178
RATS	R 2015.09	395	1,324
	R 2017.10	397	0,434
	R 2020.04	193	0,106
	R 2021.01	179	0,118

4.4. Resultados para o sistema Contiki

A Figura 5 apresenta os resultados para o sistema operacional Contiki.

Nota-se que, a partir da versão 4.0 de 2017, a quantidade total de erros e a quantidade de erros por 1k/SLOC diminuíram de forma significativa, como apontam as três ferramentas. Este fato pode ser explicado pelo fato de que a quantidade de linhas de código diminuiu à medida que novas versões do SO foram desenvolvidos. Em relação ao trabalho de [Al-Boghdady et al. 2021], as taxas se mantiveram altas em comparação com os outros sistemas, porém, variações abaixo no teste atual ocorreram devido à seleção das versões mais atualizadas, de 2015 a 2020.

4.5. Resultados para o sistema Particle

A Figura 6 apresenta os resultados para o sistema operacional Particle, sendo o sistema mais recente na época da execução dos experimentos (com lançamento em 2019).

Todas as ferramentas apontaram uma grande quantidade total de erros, no entanto, houve um decréscimo nas versões mais recentes, mesmo com a diminuição na quantidade de linhas de código destas versões, ainda positivamente houve diminuição na quantidade de erros por 1K/SLOC.

Tabela 5. Total de erros e erros por 1K/SLOC do Contiki

Ferramenta	Versão do SO	Total de Erros	Erros por 1K/ SLOC
CppCheck	R 3.0 (2015)	2510	9,421
	R v4.0 (2017)	1443	8,017
	R v4.4 (2019)	1762	8,020
	R v4.6 (2020)	1809	7,948
FlawFinder	R 3.0 (2015)	3025	11,354
	R v4.0 (2017)	1327	7,372
	R v4.4 (2019)	1454	6,618
	R v4.6 (2020)	1510	6,634
RATS	R 3.0 (2015)	719	2,699
	R v4.0 (2017)	265	1,472
	R v4.4 (2019)	252	1,147
	R v4.6 (2020)	257	1,129

Tabela 6. Total de erros e erros por 1K/SLOC do Particle

Ferramenta	Versão do SO	Total de Erros	Erros por 1K/ SLOC
CppCheck	v. 1.0.0 (2019)	3070	3,806
	v. 1.5.0 (2020)	2296	3,158
	v. 2.0.0 (2020)	2223	2,999
	v. 3.0.0 (2021)	2275	3,005
FlawFinder	v. 1.0.0 (2019)	3669	4,549
	v. 1.5.0 (2020)	2617	3,600
	v. 2.0.0 (2020)	2730	3,682
	v. 3.0.0 (2021)	2846	3,760
RATS	v. 1.0.0 (2019)	622	0,771
	v. 1.5.0 (2020)	259	0,356
	v. 2.0.0 (2020)	258	0,348
	v. 3.0.0 (2021)	265	0,350

4.6. Resultados para o sistema Zephyr

Na Figura 7 são apresentados os resultados para o sistema operacional Zephyr.

A ferramenta *RATS* apontou um redução expressiva na quantidade total de erros, de 670 para 11, e na quantidade de erros por 1K/SLOC, de 0,291 para 0,015. Por outro lado, para as ferramentas *CppCheck* e *FlawFinder*, houve uma variação entre aumento e diminuição na quantidade total de erros, ocasionando na maior taxa de erros por 1K/SLOC a cada versão do sistema. Vale ressaltar, que houve um aumento expressivo na quantidade de linhas de código da versão 1.8.0 (2017) para a versão 1.13.0 (2018) do SO, de 2.298.857 milhões para 3.722.381 (aumento aproximado de 63%).

4.7. Discussão

Cada ferramenta utiliza suas técnicas para identificar possíveis vulnerabilidades, conforme descrito na Seção 3.1. Independentemente destas diferenças, é possível, a partir dos resultados de cada ferramenta, correlacionar o erro com uma CWE (*Common Weakness Enumeration*).

Tabela 7. Total de erros e erros por 1K/SLOC do Zephyr

Ferramenta	Versão do SO	Total de Erros	Erros por 1K/SLOC
CppCheck	v. 1.8.0 (2017)	9052	3,938
	v. 1.13.0 (2018)	14615	3,926
	v. 2.1.0 (2020)	4141	6,948
	v. 2.5.0 (2021)	5451	7,271
FlawFinder	v. 1.8.0 (2017)	2800	1,218
	v. 1.13.0 (2018)	3727	1,001
	v. 2.1.0 (2020)	2822	4,735
	v. 2.5.0 (2021)	3525	4,702
RATS	v. 1.8.0 (2017)	670	0,291
	v. 1.13.0 (2018)	79	0,021
	v. 2.1.0 (2020)	69	0,116
	v. 2.5.0 (2021)	11	0,015

O CWE [4] é, conforme descrito no site oficial, uma lista de vulnerabilidades de software e hardware construída pela comunidade de desenvolvedores. O projeto define uma linguagem comum e padrão para descrições de vulnerabilidades, sendo um arcabouço para identificação, mitigação e prevenção de vulnerabilidades.

As Tabelas 8, 9 e 10 mostram, respectivamente para as ferramentas *RATS*, *FlawFinder* e *CppCheck*, com que frequência a CWE é encontrada em cada código fonte dos sistemas operacionais. Devido à limitação de espaço, cada tabela apresenta as quatro CWEs mais frequentes.

As CWEs mais frequentes encontradas pela *RATS* foram as CWE-119, CWE-120 e CWE-20; enquanto que as (CWE-119!/CWE-120)[5], CWE-120 e CWE-126 foram mais frequentes na *FlawFinder*, e as CWE-561, CWE-398 e CWE-1126 na *CppCheck*.

Comparando com o artigo base deste trabalho [Al-Boghdady et al. 2021], tem-se que para a *FlawFinder* as CWEs se mantiveram juntamente com a ordem de maior frequência entre os sistemas operacionais. Já com a *RATS*, houve a troca da terceira mais frequente, colocando a CWE-134 em quarto, com uma variação de taxa de frequência de 3% entre elas. Já com a *CppCheck*, houve a variância da terceira posição, passando a CWE-563 para a quarta posição com uma diferença de menos de 1% apenas. Porém, mesmo com algumas versões e inserção de outros sistemas operacionais nos experimentos, as vulnerabilidades continuam frequentes no desenvolvimento dos sistemas operacionais.

Outro ponto a se destacar é que apenas nos sistemas FreeRTOS e RIOT houve aumento na quantidade de vulnerabilidades. Nos demais sistemas, houve um declínio ou uma manutenção na quantidade de vulnerabilidades. Este fato mostra, de acordo com os resultados apresentados, que mesmo com o aumento no número de linhas de código, a atenção e a preocupação dos desenvolvedores com relação à segurança aumentou, havendo, por exemplo, em alguns projetos, a inclusão de sistemas automatizados de testes

[4] https://cwe.mitre.org/

[5] Esta notação significa que a CWE-120 é um sub-conjunto da CWE-119 [Al-Boghdady et al. 2021]

no controle de versões (*Git*).

4.8. Ameaças ao projeto experimental

As análises realizadas com as três ferramentas apontam um alto índice de erros e vulnerabilidades nas diferentes versões. No entanto, é importante apontar que as ferramentas limitam-se às taxas de acertos por analisarem os códigos de forma estática, apresentando falsos-positivos, que só são possíveis de confirmar averiguando alerta por alerta, ou a partir de uma grande base de dados para consulta.

Tabela 8. Erros identificados pela RATS correlacionados às CWEs

CWE	AmazonFree RTOS	Contiki	Free RTOS	Particle	RIOT	Zephyr	Totais	Frequência
CWE-119	1641	833	339	846	566	543	4768	57,163%
CWE-120	204	319	205	218	161	67	1174	14,075%
CWE-20	158	93	211	28	334	113	937	11,234%
CWE-134	49	59	117	194	8	18	445	5,335%

Tabela 9. Erros identificados pela FlawFinder correlacionados às CWEs

CWE	AmazonFree RTOS	Contiki	Free RTOS	Particle	RIOT	Zephyr	Totais	Frequência
(CWE-119!/ CWE-120)	37515	2140	45170	4081	3413	4037	96356	63,224%
CWE-120	9011	2734	6232	3019	2411	5128	28535	18,723%
CWE-126	3026	1063	2833	1158	839	1869	10788	7,079%
CWE-134	969	748	2145	656	268	524	5310	3,484%

Tabela 10. Erros identificados pela CppCheck correlacionados às CWEs

CWE	AmazonFree RTOS	Contiki	Free RTOS	Particle	RIOT	Zephyr	Totais	Frequência
CWE-561	24019	2151	34158	5969	4174	14245	84716	51,721%
CWE-398	5687	2447	16303	2217	2683	10908	40245	24,570%
CWE-1126	5821	1373	8055	558	263	4649	20719	12,649%
CWE-563	1254	170	2087	168	106	1448	5233	3,195%

5. Conclusão e Trabalhos Futuros

Independentemente dos objetivos de um sistema operacional ou do dispositivo no qual ele irá rodar e gerenciar os recursos, é importante que o desenvolvimento desses sistemas seja pautado por projetos guiados por princípios de segurança e privacidade (*Security and Privacy by Design*) [Cavoukian and Stoainov 2007]. Especificamente com relação aos princípios de privacidade, destaca-se o princípio ”Proativo e não reativo, preventivo e não remedial”, que prega o monitoramento, a predição e a inclusão de elementos que aumentem a segurança dos dados de um usuário para que se reduzam substancialmente os riscos. A evolução tecnológica, ainda que colocada como um desafio, não deve apresentar impactos negativos. No entanto, é importante ressaltar que as limitações de hardware e software de dispositivos IoT devem ser considerados.

Com os sistemas operacionais para dispositivos IoT estudados pode-se perceber uma melhora nos cuidados com o projeto, havendo redução na quantidade de erros e vulnerabilidades, porém os números ainda podem ser considerados altos pelos resultados apresentados, necessitando mais atenção para as atualizações dos sistemas.

Como trabalhos futuros, sugere-se analisar os casos de falso-positivos, havendo diversos erros apontados, porém, a análise além da estática fora limitada para diferenciar um erro real. Como complemento, outros sistemas operacionais em outras linguagens de programação podem ser analisados, evoluindo, assim, para um estudo mais robusto com relação à segurança dos sistemas operacionais para dispositivos IoT.

Referências

Al-Boghdady, A., Wassif, K., and El-Ramly, M. (2021). The presence, trends, and causes of security vulnerabilities in operating systems of iot's low-end devices. *Sensors*, pages 1–21.

Arusoaie, A., Ciobâca, S., Craciun, V., Gavrilut, D., and Lucanu, D. (2017). A comparison of open-source static analysis tools for vulnerability detection in c/c++ code. In *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 161–168.

Cavoukian, A. and Stoainov, A. (2007). *Biometric Encryption: A Positive-Sum Technology that Achieves Strong Authentication, Security and Privacy*. Information and Privacy Commissioner of Ontario. Acessado em Janeiro de 2022.

Marjamäki, D. (2007). A tool for static C/C++ code analysis. `https://cppcheck.sourceforge.io/`. Acessado em Janeiro de 2022.

Pereira, J. D. and Vieira, M. (2020). On the use of open-source c/c++ static analysis tools in large projects. *European Dependable Computing Conference*, 16.

Shiraishi, S., Mohan, V., and Marimuthu, H. (2015). Test suites for benchmarks of static analysis tools. In *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 12–15.

Solutions, S. S. (2001). Rats. `https://security.web.cern.ch/recommendations/en/codetools/rats.shtml`. Acessado em Janeiro de 2022.

Wheeler, D. (2001). Flawfinder. `https://dwheeler.com/flawfinder/`. Acessado em Janeiro de 2022.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.