Exploring the Potential of Frama-C in IoT Static Analysis

Minh Tran
minhltran1102@gmail.com
Cal State University San Marcos
San Marco, California, USA

William King
william.king@my.century.edu
St. Cloud State University
St. Cloud, Minnesota, USA

Harvey Siy
hsiy@unomaha.edu
University of Nebraska at Omaha
Omaha, Nebraska, USA

ABSTRACT

In this research, we investigated the feasibility of using static analysis for IoT applications with Frama-C. We looked at different kinds of possible IoT vulnerabilities and how static analysis specifically could be used to identify them. With certain Frama-C plugins such as Eva, we were able to run static analysis on most IoT code without modifying the code itself and catch errors that could potentially be exploited in real-world applications that would have otherwise been missed. Additionally, we created a simple IoT device, by utilizing Raspberry Pi 4 hardware with a set of different SunFounder sensors, and ran our created code for it through Frama-C to find any errors. The static analysis done gave a significant amount of potential vulnerabilities in our code, mostly consisting of integer overflows. We learned how we could use static analysis tools, like Frama-C, as a powerful way to find potential vulnerabilities with minimal changes to code.

CCS CONCEPTS

• **Software and its engineering** → *Formal software verification*; • **Computer systems organization** → Embedded software; • **Security and privacy** → *Software security engineering*.

KEYWORDS

IoT, Frama-C

ACM Reference Format:
Minh Tran, William King, and Harvey Siy. 2023. Exploring the Potential of Frama-C in IoT Static Analysis. In *The Twenty-fourth International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing (MobiHoc '23), October 23–26, 2023, Washington, DC, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3565287.3617617

1 INTRODUCTION

This work was conducted in order to answer how we can use static analysis to detect IoT application security issues. The ever growing system of connected devices and services, known as the Internet of Things (IoT), is being used in sensitive and security critical domains all over the world. This makes it a prime target for cyber attacks from bad actors, and IoT devices will continue to be the center of these attacks as their importance in our everyday lives grows. Ferrara et al. states how "IoT attacks use a common IoT device to intrude into the system, and exploit a larger network. The nature of IoT ecosystems is larger than traditional network security, with a wider attack surface from multiple interconnected devices operating at many different physical locations and network layers." [4] There is a lot of interest to formally analyze IoT code as it is being developed in order to prevent software vulnerabilities from being introduced as zero-day vulnerabilities. While many static analysis tools provide superficial syntactic analysis, we examined an experimental tool that provides a deeper semantic analysis called Frama-C.

C remains a popular language used for IoT applications due to its efficiency. This also leads to many software vulnerabilities as C code provides very few run-time checks. A notorious example is the Heartbleed bug where an attacker was able to read someone else's data from a web server's buffer. [6] While the C language may be easier to exploit with its little run-time checks, this can be mostly prevented with thorough formal static and dynamic analysis of the software. In the OWASP Top 10 Security Vulnerabilities in IoT software [4], 6 out of the 10 vulnerabilities can be covered at least partially by static analysis.

Frama-C is a framework through which a collection of different plugins for static and dynamic analysis can be run and collaborate with each other. It is aimed at verifying programs written in C code using this collective analysis. While these plugins operate with different goals and parameters, they also work together by building and sharing information to get a better picture of the software's vulnerabilities. In order to run and test Frama-C for this study we used Raspberry Pi 4s running on a Raspbian operating system. We chose this hardware for the extended purpose of acting as a sample IoT device, since we could connect different sensors to the Raspberry Pi's GPIO board.

2 BACKGROUND AND RELATED WORK

Frama-C is a C program analysis tool that provides many plugins, such as Eva, which has the capability of abstract interpretation, a static analysis technique that extracts information about potential executions of a program, computing an approximation of all possible values a variable can take [1]. In Eva, this information is then used to determine if run-time errors like division by zero, or integer or buffer overflows could occur. These errors could lead to potential exploits.

Blanchard et al. [2] presented a set of example C language code, some taken from actual IoT software, some not. They then showed how some main Frama-C plugins can be used to perform static analysis, starting with EVA (Evolved Value Analysis). They describe EVA as a value analysis tool that uses abstract reasoning to approximate potential runtime errors or prove their absence, such as invalid pointers, arithmetic overflows or division by zero. Listing

MobiHoc '23, October 23–26, 2023, Washington, DC, USA

ACM ISBN 978-1-4503-9926-5/23/10…$15.00
https://doi.org/10.1145/3565287.3617617

Listing 1: Code with division by zero.

```
int f ( int a )
{
    int x, y;
    int sum, result;
    if(a == 0) {
        x = 0;
        y = 0;
    } else {
        x = 5;
        y = 5;
    }
    sum = x + y;
    result = 10/sum; //sum can be 0
    return result;   //risk of division by 0
}
```

Listing 2: Frama-C mistakenly flags division by zero.

```
int f ( int a )
{
    int x, y;
    int sum, result;
    if(a == 0) {
        x = 0;
        y = 5;
    } else {
        x = 5;
        y = 0;
    }
    sum = x + y;
    result = 10/sum; //sum cannot be 0
    return result;   //no div. by 0
}
```

1 shows an example where EVA detects an alarm of division by zero. In order to run this we type the following command into the terminal: `frama-c -eva [name].c -main f`. The `-main f` option is to indicate to EVA that this specific function should be pointed to the function `f` in this case.

The caveat to this abstract reasoning is that it can also over-approximate and give false alarms where a potential error is reported when no error could take place, as seen in Listing 2. While in this code division by zero is not possible, EVA will still report a risk of division by zero because it computes a general domain of x and y before computing the domain of sum. There are ways to increase the precision of the EVA analysis by giving additional options at the cost of a slower analysis, though we did not explore these options in this study.

The next plugin covered was WP which is used to verify more complex properties, using ACSL annotations to give precise specifications or requirements to the intended behavior of the analyzed function. WP then takes the properties given in those ACSL annotations and proves them using deductive verification. In Listing 3

Listing 3: WP illustration. Annotations specify expected swap behavior.

```
/*@
    requires \valid(a) && \valid(b);
    requires \separated(a,b);
    assigns *a, *b;
    ensures *a == \old(*b) && *b == \old(*a);
*/
void swap(int *a, int *b){
    int tmp = *a ; *a = *b ; *b = tmp ;
}

int main(){
    int x = 2;
    int y = 4;
    swap(&x, &y);
    //@ assert x == 4 && y == 2 ;
}
```

the intended behavior is that of a simple swap function. The main function would like to take the values of x and y and switch them. There are a few different ACSL annotations needed for WP.

The first is composed of a precondition and a postcondition. The precondition is introduced by the **requires** keyword and is the expected state of the system before calling the function. The post-condition is the **ensures** keyword and is an expression of how the state of the program is modified and the properties of the results. Several of the same types of these expressions can also be written right after one another if more than one is needed. The **assigns** keyword specifies a particular class of postconditions which define what memory locations can be modified by the function. The ensures clause indicates the expected execution of the function values. The final keyword **assert** creates an assertion of a property that must be checked at that specific part of the program. While this assertion can prove that the program respects the specification we want, it doesn't mean that the program won't still fail at runtime. These specifications must be carefully written in order to be proven correctly. For example, if the assigns clause is not given, the function is assumed by WP to assign any memory location.

Lastly the plugin E-ACSL is used as runtime verification. This plugin was designed to extend Frama-C to use some forms of dynamic analysis on top of its existing static analysis. E-ACSL uses runtime assertion checking to detect errors and their locations, especially those that don't create a failure of the code during execution. It generates C code to check the assertions at runtime by executing that code, as such the assertions are checked dynamically.

In Listing 4 we took the same problem looked at from EVA (see Listing 1) and added an additional assertion that the sum cannot equal zero. E-ACSL then takes this annotated code and converts it into executable code. We run it through the command line with `frama-c -e-acsl [name].c -then-last -print -ocode monitored_main.c`. This creates a new C file with the assertion inserted into the original main file. We then run `e-acsl-gcc.sh`

Listing 4: Code for E-ACSL run-time execution.

```
int f(int a)
{
  int x, y, sum, result;
  if (a == 0) {
    x = 0;
    y = 0;
  }
  else {
    x = 5;
    y = 5;
  }
  sum = x + y;
  //@ assert Eva: division_by_zero: sum != 0;
  result = 10 / sum;
  return result;
}

int main(void){
    f(42);
    f(0);
    return 0;
}
```

Listing 5: E-ACSL output from executing Listing 4.

```
Assertion failed at line 10 in function f.
The failing predicate is:
sum !=0
Aborted (core dumped)
```

`[name].c -c -O monitored_main` to invoke the C compiler. Finally, we could run `sudo ./a.out` to print out the result, which shows the assertion fails (see Listing 5).

When the code from Listing 2 is used to replace function `f`, it shows no output as both calls do not result in a division by zero.

3 EXPERIMENTS WITH FRAMA-C

In order to better understand how Frama-C works, we experimented with multiple code examples from different Frama-C tutorial papers [2, 3].

Listing 6 is an example of using the Frama-c WP plugin [2]. The purpose of the code was to assign the loop to be decreased by any number instead of one. The command we used to run this was: `frama-c -wp [name].c -rte`. This is where the WP plugin comes in to read the special comment and ensure the loop is decreased by one at each iteration of any number beside one. Due to missing code from Blanchard et al. [2], we were missing 2 goals from the timeout. That is because the loop cannot be reset and quit.

Listing 7 is an example of using the Frama-C WP plugin on a real example of Contiki source code [2]. Contiki is an open-source operating system, and the purpose of the code was to practice defining annotations for both separate functions and loops. The command line used to run it was `frama-c -wp [name].c -rte`,

Listing 6: WP annotations for specification of loop invariants and variants.

```
#include <stdio.h>
#include <stdlib.h>

/*@
    requires 0 <= len;
    requires \valid(a + (0 .. len-1));
    assigns a[0 .. len-1];
    ensures \forall integer i;
            0 <= i < len ==> a[i] == 0;
*/
void reset_array(int* a, int len){
    /*@
        loop invariant 0 <= i <= len ;
        loop invariant \forall integer j;
                    0 <= j < i ==> a[j] == 0;
        loop assigns i, a[0 .. len-1];
        loop variant len - i ;
    */
    for(int i = 0 ; i < len ; ++i){
        a[i] = 0 ;
    }

    int i = 42;
    /*@
        loop invariant 0 <= i <= 42;
        loop assigns i;
        loop variant i;
    */
    while(i > 0){
        i = i - ((rand()%i)+1) ;
    }
}
```

which should cause the WP plugin to read the special comment and set up conditions for the functions and loops. After running, it should not return a NULL value. However, due to the complexity of the code and a timeout, at least 5 goals were missed. The problem may have come from the annotation of the loops, as the loop invariant needed to involve the loop index i. This resulted in an infinite loop, which required changing the annotations and gaining a better understanding of the loop's condition.

We ran into multiple issues working with the Frama-C E-ACSL plugin. Unlike the Eva and WP plugins, which are static analysis tools, the E-ACSL plugin is more of a dynamic analysis tool. This means the code, along with Frama-C annotations, must be compiled into an executable file using a standard C compiler. However, we had additional difficulties getting the dynamic analysis to work on our Raspberry Pi architecture. To run the plugin, we had to change the Frama-C E-ACSL files because the Raspberry Pi CPU architecture does not recognize the m32 or m64 options, which are for Intel processors, while the Raspberry Pi uses an ARM processor. To find the files, we first had to go to Frama-C's installation folder, like `/home/pi/.opam/4.14.1/bin`, then find the file

Listing 7: Example illustrating loop invariants on real code.

```
struct memb {
    unsigned short size;
    unsigned short num;
    char *count;
    void *mem;
};

/*@
    requires \valid(m)
    ensures \exists integer i;
        0 <= i < (*m).count ==>
        \old((*m).count[i] == 0) ==>
        \return != NULL
*/

void *
memb_alloc(struct memb *m)
{
    int i;
    /*@
        loop invariant loc >= 0
        loop invariant 0 <= i <= m->num
        loop invariant
            (\exists integer j;
                    0 <= j < m->num ==>
                    m->count[j] == 0) ==>
            (\forall integer k;
                    0 <= k < j ==>
                    m->count[k] == 1)
        loop variant m->num - i
    */
    for(i = 0; i < (*m).num; ++i)  {
        if(m->count[i] == 0)  {
            m->count[i] = 1 ;
            int loc = i * m->size ;
            return (void *)
                   ((char *)m->mem + loc);
        }
    }

    return NULL;
}
```

named `e-acsl-gcc.sh` and look for the line of code `GCCMACHDEP`. We changed it to `GCCMACHDEP=""`. After that, we are able to run E-ACSL successfully.

We also ran additional examples from Blanchard et al. [2, 3] and found no additional issues with Eva, WP, or E-ACSL.

4 METHODOLOGY

For our testing of Frama-C we choose two separate things to run on for analysis.

I. Frama-C on existing IoT code. We chose the Sunfounder IoT sensor kit project because it included a variety of pre-built sensor modules and multiple example programs to test with Frama-C. This gave us an easy sample base to run on that also showed possible real-world applications these programs might be used for in the form of different sensors.

II. Frama-C on our own IoT code. We designed a simple IoT device using a few of the different SunFounder sensors together to demonstrate Eva static analysis (see Figure 1). We chose to use a combination of the Ultrasonic sensor, RGB LED light, and Passive Buzzer. First by wiring the Ultrasonic sensor the exact same as in the Sunfounder examples, then wiring the RGB LED to the next open GPIO pins 27, 22, and 23. Finally wiring the Buzzer to GPIO 24. The device we designed is a proximity detector that sets off an alarm when detecting something within 40 centimeters by changing a LED color from green to red and generating a noise frequency on the Buzzer. We created the code using the existing IoT code as a starting base and developing it further to fit the needs we desired for our device. The code is available from our GitHub repository [5].

5 EVALUATION

5.1 Frama-C on existing IoT code

To better test Frama-C with more real-world like examples, we chose to run it against a set of simple programs created for the SunFounder sensor kit [7]. While starting on this, we noticed that some of the test sensors would run with the C code examples, while others only worked using the Python examples. Noticing this we narrowed down the issue to C's wiringPi library, as only the output sensors wouldn't work, and all the input sensors worked properly. After updating this library we were subsequently able to get every sensor working properly with the C language for testing with Frama-C.

When starting to run Frama-C on these example programs, we also learned that Frama-C's preprocessing will include the headers of its own standard library, instead of those installed in the user's machine. This avoids issues with non-portable, compiler-specific features. Which means if a header is included which is not available in Frama-C's library, such as our wiringPi library, preprocessing will fail by default. We could get around this problem by copying the entire wiringPi library into the same directories of the programs we were testing, and also had to use this method for a few other necessary libraries when needed. Listing 8 shows a working EVA summary output for one of the SunFounder sensor programs. We were able to analyze all the functions of this program to a 97% coverage with zero changes to the code. In doing so we found 8 possible integer overflows.

When we ran Frama-C Eva on the Sunfounder IoT sensor kit codebase, we discovered a variety of warnings (see Figure 2). For example, we received many Eva analyzer warnings, which indicate that there are errors that could potentially cause problems. Also, we noticed the Eva analyzer warnings were the highest counts. Another example is that we received integer overflow warnings, which means that the code is trying to store a value inside a holder with a maximum capacity.

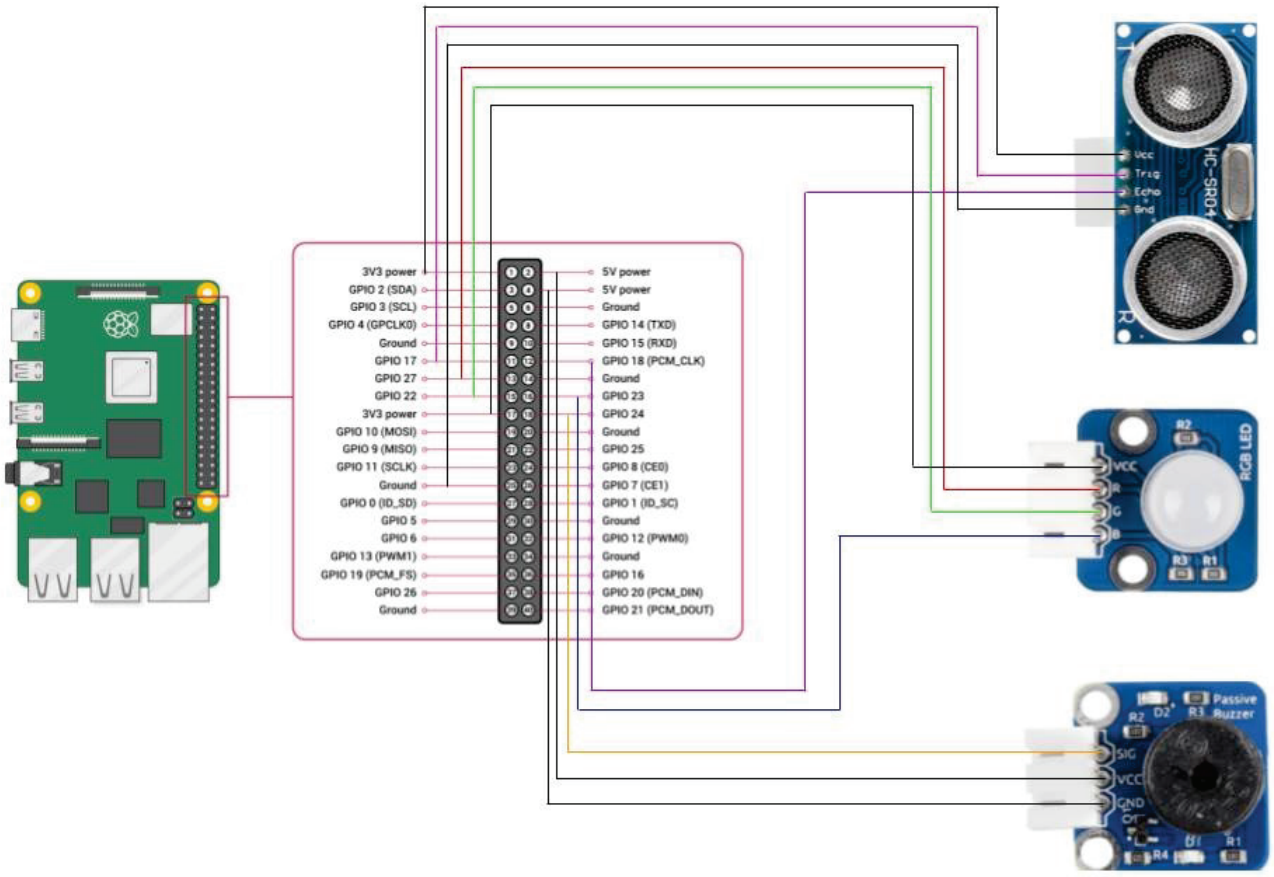

Figure 1: Schematic of our IoT application.

Listing 8: Sample Frama-C EVA output.

```
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  3 functions analyzed (out of 3): 100% coverage.
  In these functions, 35 statements reached (out of 36): 97% coverage.
  ------------------------------------------------------------------------
  Some errors and warnings have been raised during the analysis:
    by the Eva analyzer:      0 errors    0 warnings
    by the Frama-C kernel:    0 errors    6 warnings
  ------------------------------------------------------------------------
  8 alarms generated by the analysis:
       8 integer overflows
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions       0 valid    0 unknown    0 invalid     0 total
    Preconditions    2 valid    0 unknown    0 invalid     2 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

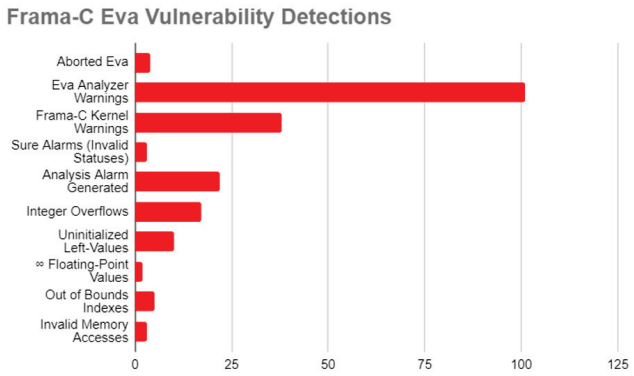

Figure 2: Eva warnings and alarms from Sunfounder IoT codebase.

Listing 9: Frama-C EVA output for our IoT application.

```
[eva:summary] ====== ANALYSIS SUMMARY ======
  ------------------------------------------------------------------------
  5 functions analyzed (out of 5): 100% coverage.
  In these functions, 56 statements reached (out of 57): 98% coverage.
  ------------------------------------------------------------------------
  Some errors and warnings have been raised during the analysis:
    by the Eva analyzer:      0 errors    0 warnings
    by the Frama-C kernel:    0 errors    6 warnings
  ------------------------------------------------------------------------
  18 alarms generated by the analysis:
      16 integer overflows
       2 illegal conversions from floating-point to integer
  ------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions       0 valid    0 unknown    0 invalid     0 total
    Preconditions    3 valid    0 unknown    0 invalid     3 total
  100% of the logical properties reached have been proven.
  ------------------------------------------------------------------------
```

5.2 Frama-C on our own IoT code

We also ran Frama-C Eva on our own IoT code. At first, the result was either integer overflows or illegal conversions from floating-point to integer warnings. Then, we added an extra line to convert distance to integer before our statement, and added Evas recommended assertions. We managed to remove some of the float-to-int warnings, and have mostly only integer overflow warnings. Our Eva's static analysis summary is shown in Listing 9. So far, using Eva allowed us to identify all the possibility of issues and exploits by giving us warnings on the code.

6 CONCLUSION AND FUTURE WORK

Frama-C has mostly been used in lab environments, but we have shown that it may be feasible to run it on real IoT device software tools for static analysis. We were able to run Frama-C with minimal changes to existing code, and identified multiple different potential vulnerabilities that may otherwise have been missed.

We uncovered a few different issues, such as integer overflow, that are often overlooked by programmers. By using Frama-C to find these issues without needing to execute a lot of tests, we can prevent software vulnerabilities before they are introduced as zero-day exploits.

In future work, we would like to explore other Frama-C plugins such as WP and E-ACSL. For WP, it requires more annotations to the code, and intimate knowledge of where and how to write them inside the program. Meanwhile, the E-ACSL is more of a dynamic analysis tool, which makes the steps to use it more complicated and requires more research.

To conclude, with the expanding use of IoT devices around the world, and with it, the increase of security concerns, there have been more attacks than ever on IoT applications. However, many people and the community have come together to work on increasing security protection for IoT applications. For example, there is Frama-C as a statistical analysis tool to identify potential exploits. Which was the result of an open-source collaborative effort. We have explored the potential of using Frama-C on possible real-world IoT devices for an extra level of vulnerability mitigation and found it to be a promising way to analyze IoT software.

ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under grant award # 2308741 and was conducted at the REU Site at University of Nebraska at Omaha. We would also like to extend a special thanks to Dr. Alfredo Perez and our mentor Dr. Harvey Siy, for going above and beyond in helping us with this research.

REFERENCES

[1] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, et al. 2021. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. *Commun. ACM* 64, 8 (2021), 56–68.

[2] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. 2018. A lesson on verification of IoT software with Frama-C. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 21–30.

[3] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. 2019. Tutorial: Formal Verification for an Internet of Secured Things. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*.

[4] Pietro Ferrara, Amit Kr Mandal, Agostino Cortesi, and Fausto Spoto. 2021. Static analysis for discovering IoT vulnerabilities. *International Journal on Software Tools for Technology Transfer* 23 (2021), 71–88.

[5] William King and Minh Tran. 2023. REU Project: Frama-C for Analyzing IoT Applications. https://github.com/WillMKing/UNOREU2023.git. Accessed [7/24/2023].

[6] Fahmida Y. Rashid. 2014. Why The Heartbleed Vulnerability Matters and What To Do About It. https://www.securityweek.com/why-heartbleed-vulnerability-matters-and-what-do-about-it/. Accessed [7/14/2023].

[7] SunFounder. 2023. Sensor Kit V2 for Raspberry Pi. https://docs.sunfounder.com/projects/sensorkit-v2-pi/. Accessed [7/13/2023].