COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

HOW TO LIFT AND VERIFY CONTROL DIFFERENTIAL EQUATIONS FROM MACHINE CODE OF CYBER-PHYSICAL SYSTEMS

Autor
Javier Jarauta Gastelu

Director
Kirill Levchenko

Madrid

May 2022

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título
How to Lift and Verify Differential Equations from Machine Code of Cyber-Physical Systems
en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2021/22 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos.
El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

[illegible]

Fdo.: Javier Jarauta Gastelu Fecha: 2022/06/05

Autorizada la entrega del proyecto
EL DIRECTOR DEL PROYECTO

[illegible]

Fdo.: Kirill Levchenko Fecha: 2022/06/05

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

HOW TO LIFT AND VERIFY CONTROL DIFFERENTIAL EQUATIONS FROM MACHINE CODE OF CYBER-PHYSICAL SYSTEMS

Autor
Javier Jarauta Gastelu

Director
Kirill Levchenko

Madrid

May 2022

Acknowledgements y Agradecimientos

En un principio, quiero agradecer a mi familia, especialmente a mis padres y a mi hermano, por haberme dado la oportunidad de llegar a este punto. Sin ellos, ni el proyecto, ni el intercambio, ni mi educación habrían sido posibles. Ellos me han apoyado a lo largo de este camino de altibajos y les estaré siempre agradecidos por haberme traido hasta aquí. Al fin y al cabo, la espinita de la ciberseguridad no ha salido de la nada.

Second, I would like to express my deepest gratitude to Kirill Levchenko, Maxwell Bland, and Tzu-Bin Yan. Their support over the past year in the laboratory and everything they have taught me made this research possible. I would like to mention Max especially, as he made this project happen. The patience to explain core concepts and the push to get it done, made this project a success, much better than I could even envision.

Third, I would like to thank the support from ICAI, Rafael Palacios, Gregorio López, Álvaro López and all of the teachers, who have provided feedback and helped throughout the research and my engineering degree. A special mention to the team at the University of Illinois to make my exchange program happen.

Por último, quería agradecer a mis amigos, desde el colegio hasta la universidad, pasando por el conservatorio, por todo el apoyo y la ayuda que me han dado.

CÓMO OBTENER Y VERIFICAR LAS ECUACIONES DIFERENCIALES DE CONTROL DESDE EL CÓDIGO MÁQUINA DE SISTEMAS CIBER-FÍSICOS

Autor: Jarauta Gastelu, Javier
Director: Levchenko, Kirill
Entidad Colaboradora: University of Illinois at Urbana-Champaign - Security and Privacy Research at Illinois (S&PR@I)

RESUMEN DEL PROYECTO

En este proyecto, se ha desarrollado una herramienta, que hemos denominado "InteGreat", que obtiene las ecuaciones de control de dispositivos embebidos. Mediante técnicas de ingeniería inversa y ejecución simbólica, se realiza la lectura de la memoria del dispositivo, y se ejecuta una inferencia de las ecuaciones de control de dicho dispositivo hasta un lenguaje de programación de alto nivel como Python. De esta manera, se representan de manera simple los cálculos y ecuaciones que gobiernan los bucles de control del dispositivo ciber-físico en cuestión, por lo que se podrán identificar vulnerabilidades y errores de programación con el objetivo de mejorar la ciberseguridad y calidad funcional del mismo.

Palabras clave: sistemas embebidos, sistemas ciber-físicos, ciberseguridad, ecuaciones de control, ejecución simbólica, decompilación, desensamblado, angr, Ghidra, radare2.

1 Introducción

Durante años, se han realizado numerosos intentos con mayor o menor éxito, para la obtención de las ecuaciones de control que gobiernan a los sistemas complejos. Desde los primeros pasos, cuando Poincaré trabajó en 1899 en el movimiento de cuerpos celestes, ha habido un gran progreso en las diferentes técnicas y metodologías. [1] El estudio de las interacciones complejas de sistemas se denomina la teoría de sistemas dinámicos, y tuvo su gran expansión a partir del año 1970. [2]

A su vez, los dispositivos embebidos y ciber-físicos, es decir, aquellos que incorporan tanto componentes de *Hardware* como de *Software* han tenido un crecimiento exponencial desde la aparición de los primeros en el año 1968. Se estima que en la actualidad existen más de 10.000 millones de dispositivos, promovidos por un abaratamiento de la tecnología y la reducción de tamaño de los componentes electrónicos, siendo una de sus principales realidades, los dispositivos IoT, así como los sistemas de ayuda a la movilidad y navegación en vehículos terrestres y aéreos. [3]

Sin embargo, a pesar de que el crecimiento de ambas tecnologías se ha potenciado enormemente en años recientes, no existen técnicas concretas ni una metodología formalizada que permitan realizar la verificación de la seguridad y correcta implementación de dichos dispositivos. La herramienta InteGreat objeto del presente proyecto pretende conseguir una mayor facilidad para dicha verificación y análisis, e implantar una metodología formal del proceso de análisis. [4]

2 Definición del proyecto

En este proyecto se busca diseñar y desarrollar una herramienta, que permita un análisis sencillo a alto nivel de sistemas ciber-físicos que posean al menos, un bucle de control. El objeto del proyecto incluye el desarrollo de la herramienta semi-automática, y el planteamiento inicial para su transformación en una herramienta automática, que será objeto de posteriores estudios de investigación. Para ello, se ha realizado la investigación y el desarrollo en dos ejes principales. En primer lugar, se ha trabajado en la obtención de las ecuaciones de control mediante técnicas de ingeniería inversa (decompilación y desensamblado) y ejecución simbólica. Y el segundo eje de investigación ha consistido en la correcta verificación y utilidad de dichas ecuaciones. De esta manera, se presenta un paquete completo, multi-arquitectura y multi-plataforma que realiza dicho análisis y presenta al usuario con un paquete que ejecuta el análisis de dispositivos embebidos.

La herramienta ha sido verificada y probada con tres dispositivos distintos, aunque uno de ellos no ha sido posible completarlo. En un principio, el primer dispositivo de análisis ha sido un quadricóptero, a través del cual se ha realizado una prueba de concepto y se ha verificado la obtención de las ecuaciones de control. En segundo lugar, se ha realizado el análisis de un PLC de control. Este dispositivo ha sido verificado y las ecuaciones de control han sido obtenidas en Python al igual que el quadricóptero. La ventaja de su análisis es que ha permitido la simulación de un ciberataque ya realizado, con resultados satisfactorios. Y finalmente, el último dispositivo, del cual se ha hecho un análisis parcial es un radar de automoción, el cual presenta una arquitectura completamente distinta a los anteriores. Sin embargo, no ha sido posible la obtención de las ecuaciones de control.

3 Descripción de la herramienta

La herramienta InteGreat presenta dos flujos principales de ejecución y una sección de análisis, dependiendo de la complejidad del sistema a evaluar y las necesidades específicas de arquitectura. De esta manera, se pueden identificar las siguientes estructuras en el flujo de datos y análisis.

(a) Flujo de ejecución manual

En el flujo manual es necesario realizar una previa investigación mediante un decompilador como Ghidra o radare2 para poder obtener tanto el bucle de control como el punto de entrada en la aplicación (aunque esto último es fácilmente reconocido por cualquiera de las dos herramientas). Así, el análisis sigue el esquema presentado en la Figura 1.

Como se observa en dicho flujo, este presenta las siguientes herramientas por orden: ghidra, pseudo-`C`, Jupyter Notebook, angr y Python. Así, este se inicia con un fichero binario, es decir la lectura de la memoria directamente desde el dispositivo ciber-físico. Prosigue con el desensamblado y decompilación

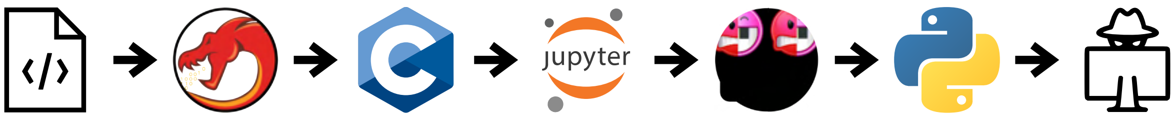


Figura 1: Flujo del programa manual

mediante la herramienta desarrollada por la NSA (*National Security Agency*) denominada Ghidra que realiza una inferencia de código pseudo-`C`. Este se utiliza para obtener información básica del sistema binario como dirección de memoria de entrada y otros parámetros como nombres de funciones. Posteriormente se utilizó Jupyter Notebook y Python para obtener tanto, registros de entrada y salida de las funciones como sus parámetros y el tamaño esperado de la variable (8/16/32 bits). A continuación se realiza la ejecución simbólica mediante angr, y finalmente se crea un fichero en Python que permite la ejecución y el análisis de las ecuaciones de control.

(b) Flujo de ejecución automático

En el caso del flujo automático, este es similar al manual, pero se simplifica al utilizar herramientas de decompilación automáticas. El orden de herramientas es: radare2/r2ghidra, angr y Python. Estas herramientas trasladan la parte manual que se realiza en Ghidra a otros decompiladores en línea de comandos, en este caso radare. Aún así, cabe destacar que cierto análisis manual siempre es necesario. Este flujo simplificado se observa en la Figura 2.

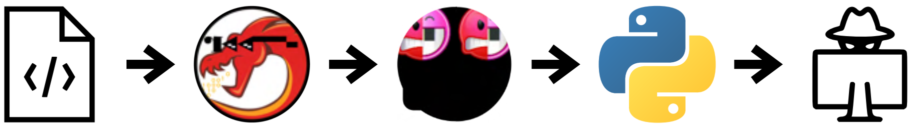


Figura 2: Flujo del programa automático

Como se observa en este flujo secundario, el análisis es mucho más sencillo de cara al usuario, y presenta un funcionamiento bastante similar, aunque, existen diferencias. En vez del uso de Ghidra, se utiliza el decompilador y desensamblador de este a través de radare2 que permite la ejecución en línea de comandos. Posteriormente, y utilizando la integración de angr y claripy se ejecuta simbólicamente el archivo binario y finalmente se guardan las ecuaciones obtenidas en un fichero Python.

(c) Análisis de accesibilidad

Para la demostración de la utilidad de la herramienta, se ha realizado un análisis de accesibilidad utilizando para ello DaDRA. Para este análisis de accesibilidad, se utilizan las ecuaciones obtenidas del quadricóptero, y se realizan una serie de gráficos para demostrar la técnica. Aún así, el análisis de accesibilidad es reducido debido a las limitaciones de tiempo y de la herramienta.

4 Resultados

Los resultados de la aplicación de la herramienta InteGreat, han sido altamente satisfactorios, habiendo obtenido en los dos principales casos analizados, es decir el quadricóptero y el PLC, resultados muy positivos en cuanto a las ecuaciones obtenidas. Estas han permitido un análisis a fondo de la ciberseguridad y calidad del sistema de control de dichos dispositivos. La metodología así como la herramienta InteGreat desarrollada, abren un camino nuevo de análisis y verificación de todo tipo de dispositivos ciber-físicos cuya masiva implantación, van a transformar nuestra sociedad en los próximos años, y es crítico que se realice tras un profundo análisis de riesgos y vulnerabilidades de los mismos, que es el objetivo final de InteGreat.

5 Conclusiones

Con los resultados del proyecto, se concluye que la inferencia de las ecuaciones de control desde la memoria del dispositivo ciber-físico hasta un lenguaje de alto nivel como Python, es posible y viable. Todo ello, se ha podido implementar mediante la herramienta InteGreat, habiendo evidenciado la simulación de un ciberataque en un dispositivo PLC y la correcta implementación de los filtros en el quadricóptero.

Hay una serie de cuestiones que se tienen que mejorar, como la automatización o la obtención más fácilmente de parámetros de ejecución. Aún así, la herramienta cumple con su función y permite analizar correctamente un dispositivo ciber-físico.

6 Referencias

[1] Chutinan, A., & Krogh, B. (2003). Computational techniques for hybrid system verification. *IEEE Transactions on Automatic Control*, 48(1), 64–75. https://doi.org/10.1109/tac.2002.806655

[2] Strogatz, S. H. (2001). *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry, and Engineering (Studies in Nonlinearity)* (1.a ed.). Westview Pr (Short Disc).

[3] Moore, S. (2021, 13 abril). *Cyber-Physical Systems Must be Part of Your Security Strategy*. Gartner. Recuperado 5 de junio de 2022, de https://www.gartner.com/smarterwithgartner/develop-a-security-strategy-for-cyber-physical-systems

[4] INCIBE-CERT. (2021, 13 abril). *Introducción a los sistemas embebidos*. INCIBE-CERT - Blog. Recuperado 5 de junio de 2022, de https://www.incibe-cert.es/blog/introduccion-los-sistemas-embebidos

HOW TO LIFT AND VERIFY CONTROL DIFFERENTIAL EQUATIONS FROM THE MACHINE CODE OF CYBER-PHYSICAL SYSTEMS

Author: Jarauta Gastelu, Javier
Supervisor: Levchenko, Kirill
Collaborating Entity: University of Illinois at Urbana-Champaign - Security and Privacy Research at Illinois (SPR@I)

ABSTRACT

In this project, a tool called "InteGreat" was developed to obtain the control equations of embedded systems. Through the use of reverse engineering and symbolic execution, the memory of the device is read and the inference of the control equations of that device up to a high-level programming language like Python is performed. By means of this tool, the equations and calculations that govern the control loops of the cyber-physical device are presented in an easy way. With this tool, it will be easy to identify vulnerabilities and programming errors with the objective of improving device cybersecurity and functional quality.

Keywords: embedded systems, cyber-physical systems, cybersecurity, control equations, symbolic execution, decompilation, disassembly, angr, Ghidra, radare2.

1 Introduction

Over the years, numerous attempts have been made with more or less success to obtain the control equations that govern complex systems. From the first steps, when Poincaré worked on 1899 on the movement of celestial bodies, there has been great progress in different techniques and methodologies [1]. The study of complex interactions within systems is called dynamical systems theory and had its great expansion from 1970 [2].

Meanwhile, embedded and cyber-physical systems, that is, those that combine both hardware and software, have had an exponential development since the first systems appeared in 1968. Some of the latest estimates believe that there are more than 10,000 million devices. Growth is facilitated by a reduction in the cost of technology and the reduction in size. Some examples of these devices are IoT systems, traffic management systems, and navigation in maritime and land transportation [3].

However, even though both technologies have expanded greatly in recent years, there are no concrete techniques or formal methodologies that allow verification and security analysis of the implementation of those devices. The tool being developed, InteGreat, aims to achieve an easier methodology for this verification and analysis and implement a formal methodology of the analysis process [4].

2 Project definition

In this project, the objective is to design and develop a tool that could be easily analyzed on high-level cyber-physical systems that have a control loop. The objective of the project includes the development of the semi-automatic tool and the initial approach to develop it into a fully automatic tool, which will be part of future research. To achieve this goal, research and development focused on two main objectives. First, we lift the control equations using reverse engineering techniques (decompilation and disassembly) and symbolic execution. Second, correct verification and use of the lifted equations. With both steps, a complete multi-architecture, multi-platform package is presented, which provides the user with packages that perform embedded system analysis.

The tool has been verified and executed with three different devices; however, the analysis of the latest was not fully completed. The device that was first analyzed was a drone through which an initial proof-of-concept and equation lifting were performed. Second, a control PLC device was analyzed. This device was verified, and the control equations were lifted to Python, just like the drone. The advantage of its analysis is that a cyberattack was simulated with adequate results. And finally, a partial analysis was done on an automotive radar system. However, the equation lifting was not completed with the radar.

3 Description of the tool

The tool Integreat has two main execution workflows and an analysis section, depending on the complexity of the system to evaluate and the specific requirements of the architecture. The following structures are then possible to identify in the data analysis.

(a) Manual workflow

In the manual analysis, it is necessary to perform some previous investigation using ghidra or radare, to obtain both the control loop and the entry point (this is easily recognizable by the tools). The analysis then follows the flow presented in Figure 1.

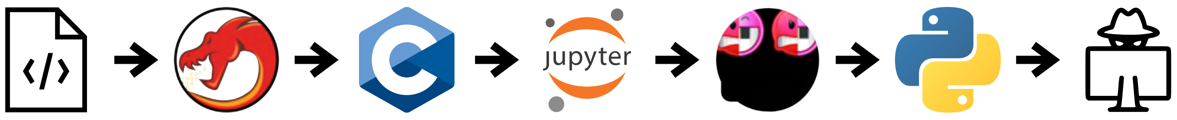


Figure 1: Manual program workflow

As can be seen, this workflow presents the following tools in order: ghidra, pseudo-C, Jupyter Notebook, angr, and Python. The workflow starts with a binary file, that is, the cyber-physical device's memory. It then continues with the disassembly and decompilation through Ghidra, developed by the NSA

(National Security Agency). Ghidra performs an inference of the binary file into pseudo-`C` code. This is used to obtain basic information from the system, such as entry address, and others like function parameters and signatures. Later, input and output registers, function parameters, and register size (8/16/32 bits) are obtained using Jupyter Notebook and Python. After that, symbolic execution is performed on the file with angr, and finally the program outputs a Python file that allows for the execution and analysis of the control equations.

(b) Automatic workflow

In the case of the automatic workflow, it is similar to that of the manual workflow, but it is easier due to the use of automatic decompilation tools. The order of the tools is: radare2/r2ghidra, angr, and Python. These tools partially transform the manual analysis done in Ghidra to other command-line decompilers. However, it is worth noting that there is always some manual component that is necessary. This simplified workflow is presented in Figure 2

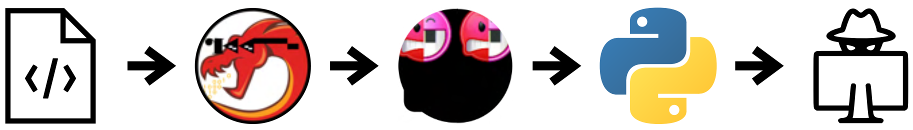


Figure 2: Automatic program workflow

As observed, the analysis is much simpler in terms of user usability; however, the workflow is similar to that of the manual steps. Instead of using Ghidra, the workflow uses its decompiler and disassembler through radare2, which allows command-line execution. Later, using the integration between angr and claripy, the program executes the binary file and saves the control equations in a Python file.

(c) Reachability analysis

To demonstrate the usefulness of the tool, a reachability analysis is performed using DaDRA. This reachability analysis uses the equations obtained from the Drone, and a series of graphs are built to demonstrate the technique. However, the scope of the reachability analysis is limited due to time and tool constraints.

4 Results

The results of the InteGreat tool are satisfactory, having obtained really positive results in the two main devices analyzed in terms of the lifted equations. These have allowed deep cybersecurity and system quality analysis. The methodology and InteGreat tool open a new path for the analysis and verification of a large number of cyber-physical systems whose massive expansion will transform the society in the next few years. Then it is critical to perform a deep vulnerability and risk analysis of those, which is the end goal of InteGreat.

5 Conclusions

Once the project is done, it is possible to conclude that inference of the control equations from the cyber-physical device's memory to a high-level programming language like Python is possible and viable. All of this was possible to combine by means of InteGreat, having evidenced the simulation of a cyberattack on a PLC device and the correct implementation of the filtering functions in a drone.

There are several technical problems that must be improved upon, such as automation or the easier acquisition of execution parameters. However, the tool performs its duty correctly and allows for the analysis of a cyber-physical system.

6 References

[1] Chutinan, A., & Krogh, B. (2003). Computational techniques for hybrid system verification. *IEEE Transactions on Automatic Control*, 48(1), 64–75. https://doi.org/10.1109/tac.2002.806655

[2] Strogatz, S. H. (2000). *Nonlinear Dynamics And Chaos: With Applications To Physics, Biology, Chemistry, And Engineering (Studies in Nonlinearity)* (1st ed.). Westview Press.

[3] Moore, S. (2021, April 13). *Cyber-Physical Systems Must be Part of Your Security Strategy*. Gartner. Retrieved June 5, 2022, from https://www.gartner.com/smarterwithgartner/develop-a-security-strategy-for-cyber-physical-systems

[4] INCIBE-CERT. (2021, April 13). *Introduction to Embedded Systems*. INCIBE-CERT - Blog. Retrieved June 5, 2022, from https://www.incibe-cert.es/en/blog/introduction-embedded-systems

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In recent years, there has been exponential growth in the number of connected devices on the Internet. According to some of the latest estimates, the number of connected appliances in 2022 is more than 26 billion [1]. However, of all these devices, more than half are devices for the Internet of Things or IoT, and most, if not all, are embedded systems. Small devices that usually serve a single purpose with low resource usage. The number of integrated systems is over 10,000 million and continues to grow at a rate of more than 15% year-on-year [2].

In this area, embedded controllers are increasingly being used to create new remotely controlled cyber-physical systems, as Gartner explains [3]. Examples of these new devices include connected home appliances, city sensorization (traffic management), smart vehicles, and avionics, to name a few. Robustness and failure tolerance are expected from these devices, without sacrificing performance or ease of use. Furthermore, the new required security standards, such as ISO/IEC 62443 and ISO/IEC 27001 [4] in industrial control systems, where embedded systems are widely used, depend on these systems to meet minimum certification requisites and security practices.

A large number of these devices may seem innocent at first, like smart connected lights, thermostats, refrigerators, or even some non-life-supporting medical devices. All of these are expected to perform adequately and without problems. After all, some are simple and should not need more than a few thousand lines of code. But when they start to fail due to vulnerabilities, attacks, or other unexpected problems, the true scale of the security problem arises. What is a house without lights? What if the refrigerator, the single appliance that keeps humans safe by maintaining edible food, no longer works? The scale of the problem is increasing exponentially as more devices are being connected, or even non-connected devices depend on complex kernels to function and may not be fault tolerant.

With this evolution of cyber-physical devices, there is a growing need for tools that can successfully and rapidly analyze these devices for correctness and vulnerabilities. As some studies show, the cost of verification and certification in the aviation industry and other safety-critical industries is more than 50%. And testing until money runs out is neither efficient nor scalable in any industrial process [5]. This is where InteGreat provides insight into the development and implementation of embedded systems and produces a versatile multi-architecture tool for software and modeling analysis. InteGreat intends to create a framework that supplies an easy-to-use tool for the verification of embedded/cyber-physical systems.

With an aim to lift the control equations of embedded and cyber-physical systems to a high-level language like Python by means of disassembly, decompilation and symbolic execution, InteGreat is set to improve on the integration between low-level bit analysis, and high-level verification languages and schemes.

This thesis project was completed at the University of Illinois Urbana-Champaign with the help of Kirill Levchenko and especially Maxwell Bland. Furthermore, this thesis is only part of the larger InteGreat tool that is being developed for a future paper submission to a conference, in conjunction with other members of the laboratory.

This first chapter presents the basic core concepts of this thesis. The development of embedded and cyber-physical systems, their history, and desirable characteristics. It also includes the theory that supports system verification, dynamical systems. And finally, we have techniques for system representation and abstraction such as continuous systems, state machines, hybrid automata models, and domain-specific languages.

1.1 Embedded Systems

An embedded system is defined as a tightly integrated device that contains hardware and software components that perform a single function. Although the concept of an embedded system is as old as electronics itself and could be applied to some of the first electromechanic devices, the modern embedded system dates back to the early 1960s. The Apollo Guidance Computer (AGC) is considered by many to be the first true modern conception of an embedded system. Developed by the MIT Instrumentation Laboratory, it used keys and actuators as a human interface and was the cornerstone of the whole guidance system in the Apollo missions.

In subsequent years, during the next decade, the appearance of integrated microchips truly revolutionized the concept of an embedded system. These devices, invented almost simultaneously by Gary Boone of Texas Instruments, Federico Faggin of Intel, and the U.S. Navy, provided tightly integrated packages with a 4-bit design CPU (Central Processing Unit), ROM (Read-Only Memory), and RAM (Random Access Memory) in a single package. The new integrated microchips opened new possibilities in terms of architecture, integration, and efficiency. Some of the later developments include Texas Instrument's TMS1000, the Intel 8080 and the Motorola 6800 [6].

1.1.1 Desirable Characteristics

The desirable characteristics and features of an embedded system are the following:

- **Reliability**: The ability of a process or device to execute its function without failure as long as it is requested. In this case, an embedded system is expected to perform reliably without unnecessary attention.
- **Efficiency**: The device should be efficient in both its computation cycle and in its overall power consumption. The use of different architectures, such as RISC (Reduced Instruction Set Computer) or CISC (Complex Set Instruction), clock speeds, interconnecting interfaces and devices, and others may directly impact the device's performance. Furthermore, code efficiency is vital in terms of reducing clocks-per-instruction, especially on real-time embedded systems, which need rapid information performance.
- **Stability**: It refers to the characteristic of a system that is constrained within the bounds of the expected performance. This means that as long as no external unexpected event occurs, the device should operate under the expected constraints [7].
- **Maintainability**: Is the ability of a system to be repaired and restored to service when maintenance is performed using specified skill levels and prescribed procedures and resources. In terms of an embedded system, the expectation is that the device will work correctly immediately and without delay after maintenance [8].
- **Low cost**: In any context, and especially in these devices, the cost of components and operation should be low. Embedded devices must be low-power, inexpensive, and easily replaceable.

1.1.2 Components

It is possible to abstract an embedded system as the following structure (Figure 1.1):

- **System inputs**: They comprise most of the time of sensor data or actuator information. They could also include signals from other devices and/or machines. Examples may be pressure / temperature information, motor status, and gyroscope information.
- **System outputs**: The system's actions or calculated values to be used in the program or by other devices. They could be instructions to an actuator such as a boiler, or packaged information through a Bus communication interface.
- **Software components**: The code that processes the inputs and obtains an output.

- **Hardware components**: Hardware that allows for the execution of the software. It may include status interfaces, such as LEDs or debug/information ports.

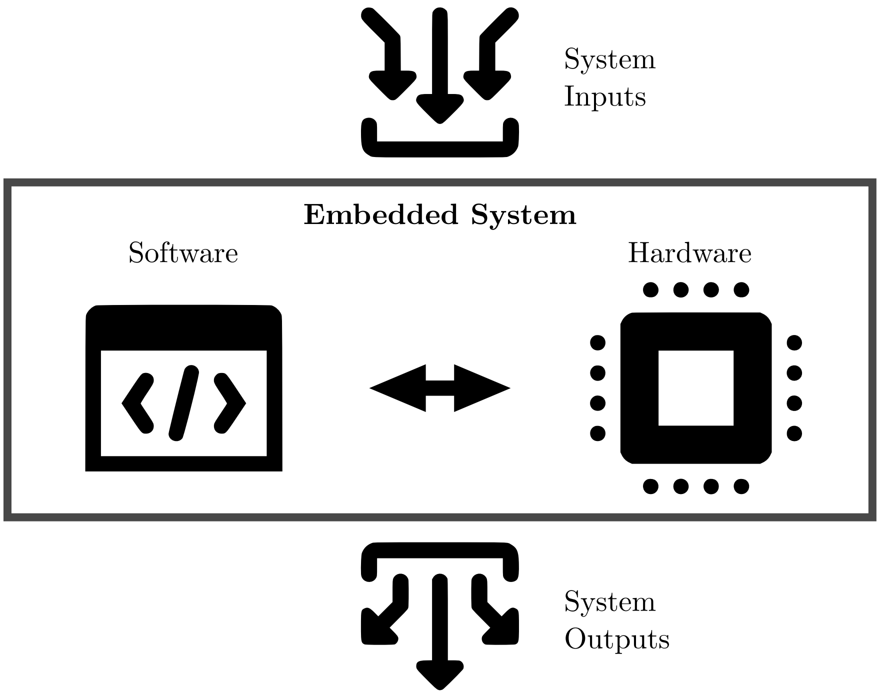


Figure 1.1: General overview of an embedded system

Logical Level

Conceptually, an embedded system has the following data / information flow graph (Figure 1.2):

- **Internal data flow**: There is an internal data flow across the microprocessor with all other internal and bridging components such as Analog-to-Digital (ADC) / Digital-to-Analog (DAC), User Interfaces, Diagnostic Ports, Software, Memory, and in some cases FPGA (Field-Programmable Gate Arrays)/ASIC (Application Specific Integrated Circuits) cards.
- **External data flow**: The flow of information between the embedded system and the environment is done through actuators, DAC/ADC, User Interfaces, and Sensors and Actuators.

Hardware

In terms of hardware components, an embedded system is usually composed of a microprocessor or microcontroller (CPU), volatile (RAM) and non-volatile memory, inputs and

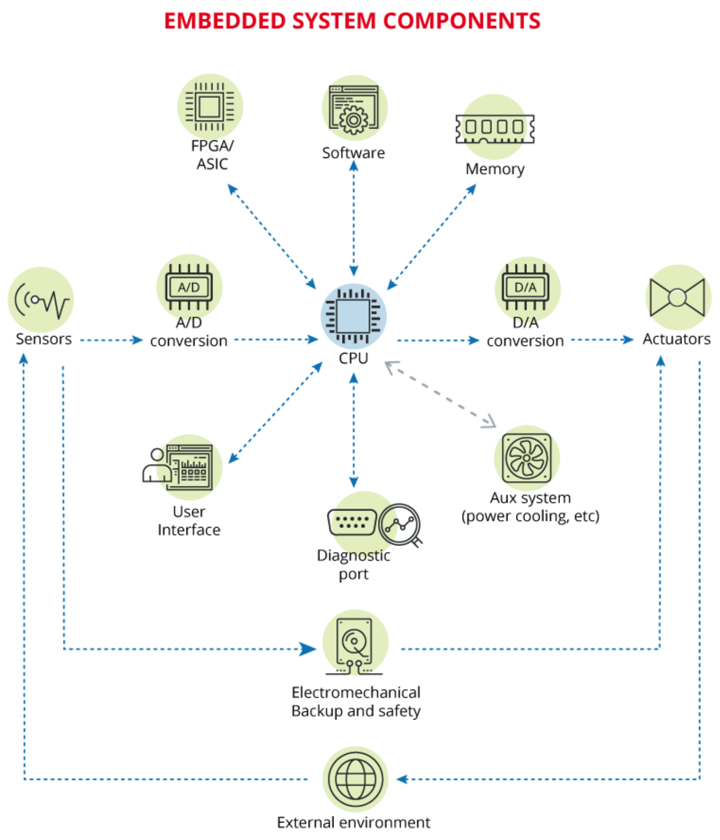


Figure 1.2: Description of an embedded system (logical level)

Source: Courtesy of INCIBE [2]

outputs (I/O) ports and interfaces, power, and in some cases it may contain a human interface such as a screen powered by a graphical processing unit (GPU). Furthermore, an embedded system may include other devices integrated in the package, such as Analog-to-Digital (ADC) / Digital-to-Analog (DAC) converters, actuators, diagnostic tools and ports, interconnecting interfaces such as keyboards, system support devices (interrupt managers), etc. as can be observed in Figure 1.3.

Software

The software is usually organized around an operating system (OS) on which applications are built. These applications are usually considered as firmware and, in most cases, are not modifiable by the user. They may sometimes be upgraded through firmware/software updates. The software, according to Jiménez et al., consists of the following components [6]:

- **System tasks**: The application is divided into different tasks that perform the device's function. These use system resources and are executed simultaneously or

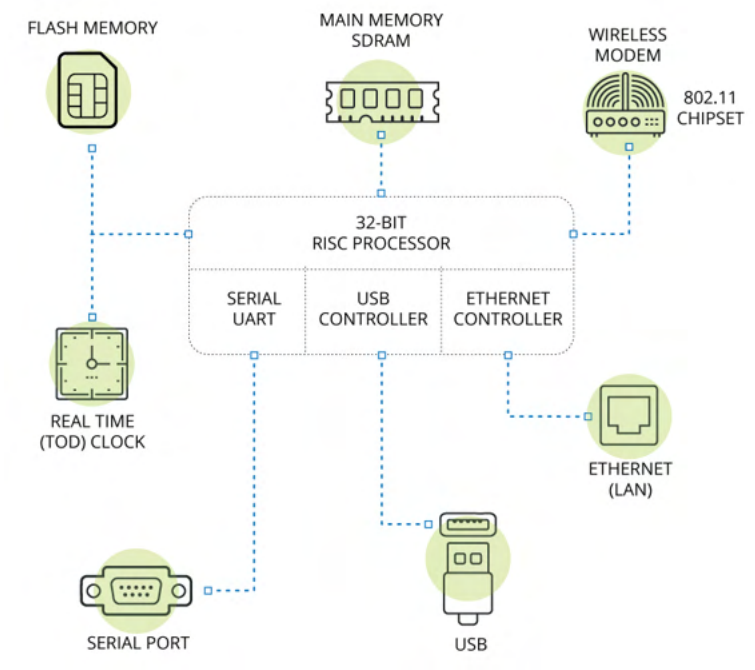


Figure 1.3: Description of an embedded system (physical level)

Source: Courtesy of INCIBE [2]

in parallel, depending on the architecture and priority. These tasks send service requests to the kernel.

- **System kernel**: This component handles the service request and manages the device's hardware. It also controls communication between components and tasks.

- **Services**: Service requests are the means to process the different tasks. They are sometimes called device drivers and have the power to create system interrupts and poll and push information from various ports and interfaces.

1.1.3 Classification

There are multiple classifications for embedded devices according to their purpose, size, reliability, and other factors. However, these are the main categories and classifications:

- **Scale**: Embedded systems can be classified according to their scale as small-scale, medium-scale, and large-scale. This classification depends on their final application, architecture complexity, and processing size [9].

- **Time constraints**: They can be embedded systems in real time or in non-real time [9].

- **Autonomy**: Depending on whether they are autonomous or not. Autonomous embedded systems do not need regular human input or intervention to function correctly or obtain data other than the initial setup [10].
- **Connections**: They can be either connected embedded systems, which output or receive to an on-line source regularly, or non-connected devices.
- **Mobility**: They can be fixed (non-movable) or mobile.

1.2 Cyber-Physical Systems

A cyber-physical system is a device that presents a combination of computation with a physical process. The behavior of the system is made up of both the physical and the cyber part [11].

The cyber-physical system, or CPS, concerns the intersection of the cyber and physical parts, instead of the union. It applies to all processes that depend on both components at the same time to work. An example of a CPS may be as simple as an actuator when a certain humidity threshold drops below a percentage or as complex as an airplane attitude control protection system. Therefore, the correct classification of a CPS is to consider it as an extension of an embedded system with specific physical and computing characteristics. In this case, instead of sending data to a port or just as data, the actual embedded system sends and receives an input from a physical device.

1.2.1 History and Development

The development of cyber-physical systems has expanded exponentially in the last 15 years, hand in hand with embedded systems. However, the concept of CPS dates back to the first iterations of computing and integrated systems. Most of the time, the study of computer science and physical sciences has been developed independently. These science studies were considered truly independent, and early researchers did not have the need to integrate and combine both disciplines. Nevertheless, as computer systems became more complex and processing capabilities were augmented, there was a growing need to combine both disciplines into one, to improve both efficiency and speed. This is the stage where the cyber-physical systems discipline started. When the cyber and physical worlds intersected, a new set of risks appeared and, as such, new techniques and methods were needed to be developed for the verification of cyber-physical systems.

Two techniques were fundamental for the development of Cyber-Physical Systems in the early stages of the discipline [12]:

- **Real-time scheduling theory**: This theory added time constraints to computational elements. It allows for the verification of the response time of computational processes and those that interact with both physical and computational elements. Furthermore, it provides expected execution times and other important parameters that can correct processes and executions [12].

- **Control theory**: Control theory allows us to verify and analyze whether the process maintains the whole system within bounds, that is, keep the system within a desired region around a specific set-point. It is a continuous-time analysis of the system and process. It is also closely integrated with the theory of real-time scheduling, as it can take into account execution times and delays to specify computation periodicity [12].

Further developments of the technique created new and complex models that closely represent the integration between physical and computational operations, and better simulate the real environment and circumstances. The creation of hybrid systems is one of those advances that allowed for the development of CPSs. A hybrid system integrates the findings and simulations from multiple integrated domains like thermodynamics, timings, electrical, physical, computational, etc.

The integration of the computing and engineering disciplines, and the greater, contemporary focus in complete and combined systems, provides an innovative framework for the development of Cyber-Physical Systems. This is in contrast to the decoupling of development and implementation that has reigned in the early years of embedded system design. This integration has driven an increase in CPS projects such as public-private partnerships for the development of cyber-physical systems such as Europe's ARTEMIS [5].

1.2.2 Verification

The verification of cyber-physical systems has been approached several times; however, fully automatic verification models have not been reached to a satisfactory degree. As systems increase in complexity, it is progressively more difficult and costly to verify and ensure the correctness of these systems. Moreover, the requirements by stakeholders and users of CPS are becoming increasingly stringent, in terms of guaranteed cyber-security and failure tolerance and resilience.

At the moment, due to the difficulty in verifying such systems, the current trend is to shift to a better design paradigm that can reduce the probability of errors and vulnerabilities in CPS from the design stage. It is necessary to assume that errors and security concerns are inevitable in any system designed, even if the best cyber-security paradigms are used and great care is taken on building safe devices.

1.2.3 Future Development

In terms of the future of cyber-physical systems, it is destined to become one of the most studied areas of research. As it stands right now, the increase in year-over-year deployment of CPS is exponential, and the future of cyber-physical devices is set to grow even further. The progressive computerization of new physical elements, such as commodities and appliances, and the modernization of industrial processes are set to become the driving gear for the development of new connected cyber-physical systems. In general, in this development, the verification of the implementation and cybersecurity analysis of these devices should be of importance and quality should take precedence over quantity.

1.3 Dynamical Systems

The study of complex systems and interactions is named *dynamical systems*. Over the years, there have been numerous attempts at obtaining the equations that govern intricate systems, and one of the first studies dates back as far back as 1899, when Poincaré worked on the movement of celestial bodies [13]. Upon his work, contemporaries such as Birkoff, Beltrami, and Strogatz developed and expanded the initial theory.

Formalizing the term *dynamics*, as Strogatz defines it, "it is the subject that deals with change, with systems that evolve in time [. . .] it is dynamics that we use to analyze the behavior." [14] Dynamics, and especially dynamical systems, have become ubiquitous in the definition and lifting of complex systems over the past 40 years.

1.3.1 History

As mentioned earlier, the study of dynamical systems dates back to Poincaré's analysis of celestial bodies. However, the study of dynamics, even without having coined the term yet, dates back to Newton's research. Table 1.1 presents a small guide to the early history of dynamical systems. The evolution of dynamical systems then reached the computing field around the 1970s. In the end, an unknown computer program is a continuous/discrete unknown system. However, several facts and limits must be considered for the concept to be applied in the engineering and computer science fields.

1.3.2 Types of Dynamical Systems

There are two main types of dynamical system: differential equations and iterated maps (also known as difference equations) [14].

1666	Newton	Invention of calculus, explanation of planetary motion
1700s		Flowering of calculus and classical mechanics
1800s	Poincaré	Geometric approach and techniques for the study of dynamical systems
1920-1950		Nonlinear oscillators in physics and engineering, invention of radio, laser, radar due to the study on those systems.
1920-1960	Birkhoff Kolmogorov Arnol'd Moser	Complex behavior in Hamiltonian mechanics
1970s	May	Chaos in logistic map

Table 1.1: Early history of dynamics [14]

- **Differential equations** describe the evolution in continuous time. They can be applied to any physical phenomenon and some continuous computer programs. They are the most widely used in the engineering and science fields. There are two types of differential equations:
  - Ordinary differential equations: It involves only ordinary derivatives dx/dt and d^2x/dt^2. The inferred equations of most embedded systems follow this pattern.
  - Partial differential equations: These present partial derivatives of multiple independent variables. Because the systems analyzed are limited, even if there are multiple variables and parameters that could be analyzed at the same time, the capabilities of the system do not allow this realization.
- **Iterated maps (difference equations)** describe problems where time is discrete. This abstraction can be applied to discrete state machines; however, its analysis can be inferred most of the time from the continuous system.

In summary, a computer system will behave in two different ways. For short periods of time or between time-steps, the program will behave as a linear system. However, when looking at long periods of time, the system behavior will map to a continuous ordinary differential equation. That is why these equations are of interest.

Overall, this behavior could also be applied to some other continuous systems, but for the scope of this paper, the sole focus will be on computer programs and systems.

1.3.3 Dynamical Systems Theory and Control Theory

Dynamical Systems Theory provides the framework for the study of dynamical systems. However, a generalization for the case of computer science is control theory. The objective of control theory is to develop an abstraction that behaves in a predictable way and drives the system to a desired state by using the inputs.

In the case of this thesis, the abstraction that dynamical system theory provides will be useful in developing and understanding the underlying concept of the technique. After all, the aim is to study the dynamical system of a computer program and obtain with it the control equations that govern such system.

1.4 System Representation and Abstractions

In this section, some other necessary concepts and abstractions necessary for the comprehension of the thesis will be presented.

1.4.1 Continuous and Discrete Systems

Building on the previous work in this chapter, the definition of both concepts is the following:

- **Continuous systems**: It is one in which the state variables change continuously over time. For example, the pressure of a chemical process.
- **Discrete systems**: It is one in which the state variables change at a discrete set of points in time. Whether with the same Δ every time or variable Δ time.

Most physical processes are continuous, even if the sampling is done at certain points in time. On the other hand, due to the architecture of computers and processors, most if not all cyber processes are discrete, since they process instructions every Δ time.

The domain that studies the conversion between discrete and continuous systems is signal analysis and processing, and it is necessary for the interaction between the physical and computational world.

1.4.2 Finite State Machines

State machines are mathematical abstractions that represent algorithms and computer programs. There are two main elements in a finite state machine.

- **State**: This is the status of a system waiting to perform a transition to another state or perform an action. They are the basic element of any state machine, and at least one is necessary for a finite-state machine to exist. Some state machines may perform multiple actions in a single state.

- **Inputs**: They are the variables that determine the transitions between the states and the actions of a state machine. These are necessary for a system to take any action. They can be internal or external.

- **Transitions**: They are the change from one state to another. They may be influenced by any of the system's inputs, and there may be multiple transitions available from one state to multiple others and vice versa.

An easy and interactive way to represent state machines is through UML (Unified Modeling Language) representations. These representations are basic in the design of any model. In the case of this thesis project, ICSREF provides a concept similar to that of state machines, which will be explored later.

1.4.3 Automata Models

Automata theory is a branch of computer science established around the 1980s, when mathematicians started developing machines that closely resemble human and real-world devices. It could be argued that it is an evolution of Dynamical Systems 1.3. The development of this theory created the concept of an automaton, which is "an abstract model of a machine that performs computations on an input by moving through a series of states or configurations" [15].

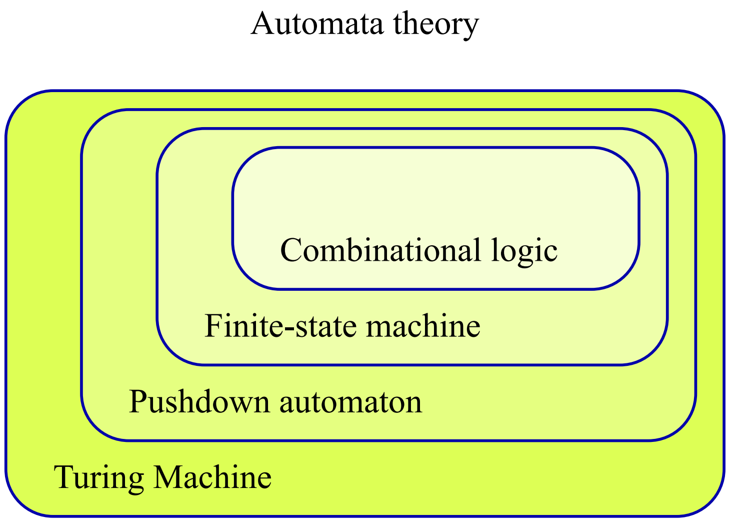


Figure 1.4: Classes of automata (non-exhaustive)
Source: Courtesy of Wikimedia Commons - Original [16]

At each state of the computation, a transition function determines the next state based on input and configurations. The most well-known automata is a Turing machine. The difference, however, between automatas/Turing Machines and finite-state machines is that the scalability of the automata concept and definition is easily abstractable to complex dynamical systems, while the finite-state machine is limited by the amount of finite states. In a computer program, except for the simplest ones, it is impossible to represent all of the states with input options, whereas automata models and, for that matter, Turing machines can correctly represent such abstraction.

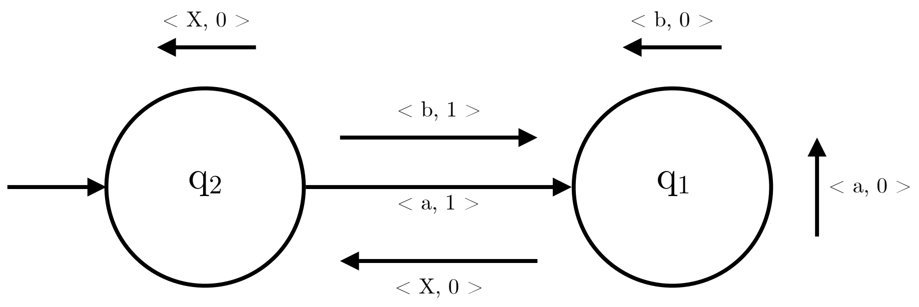


Figure 1.5: Example of an automaton state diagram

Source: Courtesy of Stanford University [15]

As mentioned above, automata models are related to dynamical systems because both try to represent complex systems; however, automata models map discrete systems in contrast to dynamical systems. The main elements of an automata are the following:

- **Inputs**: Set of symbols chosen from a finite set of input signals. A set of symbols is $\{x_1, x_2, x_3, ..., x_k\}$ where k is the number of inputs. It is a finite set I
- **Outputs**: Set of symbols selected from a finite set Z. Set Z is $\{y_1, y_2, y_3, ..., y_m\}$ where m is the number of outputs.
- **States**: It is a finite set Q.
- **State transition function**: Determines the conditions for state transitions and maps the input values to the output states, per state. It performs the action $I \rightarrow Z$

Although the above elements are a generalization of a finite-state machine, the formal definition of a finite-state machine is the following 5-tuple [15]:

- Q = finite set of states
- I = finite set of inputs
- Z = finite set of outputs
- δ = mapping of $I \times Q$ into Q called the state transition function, i.e. $I \times Q \rightarrow Q$

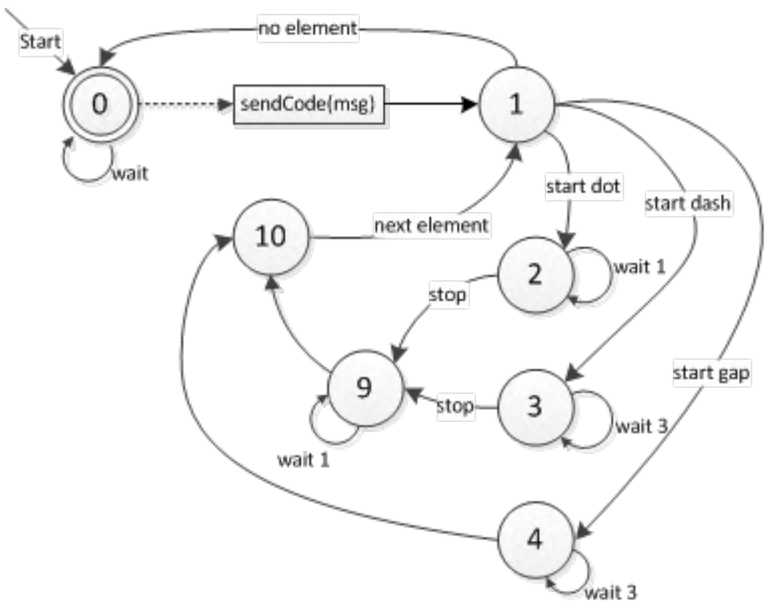


Figure 1.6: Example of a finite-state machine (Morse machine)

Source: Courtesy of mathertel.de [17]

- W = mapping W of $I \times Q$ onto Z, called the output function
- A = set of accept states where F is a subset of Q

The representation of an automaton or a finite-state machine is usually done through a graph-like schema. An example of such a representation is presented in Figure 1.6.

In general, representation is easy and universal. UML has specific guidelines for state machine representation with defined rules and verification schemes.

1.4.4 Domain Specific Languages (DSL)

Programming languages can be classified into two distinct groups: general-purpose programming languages and domain-specific languages. General-purpose programming languages are those like C, Java, or Python, and they can be used to represent and write programs in a wide range of applications and areas. These languages provide versatility, scalability, and ease of use.

However, that generality can sometimes be counterproductive. It is worth noting that not all general-purpose languages are programming languages. In this case, UML (Unified Modeling Language) is a general-purpose language, but it is not a programming language. The broader the language, the less optimized it is for a specific domain. This includes the ease of representing expressiveness in a particular domain. Here, Domain-Specific Languages or DSLs shine [18].

A Domain Specific Language is a type of programming language tailored to one specific function or modeling. In this case, it could be the representation of an embedded system, or it could be the modeling of the functioning of a PLC. All DSLs provide specific constructors and unique structures for the domain. With this, they provide ease of use and expressiveness; however, they are more complex in terms of know-how and present a steeper learning curve.

The development of a DSL is time- and resource-consuming. It requires the expertise of numerous programmers and may not be feasible for every single application. There needs to be a strong feasibility analysis to determine whether a DSL is suitable for the application and a set schedule and phase differentiation. Without these elements, the development of a DSL may not provide any gains compared to the use of general-purpose object-oriented languages [19].

History

Some of the first DSLs date back to the 1950s. BNF (Backus-Naur Form) was developed to provide a description language for programming applications and programs. It is used to describe the syntax of programming languages and other protocols. This method of language description is widely used when a technical description is necessary and there needs to be unequivocal definitions [20].

An example of a Backus-Naur form can be seen in

```
<while loop> ::= while ( <condition> ) <statement>
<assignment statement> ::= <variable> = <expression>
<statement list> ::= <statement> | <statement list> <statement>
<unsigned integer> ::= <digit> | <unsigned integer><digit>
```

Listing 1.1: Backus-Naur Form example

Coq Programming Language

Figure 1.7: Coq logo

One of the domain-specific languages that could be useful for this project is Coq.[1] Coq is a DSL or, as the developers call it, "*computer tool* that allows the verification of

[1] https://coq.inria.fr/

theorem proofs." It uses the Calculus of Inductive Construction as a backbone theory to develop the tool. Some of the applications of this language include the certification of programming languages, the formalization of mathematical theorems, and the verification of C programs.

Although the Coq language will not be used at this point in the development of the project, future iterations will be able to lift the abstractions to Coq.

Chapter 2

Tools and Techniques

This chapter will explain and introduce the main tools and technologies used for this Final Undergraduate Project. The main focus is given to reverse engineering tools and symbolic execution, since these tools comprise most of the final research thesis. However, emphasis on other really useful simulation tools and libraries is given at the end of the chapter.

2.1 Reverse Engineering

Reverse engineering is a technique that examines current systems and tries to infer, through deductive reasoning, its inner workings and construction. As M.G. Rekoff defined it, it is "the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system" [21]. The technique of reverse engineering can be applied to a variety of fields, such as computer engineering, chemical engineering, mechanical engineering, etc., and is not limited to engineering fields only.

The origins of reverse engineering date back to hardware analysis, that is, deciphering the inner workings of hardware devices. However, while the techniques for hardware inference are similar to those used for software analysis, the aim of reversing is different. The traditional hardware reversing objective is to replicate the system. However, in terms of software reverse engineering, the aim most of the time is to "gain a sufficient design-level understanding to aid maintenance, strengthen enhancement, or support replacement" [22].

In the case of this thesis, reverse engineering will be used to infer the equations that govern these cyber-physical systems. There are multiple techniques to achieve the goal of inferring control equations, but the main tools are decompilers and code interpreters. The main focus will be on the former.

2.1.1 Decompilers and Disassemblers

The aim of a decompiler is to reverse engineer the bits read from an executable file and translate those bits into a higher-level programming language. In theory, the decompiled file should be able to be compiled again to exactly the same program, or at least with the same functionality; however, due to current limitations, no decompiler is able to correctly lift the code perfectly.

Decompilers are limited tools in their performance, but in most cases they also include disassembler functionality. A disassembler is a tool that, like decompilers, is able to lift executable files into assembly instructions. The steps for any decompilation are the following [23]:

1 Decode the binary-file format.

2 Decode the machine instructions into assembly code for that machine. Extra smarts are needed to handle indirect transfers of control such as indirect calls and indexed jumps. If the targets of these are not all known, the decompilation will be incomplete for that procedure. Alternatively, human intervention may be required.

3 Perform semantic analysis to recover some low-level data types such as long variables, and to simplify the decoded instructions based on their semantics.

4 Store the information in a suitable intermediate representation If a suitable intermediate language is used, the next 2 steps can be used with any assembly language to generate any procedural HLL code.

5 Perform data flow analysis to remove low-level aspects of the intermediate representation that do not exist in HLLs, e.g. registers, condition codes, stack references.

6 Perform control flow analysis to recover the control structures available in each procedure (i.e. loops, conditionals and their nesting level)

7 Perform type analysis to recover HLL (High Level Language) data types such as arrays and structures. Recovery of classes requires extra analysis. Note: this is one of the hardest steps and may need human intervention.

8 Generate HLL code from the transformed intermediate code.

Ghidra

Ghidra[1] is a free open-source decompiler and disassembler tool developed by the NSA (National Security Agency) of the United States. It is considered a Software Reverse Engineering (SRE) Framework. The tool, released in 2019, is built using Java and `C++`.

[1] `https://github.com/NationalSecurityAgency/ghidra`

Figure 2.1: Ghidra logo

This tool has become one of the most important decompiling tools in recent years, to the detriment of other paid options such as IDA Pro or JEB decompiler. The advantage of Ghidra over other tools is first its open-source license. This allows the community to rapidly develop patches and new functionality. Furthermore, there are numerous plug-ins and add-on scripts to the program that help in the task of decompiling programs. The Ghidra community in GitHub is growing and there are more than 4,000 forks and 180 contributors to the main branch of the program.

Ghidra presents the following features: disassembly, assembly, decompilation, graphing, and scripting. All of these options provide the user with a complete suite for reverse engineering and code analysis. Furthermore, as mentioned above, Ghidra plug-ins extend the capabilities of this software tool. These plug-ins can be developed in both Java and Python (through a translation layer Jython). Some of the most important plugins for Ghidra are: ghidra_bridge, LazyGhidra, ipyghidra, pcode-emulator, and more. In this thesis, the focus will be given mainly to ghidra_bridge and ipyghidra.

- **Ghidra_Bridge:**[2] Ghidra plug-in that allows the integration of the Ghidra console with Python commands. It expands on the limited capabilities that Jython provides, and adds a Python 3 interface. In the development of this thesis, the capabilities that ghidra_bridge provides were useful in the achievement of manual binary analysis.

- **Ghidra IPython:**[3] ipyghidra is a small extension for Ghidra and IPython that builds on previous work. It adds console functionality to ghidra_bridge functionalities, and adds another API to access both internal data for the objects and decompiled functions and objects. For that extra information, it uses Ghidra's JSON information.

radare2

Radare2[4] is a command-line tool, similar to Ghidra, but focuses on command-line execution and disassembly. This tool uses an advanced disassembler framework to function and present the lifted assembly code. Its main functions are: analyzing data, disassembling,

[2] https://github.com/justfoxing/ghidra_bridge
[3] https://github.com/fmagin/ipyghidra
[4] https://rada.re/n/radare2.html

Figure 2.2: Radare2 logo

binary patching, data comparison, searching, replacing, and visualizing. Radare2 is a complete refactor of radare1, and provides with extra functionality and stability.

Radare2 supports multiple architectures and language dissasemblers that can be found in 2.1.

i386	x86-64	ARM	MIPS	PowerPC	SPARC
RISC-V	SH	m68k	m680x	AVR	XAP
S390	XCore	CR16	HPPA	ARC	Blackfin
Z80	H8/300	V810	V850	CRIS	XAP
PIC	LM32	8051	6502	i4004	i8080
Propeller	Tricore	CHIP-8	LH5801	T8200	GameBoy
SNES	SPC700	MSP430	Xtensa	NIOS II	Java
Dalvik	WebAssembly	MSIL	EBC	TMS320	Hexagon
Brainfuck	Malbolge	whitespace	DCPU16	LANAI	MCORE
mcs96	RSP	SuperH-4	VAX	KVX	Am29000

Table 2.1: Excerpt of architectures supported by radare2

The advantages of using radare2 over Ghidra are faster execution and integration with multiple languages. Although Ghidra provides a user-friendly interface, its API connection to other languages and tools is limited, and radare2 improves on that matter. Furthermore, the integration with other tools allows for a more automatic execution, and as such, it is used in the automatic workflow.

Radare2 presents an additional advantage, which is its integrated command-line management and utilities. This allows for the tool to integrate decompilers such as Ghidra's to improve on its functionality. The most important ones used for the project are the following:

r2ghidra r2pipe

Figure 2.3: R2ghidra and r2pipe logos

- **r2pm:**[5] Package manager, allows easy installation of other packages that improve the capabilities of radare2.

[5] https://github.com/radareorg/radare2-pm

- **r2pipe:**[6] API to connect to other languages. It is basically compatible with any programming language that uses pipes to connect to radare2's disassembled output. In this project, it is used to connect it to angr on some occasions for 2.2.

- **r2ghidra:**[7] Radare2's integration of the Ghidra decompiler tools. This is the backbone of the project, as it uses the decompiler to lift the code and eventually present it to the user. Its integration provides a framework to build upon. A Ghidra installation is not necessary for this package, as only the compilers, built in C++ are used.

2.2 Symbolic Execution

Symbolic execution is a technique that analyzes a computer program to determine both the inputs and outputs of that program, and tries to find the relationship between program execution and memory/variable access [24]. The aim of symbolic execution is to be able to debug programs in a similar fashion to static analysis, but provide abstraction and generalize testing for multiple cases and "branches" [25].

Symbolic execution tries to find all execution paths. In this case, each symbolic execution path represents numerous execution paths. In order to execute those paths, instead of directly reading memory or register values, it substitutes those values for variables that can then be used for further analysis. For example, if there is an assembly instruction to read register `r1` and copy it to register `r2`, a symbolic execution tool like angr would assign a variable name and size (float 32, int 16...) to `r1`, and the same variable name and size to `r2`. With this assignment, we would then continue computing further calculations and values.

With this substitution, and other useful techniques, the program is able to run through multiple machine states and infer both constrained and unconstrained variables, depending on the number of if/else statements, while-loops, for-loops, etc. In the case of this thesis, if it is known that in the control loop there are 3 `if` clauses, the final execution will indicate that there are 3 unconstrained variables.

Overall, symbolic execution is a really powerful tool, not only for debugging, but also for vulnerability search, simulations, system verification, and virtualization of cyber-physical systems.

[6] `https://rada.re/n/r2pipe.html`
[7] `https://github.com/radareorg/r2ghidra`

2.2.1 History

The concept of symbolic execution dates back to the 1970s, when a team from the Stamford Research Institute formally defined SELECT, a program whose objective is to "...provide simplified symbolic values for program variables at the output of a path..." [26]. Other early work on symbolic execution includes IBM's EFFIGY and L. Clarke's test generation and symbolic execution techniques at MIT.

In subsequent years, as computers evolved following Moore's law, symbolic execution, a computationally intensive task, became more popular in terms of debugging and analysis. This evolution in computational power made symbolic execution possible, as the time for a complete or at least partial analysis was subsequently reduced. Some of the tools that popularized this technique are z3, rosette, and angr.

2.2.2 Execution Tools

For the symbolic execution in this project, there are numerous tools that can be used. However, angr was chosen because it is an open-source community-backed project against other proprietary solutions.

angr

Figure 2.4: Angr logo

angr[8] is an open source binary analysis tool that performs symbolic execution on binaries. It is built on Python and integrates both static and dynamic symbolic ("concolic") analysis. The advantage of using angr is its support for multiple architectures, built on Python, and ease of use. Furthermore, the interfaces that angr provides creates a framework for correctly lifting the control equations. Angr supports more than double of the languages and architectures of other solutions, and those programming languages include: x86, x86-64, ARM, AARCH64, MIPS, MIPS64, PPC, PPC64, and Java. Angr's interface is through command-line execution, and it provides a fast and reliable way of performing binary analysis and program inferring.

[8] https://angr.io

Apart from symbolic execution, some of the features that angr provides are: control-flow graph recovery, disassembly and lifting to intermediate languages, and decompilation to AIS (angr intermediate language). Furthermore, angr provides extensibility for analysis, architectures, platforms, and more.

Claripy

Claripy[9] is an abstracted constraint-solving wrapper, integrated in angr. It is similar to Z3, and provides a theory solver. Claripy uses ASTs to interact with the different elements in the constraint solver. These ASTs abstract away the differences between mathematical constructs that Claripy supports. In this case ASTs are per se the variables that substitute memory and register writes/reads. There are three ASTs that Claripy supports:

- **BV**: Corresponds to a Bitvector. It can be symbolic or concrete (with a value).
- **FP**: Corresponds to a Floating-Point number. Just like BV, it can be symbolic or concrete.
- **Bool**: Corresponds to a Boolean operator. Its usage is similar to that of BV and FP.

With these three ASTs, which most of the time are not necessary to directly interact with, symbolic execution can be performed to analyze binaries and programs.

2.3 PLC Binary Analysis

One of the cyber-physical devices used for this thesis is a PLC device, built on the WAGO PLC programming language. Symbolic execution techniques were performed on this device, and control differential equations were lifted, without previously knowing information about the PLC itself.

A PLC or Programmable Logic Controller is an industrial computer adapted for the control of a manufacturing process. Its main characteristics are high reliability, process fault diagnosis, and strong physical properties. These devices are ubiquitous in the industry world and follow specific design guidelines and constraints.

[9] `https://github.com/angr/claripy`

2.3.1 PLC Architecture and Characteristics

One of the attacks of this thesis was carried out on a CODESYS[10] platform-based PLC, running binary files created with WAGO[11]. CODESYS is a hardware-independent IEC 61131 platform for industrial control systems (ICS). The decision to analyze these devices is due in part to the fact that CODESYS is widely used in industry, and partly because the aim was to simulate 2.3.3's attack with the lifted differential equations. CODESYS is used by 250 manufacturers and has a significant market share in the field of industrial control platforms, as can be seen in Table 2.2

Company	Development Platform	CODESYS-based?
Rockwell Automation	Studio 5000 Logix Designer	No
Siemens	STEP7	No
ABB	Automation Builder	Yes
Schneider Electric	SoMachine	Yes
Bosch Rexroth	Indralogic	Yes
Wago Kontakttechnik	WAGO-I/O-PRO	Yes
Eaton Industries	XSOFT-CODESYS	Yes
Beckhoff Automation	TwinCAT	Yes
Lenze Automation	PLC Designer	Yes
Owen	CODESYS	Yes
Omron	CX-One	No
SEL	acSELerator	Yes
ifm electronic	CODESYS	Yes
STW Technic	CODESYS	Yes
Berghof Automation	CODESYS	Yes

Table 2.2: Automation Platforms of ICS Vendors [27]

Tennessee Eastman Chemical Process

The target process is based on the Tennessee Eastman (TE) chemical progress which is based on a model presented from [28]. The version of the analysis is modified to meet the requirements of the ICSREF paper, [27], because the attack will be used to validate the lifting system. The TE process is a realistic simulation of a chemical process, released to the academic community as a reference process [29], which uses pressure output and conditions to model attacks on an Industrial Control System.

[10] `https://www.codesys.com`
[11] `https://www.wago.com/global/automation-technology/discover-software/codesys-2`

2.3.2 Proportional Integral Derivative (PID) Controller

For this project, it was not feasible to analyze a random binary, so it was decided that an analysis would be performed only in the `PID_FIXCYCLE` function. The `PID_FIXCYCLE` function can be used to control a Proportional Integral Derivative (PID) controller, widely used in industrial systems.

A Proportional–Integral–Derivative controller is a control loop mechanism that employs feedback in applications that require continuously modulated control. A PID controller is governed by Equation 2.1 (parallel form) [30].

$$u(t) \;=\; K_p e(t) + K_i \int_0^t e(\tau)\,\mathrm{d}\,\tau + K_d \frac{\mathrm{d}\,e(t)}{\mathrm{d}\,t} \tag{2.1}$$

In Equation 2.1, the terms K_p, K_i, and K_d represent the non-negative coefficients of the proportional, integral, and derivative. However, there is a standard form that is widely used in industry (Equation 2.2) where instead of PID coefficients, T_i and T_d are used. These represent the *integral* and *derivative* time, respectively. In this form of the equation the parameters have actual physical properties rather than being parameters. The sum of both terms is actually a single new error that is compensated for by the architecture of the algorithm. In this case, the representation is clearer.

$$u(t) \;=\; K_p(e(t) + \frac{1}{T_i} \int_0^t e(\tau)\,\mathrm{d}\,\tau + T_d \frac{\mathrm{d}}{\mathrm{d}\,t} e(t)) \tag{2.2}$$

The PID controller continuously calculates an error value $e(t)$ as the difference between a set-point and a measured variable. It then applies a correction based on the calculations set by the equations. In this case, the focus will be on `PID_FIXCYCLE`, which, as mentioned, represents a PID controller for which the cycle time can be set manually. [31]

In terms of the output and input of this function, they are presented in Table 2.3

These inputs and outputs will be inferred and can be used independently to control the lifted behavior of `PID_FIXCYCLE`. The input values are determinants for the correct execution and simulation of the attack presented in ICSREF.

2.3.3 ICSREF PLC Analysis Tool

ICSREF [27] or Industrial Control Systems Reverse Engineering Framework, "automates the reverse engineering process for ICS binaries and can provide information on the physical characteristics of a system captured in the ICS binaries controlling it, without any

Scope	Name	Type	Description
Input	ACTUAL	REAL	Actual value, process variable
	SET_POINT	REAL	Desired value, set point
	KP	REAL	Proportionality const. P
	TN	REAL	Reset time I in sec
	TV	REAL	Rate time, derivative time D in sec
	Y_MANUAL	REAL	Y is set to this value as long as MANUAL = TRUE
	Y_OFFSET	REAL	Offset for manipulated variable
	Y_MIN	REAL	Minimum value for manipulated variable
	Y_MAX	REAL	Maximum value for manipulated variable
	MANUAL	BOOL	TRUE: Manual: Y is not influenced by controller | FALSE: Controller determines Y
	RESET	BOOL	TRUE: Sets Y output to Y_OFFSET and reset integral part
	CYCLE	REAL	Time in s between two calls
Output	Y	REAL	Manipulated variable, set value
	LIMITS_ACTIVE	BOOL	TRUE: Set value would exceed limits Y_MIN, Y_MAX
	OVERFLOW	BOOL	Overflow in integral part

Table 2.3: Inputs and Outputs for `PID_FIXCYCLE`

prior knowledge of the system." This tool allows for the analysis of PLC binaries, obtaining function signatures, memory inputs and values, and correctly analyzing register values. It is also capable of creating function call maps and presenting the information through an API.

ICSREF is used to simulate an attack on the Tennessee Eastman process. They have integrated in the binaries the attack to that specific process, and in order to prove that InteGreat is able to correctly lift the control differential equations, an attack is done on the same process but using the lifted equations. This is the reason why ICSREF was useful in achieving the project objective.

ICSREF as such provided the memory values to simulate the attack and helped to infer `PID_FIXCYCLE` from the `TE.PRG` binary file.

2.4 Reachability Analysis

Reachability analysis consists of the evaluation of all possible states reachable from an initial set-point state given certain constraints and input parameters. In general, the goal is to check whether a set of final states can be reached, within reasonable input parameters, from a set of initial states [32]. Factors that may influence the reachability analysis could be additional constraints on any state, a specific requirement for reachability paths, iterative reachability, or trying to attain states by giving preference or rewards to certain states. Reachability problems can be applied to numerous fields, such as finite- and infinite-state concurrent systems, physical models, hybrid systems, and more.

The study of reachability analysis dates back to the early 1970s, with the first Petri net experiments [33]. Since then, it has been developed in recent years, especially in computer science, where the goal is to obtain complete automata models (see Section 1.4.3). However, constraints in the amount of computational power and costs of developing and obtaining new reachability analysis tools have hindered its development in the early 2000s.

Automata models are basic for correctly interpreting what a reachability analysis tool aims to do. Basically, a reachability analysis works on top of a pre-existing automata model, to compute states, transitions, and constraints in the system.

2.4.1 DaDRA

Figure 2.5: DaDRA logo

DaDRA[12] is a Python library for Data-Driven Reachability Analysis. The goal of the package is to accelerate the process of computing estimates of forward reachable sets for nonlinear dynamical systems. This package allows for the execution of a reachability analysis on different Python programs. In terms of usefulness, it helped to determine the different states that the quadcopter could reach once the input parameters were locked. This tool allows for a clear and user-friendly representation of the reachability analysis and provides an interface to that simulation.

2.4.2 JuliaReach

Figure 2.6: JuliaReach logo

[12] `https://github.com/jaredmejia/dadra`

JuliaReach[13] is an open-source reachability analysis tool that "implements reachability analysis methods for systems of ordinary differential equations (ODEs), for both continuous and hybrid dynamical systems."

JuliaReach is not used at this stage in this project; however, it is worth mentioning it as its capabilities are greater than that of DaDRA's and provide extensive reachability analysis. The aim is to integrate JuliaReach into the final binary that will be presented later for the conference.

Nevertheless, JuliaReach currently provides support for the following types of system:

- Continuous ODEs with linear dynamics
- Continuous ODEs with non-linear dynamics
- Continuous ODEs with parametric uncertainty
- Hybrid systems with piecewise-affine dynamics
- Hybrid systems with non-linear dynamics
- Hybrid systems with clocked linear dynamics

These systems fall exactly in the expected goal for this project.

2.5 Radar Simulation

For the analysis of Continental's Radar, it was necessary to simulate, using NXP's RadarSDK, the behavior of such a device in a virtual environment. Continental's radar is based on the NXP S32 family of chips, which are built on a 32-bit PowerPC architecture. This simulation of that device included the use of MATLAB and NXP's RadarSDK.

2.5.1 MATLAB

MATLAB[14] is a multi-paradigm programming language and computing environment developed in the late 1970s by Mathworks Inc. The strengths of MATLAB stem from its numeric computing features and the presence of toolboxes, such as the Signal Processing Toolbox, with multiple applications in the signal and processing fields. For this thesis, the following additional features were used.

[13] https://juliareach.github.io/
[14] https://www.mathworks.com/products/matlab.html

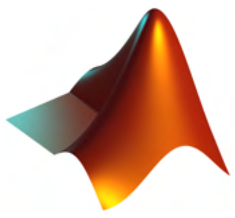


Figure 2.7: MATLAB logo

Signal Processing Toolbox™

The signal processing toolbox for MATLAB, "provides functions and apps to manage, analyze, preprocess, and extract features from uniformly and non-uniformly sampled signals" [34]. This toolbox provides efficient and optimized functions for signal processing, such as FFT, refactoring, filters, frequency domain visualizations, and more. NXP's Radar Demo uses this library to provide fast operations while generating `3D FFTs` and other processing elements required for the example.

The advantage of using this toolbox is that it is compatible with GPU/CUDA© acceleration and its functions can be exported to C/C++, with MATLAB's Coder App, which is used by NXP's software to obfuscate the contents of their signal processing code.

NXP RADAR Toolbox for S32R

Figure 2.8: NXP logo

NXP's Radar Toolbox is a complementary development environment for NXP's S32 family of chips. It offers a path of integration between advanced radar signal processing capabilities and microcontroller capabilities, for generic software task and car bus interfacing. It also provides multicore architecture support and signal processing acceleration [35].

2.5.2 NXP Premium RadarSDK

RadarSDK[15] is part of the NXP Semiconductor Automotive Radar System package. It provides capabilities to program and build tools for NXP's automotive radars (S32R45 and S32R41 families) and tools for debugging. While access to RadarSDK is limited to

[15] https://www.nxp.com/design/automotive-software-and-tools/premium-radar-sdk-advanced-radar-processing:PREMIUM-RADAR-SDK

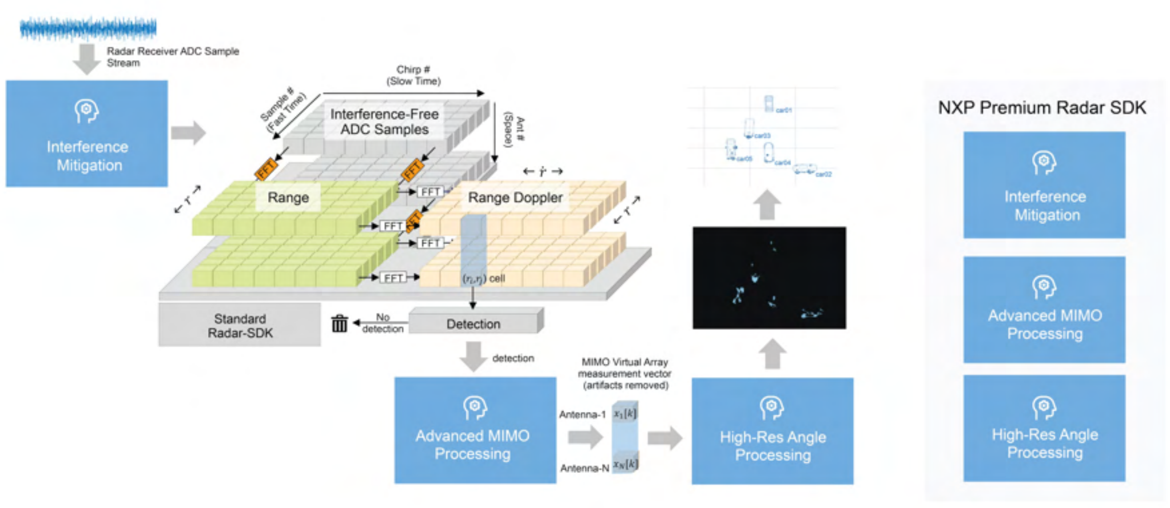


Figure 2.9: Premium RadarSDK Processing Chain

Source: Courtesy of NXP Semiconductors

paying customers, some of the functionality and examples are available, while following NXP'S terms and conditions.

RadarSDK is used as the third target for the program. The advantage of using this development environment as a target is the structure and availability of kernels for MATLAB. It includes the following elements [36].

- **SPTDriver**
- **SPTKernels**
- **Matlab bitexact model** for SPT kernels
- **RF Abstract API** for NXP FE (incl SPI & CSI2 I/F)

2.6 Parallel Execution

Some other useful tools used for the development of this thesis project include these command-line tools, that helped both in the execution and evaluation of the final tool.

2.6.1 tmux

Figure 2.10: Tmux logo

Tmux[16] is a terminal multiplexer, that is, a program that allows multiple terminal processes to be run at the same time. Tmux sessions persist in time, even if there is no connection, and it is especially useful for headless machines. Tmux is easily accessible though a terminal connection, and provides an easy interface for the development of programs that required extended execution times. Some of the main features of tmux are:

- **Session persistence**: tmux sessions, unless closed, persist even if a user is not connected. This allows for the continuation of previous work and the ability to connect from multiple ssh sessions to the same tools and working environment.
- **Session management**: tmux allows to execute multiple "individual" command-line processes at the same time without having to interrupt any one or having to create new connections.

This tool was really useful for the execution of large binary analysis, using the University of Illinois S&PR@I servers. Furthermore, the ability to maintain sessions over different computers allowed for the execution both at the laboratory and remotely.

2.6.2 keep

Figure 2.11: Keep logo

Keep[17] is a simple command-line toolkit that allows saving terminal commands across for ease of use. Commands are saved with a small description, and powerful search patterns allow for the easy execution of repetitive/tedious command-line commands. Furthermore, keep allows the synchronization of those commands by means of a GitHub gist.

[16] `https://github.com/tmux/tmux/`
[17] `https://github.com/OrkoHunter/keep`

2.7 Memory read

Techniques for reading memory chips, such as the one present in the coffee maker, were necessary. Although these devices were used, actual memory acquisition was unsuccessful due to unforeseen reasons.

2.7.1 Arduino

Figure 2.12: Arduino logo

An Arduino[18] is an open-source integrated single-board computer that contains, in a single package, a wide array of components for low-power computation. Arduino boards are widely used in embedded systems due to their cost and capabilities.

In this thesis, an Arduino Mega 2560[19] was used to try to obtain memory from a Winbond chip from the coffee maker, with no success.

2.7.2 Arduino libraries

The library used for SPI memory acquisition is SPIMemory[20]. It contains easy-to-use commands and compatibility with a large amount of SPI memory vendors.

2.8 Other tools

These tools and programs are necessary for the execution of the project and helped in achieving the goals set out at the beginning. Some of these are basic, such as programming languages and IDEs, and some others are more task-specific and present a steep learning curve.

[18] `https://www.arduino.cc/`
[19] `http://store.arduino.cc/products/arduino-mega-2560-rev3`
[20] `https://www.arduino.cc/reference/en/libraries/spimemory/`

2.8.1 Python

Figure 2.13: Python logo

Python[21] is a high-level programming language. It is interpreted and general purpose, and provides one of the most extensive documentations and toolkits, through add-on libraries. Its popularity has risen over the past few years, holding the first position as the most popular programming language in some of the latest statistics [37].

For the development of this project, Python3 and Python2 and a multitude of Python libraries and resources were used. Some of those have already been mentioned, like angr, DaDRA, however, an excerpt of the most important packages follows in Table 2.4.

Name	Description
matplotlib	Comprehensive library for creating static, animated, and interactive visualizations in Python
regex	String matching utility
numpy	Mathematical library that provides multidimensional array objects, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays

Table 2.4: Excerpt of Python packages used in the project

2.8.2 Jupyter Notebook

Figure 2.14: Jupyter Notebook logo

[21] https://www.python.org

Jupyter Notebooks[22], formerly a spin-off of IPython, is a web-based interactive computer environment to create notebooks. The advantage of Jupyter Notebook is that it can be integrated with different programming language kernels and provides interactive support for a multitude of programming languages.

In this project, Jupyter notebooks were used to prototype and verify the correctness of the inferred functions decompiled and disassembled from Ghidra and, as such, provided an easy way to verify the initial concepts. The notebook was used to first familiarize with the developing environment and to perform checks and verification. However, its use was limited to the beginning of the project while prototyping a solution.

2.8.3 IPython

IP[y]:

Figure 2.15: IPython logo

IPython is a command-line shell for interactive computing for Python that provides a kernel for the execution of user code from Jupyter Notebook and other programs. It is also responsible for computing the completions of the Qt console. In this project, IPython is used in conjunction with the Jupyter Notebook, moreover some of its features are used to communicate between the kernel and the specific functionalities needed for symbolic execution and binary analysis [38].

2.8.4 MPLAB X IDE

MPLAB®
X IDE

Figure 2.16: MPLAB X IDE logo

MPLAB X[23] is a proprietary IDE (Integrated Development Environment) released by Microchip Technologies. It is designed to be able to program all of Microchip's PIC-based microprocessors. It supports 8/16/32 bit microcontrollers and supports external programmers.

[22] `https://jupyter.org`
[23] `https://www.microchip.com/en-us/tools-resources/develop/mplab-x-ide`

This program was used in the early stages of binary analysis, in conjunction with the PICKit 4 Programmer, to try to obtain the memory of the PIC32 coffee maker chip. However, due to hardware locks, its use was limited and short.

MPLAB PICkit 4 In-Circuit Debugger

The PICkit 4[24] is a microcontroller debugger, created by Microchip Technologies, that works in conjunction with MPLAB X. It was used to connect to the debug port of the PIC 32 through a self-made adapter to RJ11. However, its use was unsuccessful as mentioned above.

2.8.5 Git

Figure 2.17: Git logo

Git[25] is a distributed version control system. It is an open-source project that is capable of managing small to large projects efficiently. Version control systems are focused on the ability to control multiple files and versions concurrently and provide with the ability to test features before deploying them through means of branches and forks.

For this project, it was used to collaborate between team members and has an easy way to collaborate with external parties that have helped in this project.

GitHub

Figure 2.18: GitHub logo

In order to correctly implement and correctly use all of Git's features, GitHub[26] was used for source control. GitHub is a web-based interface that deploys all of Git's features

[24] https://www.microchip.com/en-us/development-tool/PG164140
[25] https://git-scm.com/
[26] https://github.com/

and provides an easy user interface. It can be synchronized between devices and team members.

2.8.6 Visual Studio Code

Figure 2.19: Visual Studio Code logo

Visual Studio Code[27] is a source code editor, built by Microsoft Corporation and released in 2016. Most of its code is open-source under the MIT license, and according to the 2021 Stack Overflow Developer Survey, it was the most widely used IDE [39].

The advantage of VS Code over other editors is the possibility of adding plug-ins, both official and community-made plug-ins. This allows for the integration of the IDE with Git version control in an easy way, TeX editor, Markdown editor, SQL Query manager, multiple language integration, Jupyter Notebooks, etc.

The ability to provide such integration and compatibility with languages makes the development of code and this thesis easy.

2.8.7 Servers and Physical Devices

Finally, this project needed both computing power and an environment to deploy solutions and tests. This was possible by the use of multiple servers provided by the director of this thesis and an additional server provided by the author hosted at Hetzner[28].

The use of all of these servers was done through ssh and headless execution.

[27] `https://code.visualstudio.com/`
[28] `https://www.hetzner.com/`

Chapter 3

State of the Art

This chapter introduces some of the previous work in the fields of system verification and analysis in the computer field. Furthermore, the state-of-the-art of some of the tools presented in Section 2 is explained in further detail, especially the concepts and theory behind them.

As of right now, there are few or no tools that allow for the integration between high- and low-level code. Furthermore, there has not been much work done in terms of a security-focused tool that builds its inference and verification schemes from the ground up, that is, from bits to high-level programming languages and domain-specific languages. InteGreat aims to bridge that gap and be a cybersecurity and system verification-focused tool, by combining some of the most state-of-the-art techniques, libraries, and methodologies.

3.1 Previous Work

Since the introduction of the first embedded systems, there has been a tremendous amount of work on both their development and evolution. As was mentioned, embedded systems, and for that matter, cyber-physical systems, have become omnipresent in our world. But with the increase in popularity, there is also an increase in vulnerabilities discovered and the need for system verification.

Work on system verification has also been growing in recent years. With more compute power, testing through "brute force" has become cheaper as systems have become more powerful and can be done in some specific devices. These devices typically require high levels of safety and stability in their systems. However, as embedded systems are becoming increasingly cheaper, there is a need to deploy novel solutions and products as quickly as possible, and if that requires reducing testing times and cases, it is sometimes preferred. The clear example is the number of devices that continuously have patches applied to their firmware or software due to vulnerabilities.

The work in dynamical systems, as explained in Section 1.3, has evolved from the attempt to infer complex physical systems into a shift to the computational world. These techniques, designed at first for continuous systems, can be applied to discrete state-machine systems, by means of hybrid automata models. Some work has been done on this domain in [13]. However, little has been done in terms of verifying systems from the ground up to a high-level programming language. Most, if not all, verification schemes for cyber-physical systems depend on lifting the models to `C` code, or at least pseudo-`C`. Nevertheless, for the most part, these implementations are neither comprehensive nor implemented in real life in practice.

The shortcomings of some of the previous work stem from the lack of microarchitecture and low-level programming adjustments. Techniques for high-level analysis have been developed and are well known, but there is a lack of detailed studies of lower-level code analysis in their properties and specific characteristics. Here is a more detailed analysis of previous work done in some of the fields concerning this thesis.

3.1.1 Dissasemblers and Decompilers

The work on decompilers has improved greatly in the last 5 years or so. Since Ghidra was released to the public in 2019, including the decompiler and disassemble software, its implementations have been numerous. However, the work on decompilers is still on-going. New sleigh specifications and different architecture support, to translate binary code into actual assembly code instructions, are becoming more precise, especially for CISC architectures, which present variable-size instructions [40].

3.1.2 Hybrid System Verification

In terms of hybrid system verification, work has been done on system verification, using domain-specific languages such as Coq and Coq's proof assistant. An example of this analysis is given in the VERIDRONE paper [41]. The purpose of this verification is to provide safety for programming operations. In addition to the VERIDRONE article, further analysis of the code has been carried out with the development of a new DSL as seen in [42]. However, there is no further work in terms of low-level verification of hybrid automata systems.

3.1.3 Function Matching

Function matching development has been extensive, especially in recent years, as seen in [43], [44], [45]. Work on existing vulnerable function matching and function research. These studies can be helpful in terms of integration into the program. The work on this tool pretends to find another way to match functions and discover vulnerabilities.

3.1.4 Reachability Analysis

Reachability analysis has been extensively studied in terms of high-level code, but the development of this analysis, taking into consideration the limitations and difficulty of low-level code, such as assembly, has not been widely studied. As studied in [46], [47], these provide safety verification in continuous domains but lack some considerations on the more intricate parts of lower-level execution.

3.2 Improvements and Novelties that the Project Introduces

This project bridges the gap that currently exists between low-level code analysis and high-level representation. The ability to represent, in clear Python statements, complex programming functions provides a great tool for vulnerability and stability analysis.

Moreover, its ability to provide support for multi-architecture, multi-platform system verification adds to the usefulness of the tool. In future iterations, the configuration package for each architecture will be programmed using a set of `.json` files. With this abstraction in mind, it is not far fetched that apart from the three studied architectures (arm, x86, PowerPC), the system could be easily expanded to other architectures and useful devices.

One of the limitations that the project has, the inability to produce consistent binary analysis for control loop functions, could be a feature in the future. There is work being done at the University of Illinois in terms of function identification and characterization for multiple architectures. Although the work focuses on the search for library vulnerabilities, this ID technology could prove useful for the recognition of known control loop functions, similar to what ICSREF does with the identification of PLC functions `PID_FIXCYCLE` [27].

Finally, the tool provides ease of use. For programs that have identified functions, the work for the end user is easy. However, there needs to be some understanding of binary file disassembly and program structure, as most programs are not fully compatible with automatic disassembly and equation lifting.

Chapter 4

Project Definition

In this chapter, the motivation, objectives, and plan of the project are presented. The aim of a fully developed system verification platform will be emphasized, even if the extent of such a task is not achievable in the scope of this project.

4.1 Motivation

The motivation for this project comes from the search for new and novel techniques to verify and analyze the cybersecurity of embedded systems, and more specifically cyber-physical systems. However, this verification scheme can be expanded to other devices that rely on continuous control loops to perform their actions. This system verification is vital due to the increasing number of devices that, in the name of cheapness, lack system and security verification.

Once an embedded system goes through the first stages of development, problem definition, system design, coding, debugging, and testing, there is always a risk that an already programmed and used system may have implementation error or an unknown bug may affect its utility. These errors could even cause a catastrophic failure in device operation. Therefore, it is necessary to verify and correctly assess that the performance of a gadget is as expected and that no known or unknown situations may arise from its operation. With this tool, after the lifting of differential equations, a further analysis of state reachability, provides a study of unstable states and unexpected behavior. Further analysis may provide cybersecurity vulnerabilities through privilege execution and malicious firmware upload to the device with modified memory values.

With complete development of the lifting tool, it will be possible to analyze binary code from an embedded system, obtain looping functions, lift control equations, perform a reachability analysis, and find vulnerabilities in the code. With all this information, new techniques can be developed to further improve the tool.

4.2 Objectives

The objective of this project is the search for a multi-functional, multi-platform tool that can provide advanced analysis for cyber-physical system binaries. The aim is to be able to semi-automatically analyze a binary file from different architectures and to be able to lift the control routine or loop for that device. To achieve this goal, there are four main sections of the project that need to be completed.

1 **Binary analysis and function identification**. Correctly obtain and decompile the binary file into separate functions and be able to identify the main recursive control loops within the file. This section may be done partially manually, as the current function identification is limited by technology and lost naming conventions in the binaries. Furthermore, the entry points to loops and functions are sometimes dependent on the type of function calls the program has, either dynamic or static, and have to be manually adjusted for each case individually.

2 **Symbolic execution and equation lifting**. The binary files will be executed symbolically, using angr and claripy to correctly model interactions with read and written memory and register values. Once these operations are lifted symbolically, they are exported to a Python file that needs minimal modification to create a class. Then a verifier is built manually to verify the correctness.

3 **Correctness of equation lifting**. Verify and assess the correctness of the lifted equations. For this step, the behavior of the equation will be compared with the expected behavior of the code using graphs and variable simulation.

4 **State reachability analysis**. To finalize the acquisition of the control equations,a reachability analysis will be done to some of the binaries studied, using tools such as DaDRA. With these tools, the objective is to demonstrate that systems may reach an unexpected state in which their operation is unstable and no longer reliable. With this reachability analysis, it is possible to identify the memory and sensor input values that can cause errors in the system, thus creating vector cyberattacks.

4.3 Methodology

In terms of work methodology, an Agile-like scheme is used; however, due to the nature of the progress and development of the tool, some deviations were taken.

Weekly meetings were held every Monday to review progress. In these meetings, the current situation and short-term objectives for that week were established. Furthermore, there were no daily meetings, but due to the ability to work in the SPR@I (Security and Privacy Research at Illinois) laboratory in the CSL (Computer Science Laboratory) building, concerns and questions were practically addressed immediately with the team and the project director.

4.4 Plan

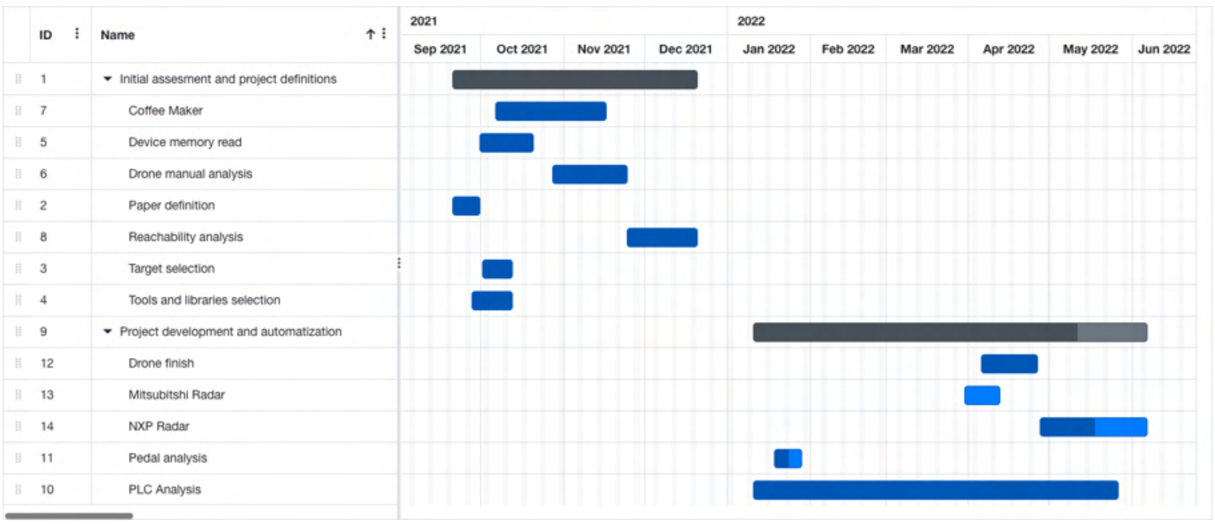
The schedule and plan of the project are explained in this section. Furthermore, the completion of the objectives (only the main objectives are shown) is shown in a Gantt chart (Figure 4.1).

1 Initial assessment and project definitions

- **1.1 September 2021**. Definition of the paper. Target selection: coffee maker and open-source drone. Objectives of the project, tools and libraries, concepts, and introduction. Setup of the environment and compatibility checks.
- **1.2 October - November 2021**. Start of the device memory reading and collection. The coffee maker exploded. Drone manual analysis and initial equation lifting.
- **1.3 December 2021**. Final proof of concept. Reachability analysis of drone binaries and acceptance of the results of the proof-of-concept.

2 Project development and automatization

- **2.1 January 2022**. New objectives chosen due to the lack of resources in the initial devices. A PLC and a guitar pedal were chosen for the development of the research.
- **2.2 February – April 2022**. Further automatization and independence of the system. PLC analysis. An automotive radar system is chosen as the third target and the memory is read from it.
- **2.3 May 2022**. Summary of the project. Code cleaning and further automatization and interface. Presentation of the paper at a workshop and a written project.



ID	Name
1	Initial assesment and project definitions
7	Coffee Maker
5	Device memory read
6	Drone manual analysis
2	Paper definition
8	Reachability analysis
3	Target selection
4	Tools and libraries selection
9	Project development and automatization
12	Drone finish
13	Mitsubitshi Radar
14	NXP Radar
11	Pedal analysis
10	PLC Analysis

Figure 4.1: Gantt chart of the project plan

Chapter 5

Development and Research

This chapter will focus on the methodology, devices, techniques, and results of the research, that is, how the project was built and what accomplishments or milestones were reached.

5.1 Analyzed Devices

Three main devices were analyzed for this project. However, work was done on several others that did not make it to the analysis, due to unexpected shortcomings or impossibilities. The focus will be on presenting a description of the devices, with specifications, and some of the problems encountered in terms of obtaining memory files. A further detailed analysis of the binary file specifics will be presented in Section 5.2.

5.1.1 Drone - Quadcopter

The binaries analyzed belong to an open-source quadcopter project whose aim is to develop and study control algorithms for a flying device. The objective of the original project was to obtain new control methods and apply them to the development of quadcopters. The new methods take into account the new algorithms in motion and dynamics done in recent work [48].

One technique that was implemented in the quadcopter software that will be analyzed in this thesis is a quaternion update function, specifically `Madgwick Filter`[1]. Madgwick Filter is a novel way of making quaternion updates for drones and other aerial systems, which takes into consideration bias and sensor error by adapting recursively filters in the

[1]`https://github.com/bjohnsonfl/Madgwick_Filter`

update functions. It uses the input of an integrated IMU (Inertial Measurement Unit) consisting of tri-axis gyroscopes and accelerometers, and MARG sensor arrays that also include tri-axis magnetometers [49].

The advantage of analyzing this drone binary file is that the drone binary files, the drone code, and the original Madgwick filter are available. This allows for both extensive reachability analysis and, most importantly, for checking that the lifted equations are correct and correspond to the original plan.

With all this, quadcopter binaries were used to evaluate the lifting of control equations for devices of `arm` architecture. As mentioned, not all binary functions will be studied, as the project relates only to control equations. As such, the focus will only be on `Madgwick Quaternion Update`.

Figure 5.1: Image of a Pluto drone

Source: Courtesy of Pluto/Drona Aviation & amazon.in [50]

Characteristics and Data

- **Device**: Quadcopter Pluto Drone (simulated but real binaries)
- **Manufacturer**: Drona Aviation
- **Architecture**: ARM
- **Microcontroller/Microprocessor**: STMicroelectronics STM32F103 (Cortex-M3) [32-bit]
- **Programming language used**: `C`
- **Analyzed binary**: `Madgwick Quaternion Update` function on `drone.bin` binary file
- **Open Source Project**: Certain libraries used are closed sourced
- **Resource link**: `https://github.com/heethesh/eYSIP-2017_Control_and_Algorithms_development_for_Quadcopter`
- **Analysis scheme**: Manual

Problems

The advantage of this device is that the code did not have to be read from physical memory. This allows for easy implementation and analysis of the different lifting schemes and greatly improved the speed of binary analysis. Some problems arose in terms of binary lifting, but these will be explained in Section 5.2.

5.1.2 WAGO PLC

The second device that was analyzed for control equation lifting is a WAGO based PLC. The file was taken directly from the ICSREF binary example files [27]. Although some minor modifications were necessary to open it in Ghidra for initial evaluation and analysis, the behavior is exactly the same. However, the final version of the binary is explicitly presented by ICSREF.

Figure 5.2: Image of a WAGO PLC

Source: Courtesy of WAGO

Characteristics and Data

- **Device**: WAGO PLC (simulated but real binaries)
- **Manufacturer**: WAGO
- **Architecture**: x86-32
- **Programming language used**: `WAGO`
- **IDE**: CODESYS
- **Analyzed binary**: `PID_FIXCYCLE` function on `TE.PRG` binary file
- **Open/Closed Source Project** (Some elements are propietary)

- **Resource link**: `https://github.com/momalab/ICSREF`
- **Analysis scheme**: Automatic

5.1.3 Continental Radar System

The latest device analyzed is a Continental ARS4-B Automotive Radar. Note that although this device was partially analyzed, the control equations were not lifted at the time of writing this thesis. However, it will be done before a future release of the conference paper.

Figure 5.3: Image of a Continental ARS4-B (Note that it is a Tesla Inc. branded part)
Source: Courtesy of SystemPlus Consulting [51]

Characteristics and Data

- **Device**: Continental ARS4-B Automotive Radar
- **Manufacturer**: Continental (ADC Automotive Distance Control Systems GmbH)
- **Architecture**: PowerPC
- **Microcontroller/Microprocessor**: NXP MPC5775/S32R274 [32-bit]
- **Programming language used**: `C, NXP Proprietary`
- **IDE**: NXP RadarSDK
- **Analyzed binary**: `RSDK_offline_example.m` from NXP's Radar SDK Library for `MATLAB`
- **Closed Source Project**
- **Resource link**: `https://www.continental-automotive.com/en-gl/Passenger-Cars/Autonomous-Mobility/Enablers/Radars/Long-Range-Radar/ARS441`
- **Analysis scheme**: Partial

5.1.4 Other Devices

Some other devices that were not useful for the final result were also analyzed. However, by studying these, numerous concepts were learned from their analysis and memory acquisition.

Keurig© K-Elite Coffee Maker

The first device studied in this project, even before any work on the initial analysis of the drone, is a Keurig semi-automatic coffee maker. This coffee machine was already disassembled and the board and components were already detached from the body and components of the coffee maker.

Figure 5.4: Keurig K-Elite coffee maker (model not exact)
Source: Courtesy of Keurig

The purpose of this device was to obtain the memory values and to start performing embedded system analysis on the control loops that govern the functioning. The aim was to obtain and simulate an attack on the boiler system, to make it exceed parameters. There were two units available, one fully disassembled and the other functioning.

- **Device**: K-Elite© Coffee Maker
- **Manufacturer**: Keurig©
- **Architecture**: MIPS32
- **Microcontroller/Microprocessor**: Microchip PIC32MX360F512L
- **Programming language used**: Unknown (`C` possibly)
- **IDE**: MPLAB X
- **Non-volatile memory make and model**: Winbond W25Q128FV (SPI interface)

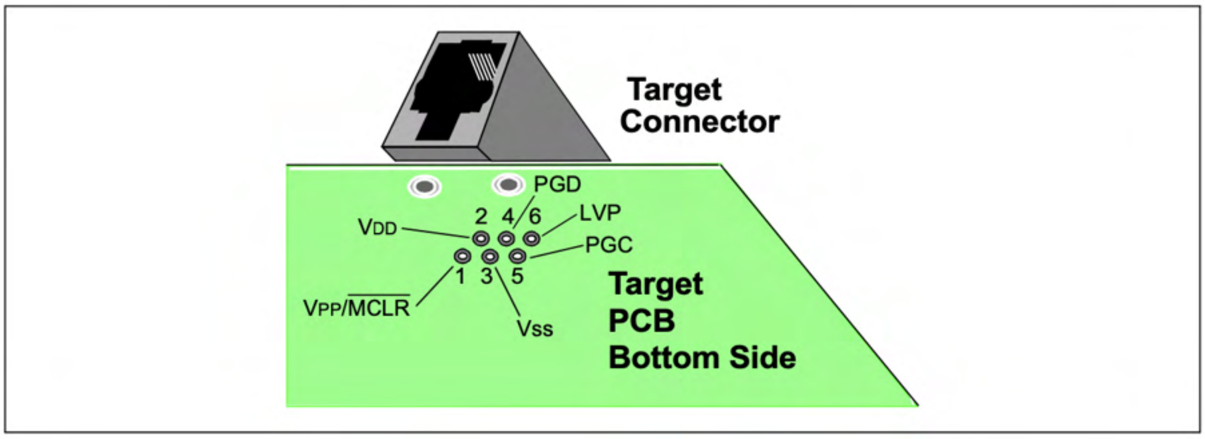


Figure 5.5: Debugging port interface pinout

Source: Courtesy of MICROCHIP

- **Analyzed binary**: `RSDK_offline_example.m` from NXP's Radar SDK Library for `MATLAB`
- **Closed Source**
- **Resource link**: `https://www.continental-automotive.com/en-gl/Passenger-Cars/Autonomous-Mobility/Enablers/Radars/Long-Range-Radar/ARS441`
- **Analysis scheme**: Partial

When analyzing the coffee maker, two catastrophic failures occurred that impeded memory read operations. The first disassembled board device was in a precarious state, and due to some unfortunate reasons, the board's power supply short-circuited with the metal plate, causing it to fail. It was possible to repair it through some reverse engineering work on that board, but unfortunately another explosion of two capacitors, due to a completely different reason, impeded the acquisition of data from that coffee maker board.

On the second unit, no disassembly was performed. A PIC Kit 4 Microchip Debugger was used to connect to an RJ-11 style port using a self-made adapter. An example of the pins of the debugging port is shown in Figure 5.5.

However, that debug was not completed because the microprocessor is hardware-locked, by means of a set pin. Therefore, at that point, it was decided that the coffee maker would not be used for the project.

Guitar Pedal

In some of the first stages of the project, a guitar pedal was used to verify the proof of concept. The advantage of using an electronic guitar pedal is the number of repetitive control loops it presents, according to the different presets and desired sounds. Note that this sound morphing is done with computer processing rather than analogous processes.

Although the initial decoding was promising, it was decided against because it was too simplistic and was not really useful for consideration as a connected embedded system or a safety-critical system.

Mitshubishi Radar System

Before analyzing the Continental ARS4-B system, a Mitsubishi-built radar system was acquired for possible analysis. However, once the device was physically in the laboratory and the microprocessors were analyzed, it was determined that it would be unwise to obtain the data from it.

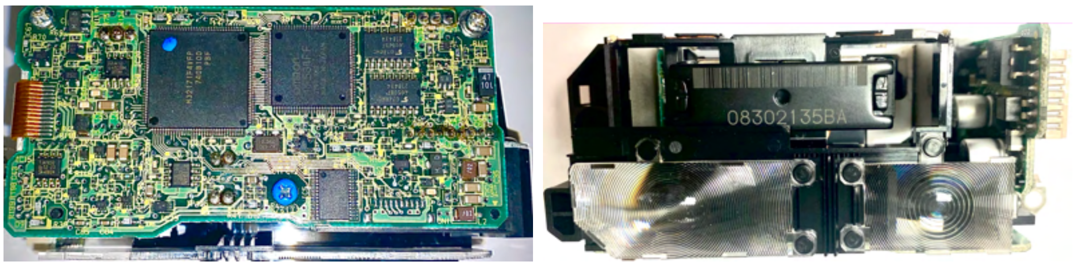


Figure 5.6: Mithubishi Automotive Radar (front & side)

Source: Author's work

Mitsubishi uses its own internal microprocessors and architecture. It presents a RISC based architecture with 84 instructions. At that stage of development, it was considered that a disassembly layer should be built for the specific device, but the director defended against it. It was not worth the progress on creating a new disassembler without knowing if the actual memory of the device could be read.

- **Device**: Mitsubishi Automotive Radar
- **Manufacturer**: Mitsubishi
- **Architecture**: RISC Based [32-bit]
- **Microcontroller/Microprocessor**: Mitsubishi M32171F4VFP
- **Non-volatile memory make and model**: Integrated in the microcontroller
- **Closed Source**
- **Analysis scheme**: Not completed.

At this point, it was decided to proceed with three devices, the drone, the PLC, and Continental/NXP's Automotive Radar system.

5.2 Used Techniques

The technique for analyzing binaries is similar in all three cases. However, there are specific concerns for each of the devices, due to their architecture, available information, and other factors.

5.2.1 General Technique and Methodology

The Drone and PLC were fully analyzed, consisting of a step-by-step analysis schema. However, the Automotive Radar has not been fully analyzed, as mentioned earlier, due to time constraints and difficulties. General techniques include the following steps:

1 **Initial binary loading**. The binary files were loaded into Ghidra for an initial assessment. Disassembly and decompilation were performed with the objective of obtaining the structure of the binary.

2 **Binary analysis and function ID**. The binary file is analyzed for function structures, memory structure, and main program execution.

3 **Depth limit setting**. It is at this point that the function to analyze is set and the decision is made about the depth level of the analysis. It is decided whether functions should be symbolically executed and substituted for their purpose, e.g. a floating-point division function substituted for the division operation on claripy, or whether the symbolic execution goes to lower levels, down to assembly code.

4 **Analyzer code modification**. The general analyzer code is then modified to adapt it to the specifics of the binary file, such as the entry point or certain execution patterns that need to be executed.

5 **Equation lifting**. Running the analyzer and acquisition of partial/complete lifted equations.

6 **Verification and validation**. Verification of the outputted files and checks that the files are executable and the equations are correct based on the initial expected behavior.

5.2.2 Specifics for the Drone Analysis

Analysis Steps

For the analysis of this device, the binary was first loaded in Ghidra. After an initial analysis of the binary file, problems with decompilation into pseudo-`C` were encountered.

After determining the function mapping in terms of register inputs and outputs using functions created by the author, it was then possible to correctly assess functions, function trees, and function calls using Ghidra, ghidra_bridge and Jupyter Notebook.

After completing the initial analysis, it was then possible to obtain expression maps corresponding to the inputs and outputs of `Madgwick Quaternion Update` by means of Python-made execution programs. However, unlike the complete classes that the PLC binary analyzer produces, it was necessary to minimally modify these documents to make them executable and obtain reasonable expression maps for the binary.

After the latest adjustments in terms of expression map execution, it was then possible to perform a reachability analysis on the function using DaDRA. For simplicity, the reachability analysis was performed only by changing one of the input values of the gyroscope.

Problems Encountered

`Madgwick Quaternion Update`, and for that matter, the whole drone binary file uses SEGGER's floating point operations library (emFloat).[2] SEGGER emFloat is a highly optimized library component for SEGGER's `C` Runtime Library. It provides highly efficient floating points operations following IEEE 754 rules, designed specifically for embedded systems. The advantages for using this library in the embedded system is its efficiency, fast operations and ability to compute in a small number of instructions floating point operations. However while analyzing `Madgwick Quaternion Update`, numerous problems developed from the use of this library.

The SEGGER library uses two distinct registers to represent a 32-bit value. Instead of using the architecture's full 32 bit registers, it uses 16 bits out of each register and operates on those reduced vectors. While in terms of efficiency, according to their data, the library is extremely efficient, the disassembler and decompiler have a hard time interpreting both input and output registers. This problem required that any call to a floating-point function be symbolically executed.

With this compromise, a decision was made that instead of lifting the equations down to the instruction set, they will be inferred down to the floating-point operation library. The advantage in this case of the arm architecture and the compiler used for the drone binary is that a lot of information was not lost on compilation in terms of function names and IDs. Therefore, Ghidra was able to easily identify the function names, and with that, while symbolically executing it with angr, it was a matter of simulating the floating-point operations.

Although this inconvenience is limiting in terms of the universality of the tool, it is an

[2] `https://www.segger.com/products/development-tools/runtime-library/technology/floating-point-library/`

advantage that the inference of equations can be done for different depth levels. In this matter, the improvements in function ID and the future database that is being developed for that purpose in the research laboratory at the University of Illinois will be useful in terms of future configuration parameters of the tool.

5.2.3 Specifics for the PLC Analysis

Analysis Steps

In this case, the binary analysis of the drone was similar to that of the drone. However, there were several key differences throughout the system. The file was first imported into Ghidra, but because the instructions in x86-32 are variable in size, Ghidra is unable to disassemble any function. It was necessary to use radare2 and find that if a forced decompilation was done on the last function of the binary, there is a call to the main function from that point. Forcing the decompilation with some non-default parameters in Ghidra, it was possible to obtain those values.

Once these were lifted, it was only necessary to port and abstract those memory values to angr/Python. Knowing the entry point of the main control loop was really useful for angr, as the state machine could easily be symbolically executed.

After the binaries were symbolically executed, the equations were lifted into Python.

Problems Encountered

In the PLC analysis, most of the problems came from the initial analysis. Due to the variable instruction size of x86, Ghidra presents difficulties in disassembling instructions. When the initial binary was loaded for the first time, Ghidra needed specific commands to correctly discover and decompile those functions.

It was decided that the analysis should start with the latest and only function that Ghidra could easily disassemble. In that function, subsequent calls to one function are made, and with the aid of radare2, it was possible to determine that decompilation of that function call, provided with the initialization function, by means of the binary structure.

Once that initialization function was performed, it was a matter of further disassembling in Ghidra to obtain the main control loop, in this case `PID_FIXCYCLE`. The information obtained in Ghidra was also corroborated with the decompiling function that the ICSREF tool presents. It includes an experimental identification of `PID_FIXCYCLE` that was able to determine the initial function `PID_FIXCYCLE` at `0x1624`.

With all of this, when trying to decompile the functions in radare2, two other problems

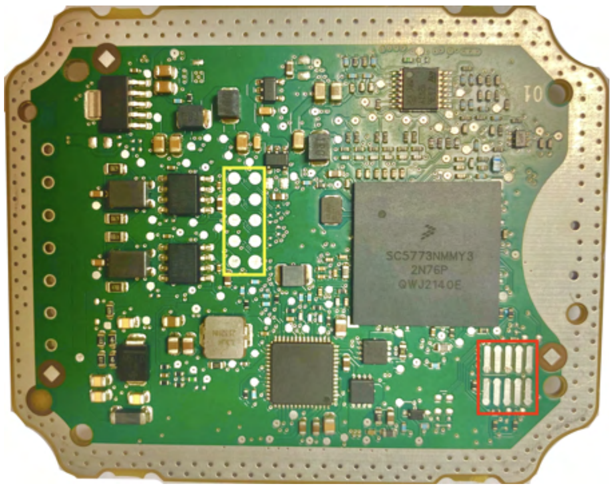


Figure 5.7: Main radar board with markings

Source: Author's work

arose that required the modification of several files and especially adjustments in the analyzer (`analyzer.py`). For some reason, the function calls to `PID_FIXCYCLE`, `DERIVATIVE`, and `INTEGRAL` did not have the correct memory address assigned to the call. It is unknown whether it is due to the decompiler or human error. The other problem came from the location of the last routine. Ghidra arguments that it does not exist, so it is necessary to force it to exist, and then it is possible to execute symbolically the binary.

5.2.4 Specifics for the Automotive Radar Analysis

In the last device, the team first tried to read the binary files directly from the device memory. The device presented what appeared like two debugging ports, although they were not readily accessible, as there were only traces on the radar's PCB. Both expected ports are shown in Figure 5.7. We expected the port encircled in red would be JTAG and the port encircled in yellow would be BDM. After a long week of analysis and trial and error, it was determined that no standard protocol was present in any of the debugging ports, as the traces and ground positions did not correspond to any known interface, whether it was JTAG, BDM, or SPI. The microcontroller included in the package supports the Freescale (NXP) BDM interface and contained what was possibly an interface, but the layout and pin connections were not satisfactory. In the end, it was decided not to spend more time on memory acquisition, as this project is more focused on the lifting of the bytecode, rather than the whole memory reading process.

With this change in plans, it was decided that for the analysis of automotive radars, it was necessary to use the complex licensing scheme for NXP's libraries. With that, NXP's RadarSDK will be used, and the MATLAB development environment, along with the Signal Processing toolboxes, will be the final test bench. Then it was necessary to change the analysis steps and technique.

The files are present in an example RSDK folder. They have sample data and output several MATLAB graphs with the calculated values, instead of actual CAN bus data. Therefore, after some exhaustive and manual analysis of binaries and data structure, it was determined that a full compilation must be made into `C` code using Matlab's Code Generator.

5.3 Research Results and Conclusions

After exhaustive research of the three devices, some conclusions were reached.

- **Manual analysis is required**. Except for basic RISC-based architectures, which keep function information and have "perfect" decompilations, some manual processing is always necessary. Whether it is selecting the entry point of the main function or choosing what depth level the analysis will reach, it is inevitable. Furthermore, depending on the depth level, complex symbolic execution of functions may be performed, which includes function ID and library simulation.
- **There is possibility of complex binary analysis**. Although computationally intensive, the tool is expected to perform adequately on larger binaries and when performing a full binary analysis. Some of the worries are that all functions are correctly mapped in case symbolic execution is performed at the function level, and that symbolic analysis gets exponentially more complex when different conditions are used. Therefore, some of the more complex programs that have multiple "branches" could easily overload execution and not work properly.
- **Verification schemes**. The verification schemes for this type of lifted equation are complex but necessary. At this point, some of the best examples of system verification are reachability analysis and graph plotting. However, there is a need to generate better tools for verification and correctness, in order to improve the credibility and safety of the tool.

Chapter 6

Developed Tool - InteGreat

In this chapter, we present InteGreat. This is the tool that has been developed for this thesis. It is developed in Python and uses other tools presented previously in Section 2. The structure, capabilities, characteristics, and finally its usage of components and some recommendations are described in great detail. The shortcomings and problems are explained in Section 7. An installation guide for all tools used is presented in Annex B.

6.1 Structure

The structure of the program follows a paradigm that was briefly explained in Section 5.2.1, but will be explained in greater detail.

In general, there are two main schemes of decompilation. There is a manual version, used by the drone binaries, and a more automatic version, used by the PLC. Decompilation and lifting follow these patterns, as shown in Figures 6.1 and Figure 6.2, respectively.

6.1.1 Common Steps between Workflows

While those two workflows are distinct in nature, they follow similar steps. In both, it is necessary to first perform an initial analysis in Ghidra, or radare2. This is necessary to

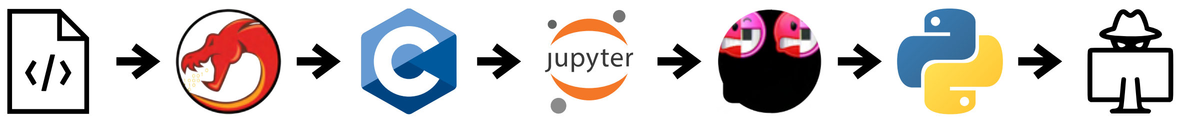


Figure 6.1: Manual workflow

Figure 6.2: Automatic workflow

find the main function, any control loop, or any other function of interest. As mentioned, there are limitations to symbolic execution and the number of unconstrained variables that could be handled; therefore, it is necessary to take into account the limitations. The necessity is for the identification and analysis of functions.

In the search for new looping functions, it would be preferable to look for the following.

- Recursive/looping functions that have arithmetical operations and depend on some inputs and outputs.
- Program entry points.
- Function calls.
- Function maps.

All these items can be searched with Ghidra; however, there are companion programs that were built specifically for this research that aid in looking up those characteristics, by means of ghidra_bridge and IPython, such as A.1.

After finding this information, both the automatic and manual flows diverge.

6.1.2 Manual Workflow

The manual workflow continues with the pseudo-C analysis presented by Ghidra. This workflow was used for drone analysis. Although not strictly necessary, some assumptions are made for its execution. The code to analyze should have been partially decoded by Ghidra, and there needs to be assurance that the code is partially disassembled and decompiled. In the case of the drone binaries, it was necessary that the decompilation be successful, as the execution of the project depended on the correct assumption on SEGGER's floating-point operation library.

Once the programmer is certain that the decompilation is relatively successful, the `analyzer.py` program is called. The inputs to that program are:

- **Firmware path**. The binary file

- **Function call file**. A file with the following structure `function_name` and `function_call_address` per each line. The function that calls the address is the one where `function_name` is called.

The advantage of that binary and function calling file is that there is a set of functions that develop this output. The code for this is presented in the Listing A.1. Note that in line 238 the function name that will be analyzed is chosen. Furthermore, because most of the representations and files that this program outputs are general, the analysis could be done as far up or down the hierarchy as necessary.

After running this program through IPython and ghidra_bridge, it is a matter of using the `drone_toplevel.py` executable. There are certain parameters that must be studied for the execution to be successful. These should be changed in the analyzer and they are as follows:

- `a.enter_function`. The function address that will be analyzed
- `a.init_sym_regs`. Fill in information about the registers used in the program.
- `unconstr_stack/a.init_sym_stack`. Choose the memory values for the unconstrained stack.
- `a.init_sym_memory`. System memory values initialization
- `a.init_time_field`. Non-zero time-filled value for the program to work
- `a.init_control_state`. Control state loops
- `a.set_concrete_stack`. Set a stack for the program to use
- `mem_locs`. Generate memory locks because the output of the function corresponds to certain memory positions.

After all these values are set, a file with the lifted differential equations is presented. With that Python file, it is possible to execute and obtain value implementations. Furthermore, for the drone, an internal analyzer was specifically built for the verification of the binaries.

The last step is to perform the reachability analysis using the analyzer files. Although little emphasis will be placed on reachability analysis at this stage of the project, some reachability results are shown in the next chapter.

A detailed example of this process is presented in Section 6.2, and examples of the code will be presented in Appendix A.

6.1.3 Automatic Workflow

The automatic workflow follows a different, yet similar path to that of the manual workflow. This workflow was tested with the PLC device and proved to be more consistent in terms of reaching the same result in each execution run. The advantage of the workflow is that no user-made analysis is needed except for basic function inferring, which is necessary in any device analysis. The analyzer `ASTA_analyzer.py`, developed by Maxwell Bland[1] in collaboration with the author, handles symbolic execution and equation lifting in one package. The analyzer works with three main parameters.

- **Firmware path**. The firmware path of the binary file
- **Program start address**. The initial program address.
- **Resolved function address**. The decompiled function address map
- **Hooks for 3 functions** that Ghidra does not decompile well (specific for the WAGO PLC binary).

When all these values are set, it is a matter of running the `plc_toplevel.py` file, with the values discovered in the last step. After that, an executable Python file with the lifter equations is given in around 5-10 minutes, once disassembly, decompilation, and symbolic execution are finished.

6.2 Usage and execution

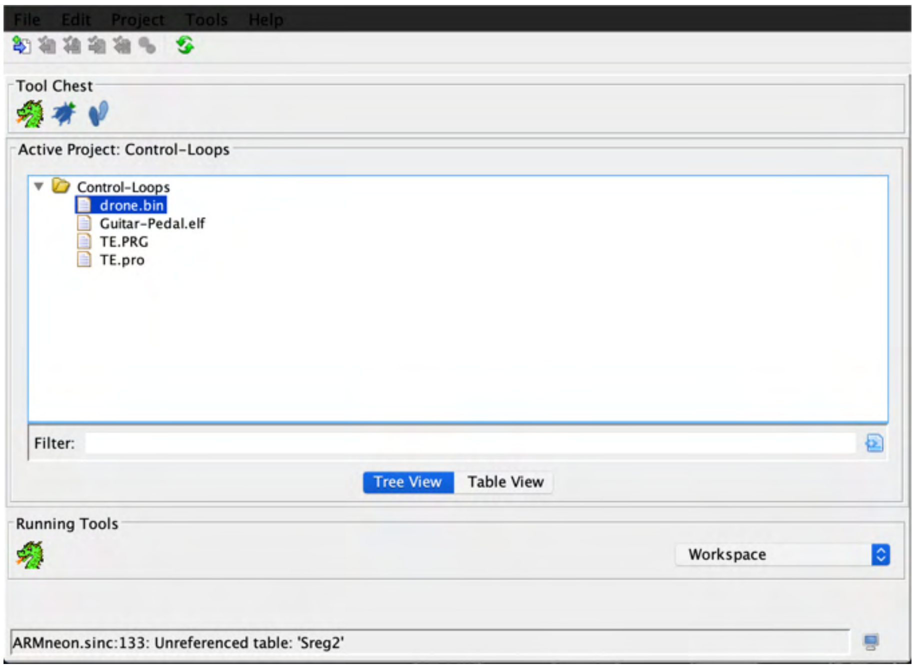
The usage of this tool is not as simple as first envisioned. Some binary analysis is required before a reasonable execution and differential equation lifting can be made. However, there is much to improve on in the final conference presentation.

In this section, the usage of the main tools is explained. The focus will be on the tool usage, because the execution of the tool is rather simplistic, as it is just executing Python scripts.

6.2.1 Virtual Machine Execution

It is encouraged and recommended to run InteGreat and the entire process on a virtual machine or a brand new installation, due to some compatibility errors that may exist. Although a graphical interface is preferred for Ghidra analysis, which greatly helps to

[1] https://bland.web.illinois.edu/



ARMneon.sinc:133: Unreferenced table: 'Sreg2'

Figure 6.3: Screenshot of Ghidra welcome screen

understand the binaries, it is also possible to disassemble the code on the command line with a worse user experience.

The virtual machine should run either a Linux distribution or Mac OS X. Windows is not supported at this time. It should also have a package manager and Python installed. However, an installation guide is provided in Appendix B.

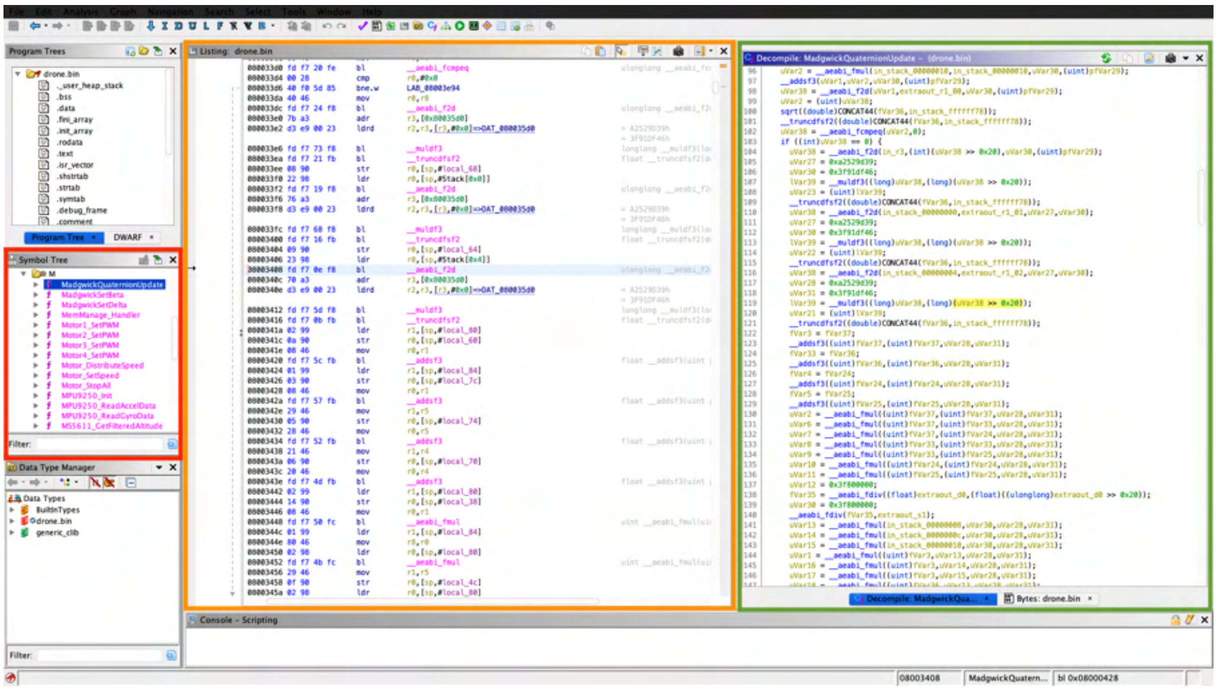
6.2.2 Ghidra

In order to analyze a binary file in Ghidra, follow these steps. In the example, the binary file `drone.bin` is loaded due to its ease and the advantage of using ARM as an architecture.

To load a binary file in Ghidra for initial analysis, the following steps should be taken:

1 **Open Ghidra and create a new project**. A new project is created in `File > New Project...` as shown in Figure 6.3. Import the binary files to the project and select both the Format, Language (ARM:LE:32:v8:default for the drone); and once imported double-click in the file.

2 **Analyze and decompile the file**. A pop-up window should appear asking you to auto-analyze the binary file. The default configuration should be fine. However, if that pop-up window is not available, go to `Analysis > Auto Analyze 'file'...`, press `Analyze` and wait for it to finish.



```
080033d0 fd f7 20 fe     bl        __aeabi_fcmpeq                        ulonglong __aeabi_fc
080033d4 00 28           cmp       r0,#0x0
080033d6 40 f0 5d 85     bne.w     LAB_08003e94
080033da 40 46           mov       r0,r8
080033dc fd f7 24 f8     bl        __aeabi_f2d                           ulonglong __aeabi_f2
080033e0 7b a3           adr       r3,[0x80035d0]
080033e2 d3 e9 00 23     ldrd      r2,r3,[r3,#0x0]=>DAT_080035d0         = A2529D39h
                                                                         = 3F91DF46h
080033e6 fd f7 73 f8     bl        __muldf3                              longlong __muldf3(lo
080033ea fd f7 21 fb     bl        __truncdfsf2                          float __truncdfsf2(d
080033ee 08 90           str       r0,[sp,#local_68]
080033f0 22 98           ldr       r0,[sp,#Stack[0x0]]
080033f2 fd f7 19 f8     bl        __aeabi_f2d                           ulonglong __aeabi_f2
080033f6 76 a3           adr       r3,[0x80035d0]
080033f8 d3 e9 00 23     ldrd      r2,r3,[r3,#0x0]=>DAT_080035d0         = A2529D39h
                                                                         = 3F91DF46h
080033fc fd f7 68 f8     bl        __muldf3                              longlong __muldf3(lo
08003400 fd f7 16 fb     bl        __truncdfsf2                          float __truncdfsf2(d
08003404 09 90           str       r0,[sp,#local_64]
08003406 23 98           ldr       r0,[sp,#Stack[0x4]]
08003408 fd f7 0e f8     bl        __aeabi_f2d                           ulonglong __aeabi_f2
0800340c 70 a3           adr       r3,[0x80035d0]
0800340e d3 e9 00 23     ldrd      r2,r3,[r3,#0x0]=>DAT_080035d0         = A2529D39h
                                                                         = 3F91DF46h
08003412 fd f7 5d f8     bl        __muldf3                              longlong __muldf3(lo
08003416 fd f7 0b fb     bl        __truncdfsf2                          float __truncdfsf2(d
0800341a 02 99           ldr       r1,[sp,#local_80]
0800341c 0a 90           str       r0,[sp,#local_60]
0800341e 08 46           mov       r0,r1
08003420 fd f7 5c fb     bl        __addsf3                              float __addsf3(uint
08003424 01 99           ldr       r1,[sp,#local_84]
08003426 03 90           str       r0,[sp,#local_7c]
08003428 08 46           mov       r0,r1
0800342a fd f7 57 fb     bl        __addsf3                              float __addsf3(uint
0800342e 29 46           mov       r1,r5
08003430 05 90           str       r0,[sp,#local_74]
08003432 28 46           mov       r0,r5
08003434 fd f7 52 fb     bl        __addsf3                              float __addsf3(uint
08003438 21 46           mov       r1,r4
0800343a 06 90           str       r0,[sp,#local_70]
0800343c 20 46           mov       r0,r4
0800343e fd f7 4d fb     bl        __addsf3                              float __addsf3(uint
08003442 02 99           ldr       r1,[sp,#local_80]
08003444 14 90           str       r0,[sp,#local_38]
08003446 08 46           mov       r0,r1
08003448 fd f7 50 fc     bl        __aeabi_fmul                          uint __aeabi_fmul(ui
0800344c 01 99           ldr       r1,[sp,#local_84]
0800344e 80 46           mov       r8,r0
08003450 02 98           ldr       r0,[sp,#local_80]
08003452 fd f7 4b fc     bl        __aeabi_fmul                          uint __aeabi_fmul(ui
08003456 29 46           mov       r1,r5
08003458 0f 90           str       r0,[sp,#local_4c]
0800345a 02 98           ldr       r0,[sp,#local_80]
```

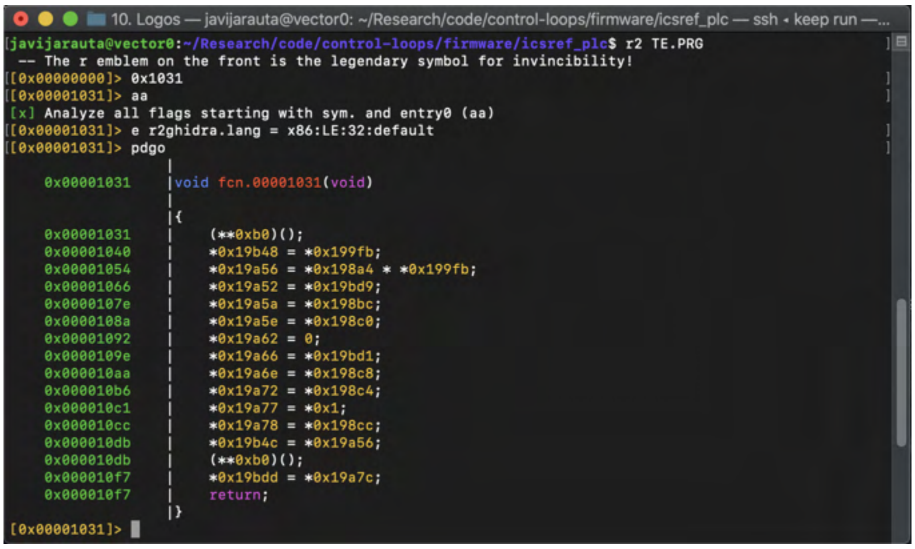
```
  uVar2 = __aeabi_fmul(in_stack_00000010,in_stack_00000010,uVar30,(uint)pfVar29);
  __addsf3(uVar1,uVar2,uVar30,(uint)pfVar29);
  uVar38 = __aeabi_f2d(uVar1,extraout_r1_00,uVar30,(uint)pfVar29);
  uVar2 = (uint)uVar38;
  sqrt((double)CONCAT44(fVar36,in_stack_ffffff78));
  __truncdfsf2((double)CONCAT44(fVar36,in_stack_ffffff78));
  uVar38 = __aeabi_fcmpeq(uVar2,0);
  if ((int)uVar38 == 0) {
    uVar38 = __aeabi_f2d(in_r3,(int)(uVar38 >> 0x20),uVar30,(uint)pfVar29);
    uVar27 = 0xa2529d39;
    uVar30 = 0x3f91df46;
    lVar39 = __muldf3((long)uVar38,(long)(uVar38 >> 0x20));
    uVar23 = (uint)lVar39;
    __truncdfsf2((double)CONCAT44(fVar36,in_stack_ffffff78));
    uVar38 = __aeabi_f2d(in_stack_00000000,extraout_r1_01,uVar27,uVar30);
    uVar27 = 0xa2529d39;
    uVar30 = 0x3f91df46;
    lVar39 = __muldf3((long)uVar38,(long)(uVar38 >> 0x20));
    uVar22 = (uint)lVar39;
    __truncdfsf2((double)CONCAT44(fVar36,in_stack_ffffff78));
    uVar38 = __aeabi_f2d(in_stack_00000004,extraout_r1_02,uVar27,uVar30);
    uVar28 = 0xa2529d39;
    uVar31 = 0x3f91df46;
    lVar39 = __muldf3((long)uVar38,(long)(uVar38 >> 0x20));
    uVar21 = (uint)lVar39;
    __truncdfsf2((double)CONCAT44(fVar36,in_stack_ffffff78));
    fVar3 = fVar37;
    __addsf3((uint)fVar37,(uint)fVar37,uVar28,uVar31);
    fVar33 = fVar36;
    __addsf3((uint)fVar36,(uint)fVar36,uVar28,uVar31);
    fVar4 = fVar24;
    __addsf3((uint)fVar24,(uint)fVar24,uVar28,uVar31);
    fVar5 = fVar25;
    __addsf3((uint)fVar25,(uint)fVar25,uVar28,uVar31);
    uVar2 = __aeabi_fmul((uint)fVar37,(uint)fVar37,uVar28,uVar31);
    uVar6 = __aeabi_fmul((uint)fVar37,(uint)fVar33,uVar28,uVar31);
    uVar7 = __aeabi_fmul((uint)fVar37,(uint)fVar24,uVar28,uVar31);
    uVar8 = __aeabi_fmul((uint)fVar33,(uint)fVar33,uVar28,uVar31);
    uVar9 = __aeabi_fmul((uint)fVar33,(uint)fVar25,uVar28,uVar31);
    uVar10 = __aeabi_fmul((uint)fVar24,(uint)fVar24,uVar28,uVar31);
    uVar11 = __aeabi_fmul((uint)fVar25,(uint)fVar25,uVar28,uVar31);
    uVar12 = 0x3f800000;
    fVar35 = __aeabi_fdiv((float)extraout_d0,(float)((ulonglong)extraout_d0 >> 0x20));
    uVar30 = 0x3f800000;
    __aeabi_fdiv(fVar35,extraout_s1);
    uVar13 = __aeabi_fmul(in_stack_00000008,uVar30,uVar28,uVar31);
    uVar14 = __aeabi_fmul(in_stack_0000000c,uVar30,uVar28,uVar31);
    uVar15 = __aeabi_fmul(in_stack_00000010,uVar30,uVar28,uVar31);
    uVar1 = __aeabi_fmul((uint)fVar3,uVar13,uVar28,uVar31);
    uVar16 = __aeabi_fmul((uint)fVar3,uVar14,uVar28,uVar31);
    uVar17 = __aeabi_fmul((uint)fVar3,uVar15,uVar28,uVar31);
```

Figure 6.4: Screenshot of the Ghidra main development screen

3 **Start exploring in the main Ghidra screen**. An example of a binary loaded into Ghidra is shown in Figure 6.4. There are three main sections in Ghidra that will be useful to the user. The area boxed in red shows all of the decompiled functions, ordered by name. The section surrounded by an orange box represents the disassembled code. There are multiple configurations and language abstractions that can be added to that view by selecting them on the `Listing` button. On the right, the section boxed in green shows the pseudo-`C` code decompiled by Ghidra. In that window, some of the main functionalities are to change the signatures of the functions, the variables, recompiling, and modifying the decompiled code.
A double click on either main screen will redirect you to the decompiled function or the lifted assembly instructions, and vice versa.

4 **Activate ghidra_bridge and analyze externally**. This last step, optional if radare2 is being used, consists of activating the script ghidra_bridge, previously installed, to be able to access the ghidra commands and the interface from an external programming languages. Note that the execution time will be slow for large binaries and consultations. To activate ghidra_bridge, you should first open `Window > Script Manager`, search for ghidra_bridge and select `ghidra_bridge_server _backgorund.py`. Once it is up and running, access is available through the Python console, Jupyter, etc.

Although there are specific actions that can be taken to correctly analyze a binary file, each one is independent, and there is no generalization for any analysis in particular. In this thesis, techniques such as last function searching, for the PLC binary, or function simulation were used in order to infer the parameters necessary for manual execution.



10. Logos — javijarauta@vector0: ~/Research/code/control-loops/firmware/icsref_plc — ssh ◂ keep run —...

```
javijarauta@vector0:~/Research/code/control-loops/firmware/icsref_plc$ r2 TE.PRG
 -- The r emblem on the front is the legendary symbol for invincibility!
[0x00000000]> 0x1031
[0x00001031]> aa
[x] Analyze all flags starting with sym. and entry0 (aa)
[0x00001031]> e r2ghidra.lang = x86:LE:32:default
[0x00001031]> pdgo
                  |
    0x00001031    |void fcn.00001031(void)
                  |
                  |{
    0x00001031    |    (**0xb0)();
    0x00001040    |    *0x19b48 = *0x199fb;
    0x00001054    |    *0x19a56 = *0x198a4 * *0x199fb;
    0x00001066    |    *0x19a52 = *0x19bd9;
    0x0000107e    |    *0x19a5a = *0x198bc;
    0x0000108a    |    *0x19a5e = *0x198c0;
    0x00001092    |    *0x19a62 = 0;
    0x0000109e    |    *0x19a66 = *0x19bd1;
    0x000010aa    |    *0x19a6e = *0x198c8;
    0x000010b6    |    *0x19a72 = *0x198c4;
    0x000010c1    |    *0x19a77 = *0x1;
    0x000010cc    |    *0x19a78 = *0x198cc;
    0x000010db    |    *0x19b4c = *0x19a56;
    0x000010db    |    (**0xb0)();
    0x000010f7    |    *0x19bdd = *0x19a7c;
    0x000010f7    |    return;
                  |}
[0x00001031]>
```

Figure 6.5: Screenshot of a radare2 decompilation example

For extended Ghidra learning, see "The Ghidra Book: The definitive guide"[2]. This book teaches the basics of Ghidra, up to high-level complex analysis, and has been a really useful tool for learning the techniques necessary.

6.2.3 radare2

The use of radare is easier than that of Ghidra. After all, it is a command-line tool that has impressive decompilation and disassembling capabilities, integrated with r2pipe and r2ghidra.

The most basic example was a simple decompilation. In this case, the example is the PLC's binary file (`TE.PRG`). As mentioned, due to the complexity of disassembly of x86 language instructions, radare2 is not able to identify functions as easily as with arm. However, a basic example of function decompilation is shown in Figure 6.5.

Some of the most important flags and commands that can be seen in Figure 6.5 are shown in Table 6.1.

Other commands and basic compilation can be found in their GitHub repositories[3] [4]

The use of radare2, r2ghidra, and r2pipe allowed for headless execution of the binaries. These tools were used mainly for function recognition. As mentioned, in the PLC binary, there were quite a few problems in terms of getting Ghidra to disassemble and decompile the binaries. Therefore, it is necessary to generate and create new checking functions in

[2] `https://www.ghidrabook.com/`
[3] `https://github.com/radareorg/radare2/blob/master/doc/intro.md`
[4] `https://github.com/radareorg/r2ghidra`

Command	Category	Description
`0xNNNNNN`	Basic	Go to address 0xNNNNNN
`e`	Basic	Show all program variables and settings
`aa`	Analyze	Analyze all, just like Ghidra, analyze the binary
`afl`	Analyze	List all functions
`e r2ghidra.lang = x86...`	r2ghidra	Assign a language to the ghidra decompiler
`pdgo`	r2ghidra	Decompile current function side by side with offsets

Table 6.1: Basic commands for radare2 and r2ghidra

the `analyzer.py`, that make use of these two libraries, to first, obtain function calls, and secondly, make the function calls point to the correct address.

6.2.4 angr

In this project, angr was used to symbolically execute the binary files, and obtain the lifted equations. Angr's use is just like that of radare2. It is a commnand-line tool that presents a myriad of tools for symbolic execution. Both angr and claripy need to be imported into the project for both to work.

To perform symbolic execution, there are several main steps, but this code example is basic to show how symbolic execution is performed. It is worth noting that variable, memory, register, and stack symbols should be added through claripy if they are known beforehand.

```

#!/usr/bin/env python3
import angr #the main framework
import claripy #the solver engine

self.project = angr.project.Project(firmware,     # Load the project
    firmware
                                    load_options={
                                        'main_opts': {
                                        'backend': 'blob',
                                        'arch': 'i386',
                                        'base_addr': 0,
                                        'entry_point': addr
                                        },
                                    'auto_load_libs': False
                                    })
self.entry_state = self.project.factory.full_init_state()    # Determine the
     entry state for the project
self.sm = self.project.factory.simulation_manager(self.entry_state) #
    Create a new simulation manager with its entry state
```

```
self.hook = Hook()        # Hook function to symbolically execute SEGGER's
    functions
self.init_hooks(func_call_list_file)        # Hook function list

p.hook(0xdd2, nop, length=6)
p.hook(0x1031, lambda s: call_fixup(s, 0x1031), length=6)
p.hook(0x10db, lambda s: call_fixup(s, 0x10db), length=6)

s = p.factory.entry_state()
sm = p.factory.simulation_manager(s)
run(sm)


# Run until the program finds all unconstrained variables
def run(sm):
    while sm.active:
        print(sm.active)
        sm.step()
```

Listing 6.1: Example code for angr symbolic execution
Courtesy of angr, Maxwell Bland

There needs to be more companion functions to perform symbolic execution. However, with this, certain analysis and lifting can be performed, especially to familiarize yourself with the environment.

6.3 Characteristics

Although the tool presents many capabilities, after extensive research and studies, here are some of the main characteristics of the tool.

- **Versatility and multi-architecture**. With adequate disassemblers and correct function information, the expansion of the tool and the research is limitless. As long as there are frameworks for decompiling in different architectures, it is possible to raise the control equations at least to a higher-level code.
- **Scalability**. Disassembly and decompilation is not as computationally intensive as it used to be, but some modern and complex decompilation schemes may use extended periods of time for large binaries. The same phenomenon occurs while symbolically executing the binaries. However, this tool has shown that, with enough time, the robustness of the tool is unparalleled.
- **Novelty**. Although a lot of previous work has been done on decompiling techniques, in this case, it was possible to obtain actual executable files that have the same behavior as the embedded device that was lifted. This ability is unparalleled and generates a new paradigm of system verification and simulation.

6.4 Limitations

The tool, like any complex system, has several shortcomings and limitations, which will be improved upon in the final product.

First of all, the version presented in this thesis is limited by its capabilities. It has been adjusted to lift the equations of both the drone and the PLC. However, as mentioned in previous chapters, the aim is to expand on the tool and be able to analyze more architectures, and to formally present with a multi-architecture device.

Moreover, a lot of manual analysis is needed for the verification of the equations. After all, the loop functions are chosen manually, and the depth level is also chosen in the same way. Therefore, when the complete tool is done, it will have some kind of mechanism to recognize, depending on the device and function, these kinds of functions. Loop recognition could be aided by developments in function ID schemes.

Another limitation is that the analyzer has to be made specifically for the binary, or at least partially. While the aim is to be more universal, the constraints in the binaries did not allow for that. The only functions that can be used for both, when or not a successful decompilation has been performed, are the function helper tools (Listing A.1). These tools, which include function trees and maps, function calls, and function information, can be performed if the decompilation was successful for the most part. In this case, that would be decompilation for RISC-based systems which present instructions of the same size.

Finally, the tool is limited by its performance. At this point, some of the symbolic execution takes some time; however, if the level of analysis is increased to a higher function, the limitations of symbolic execution could start to appear at some point, especially if a large number of functions and variables have to be hooked.

Although there are limitations and constraints, in Chapter 7, there is a more detailed explanation of performance and execution.

Chapter 7

Results

This chapter provides an explanation of the main results of this research thesis. The focus is on the accuracy and development of the actual verification and output of the files.

7.1 Initial Results

The results were overall very satisfactory. Even with difficulties early on in terms of control equation lifting, further analysis demonstrated that the inferring was adequate and that it is possible, in fact, to lift control equations from cyber-physical systems up to a high-level programming language.

7.1.1 Initial Function Analysis

When using the first initial binary analysis, using the function helper, it was possible to correctly assess all function calls and the function tree. This data were then either used directly (for the Drone analysis), or referenced indirectly in the program constraints (for the PLC analysis). An example of the partial function call list for the Drone is shown in Listing 7.1.

```
__muldf3       0x080033e6
__muldf3       0x080033fc
__muldf3       0x08003412
FUN_080006ac       0x080004ec
__aeabi_fcmpeq   0x0800338a
__aeabi_fcmpeq   0x080033d0
__aeabi_cfcmpeq  0x08001018
__cmpsf2       0x08001006
__divdf3       0x08003d36
FUN_08000892       0x08000740
```

```
__aeabi_f2d 0x0800337a
__aeabi_f2d 0x080033c0
__truncdfsf2     0x08003382
__truncdfsf2     0x080033c8
asin     0x08003d24
__errno 0x080058bc
__errno 0x080058d4
__ieee754_asin   0x0800585e
__muldf3     0x08005c9c
__muldf3     0x08005cae
```

Listing 7.1: Extract of `func_call_addr.txt` for the Drone binary

As is shown, the binary file consists of the function name and all of the addresses where that function is called. In the case of the drone binary analysis, it was really useful because for the drone symbolic execution to work, it was necessary to hook or symbolically change all floating-point operation functions (`__aeabi`) to its corresponding operation. That is, instead of executing `__aeabi_fmul` in the symbolic execution, it was set that the claripy operation “floating point" multiplication should be executed.

Apart from the output of the basic function helper `func_call_addr.txt`, there are other useful files for rapid analysis. For example, some other excerpts from the function helper are shown in Listing 7.2 (Function return registers) and Listing 7.3 (Function input registers). Those files show, respectively, the function return register, that is, when a function is done, the value that will be returned to its superior routine. And in the case of 7.3, the input registers for the function are shown.

Although the results obtained in both files are correct according to the Ghidra decompiler and disassembler, there are certain errors that were known to the author in that decompilation. Due to the use of the SEGGER floating-point library, it is known that some of the input registers are combined, which means that, for example, in `__aeabi_f2d`, "Extend float to double", registers `r0`, `r1`, `r2`, `r3` are a single number which should be combined into one register, `r0` [52]. This refactoring, although not reflected in this example, was later done through the execution by means of `json` structures.

```
__aeabi_dadd     r1
__floatunsidf    r1
__aeabi_i2d  r1
__aeabi_f2d  r1
__floatundidf    r1
__aeabi_l2d  r1
__muldf3     r1
FUN_080006ac     r1
__divdf3     r1
FUN_08000892     r1
__gedf2  r0
__ledf2  r0
__nedf2  r0
__aeabi_cdrcmple     r0
__aeabi_cdcmpeq  r0
__aeabi_dcmpeq   r0
```

```
__aeabi_dcmplt   r0
__aeabi_dcmple   r0
__aeabi_dcmpge   r0
__aeabi_dcmpgt   r0
```

Listing 7.2: Extract of `func_return.txt`

```
__aeabi_dadd
__floatunsidf    r0
__aeabi_i2d  r0
__aeabi_f2d  r0    r1    r2    r3
__floatundidf    r0    r1
__aeabi_l2d  r0    r1
__muldf3     r0    r1
FUN_080006ac     r0    r1    r2    r3
__divdf3     r0    r1
FUN_08000892     r0    r1    r2    r3
__gedf2  r0    r1    r2    r3
__ledf2  r0    r1    r2    r3
__nedf2  r0    r1    r2    r3
__aeabi_cdrcmple     r0    r1    r2    r3
__aeabi_cdcmpeq  r0    r1    r2    r3
__aeabi_dcmpeq   r0    r1    r2    r3
__aeabi_dcmplt
__aeabi_dcmple   r0    r1    r2    r3
__aeabi_dcmpge   r0    r1    r2    r3
__aeabi_dcmpgt
```

Listing 7.3: Extract of `func_param.txt`

Overall, the output of this helper is very positive and is integrated into the analyzer code that provides extended equation lifting capabilities.

7.1.2 Lifted Control Equations

In this section, examples of the PLC binary are shown. The advantage of analyzing in this section the PLC is that the original programming code is available, and it is useful to show the whole process, from binary analysis to equation lifting.

First, an excerpt of the WAGO file `TE.pro` that is compiled into the PLC is shown in Listing 7.4

```
(* Control Loops *)
(*IF( Trigger_Handle.Q OR AP_plc_reset) THEN*)
    Trigger_Count    := Trigger_Count + 1;
    Reset_Count          := Reset_Count + BOOL_TO_REAL(AP_plc_reset);

    (* Pressure PI Loop *)
    Pressure_Loop(
        ACTUAL           :=  SIM_xmeas07,
        SET_POINT        :=  Pressure_SetPoint,
        KP                      :=   Pressure_KP,
        TN                      :=   Pressure_KI,
        TV                      :=   0.0,(*No Derivative*)
        Y_MANUAL           :=   Pressure_Manual,
        Y_OFFSET           :=        ,
        Y_MIN                   :=        Pressure_Output_Min,
        Y_MAX                   :=        Pressure_Output_Max,
        MANUAL             :=   ,
        RESET              :=        AP_plc_reset,
        CYCLE              :=   Cycle_Time,
        Y                       =>   Pressure_Output,
        LIMITS_ACTIVE      =>  ,
        OVERFLOW           =>
    );

    Pressure_Output_Fp := Pressure_Output*SIM_Fp_Mult;

    (* Flow-rate PI Loop *)
    Purge_Loop(
        ACTUAL             :=   SIM_xmeas10,
        SET_POINT          :=   Pressure_Output_Fp,
        KP                      :=   Purge_KP,
        TN                      :=   Purge_KI,
        TV                      :=   0.0,(*No Derivative*)
        Y_MANUAL           :=   Purge_Manual,
        Y_OFFSET           :=   ,
        Y_MIN                   :=        Purge_Output_Min,
        Y_MAX                   :=        Purge_Output_Max,
        MANUAL             :=   ,
        RESET              :=   AP_plc_reset,
        CYCLE              :=   Cycle_Time,
        Y                       =>   Purge_Output,
        LIMITS_ACTIVE      =>  ,
        OVERFLOW           =>
    );
```

Listing 7.4: Extract of `TE.pro`, a WAGO programming file

In that WAGO code file, it is possible to observe two control loops. `Purge_loop`, which will not be of interest for our program, as it is an "error" loop, and `Pressure_loop`, which is the `PID_FIXCYCLE` loop analyzed later. This `Pressure_loop` has other code, adjusting the `derivative` and `integral` functions, but it is not shown because its representation is not standard UTF-8.

Once that file is compiled using the WAGO/CODESYS compiler, then it is loaded into the PLC, as `TE.PRG`. That file is the one that is loaded into Ghidra/radare2 which the analysis is done on.

Once the program is fully executed analyzing the `TE.PRG` file, there is an output with the lifted control equations, as seen in the Listing 7.5. corresponding to the PLC analysis. The complete file is present in the Listing A.3. Note that the output of the actual program is all functions, and the comments were added later on for simplicity and analysis.

```
def generate_FP_integral_out_0_27_32(FP_integral_out_0_27_32):
    assignment_1_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19aad_6_16
    assignment_2_0 = assignment_1_0 + -0.0046
    assignment_3_0 = assignment_2_0 / 0.9876000000000001
    assignment_4_0 = 1000.0 * assignment_3_0
    assignment_5_0 = assignment_4_0 + 2000.0
    assignment_6_0 = mem_198a8_14_32 - assignment_5_0
    assignment_7_0 = mem_198cc_20_32 * 1000.0
    assignment_8_0 = assignment_6_0 * assignment_7_0
    assignment_9_0 = assignment_8_0 / 1000.0
    assignment_10_0 = FP_integral_out_0_27_32 + assignment_9_0
    FP_integral_out_0_27_32 = assignment_10_0

def generate_FP_pid_fixcycle_y_0_21_32(FP_pid_fixcycle_y_0_21_32,
   FP_derivative_t1_0_26_32, FP_derivative_x2_0_23_32,
   FP_integral_out_0_27_32, FP_derivative_t2_0_25_32,
   FP_derivative_x1_0_22_32, FP_derivative_x3_0_24_32):
    assignment_1_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_2_0 = assignment_1_0 + -0.0046
    assignment_3_0 = assignment_2_0 / 0.9876000000000001
    assignment_4_0 = 1000.0 * assignment_3_0
    assignment_5_0 = assignment_4_0 + 2000.0
    assignment_6_0 = mem_198a8_14_32 - assignment_5_0
    [...]
    assignment_46_0 = assignment_45_0 + 2000.0
    assignment_47_0 = mem_198a8_14_32 - assignment_46_0
    assignment_48_0 = mem_198ac_15_32 * assignment_47_0
    assignment_49_0 = assignment_41_0 - assignment_48_0
    assignment_50_0 = assignment_40_0 + assignment_49_0
    FP_pid_fixcycle_y_0_21_32 = assignment_50_0
```

Listing 7.5: Result of the Drone differential equation lifting

First of all, upon analyzing the output file, two characteristics stand out. InteGreat is capable of lifting individual functions. As explained in Section 2.3.2, a PID PLC is composed of a function `derivative`, `integral` and `PID`. The code was able to correctly lift the three functions, with inputs from memory, that correspond to the inputs shown on the `TE.pro` file in Listing 7.4. The program presents the expected behavior.

The variables named `assignment_x_0` correspond to the symbolically executed variables of the binary. That is, each assignment variable is a register that is used for an assembly operation such as `mul`. This is what is expected from the actual disassembly and decompiling functions of the tool.

As can be seen, the tool is also able to generate multiple `FP_pid_fixcycle` functions depending on the `if` and looping conditions that the original disassembled function had. If the original program had several routines that depended on conditions, such as the input pressure (in or out of bounds), or depending on whether some input variables are set to `true` or `false`, the output file will have different `FP_pid_fixcycle` functions. The output of InteGreat is readily executable from Python, even if some minor modifications have to be made to the file.

The other advantage of this output, as mentioned, is that it also specifies if the variable being simulated is a memory value with `mem_`, an internal variable with `assignment_`, or if it comes from another output of a function, in which case it will keep its name for similarities. This allows for a faster interpretation of the values in hand, and makes comparing the lifted code against the disassembled code much easier.

In another example, for the drone binaries, the output is very similar to that seen in the PLC. The result can be found in Listing A.4. In this case, an exhaustive analysis was performed using simulation and reachability analysis.

Overall, the advantage of this methodology is the ability to execute and integrate the functions and lifted equations into any verifier, to check for the correctness and expected behavior.

7.2 Verification and Correctness

To verify the correctness of the lifted control equations, two distinct processes were evaluated, depending on the device and target.

7.2.1 Drone

For the drone analysis, it was determined that it would be adequate to simulate the same behavior and parameters between the lifted equations and the existing code. A test setup was set to verify the claims and prove the accuracy of the model.

First, it should be noted that a preventive verification analysis was performed for both the `Madgwick filter` implementation and the drone implementation. There is some discrepancy between the filter theory and the actual implementation of C in the drone, as checked by the code. However, the behavior is really similar, but, as will be explained, some slight differences are expected.

Then it was decided to plot the quaternion update function. The input of that function is stable; that is, the drone is in a slow moving position, and the sensor values do not have any bias. In that case, when the inputs of the inertial sensors are stable and only the gyroscope value is changed, as could be possible in a real environment, a deviation on the quaternion update is seen in the figures below. Both the original simulation (Figure 7.1) and the simulation of lifted equations (Figure 7.2) are represented.

As can be seen, the graphs shown are "significantly different." However, as mentioned

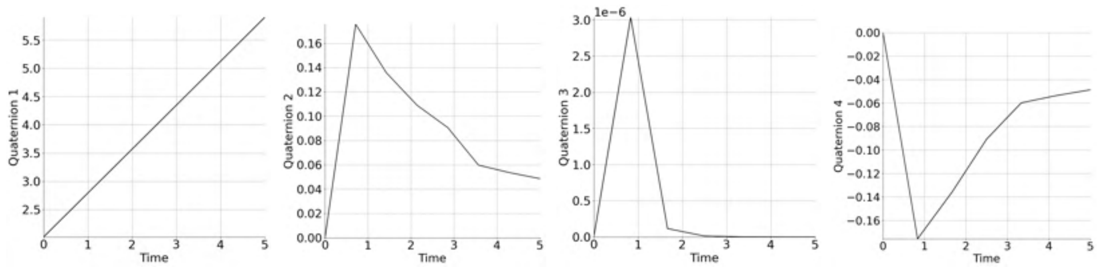


Figure 7.1: Quaternion updates from the original binaries

Source: Author's work and Maxwell Bland

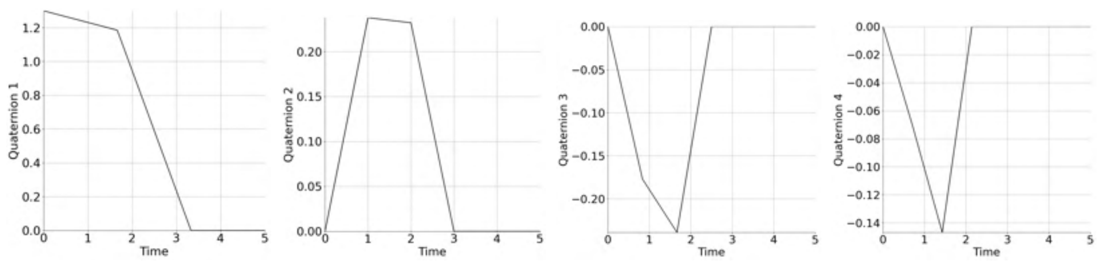


Figure 7.2: Quaternion updates from the lifted equations

Source: Author's work and Maxwell Bland

before, the real drone implementation is different from that of the theoretical implementation and, therefore, these deviations are expected. This is due to the fact that the drone starts at a different initial state than the model, so there is a discrepancy between both.

Except for the first quaternion, which will be studied in future iterations of the tool and its behavior is not clear, the representation of the other three quaternions is within the expected error bounds. Note that the third quaternion is inverted, but that is due to the mounting position of the sensor.

Overall, this interpretation and simulation between the lifted and real equations is satisfactory and demonstrates how the technique could be applied for further analysis.

7.2.2 PLC

For the verification of the PLC, it was decided to simulate the ICSREF attack [27]. For that matter, the objective is to reduce the pressure of the chemical process from around 2800 kPa to 2400 kPa. As explained in the ICSREF paper, the change in pressure from those values would not trigger an alarm in the control system, but it would drastically decrease the efficiency of the process.

The attack graph is in Figure 7.3. The actual attack presents a smoother graph. That behavior is expected because the simulation is done in real life with a physical device.

The graph clearly shows a drop in pressure from the desired value. Following the analysis,

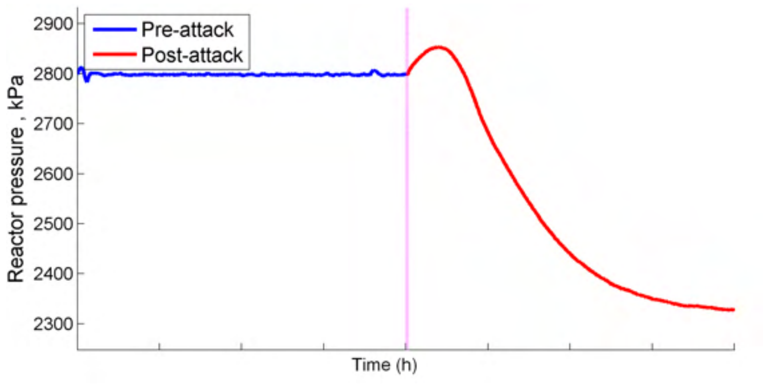


Figure 7.3: ICSREF's attack, reactor pressure is shown in kPa

Source: Courtesy of Anastasis Keliris, Michail Maniatakos at [27]

the pressure graph of the lifted equations is shown in Figure 7.4. The results are similar to those of the ICSREF paper, but due to the nature of the simulation, there are some differences.

First of all, the drop is more pronounced. That behavior is expected partially because, by simulating a discrete system, rather than a continuous system like the original PLC, the time steps are variable in terms of the equations. This tool was unable to correctly simulate the time steps set in the `PID_FIXCYCLE` function; however, its improvement is part of future work for PLC analysis. However, the objective of lowering the pressure from 2800 kPa to 2400 kPa is achieved completely. The drop is actually achieved to 2450 kPa, which is slightly lower than expected, but within the limits.

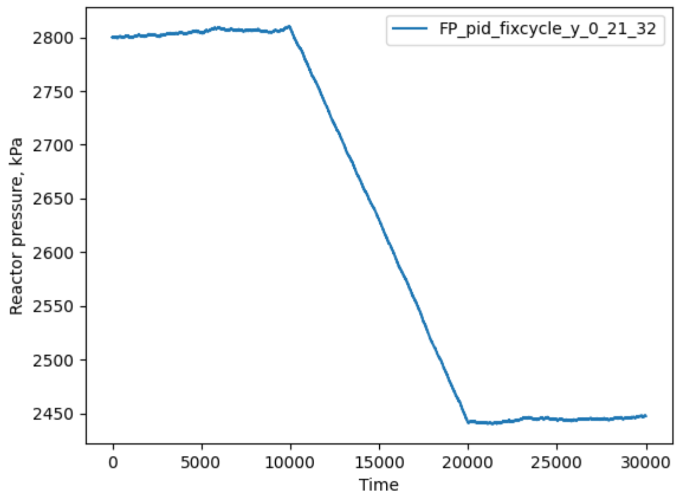


Figure 7.4: ICSREF's attack simulated with the lifted equations

Overall, the behavior is satisfactory in terms of verifying that the technique is complete and represents the real expected behavior. As mentioned in the Drone analysis, future iterations of the tool will take into consideration more parameters for correct time-step simulation and continuous system simulation.

With this two examples, the verification is completed. Although it is not 100% satisfactory, it is well within the bounds and expected behavior and is considered successful.

7.3 Reachability

Finally, in terms of reachability analysis, the tool built by Jared Mejia, DaDRA, has been really useful for that simulation. Reachability analysis was performed on the lifted drone equations, as the PLC has limited input variables and its expected output is very large.

However, due to the nature of the drone simulation and the limitations of the tool, it was not possible to fully plot all the necessary variables. An example of reachability analysis is shown in Figure 7.5.

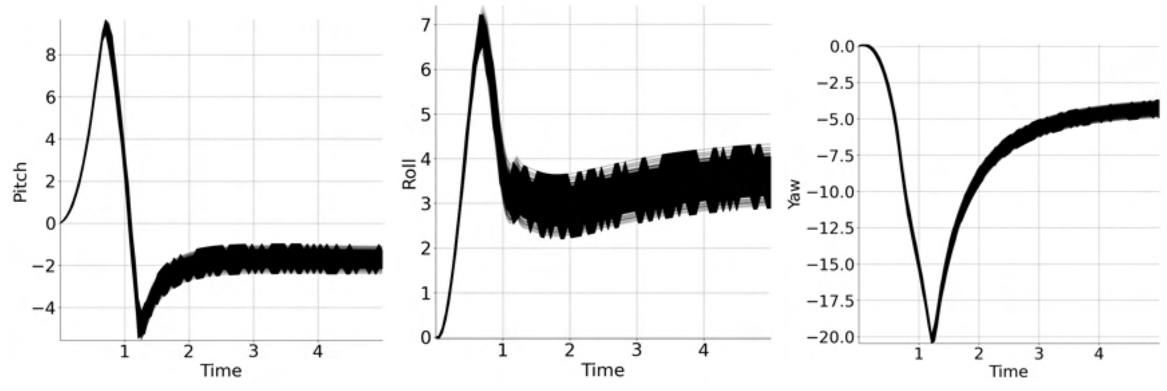


Figure 7.5: Reachability analysis with DaDra

Source: Author's work and Maxwell Bland

Analyzing the graphs, we see a correct and expected output, as this reachability analysis has better results than the simulations explained above. As observed, the `Madgwick filter` performs as expected in its lifted version, by dampening and finally eliminating bias. For this reachability analysis, a bias value of `0.1º` was added to the gyroscope. For the first second/second and a half, the behavior of the drone is unexpected, and if left with a non-filtered function, it could reach an instability point. However, as the behavior is anticipated, `Madgwick filters` out that bias and keeps the drone in a stable position.

Once that drone is in a stable position, the alternating line seen in the reachability graphs is only a few degrees, which represents that the function has reached the limits of the resolution of the sensor and filtering capabilities. That behavior in a physical device, with varying inputs, is expected. At this point, the actual utility of the tool is shown. InteGreat was able to demonstrate that, even if the implementation of the filtering function present in the drone is different to that of the theory, its functioning is as anticipated and correctly filters bias and errors.

Overall, the reachability analysis was really successful. Even if it is not the main focus of the paper, it is a way to show the usability of this tool and how it could analyze the software for embedded systems after being implemented in the physical device.

Chapter 8

Conclusions and Future Developments

In this chapter, the most important conclusions of the development of this thesis are presented, analyzing the initial objectives and results. Moreover, some future work will be explained for the further development of this tool, with the objective of presenting it at a conference or workshop, as part of the work performed at the University of Illinois.

8.1 Conclusions

The general objectives of the project set forth at the beginning of the year were reached for the most part; that is, to make a multi-architecture, multi-platform tool that is capable of analyzing binaries and lifting them to a high-level programming language. In terms of the goals set in Section 4.2, they were also achieved at a great level.

1 **Binary analysis and function identification**. This is arguably the goal that was not reached to the expected level. While the tool disassembles and decompiles adequately to obtain control differential equations, it is not able to perform the initial analysis for looping functions on its own. Furthermore, some of the manual analysis, although it was known that it was necessary, exceeds the extent that was originally envisioned. This analysis included function entry points, hooks, and the main program entry. However, in general, this objective could be considered complete around 50% of the time, because most problems arise from variable-size instruction architectures, such as x86.

2 **Symbolic execution and equation lifting**. This objective is achieved around 80%. The binary files are executed symbolically, and the equations are lifted to great precision. It is possible to ensure that the correctness is maintained most of the time per the verification schemes performed and that the results are what is expected from the devices. However, just like in the previous objective, there is some manual

component that was not originally intended, because there is a necessary set-up of function entry points and other parameters for a correct compilation of the lifted equations. In this case, depth-level analysis of the functions is also included as an improvement on the original goals and exceeds what was needed.

3 **Correctness of equation lifting**. The correctness of the equations lifted is completed, and except in error situations of the tool, it is acceptable and guaranteed. This point is completed to 95%.

4 **State reachability analysis**. Although this section was only done for the drone, the gradual improvement of the design of DaDRA and other reachability tools will allow further and more extensive reachability analysis. Furthermore, this section could at some point be integrated into the final tool. However, the completion level is around 75% due to the success of correctly demonstrating that the drone's implementation of `Madgwick` works as expected and that the tool is useful for the analysis of cyber-physical systems at a high level.

In general, the general objectives have been achieved to a great extent. However, some complex tasks, such as automation and standalone analysis, are not yet readily available due to technology and time constraints. Nevertheless, this improvement is part of the future development and improvement of InteGreat.

8.2 Future Work

As explained, there are three main sections where the tool could be greatly improved in future versions.

- **Binary analysis automation**. Using `.json` files to determine the specific characteristics of each architecture decompilation, it is possible to generate and have a different binary decompilation scheme for each architecture/embedded system device. A future goal could be to provide more support for other languages, without having to manually select entry points and decompile the functions in radare2 or Ghidra. Furthermore, advancements in function ID techniques may allow automatic binary recognition and, especially, floating-point operation library identification, just like the case with SEGGER's floating-point operation libraries. With this automatic binary identification, it is then possible to add an automatic abstraction level depending on the preferences of the user.

- **Automatic verification**. Apart from general binary automation, a further improvement could be an automatic verifier that uses both real execution and the lifted equation execution to determine if the lifting has been done correctly. With this tool, there could be an easy way for verification and assurance; and a correctness report could be generated for each binary decompilation.

- **Abstraction to Domain Specific Languages**. Just as lifting is done to a high-level programming language like Python, it could be possible, using `.json` files, to lift the equation to a DSL. In this case, the versatility that this tool should provide developers and testing personnel could increase greatly and become widely adopted.
- **Vulnerability searches**. By means of vulnerable function searches and reachability analysis, it could be easy to determine unstable states in which the system would not perform as expected. This vulnerability and stability analysis may be really useful in safety-critical devices and embedded systems. As the number of these grows year over year, it is possible that instead of only verifying the code, tests could be done through control equation lifting.
- **Cyber-security vulnerability searches**. By simulating known vulnerable devices and saving the behavior to a file, exploits and equation patterns could be compared to find vulnerable devices even before having performed a detailed forensic analysis.

There are several other improvements, such as efficiency, ease of use, and user interface. However, it is preferable to have a strong verifiable tool rather than a fancy one with less functionality. Moreover, those improvements are integrated into the normal tool workflow and development.

Overall, these improvements should improve the system toward a more functional and robust solution, closer to reality and the expected usage of the tool.

Bibliography

[1] Christian Buck and Chris Winkler. *The IoT story*. en. publisher: Siemens Research. Jan. 2020. URL: https://new.siemens.com/global/en/company/stories/research-technologies/digitaltwin/iot-story.html (visited on 05/29/2022).

[2] INCIBE. *Introduction to Embedded Systems*. Aug. 2018. URL: https://www.incibe-cert.es/en/blog/introduction-embedded-systems.

[3] Moore. *Cyber-Physical Systems Must be Part of Your Security Strategy*. Apr. 13, 2021. URL: https://www.gartner.com/smarterwithgartner/develop-a-security-strategy-for-cyber-physical-systems (visited on 06/05/2022).

[4] Sahar Bukhari and Muhammad Hasan Islam. "Security of Embedded Systems Using "ISO 27002" Standards". In: *International Journal of Scientific & Engineering Research* 7.12 (Dec. 2016). ISSN: 2229-5518. URL: https://www.ijser.org/researchpaper/Security-of-Embedded-Systems-Using-ISO-27002-Standards.pdf.

[5] Radhakisan Baheti and Helen Gill. "Cyber-physical systems". In: *The impact of control technology* 12.1 (2011), pp. 161–166.

[6] Manuel. Jiménez, Rogelio. Palomera, and Isidoro. Couvertier. *Introduction to Embedded Systems Using Microcontrollers and the MSP430*. 1st ed. 2014. Springer New York, 2014. ISBN: 1-4614-3143-3. DOI: 10.1007/978-1-4614-3143-5.

[7] A Aminifar et al. "Stability-aware analysis and design of embedded control systems". In: 2013, pp. 1–10. DOI: 10.1109/EMSOFT.2013.6658601.

[8] Michael Tortorella. *Reliability, maintainability, and supportability : best practices for systems engineers*. Includes bibliographical references and index. John Wiley & Sons Inc., 2015. ISBN: 1-119-05882-1.

[9] Jiacun Wang. *Real-time embedded systems*. Includes bibliographical references and index. Wiley, 2017. ISBN: 9781119420705.

[10] Clemens. Holzmann. *Spatial Awareness of Autonomous Embedded Systems*. eng. 1st ed. 2009. Wiesbaden: Vieweg+Teubner Verlag, 2009. ISBN: 1-283-17243-7.

[11] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. Second Edition. Accession Number: MR3616878; Authors: 675768; 739666 Author Affiliation: 1-CA-C; Department of Electrical Engineering and Computer Sciences (EECS), University of California; 1-CA-C; Department of Electrical Engineering and Computer Sciences (EECS),

University of California. Copyright: ©Copyright 2022, American Mathematical Society. Document Type: Book. Publication Type: Book Date: 20170101. Pages: xxii+537 pp. MIT Press, 2017. ISBN: 978-0-262-53381-2. URL: http://www.library.illinois.edu/proxy/go.php?url=https://search.ebscohost.com/login.aspx?direct=true&db=msn&AN=MR3616878&site=eds-live&scope=site%20http://www.ams.org/mathscinet/MRAuthorID/739666%20http://www.ams.org/mathscinet/MRAuthorID/675768%20http://www.ams.org/mathscinet-getitem?mr=3616878.

[12] Raj Rajkumar. *Cyber-physical systems.* eng. 1st edition. The SEI series in software engineering. Boston: Addison-Wesley, 2017. ISBN: 0-13-341616-X.

[13] A Chutinan and B H Krogh. "Computational techniques for hybrid system verification". In: *IEEE Transactions on Automatic Control, Automatic Control, IEEE Transactions on, IEEE Trans. Automat. Contr.* 48 (1 Jan. 2003). Item Citation: IEEE Transactions on Automatic Control IEEE Trans. Automat. Contr. Automatic Control, IEEE Transactions on. 48(1):64-75 Jan, 2003 Document Subtype: IEEE Transaction Sponsored by: IEEE Control Systems Society Date of Current Version: 2003 AMSID: 1166525 Accession Number: edseee.1166525; Publication Type: Academic Journal; Source: IEEE Transactions on Automatic Control, Automatic Control, IEEE Transactions on, IEEE Trans. Automat. Contr.; Language: English; Publication Date: 20030101; Rights: Copyright 1963-2012, IEEE; Imprint: USA: IEEE, pp. 64–75. ISSN: 0018-9286. DOI: 10.1109/TAC.2002.806655. URL: http://www.library.illinois.edu/proxy/go.php?url=https://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.1166525&site=eds-live&scope=site.

[14] Steven Strogatz. *Nonlinear dynamics and chaos: with applications to physics, biology, chemistry, and engineering.* eng. Repr. Studies in nonlinearity. Cambridge, Mass: Westview Press, 2007. ISBN: 9780738204536.

[15] Stanford University. *Basics of Automata Theory.* publisher: Stanford university. 2004. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html.

[16] Joaquín Aranda Almansa et al. *Fundamentos de lógica matemática y computación.* Spanish. 1st ed. OCLC: 71821342. Madrid: Sanz y Torres, 2006. ISBN: 9788496094741.

[17] mathertel.de. *Programming Finite State Machines.* Dec. 2011. URL: http://www.mathertel.de/Arduino/FiniteStateMachine.aspx.

[18] Martin Fowler. *Domain Specific Languages.* Addison-Wesley Professional, 2010.

[19] Marjan Mernik. *Formal and Practical Aspects of Domain-specific Languages.* Information Science Reference, 2013. DOI: 10.4018/978-1-4666-2092-6.

[20] David Gries. *The Science of Programming.* New York, United States: Springer Publishing, 2012. DOI: 10.1007/978-1-4612-5983-1.

[21] M.G. Rekoff. “On reverse engineering.” In: *IEEE Transactions on Systems, Man, and Cybernetics, Systems, Man and Cybernetics, IEEE Transactions on, IEEE Trans. Syst., Man, Cybern* SMC-15.2 (1985), pp. 244–252. ISSN: 2168-2909. URL: http://www.library.illinois.edu.proxy2.library.illinois.edu/proxy/go.php?url=https://search-ebscohost-com.proxy2.library.illinois.edu/login.aspx?direct=true&db=edseee&AN=edseee.6313354&site=eds-live&scope=site.

[22] E.J. Chikofsky and J.H. Cross. “Reverse engineering and design recovery: a taxonomy”. In: *IEEE Software* 7.1 (Jan. 1990), pp. 13–17. ISSN: 1937-4194. DOI: 10.1109/52.43044.

[23] Mike Van Emmerik. *Program Transformation Wiki / Decompilation Process*. Feb. 2005. URL: http://www.program-transformation.org/Transform/DecompilationProcess.html.

[24] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. “Demand-driven compositional symbolic execution”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 367–381.

[25] Michael Hicks. *Symbolic Execution for finding bugs.*

[26] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. “SELECT—a Formal System for Testing and Debugging Programs by Symbolic Execution”. In: *SIGPLAN Not.* 10.6 (Apr. 1975), pp. 234–245. ISSN: 0362-1340. DOI: 10.1145/390016.808445. URL: https://doi-org.proxy2.library.illinois.edu/10.1145/390016.808445.

[27] Anastasis Keliris and Michail Maniatakos. “ICSREF: A Framework for Automated Reverse Engineering of Industrial Control Systems Binaries”. In: San Diego, CA, 2019. ISBN: 1-891562-55-X. DOI: 10.14722. URL: https://dx.doi.org/10.14722/ndss.2019.23271.

[28] N. Lawerence Ricker. *Tennessee Eastman challenge archive*. 2015. URL: https://depts.washington.edu/control/LARRY/TE/download.html (visited on 05/31/2022).

[29] J.J. Downs and E.F. Vogel. “A plant-wide industrial process control problem.” In: *Computers and Chemical Engineering* 17.3 (1993), pp. 245–255. ISSN: 00981354. URL: http://www.library.illinois.edu.proxy2.library.illinois.edu/proxy/go.php?url=https://search-ebscohost-com.proxy2.library.illinois.edu/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-0027561446&site=eds-live&scope=site.

[30] Nicolas Minorsky. “Directional stability of automatically steered bodies”. In: *Journal of the American Society for Naval Engineers* 34.2 (1922), pp. 280–309.

[31] CODESYS GmbH. *CODESYS Online Help - Util.* publisher: Schneider Electric. 2019. URL: https://product-help.schneider-electric.com/Machine%20Expert/V1.1/en/Util/index.htm?#t=topics%2Fpid_fixcycle.htm%23CSH_106 (visited on 05/31/2022).

[32] Matthias Althoff, Goran Frehse, and Antoine Girard. “Set Propagation Techniques for Reachability Analysis”. In: *Annual Review of Control, Robotics, and Autonomous Systems* 4 (1 May 2021). doi: 10.1146/annurev-control-071420-081941, pp. 369–395. ISSN: 2573-5144. DOI: 10.1146/annurev-control-071420-081941. URL: https://doi.org/10.1146/annurev-control-071420-081941.

[33] Richard Lipton. “The reachability problem requires exponential space”. In: *Department of Computer Science. Yale University* 62 (1976).

[34] Inc. The Mathworks. *Signal Processing Toolbox - Perform signal processing and analysis*. publisher: Mathworks. 2022. URL: https://www.mathworks.com/products/signal.html.

[35] NXP Model-Based Design Toolbox Team (2022). *NXP Support Package S32R*. 2022. URL: https://www.mathworks.com/matlabcentral/fileexchange/72232-nxp_support_package_s32r.

[36] Silion. *RADAR SDK FOR S32R PROCESSORS*. June 2017. URL: https://community.nxp.com/pwmxy87654/attachments/pwmxy87654/connects/225/1/AMF-AUT-T2723.pdf.

[37] TIOBE. *TIOBE Index for May 2022*. May 2022.

[38] Fernando Pérez and Brian E. Granger. “IPython: a System for Interactive Scientific Computing”. In: *Computing in Science and Engineering* 9.3 (May 2007), pp. 21–29. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.53. URL: https://ipython.org.

[39] Stack Overflow. *Stack Overflow Developer Survey 2021*. en. 2021. URL: https://insights.stackoverflow.com/survey/2021/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2021 (visited on 06/01/2022).

[40] Ric Messier. *Operating system forensics*. eng. First edition. Waltham, MA: Elsevier, 2016. ISBN: 0-12-801963-8.

[41] Gregory Malecha et al. “Towards foundational verification of cyber-physical systems”. In: *2016 Science of Security for Cyber-Physical Systems Workshop (SOSCYPS)*. Vienna, Austria: IEEE, Apr. 2016, pp. 1–5. ISBN: 9781509043040. DOI: 10.1109/SOSCYPS.2016.7580000. URL: http://ieeexplore.ieee.org/document/7580000/ (visited on 06/01/2022).

[42] Michael Sammler et al. “RefinedC: automating the foundational verification of C code with refined ownership types”. en. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Virtual Canada: ACM, June 2021, pp. 158–174. ISBN: 9781450383912. DOI: 10.1145/3453483.3454036. URL: https://dl.acm.org/doi/10.1145/3453483.3454036 (visited on 06/01/2022).

[43] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. “discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code”. en. In: 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016. Internet Society, 2016, pp. 1–15. ISBN: 1-891562-41-X. DOI: 10.14722/ndss.2016.23185.

[44] Dennis Andriesse, Asia Slowinska, and Herbert Bos. "Compiler-Agnostic Function Detection in Binaries." In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P), Security and Privacy (EuroS&P), 2017 IEEE European Symposium on, EUROS-P* (2017), pp. 177–189. ISSN: 978-1-5090-5762-7. URL: `http://www.library.illinois.edu.proxy2.library.illinois.edu/proxy/go.php?url=https://search-ebscohost-com.proxy2.library.illinois.edu/login.aspx?direct=true&db=edseee&AN=edseee.7961979&site=eds-live&scope=site`.

[45] Y David, N Partush, and E Yahav. "Similarity of Binaries through re-Optimization". In: vol. Part F128414. Accession Number: edselc.2-52.0-85025120959; (Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 14 June 2017, Part F128414:79-94) Publication Type: Conference Proceeding; Rights: Copyright 2018 Elsevier B.V., All rights reserved. Association for Computing Machinery, 2017, pp. 79–94. ISBN: 9781450349888. DOI: `10.1145/3062341.3062387`. URL: `http://www.library.illinois.edu/proxy/go.php?url=https://search.ebscohost.com/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-85025120959&site=eds-live&scope=site`.

[46] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. "Flow*: An Analyzer for Non-linear Hybrid Systems". In: *Computer Aided Verification*. Ed. by David Hutchison et al. Vol. 8044. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 258–263. ISBN: 9783642397981. DOI: `10.1007/978-3-642-39799-8_18`. URL: `http://link.springer.com/10.1007/978-3-642-39799-8_18` (visited on 06/01/2022).

[47] J.A. Agirre et al. "The VALU3S ECSEL project: Verification and validation of automated systems safety and security". In: *Microprocessors and Microsystems* 87 (2021), p. 104349. ISSN: 0141-9331. DOI: `https://doi.org/10.1016/j.micpro.2021.104349`. URL: `https://www.sciencedirect.com/science/article/pii/S0141933121005068`.

[48] Heethesh Vhavle, Sanam Shakya, and Pushkar Raj. *eYSIP-2017 Control and Algorithms Development for Quadcopter*. May 2017.

[49] Sebastian Madgwick et al. "An efficient orientation filter for inertial and inertial/-magnetic sensor arrays". In: *Report x-io and University of Bristol (UK)* 25 (2010), pp. 113–118.

[50] Drona Aviation. *Pluto DIY Nano Drone Kit 1.2 - for Learning Stem & Tinkering, Crash Resistant, Smartphone Controlled with Rechargeable Battery by Drona Aviation*. 2021.

[51] SYSTEMPlus Consulting. *Continental ARS4-B / ARS41Continental ARS4-B / ARS410 Long and Short Range Radar (1108647-00-D)*. 2020.

[52] SEGGER Microcontroller GmbH. *emFloat User Guide & Reference Manual*. 2021. URL: `https://www.segger.com/doc/UM12008_FloatingPointLibrary.html`.

[53] United Nations. *Sustainable Development Goals*. June 2, 2022. URL: `https://www.un.org/sustainabledevelopment/`.

[54] *WCED .:. Sustainable Development Knowledge Platform*. Aug. 4, 1987. URL: `https://sustainabledevelopment.un.org/milestones/wced`.

[55] Rockström, Sukhdev, and Stockholm University. *Contributions to Agenda 2030*. Feb. 28, 2017. URL: https://www.stockholmresilience.org/research/research-news/2017-02-28-contributions-to-agenda-2030.html (visited on 06/05/2022).

Appendix A

Code and Execution Examples

A.1 Function Helper

```
import ghidra_bridge
from binascii import hexlify

# Create the ghidra_bridge space
gb = ghidra_bridge.GhidraBridge(namespace=globals())

# Importing function analyzing tools (decompiler options,
# task monitor and the decompiler interface)
options = ghidra.app.decompiler.DecompileOptions()
monitor = ghidra.util.task.ConsoleTaskMonitor()
ifc = ghidra.app.decompiler.DecompInterface()
ifc.setOptions(options)

# Obtaining the current program functions
fm = currentProgram.getFunctionManager()
functions = fm.getFunctions(True)

# Create a function f_listed
function_list = []

# Getting the program listing
listing = currentProgram.getListing()

# Dictionary where all function calls are max_depth
func_calls_dictionary = {}


#-----------------------------------------------------------------------------
# @param f -> FunctionDB variable
#-----------------------------------------------------------------------------
def getFunctionAnalysis(f):
    """Prints information and analysis of a given function
    including symbols, internal variables, Program Counter Addresses
    and others
    """
    print("\n\n--------------------------------------------------------------------------------")
    print("| Function '%s' @%s |" % (f.getSignature().getPrototypeString(), f.getEntryPoint()))
    print("---------------------------")

    # Opening the function and decompiling it
    ifc.openProgram(f.getProgram())
    res = ifc.decompileFunction(f, 30, monitor)
    high_func = res.getHighFunction()
    lsm = high_func.getLocalSymbolMap()
    symbols = lsm.getSymbols()

    for i, symbol in enumerate(symbols):
        print("\nSymbol {}:".format(i+1))
        print("  name:         {}".format(symbol.name))
        print("  dataType:     {}".format(symbol.dataType))
        hs = symbol.getHighVariable()
        instances = hs.getInstances()
        for instance in instances:
            print("\n  instance:     {}".format(instance))
            print("    type:         {}".format(type(instance)))
            print("    uniqueID:     {}".format(instance.uniqueId))
            print("    PCAddress:    {}".format(instance.getPCAddress()))
            for desc in instance.getDescendants():
                print("     Descendant:    {}".format(desc))


#--------------------------------------------------------------------------------
# Function that returns the first FuntionDB object when given a name
#--------------------------------------------------------------------------------
def getFunctionFromName(f_name):
    """Returns the FunctionDB variable for a given function with the f_name
    """
    for f_listed in functions:
        if(f_name in f_listed.getSignature().getPrototypeString()):
            print("Found a function")
            return f_listed

    #return False


#--------------------------------------------------------------------------------
# Function to obtain and print function information such as input parameters and
#--------------------------------------------------------------------------------
def getFunctionNameandInfo(f_name):
    """Prints basic function information given a function name and also returns the
    actual FunctionDB variable. Duplicate method"""
    for f_listed in functions:
        if(f_name in f_listed.getSignature().getPrototypeString()):

            print("-----------------------------------------------")
            print("| Function: ", f_listed.getSignature().getPrototypeString(), " |")
            print("-----------------------------------------------")

            print("Entry point: ", f_listed.getEntryPoint())
            print("Entry variables: ", f_listed.getParameters())
            print("Return variable: ", f_listed.getReturn().getFormalDataType(), " Addr: ", f_listed.getReturn())

            return f_listed


#----------------------------------------------------------------------------
# Function to iterate get thorugh
# F is the function that is being called
# calling_function is the fucntion where it is beign called
#----------------------------------------------------------------------------
def getFunctionCallingAddresses(f_pcpal, f_hoja):
    #addr_set = f.getBody()
    #codeUnits = listing.getCodeUnits(addr_set, True)

    entry_point = f_hoja.getEntryPoint()
    references = getReferencesTo(entry_point)

    for xref in references:
        print(f_pcpal.getBody().getMaxAddress(), "  min ", f_pcpal.getBody().getMinAddress())
        if(f_pcpal.getBody().contains(xref.getFromAddress())):
            print(f_hoja.getName(), " is called at: ", xref.getFromAddress())
            calls_file.write(str(f_hoja.getName())+"\t0x"+str(xref.getFromAddress())+"\n")
            if(str(xref.getFromAddress()) not in func_calls_dictionary):
                func_calls_dictionary[str(xref.getFromAddress())] = f_hoja.getName()

    # Uncomment this section to print in the command-line function information
    """for unit in codeUnits:
        print(unit.toString())
        if(calling_func in unit.toString()):
            print(calling_func+" is located at "+ str(unit.getAddress()) + " in " + "f.getName()")
            #print(unit.toString())
            #print(f.getAddress())
            #print("Function: ", f.getName(), " has a return address of: ", unit.getAddress())
            return unit.getAddress()"""


#----------------------------------------------------------------------------
# Function to write Function -> Calling address
#----------------------------------------------------------------------------
```

```python
def writeFunctionCalls():
    string_write_dictionary = ""            # String where the dictionary
 between addresses and call locations

    for key, value in func_calls_dictionary.items():
        string_write_dictionary = string_write_dictionary + str(value)+"\
t0x"+str(key)+"\n"

    #print(string_write_dictionary)

    with open("func_call_addr.txt", "w") as fc:
        fc.write(string_write_dictionary)

#-------------------------------------------------------------------------------
# Function to write Function -> Return register
#-------------------------------------------------------------------------------
def writeFunctionReturns(string):
    with open("func_return.txt", "w") as fr:
        fr.write(string)


#-------------------------------------------------------------------------------
# Function to write Function -> Input parameters
#-------------------------------------------------------------------------------
def writeFunctionParameters(string):
    with open("func_param.txt", "w") as fp:
        fp.write(string)            # TODO acabar esto


def writeFunctionTreeList(string):
    with open("func_tree_list.txt", "w") as ftl:
        ftl.write(string)           # TODO Acabar esto


#-------------------------------------------------------------------------------
# Define a function to iterate and get through all of the leaf functions
#-------------------------------------------------------------------------------
def generateFunctionTree(f, max_depth, depth):
    """Generates a list of all called functions by one recursively
    """

    if f not in function_list:
        function_list.append(f)

    if(depth == 0):
        return

    called_func = f.getCalledFunctions(monitor)

    if not bool(called_func):
        return

    #print("Function ", f.getName(), " calls: ", called_func)
    for func_el in called_func:
        getFunctionCallingAddresses(f, func_el)
        print("Function name: ", func_el)
        #generateFunctionTree(func_el, max_depth, depth-1)

    return

#-----------------------------------------------------------------------------
# Generates all function return registers, omitting those that don't
# return anything
#-----------------------------------------------------------------------------
def getFunctionsReturnRegisters():
    """Gets and writes all of the function's return registers.

    Obtains the return register for all of the program functions
    and writes them to a file <func_return.txt with the following format:
    <function name> \\t <return register>."""
    string_write_registers = ""
    func = getFirstFunction()                   # Obtaining the first function
    while func is not None:
        if("None" not in str(func.getReturn().getRegister())):
            #print("None no esta en ", func.getReturn().getRegister())
            string_write_registers = string_write_registers + str(func.
   getName())+"\t"+str(func.getReturn().getRegister())+"\n"
        func = getFunctionAfter(func)

    writeFunctionReturns(string_write_registers)

#-----------------------------------------------------------------------------
# Generates all of the functions input registers, omitting those that don't
# exist
#-----------------------------------------------------------------------------
def getFunctionsInputParams():
    """Gets and writes all of the fucntion's input parameter registers.

    Obtains all of the input parameters registers for all of the program
   functions
    and rewrites them to a file <func_param.txt> with the following format:
    <function name> \\t <input registers (\\t)>. """
    string_write_params = ""
    func = getFirstFunction()
    while func is not None:
        string_write_params = string_write_params + str(func.getName()) + "
   \t"
        for param in func.getParameters():
            if "None" not in str(param.getRegister()):
                string_write_params = string_write_params + str(param.
   getRegister()) + "\t"
        string_write_params = string_write_params + "\n"
        func = getFunctionAfter(func)

    writeFunctionParameters(string_write_params)


#-----------------------------------------------------------------------------
#-----------------------------------------------------------------------------
# |                          MAIN EXECUTION                                  |
#------------------------------------------------------------------------------
#------------------------------------------------------------------------------

# Analizing the function selected
function_name = "MadgwickQuaternionUpdate"
fu = getFunctionNameandInfo(function_name)

# Generating the function tree for "MadgwickQuaternionUpdate"
# Also generates a dictionary with all of the function calls
print("INFO: function tools: generating the function tree.")
generateFunctionTree(fu, 10, 10)


# Writing the function tree to a file
writeFunctionCalls()

# Writing the function Parameters and Returns
print("INFO: function tools: generating the function input parameters.")
getFunctionsInputParams()
print("INFO: function tools: generating the function return registers.")
getFunctionsReturnRegisters()

print("INFO: function tools: done.")
```

Listing A.1: Code of the function helper tool.

A.2 Drone Top-Level

```
#!/usr/bin/env python3
"""
Analyzes the control loop of the quadcopter firmware

argv[1] is the firmware path
"""
import random
import sys
import angr
import claripy
import pprint
import IPython
from analyzer import Analyzer
import os

DIR = os.path.dirname(os.path.realpath(__file__))

a = Analyzer(sys.argv[1], DIR + "/../notebooks/func_call_addr.txt")
a.enter_function(0x8003331)

a.init_sym_regs({
    "r0": "ax",
    "r1": "ay",
    "r2": "az",
    "r3": "gx",
    "r4": "gy",
    "r5": "my",
    "r6": None,
    "r7": None,
    "r8": None,
    "r9": None,
    "r10": None,
    "r11": None,
})

unconstr_stack = {}
for i in range(-0x84, 0x2C, 4):
    unconstr_stack[i] = None

unconstr_stack[0] = "gy"
unconstr_stack[4] = "gz"
unconstr_stack[8] = "mx"
unconstr_stack[12] = "my"
unconstr_stack[16] = "mz"

a.init_sym_stack(unconstr_stack)

a.init_sym_memory(0x2000035C], "FPstate_beta")
a.init_sym_memory(0x20000360], "FPstate_deltat")
a.init_sym_memory(0x2000003C], "FPstate_q0")
a.init_sym_memory(0x20000040], "FPstate_q1")
a.init_sym_memory(0x20000044], "FPstate_q2")
a.init_sym_memory(0x20000048], "FPstate_q3")

a.init_time_field(0x20000360)

a.init_control_state([
0x2000003C, 0x20000040, 0x20000044, 0x20000048
])

a.set_concrete_stack(0x14, 0x30000000)

a.run()

unc = a.get_results()
mem_locs = [
    ["q0_out", unc[-1].mem.float[0x2000003C]],
    ["q1_out", unc[-1].mem.float[0x20000040]],
    ["q2_out", unc[-1].mem.float[0x20000044]],
    ["q3_out", unc[-1].mem.float[0x20000048]],
    ["theta_out", unc[-1].mem.float[unc[-1].regs.r3]],
    ["phi_out", unc[-1].mem.float[unc[-1].regs.r3 + 4]],
    ["psi_out", unc[-1].mem.float[unc[-1].regs.r3 + 8]]
]

a.analyze(mem_locs)
```

Listing A.2: Code of `drone_toplevel.py`
Courtesy of Maxwell Bland

A.3 PLC Differential Equations Output

```

#!/usr/bin/env python3

# Verification of the equations presented for the analysis of the PLC

# Input variables to simulate 1
mem_19aad_6_16 = 1
mem_198a8_14_32 = 1
mem_198cc_20_32 = 1
mem_19898_7_32 = 1
mem_198b0_16_32 = 1
mem_198ac_15_32 = 1

# Input variables to simulate 2
mem_19aaf_8_16 = 1
mem_198a4_37_32 = 1

mem_198c0_39_32 = 1
mem_198bc_38_32 = 1


# Input variables to simulate 3
mem_199fb_50_32 = 1
mem_198a4_51_32 = 1


# Function execution and verification

# Setting up memory values


# Gen_output:  FP_integral_out_0_27_32
# Elements to simulate mem_19aad_6_16, mem_198a8_14_32, mem_198cc_20_32
def generate_FP_integral_out_0_27_32(FP_integral_out_0_27_32):
    assignment_1_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19aad_6_16
    assignment_2_0 = assignment_1_0 + -0.0046
    assignment_3_0 = assignment_2_0 / 0.9876000000000001
    assignment_4_0 = 1000.0 * assignment_3_0
    assignment_5_0 = assignment_4_0 + 2000.0
    assignment_6_0 = mem_198a8_14_32 - assignment_5_0
    assignment_7_0 = mem_198cc_20_32 * 1000.0
    assignment_8_0 = assignment_6_0 * assignment_7_0
    assignment_9_0 = assignment_8_0 / 1000.0
    assignment_10_0 = FP_integral_out_0_27_32 + assignment_9_0
    FP_integral_out_0_27_32 = assignment_10_0


#Gen_output:  FP_derivative_x3_0_24_32
def generate_FP_derivative_x3_0_24_32(FP_derivative_x2_0_23_32):
    FP_derivative_x3_0_24_32 = FP_derivative_x2_0_23_32


```

```
#Gen_output:   FP_derivative_x2_0_23_32
def generate_FP_derivative_x2_0_23_32(FP_derivative_x1_0_22_32):
    FP_derivative_x2_0_23_32 = FP_derivative_x1_0_22_32


#Gen_output:   FP_derivative_x1_0_22_32
def generate_FP_derivative_x1_0_22_32():
    assignment_1_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_2_0 = assignment_1_0 + -0.0046
    assignment_3_0 = assignment_2_0 / 0.9876000000000001
    assignment_4_0 = 1000.0 * assignment_3_0
    assignment_5_0 = assignment_4_0 + 2000.0
    assignment_6_0 = mem_198a8_14_32 - assignment_5_0
    FP_derivative_x1_0_22_32 = assignment_6_0
    return FP_derivative_x1_0_22_32


#Gen_output:   FP_derivative_t2_0_25_32
def generate_FP_derivative_t2_0_25_32(FP_derivative_t1_0_26_32):
    FP_derivative_t2_0_25_32 = FP_derivative_t1_0_26_32


#Gen_output:   FP_derivative_t1_0_26_32
def generate_FP_derivative_t1_0_26_32():
    assignment_1_0 = mem_198cc_20_32 * 1000.0
    FP_derivative_t1_0_26_32 = assignment_1_0


#Gen_output:   FP_pid_fixcycle_y_0_21_32
def generate_FP_pid_fixcycle_y_0_21_32(FP_pid_fixcycle_y_0_21_32,
   FP_derivative_t1_0_26_32, FP_derivative_x2_0_23_32,
   FP_integral_out_0_27_32, FP_derivative_t2_0_25_32,
   FP_derivative_x1_0_22_32, FP_derivative_x3_0_24_32):
    assignment_1_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_2_0 = assignment_1_0 + -0.0046
    assignment_3_0 = assignment_2_0 / 0.9876000000000001
    assignment_4_0 = 1000.0 * assignment_3_0
    assignment_5_0 = assignment_4_0 + 2000.0
    assignment_6_0 = mem_198a8_14_32 - assignment_5_0
    assignment_7_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_8_0 = assignment_7_0 + -0.0046
    assignment_9_0 = assignment_8_0 / 0.9876000000000001
    assignment_10_0 = 1000.0 * assignment_9_0
    assignment_11_0 = assignment_10_0 + 2000.0
    assignment_12_0 = mem_198a8_14_32 - assignment_11_0
    assignment_13_0 = mem_198cc_20_32 * 1000.0
    assignment_14_0 = assignment_12_0 * assignment_13_0
    assignment_15_0 = assignment_14_0 / 1000.0
    assignment_16_0 = FP_integral_out_0_27_32 + assignment_15_0
    assignment_17_0 = assignment_16_0 / mem_198b0_16_32
    assignment_18_0 = assignment_6_0 + assignment_17_0
    assignment_19_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_20_0 = assignment_19_0 + -0.0046
    assignment_21_0 = assignment_20_0 / 0.9876000000000001
    assignment_22_0 = 1000.0 * assignment_21_0
    assignment_23_0 = assignment_22_0 + 2000.0
    assignment_24_0 = mem_198a8_14_32 - assignment_23_0
    assignment_25_0 = assignment_24_0 - FP_derivative_x3_0_24_32
    assignment_26_0 = 3.0 * assignment_25_0
    assignment_27_0 = assignment_26_0 + FP_derivative_x1_0_22_32
    assignment_28_0 = assignment_27_0 - FP_derivative_x2_0_23_32
    assignment_29_0 = 3.0 * FP_derivative_t2_0_25_32
    assignment_30_0 = 4.0 * FP_derivative_t1_0_26_32
    assignment_31_0 = assignment_29_0 + assignment_30_0
    assignment_32_0 = mem_198cc_20_32 * 1000.0
    assignment_33_0 = 3.0 * assignment_32_0
    assignment_34_0 = assignment_31_0 + assignment_33_0
    assignment_35_0 = assignment_28_0 / assignment_34_0
    assignment_36_0 = assignment_35_0 * 1000.0
    assignment_37_0 = assignment_36_0 * 0.0
    assignment_38_0 = assignment_18_0 + assignment_37_0
    assignment_39_0 = mem_198ac_15_32 * assignment_38_0
    assignment_40_0 = 0.0 + assignment_39_0
    assignment_41_0 = FP_pid_fixcycle_y_0_21_32 - 0.0
    assignment_42_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_43_0 = assignment_42_0 + -0.0046
    assignment_44_0 = assignment_43_0 / 0.9876000000000001
    assignment_45_0 = 1000.0 * assignment_44_0
    assignment_46_0 = assignment_45_0 + 2000.0
    assignment_47_0 = mem_198a8_14_32 - assignment_46_0
    assignment_48_0 = mem_198ac_15_32 * assignment_47_0
    assignment_49_0 = assignment_41_0 - assignment_48_0
    assignment_50_0 = assignment_40_0 + assignment_49_0
    FP_pid_fixcycle_y_0_21_32 = assignment_50_0


#-----------------------------------------------------------
#-----------------------------------------------------------
#-----------------------------------------------------------


#Gen_output:  FP_integral_out_1_49_32
def generate_FP_integral_out_1_49_32(FP_pid_fixcycle_y_0_21_32,
   FP_integral_out_1_49_32, FP_derivative_t1_0_26_32,
   FP_derivative_x2_0_23_32, FP_integral_out_0_27_32,
   FP_derivative_t2_0_25_32, FP_derivative_x1_0_22_32,
   FP_derivative_x3_0_24_32):
    assignment_1_0 = ((0x0 << 32) | mem_19aaf_8_16) / mem_19898_7_32
    assignment_2_0 = assignment_1_0 + -0.0024000000000000002
    assignment_3_0 = assignment_2_0 / 0.9936
    assignment_4_0 = 3.0 * assignment_3_0
    assignment_5_0 = assignment_4_0 + -0.5
    assignment_6_0 = 0.5 * assignment_5_0
    assignment_7_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_8_0 = assignment_7_0 + -0.0046
    assignment_9_0 = assignment_8_0 / 0.9876000000000001
    assignment_10_0 = 1000.0 * assignment_9_0
    assignment_11_0 = assignment_10_0 + 2000.0
    assignment_12_0 = -assignment_11_0
    assignment_13_0 = mem_198a8_14_32 + assignment_12_0
    assignment_14_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_15_0 = assignment_14_0 + -0.0046
    assignment_16_0 = assignment_15_0 / 0.9876000000000001
    assignment_17_0 = 1000.0 * assignment_16_0
    assignment_18_0 = assignment_17_0 + 2000.0
    assignment_19_0 = -assignment_18_0
    assignment_20_0 = mem_198a8_14_32 + assignment_19_0
    assignment_21_0 = mem_198cc_20_32 * 1000.0
    assignment_22_0 = assignment_20_0 * assignment_21_0
    assignment_23_0 = assignment_22_0 / 1000.0
    assignment_24_0 = FP_integral_out_0_27_32 + assignment_23_0
    assignment_25_0 = assignment_24_0 / mem_198b0_16_32
    assignment_26_0 = assignment_13_0 + assignment_25_0
    assignment_27_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_28_0 = assignment_27_0 + -0.0046
    assignment_29_0 = assignment_28_0 / 0.9876000000000001
    assignment_30_0 = 1000.0 * assignment_29_0
    assignment_31_0 = assignment_30_0 + 2000.0
    assignment_32_0 = -assignment_31_0
    assignment_33_0 = mem_198a8_14_32 + assignment_32_0
    assignment_34_0 = -FP_derivative_x3_0_24_32
    assignment_35_0 = assignment_33_0 + assignment_34_0
    assignment_36_0 = 3.0 * assignment_35_0
    assignment_37_0 = assignment_36_0 + FP_derivative_x1_0_22_32
    assignment_38_0 = -FP_derivative_x2_0_23_32
    assignment_39_0 = assignment_37_0 + assignment_38_0
    assignment_40_0 = 3.0 * FP_derivative_t2_0_25_32
    assignment_41_0 = 4.0 * FP_derivative_t1_0_26_32
    assignment_42_0 = assignment_40_0 + assignment_41_0
    assignment_43_0 = mem_198cc_20_32 * 1000.0
    assignment_44_0 = 3.0 * assignment_43_0
    assignment_45_0 = assignment_42_0 + assignment_44_0
    assignment_46_0 = assignment_39_0 / assignment_45_0
    assignment_47_0 = assignment_46_0 * 1000.0
    assignment_48_0 = assignment_47_0 * 0.0
    assignment_49_0 = assignment_26_0 + assignment_48_0
    assignment_50_0 = mem_198ac_15_32 * assignment_49_0
    assignment_51_0 = 0.0 + assignment_50_0
    assignment_52_0 = FP_pid_fixcycle_y_0_21_32 + -0.0
    assignment_53_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_54_0 = assignment_53_0 + -0.0046
    assignment_55_0 = assignment_54_0 / 0.9876000000000001
    assignment_56_0 = 1000.0 * assignment_55_0
    assignment_57_0 = assignment_56_0 + 2000.0
    assignment_58_0 = -assignment_57_0
    assignment_59_0 = mem_198a8_14_32 + assignment_58_0
    assignment_60_0 = mem_198ac_15_32 * assignment_59_0
    assignment_61_0 = -assignment_60_0
    assignment_62_0 = assignment_52_0 + assignment_61_0
    assignment_63_0 = assignment_51_0 + assignment_62_0
    assignment_64_0 = assignment_63_0 * mem_198a4_37_32
    assignment_65_0 = assignment_6_0 - assignment_64_0
    assignment_66_0 = mem_198cc_20_32 * 1000.0
    assignment_67_0 = assignment_65_0 * assignment_66_0
    assignment_68_0 = assignment_67_0 / 1000.0
```

```python
    assignment_69_0 = FP_integral_out_1_49_32 + assignment_68_0
    FP_integral_out_1_49_32 = assignment_69_0


#Gen_output:  FP_derivative_x3_1_46_32
def generate_FP_derivative_x3_1_46_32(FP_derivative_x2_1_45_32):
    FP_derivative_x3_1_46_32 = FP_derivative_x2_1_45_32


#Gen_output:  FP_derivative_x2_1_45_32
def generate_FP_derivative_x2_1_45_32(FP_derivative_x1_1_44_32):
    FP_derivative_x2_1_45_32 = FP_derivative_x1_1_44_32


#Gen_output:  FP_derivative_x1_1_44_32
def generate_FP_derivative_x1_1_44_32(FP_pid_fixcycle_y_0_21_32,
   FP_derivative_t1_0_26_32, FP_derivative_x2_0_23_32,
   FP_integral_out_0_27_32, FP_derivative_t2_0_25_32,
   FP_derivative_x1_0_22_32, FP_derivative_x3_0_24_32):
    assignment_1_0 = ((0x0 << 32) | mem_19aaf_8_16) / mem_19898_7_32
    assignment_2_0 = assignment_1_0 + -0.0024000000000000002
    assignment_3_0 = assignment_2_0 / 0.9936
    assignment_4_0 = 3.0 * assignment_3_0
    assignment_5_0 = assignment_4_0 + -0.5
    assignment_6_0 = 0.5 * assignment_5_0
    assignment_7_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_8_0 = assignment_7_0 + -0.0046
    assignment_9_0 = assignment_8_0 / 0.9876000000000001
    assignment_10_0 = 1000.0 * assignment_9_0
    assignment_11_0 = assignment_10_0 + 2000.0
    assignment_12_0 = -assignment_11_0
    assignment_13_0 = mem_198a8_14_32 + assignment_12_0
    assignment_14_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_15_0 = assignment_14_0 + -0.0046
    assignment_16_0 = assignment_15_0 / 0.9876000000000001
    assignment_17_0 = 1000.0 * assignment_16_0
    assignment_18_0 = assignment_17_0 + 2000.0
    assignment_19_0 = -assignment_18_0
    assignment_20_0 = mem_198a8_14_32 + assignment_19_0
    assignment_21_0 = mem_198cc_20_32 * 1000.0
    assignment_22_0 = assignment_20_0 * assignment_21_0
    assignment_23_0 = assignment_22_0 / 1000.0
    assignment_24_0 = FP_integral_out_0_27_32 + assignment_23_0
    assignment_25_0 = assignment_24_0 / mem_198b0_16_32
    assignment_26_0 = assignment_13_0 + assignment_25_0
    assignment_27_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_28_0 = assignment_27_0 + -0.0046
    assignment_29_0 = assignment_28_0 / 0.9876000000000001
    assignment_30_0 = 1000.0 * assignment_29_0
    assignment_31_0 = assignment_30_0 + 2000.0
    assignment_32_0 = -assignment_31_0
    assignment_33_0 = mem_198a8_14_32 + assignment_32_0
    assignment_34_0 = -FP_derivative_x3_0_24_32
    assignment_35_0 = assignment_33_0 + assignment_34_0
    assignment_36_0 = 3.0 * assignment_35_0
    assignment_37_0 = assignment_36_0 + FP_derivative_x1_0_22_32
    assignment_38_0 =-FP_derivative_x2_0_23_32
    assignment_39_0 = assignment_37_0 + assignment_38_0
    assignment_40_0 = 3.0 * FP_derivative_t2_0_25_32
    assignment_41_0 = 4.0 * FP_derivative_t1_0_26_32
    assignment_42_0 = assignment_40_0 + assignment_41_0
    assignment_43_0 = mem_198cc_20_32 * 1000.0
    assignment_44_0 = 3.0 * assignment_43_0
    assignment_45_0 = assignment_42_0 + assignment_44_0
    assignment_46_0 = assignment_39_0 / assignment_45_0
    assignment_47_0 = assignment_46_0 * 1000.0
    assignment_48_0 = assignment_47_0 * 0.0
    assignment_49_0 = assignment_26_0 + assignment_48_0
    assignment_50_0 = mem_198ac_15_32 * assignment_49_0
    assignment_51_0 = 0.0 + assignment_50_0
    assignment_52_0 = FP_pid_fixcycle_y_0_21_32 + -0.0
    assignment_53_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_54_0 = assignment_53_0 + -0.0046
    assignment_55_0 = assignment_54_0 / 0.98760000000000001
    assignment_56_0 = 1000.0 * assignment_55_0
    assignment_57_0 = assignment_56_0 + 2000.0
    assignment_58_0 =-assignment_57_0
    assignment_59_0 = mem_198a8_14_32 + assignment_58_0
    assignment_60_0 = mem_198ac_15_32 * assignment_59_0
    assignment_61_0 =-assignment_60_0
    assignment_62_0 = assignment_52_0 + assignment_61_0
    assignment_63_0 = assignment_51_0 + assignment_62_0
    assignment_64_0 = assignment_63_0 * mem_198a4_37_32
    assignment_65_0 = assignment_6_0 - assignment_64_0
    FP_derivative_x1_1_44_32 = assignment_65_0


#Gen_output:  FP_derivative_t2_1_47_32
def generate_FP_derivative_t2_1_47_32(FP_derivative_t1_1_48_32):
    FP_derivative_t2_1_47_32 = FP_derivative_t1_1_48_32


#Gen_output:  FP_derivative_t1_1_48_32
def generate_FP_derivative_t1_1_48_32():
    assignment_1_0 = mem_198cc_20_32 * 1000.0
    FP_derivative_t1_1_48_32 = assignment_1_0


#Gen_output:  FP_pid_fixcycle_y_1_43_32
def generate_FP_pid_fixcycle_y_1_43_32(FP_pid_fixcycle_y_0_21_32,
   FP_derivative_x2_1_45_32, FP_pid_fixcycle_y_1_43_32,
   FP_integral_out_1_49_32, FP_derivative_x1_1_44_32,
   FP_derivative_t1_0_26_32, FP_derivative_x2_0_23_32,
   FP_integral_out_0_27_32, FP_derivative_t2_1_47_32,
   FP_derivative_t2_0_25_32, FP_derivative_x1_0_22_32,
   FP_derivative_t1_1_48_32, FP_derivative_x3_1_46_32,
   FP_derivative_x3_0_24_32):
    assignment_1_0 = ((0x0 << 32) | mem_19aaf_8_16) / mem_19898_7_32
    assignment_2_0 = assignment_1_0 + -0.0024000000000000002
    assignment_3_0 = assignment_2_0 / 0.9936
    assignment_4_0 = 3.0 * assignment_3_0
    assignment_5_0 = assignment_4_0 + -0.5
    assignment_6_0 = 0.5 * assignment_5_0
    assignment_7_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_8_0 = assignment_7_0 + -0.0046
    assignment_9_0 = assignment_8_0 / 0.9876000000000001
    assignment_10_0 = 1000.0 * assignment_9_0
    assignment_11_0 = assignment_10_0 + 2000.0
    assignment_12_0 = -assignment_11_0
    assignment_13_0 = mem_198a8_14_32 + assignment_12_0
    assignment_14_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_15_0 = assignment_14_0 + -0.0046
    assignment_16_0 = assignment_15_0 / 0.9876000000000001
    assignment_17_0 = 1000.0 * assignment_16_0
    assignment_18_0 = assignment_17_0 + 2000.0
    assignment_19_0 = -assignment_18_0
    assignment_20_0 = mem_198a8_14_32 + assignment_19_0
    assignment_21_0 = mem_198cc_20_32 * 1000.0
    assignment_22_0 = assignment_20_0 * assignment_21_0
    assignment_23_0 = assignment_22_0 / 1000.0
    assignment_24_0 = FP_integral_out_0_27_32 + assignment_23_0
    assignment_25_0 = assignment_24_0 / mem_198b0_16_32
    assignment_26_0 = assignment_13_0 + assignment_25_0
    assignment_27_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_28_0 = assignment_27_0 + -0.0046
    assignment_29_0 = assignment_28_0 / 0.9876000000000001
    assignment_30_0 = 1000.0 * assignment_29_0
    assignment_31_0 = assignment_30_0 + 2000.0
    assignment_32_0 = -assignment_31_0
    assignment_33_0 = mem_198a8_14_32 + assignment_32_0
    assignment_34_0 = -FP_derivative_x3_0_24_32
    assignment_35_0 = assignment_33_0 + assignment_34_0
    assignment_36_0 = 3.0 * assignment_35_0
    assignment_37_0 = assignment_36_0 + FP_derivative_x1_0_22_32
    assignment_38_0 = -FP_derivative_x2_0_23_32
    assignment_39_0 = assignment_37_0 + assignment_38_0
    assignment_40_0 = 3.0 * FP_derivative_t2_0_25_32
    assignment_41_0 = 4.0 * FP_derivative_t1_0_26_32
    assignment_42_0 = assignment_40_0 + assignment_41_0
    assignment_43_0 = mem_198cc_20_32 * 1000.0
    assignment_44_0 = 3.0 * assignment_43_0
    assignment_45_0 = assignment_42_0 + assignment_44_0
    assignment_46_0 = assignment_39_0 / assignment_45_0
    assignment_47_0 = assignment_46_0 * 1000.0
    assignment_48_0 = assignment_47_0 * 0.0
    assignment_49_0 = assignment_26_0 + assignment_48_0
    assignment_50_0 = mem_198ac_15_32 * assignment_49_0
    assignment_51_0 = 0.0 + assignment_50_0
    assignment_52_0 = FP_pid_fixcycle_y_0_21_32 + -0.0
    assignment_53_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_54_0 = assignment_53_0 + -0.0046
    assignment_55_0 = assignment_54_0 / 0.9876000000000001
    assignment_56_0 = 1000.0 * assignment_55_0
    assignment_57_0 = assignment_56_0 + 2000.0
    assignment_58_0 = -assignment_57_0
    assignment_59_0 = mem_198a8_14_32 + assignment_58_0
    assignment_60_0 = mem_198ac_15_32 * assignment_59_0
    assignment_61_0 = -assignment_60_0
    assignment_62_0 = assignment_52_0 + assignment_61_0
    assignment_63_0 = assignment_51_0 + assignment_62_0
    assignment_64_0 = assignment_63_0 * mem_198a4_37_32
    assignment_65_0 = assignment_6_0 - assignment_64_0
    assignment_66_0 = ((0x0 << 32) | mem_19aaf_8_16) / mem_19898_7_32
    assignment_67_0 = assignment_66_0 + -0.0024000000000000002
    assignment_68_0 = assignment_67_0 / 0.9936
    assignment_69_0 = 3.0 * assignment_68_0
    assignment_70_0 = assignment_69_0 + -0.5
    assignment_71_0 = 0.5 * assignment_70_0
    assignment_72_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_73_0 = assignment_72_0 + -0.0046
    assignment_74_0 = assignment_73_0 / 0.9876000000000001
    assignment_75_0 = 1000.0 * assignment_74_0
    assignment_76_0 = assignment_75_0 + 2000.0
    assignment_77_0 = -assignment_76_0
    assignment_78_0 = mem_198a8_14_32 + assignment_77_0
    assignment_79_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_80_0 = assignment_79_0 + -0.0046
    assignment_81_0 = assignment_80_0 / 0.9876000000000001
    assignment_82_0 = 1000.0 * assignment_81_0
    assignment_83_0 = assignment_82_0 + 2000.0
    assignment_84_0 = -assignment_83_0
    assignment_85_0 = mem_198a8_14_32 + assignment_84_0
    assignment_86_0 = mem_198cc_20_32 * 1000.0
    assignment_87_0 = assignment_85_0 * assignment_86_0
    assignment_88_0 = assignment_87_0 / 1000.0
    assignment_89_0 = FP_integral_out_0_27_32 + assignment_88_0
    assignment_90_0 = assignment_89_0 / mem_198b0_16_32
    assignment_91_0 = assignment_78_0 + assignment_90_0
    assignment_92_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_93_0 = assignment_92_0 + -0.0046
    assignment_94_0 = assignment_93_0 / 0.9876000000000001
    assignment_95_0 = 1000.0 * assignment_94_0
    assignment_96_0 = assignment_95_0 + 2000.0
    assignment_97_0 = -assignment_96_0
    assignment_98_0 = mem_198a8_14_32 + assignment_97_0
    assignment_99_0 = -FP_derivative_x3_0_24_32
    assignment_100_0 = assignment_98_0 + assignment_99_0
    assignment_101_0 = 3.0 * assignment_100_0
    assignment_102_0 = assignment_101_0 + FP_derivative_x1_0_22_32
    assignment_103_0 = -FP_derivative_x2_0_23_32
    assignment_104_0 = assignment_102_0 + assignment_103_0
    assignment_105_0 = 3.0 * FP_derivative_t2_0_25_32
    assignment_106_0 = 4.0 * FP_derivative_t1_0_26_32
    assignment_107_0 = assignment_105_0 + assignment_106_0
    assignment_108_0 = mem_198cc_20_32 * 1000.0
    assignment_109_0 = 3.0 * assignment_108_0
    assignment_110_0 = assignment_107_0 + assignment_109_0
    assignment_111_0 = assignment_104_0 / assignment_110_0
    assignment_112_0 = assignment_111_0 * 1000.0
    assignment_113_0 = assignment_112_0 * 0.0
    assignment_114_0 = assignment_91_0 + assignment_113_0
    assignment_115_0 = mem_198ac_15_32 * assignment_114_0
    assignment_116_0 = 0.0 + assignment_115_0
    assignment_117_0 = FP_pid_fixcycle_y_0_21_32 + -0.0
    assignment_118_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_119_0 = assignment_118_0 + -0.0046
    assignment_120_0 = assignment_119_0 / 0.9876000000000001
    assignment_121_0 = 1000.0 * assignment_120_0
    assignment_122_0 = assignment_121_0 + 2000.0
    assignment_123_0 = -assignment_122_0
    assignment_124_0 = mem_198a8_14_32 + assignment_123_0
    assignment_125_0 = mem_198ac_15_32 * assignment_124_0
    assignment_126_0 = -assignment_125_0
    assignment_127_0 = assignment_117_0 + assignment_126_0
    assignment_128_0 = assignment_116_0 + assignment_127_0
    assignment_129_0 = assignment_128_0 * mem_198a4_37_32
    assignment_130_0 = assignment_71_0 - assignment_129_0
    assignment_131_0 = mem_198cc_20_32 * 1000.0
    assignment_132_0 = assignment_130_0 * assignment_131_0
    assignment_133_0 = assignment_132_0 / 1000.0
    assignment_134_0 = FP_integral_out_1_49_32 + assignment_133_0
    assignment_135_0 = assignment_134_0 / mem_198c0_39_32
    assignment_136_0 = assignment_65_0 + assignment_135_0
    assignment_137_0 = ((0x0 << 32) | mem_19aaf_8_16) / mem_19898_7_32
    assignment_138_0 = assignment_137_0 + -0.0024000000000000002
    assignment_139_0 = assignment_138_0 / 0.9936
    assignment_140_0 = 3.0 * assignment_139_0
    assignment_141_0 = assignment_140_0 + -0.5
    assignment_142_0 = 0.5 * assignment_141_0
    assignment_143_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_144_0 = assignment_143_0 + -0.0046
    assignment_145_0 = assignment_144_0 / 0.9876000000000001
    assignment_146_0 = 1000.0 * assignment_145_0
    assignment_147_0 = assignment_146_0 + 2000.0
    assignment_148_0 = -assignment_147_0
    assignment_149_0 = mem_198a8_14_32 + assignment_148_0
    assignment_150_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_151_0 = assignment_150_0 + -0.0046
    assignment_152_0 = assignment_151_0 / 0.9876000000000001
    assignment_153_0 = 1000.0 * assignment_152_0
    assignment_154_0 = assignment_153_0 + 2000.0
    assignment_155_0 = -assignment_154_0
    assignment_156_0 = mem_198a8_14_32 + assignment_155_0
    assignment_157_0 = mem_198cc_20_32 * 1000.0
    assignment_158_0 = assignment_156_0 * assignment_157_0
    assignment_159_0 = assignment_158_0 / 1000.0
    assignment_160_0 = FP_integral_out_0_27_32 + assignment_159_0
    assignment_161_0 = assignment_160_0 / mem_198b0_16_32
    assignment_162_0 = assignment_149_0 + assignment_161_0
    assignment_163_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_164_0 = assignment_163_0 + -0.0046
    assignment_165_0 = assignment_164_0 / 0.9876000000000001
    assignment_166_0 = 1000.0 * assignment_165_0
    assignment_167_0 = assignment_166_0 + 2000.0
    assignment_168_0 = -assignment_167_0
    assignment_169_0 = mem_198a8_14_32 + assignment_168_0
    assignment_170_0 = -FP_derivative_x3_0_24_32
    assignment_171_0 = assignment_169_0 + assignment_170_0
    assignment_172_0 = 3.0 * assignment_171_0
    assignment_173_0 = assignment_172_0 + FP_derivative_x1_0_22_32
    assignment_174_0 = -FP_derivative_x2_0_23_32
    assignment_175_0 = assignment_173_0 + assignment_174_0
    assignment_176_0 = 3.0 * FP_derivative_t2_0_25_32
    assignment_177_0 = 4.0 * FP_derivative_t1_0_26_32
    assignment_178_0 = assignment_176_0 + assignment_177_0
    assignment_179_0 = mem_198cc_20_32 * 1000.0
    assignment_180_0 = 3.0 * assignment_179_0
    assignment_181_0 = assignment_178_0 + assignment_180_0
    assignment_182_0 = assignment_175_0 / assignment_181_0
    assignment_183_0 = assignment_182_0 * 1000.0
    assignment_184_0 = assignment_183_0 * 0.0
    assignment_185_0 = assignment_162_0 + assignment_184_0
    assignment_186_0 = mem_198ac_15_32 * assignment_185_0
    assignment_187_0 = 0.0 + assignment_186_0
    assignment_188_0 = FP_pid_fixcycle_y_0_21_32 + -0.0
    assignment_189_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_190_0 = assignment_189_0 + -0.0046
    assignment_191_0 = assignment_190_0 / 0.9876000000000001
    assignment_192_0 = 1000.0 * assignment_191_0
    assignment_193_0 = assignment_192_0 + 2000.0
    assignment_194_0 = -assignment_193_0
    assignment_195_0 = mem_198a8_14_32 + assignment_194_0
    assignment_196_0 = mem_198ac_15_32 * assignment_195_0
    assignment_197_0 = -assignment_196_0
    assignment_198_0 = assignment_188_0 + assignment_197_0
    assignment_199_0 = assignment_187_0 + assignment_198_0
    assignment_200_0 = assignment_199_0 * mem_198a4_37_32
    assignment_201_0 = assignment_142_0 - assignment_200_0
    assignment_202_0 = assignment_201_0 - FP_derivative_x3_1_46_32
    assignment_203_0 = 3.0 * assignment_202_0
    assignment_204_0 = assignment_203_0 + FP_derivative_x1_1_44_32
    assignment_205_0 = assignment_204_0 - FP_derivative_x2_1_45_32
    assignment_206_0 = 3.0 * FP_derivative_t2_1_47_32
    assignment_207_0 = 4.0 * FP_derivative_t1_1_48_32
    assignment_208_0 = assignment_206_0 + assignment_207_0
    assignment_209_0 = mem_198cc_20_32 * 1000.0
    assignment_210_0 = 3.0 * assignment_209_0
    assignment_211_0 = assignment_208_0 + assignment_210_0
    assignment_212_0 = assignment_205_0 / assignment_211_0
    assignment_213_0 = assignment_212_0 * 1000.0
    assignment_214_0 = assignment_213_0 * 0.0
    assignment_215_0 = assignment_136_0 + assignment_214_0
    assignment_216_0 = mem_198bc_38_32 * assignment_215_0
    assignment_217_0 = 0.0 + assignment_216_0
    assignment_218_0 = FP_pid_fixcycle_y_1_43_32 - 0.0
    assignment_219_0 = ((0x0 << 32) | mem_19aaf_8_16) / mem_19898_7_32
    assignment_220_0 = assignment_219_0 + -0.0024000000000000002
    assignment_221_0 = assignment_220_0 / 0.9936
    assignment_222_0 = 3.0 * assignment_221_0
    assignment_223_0 = assignment_222_0 + -0.5
    assignment_224_0 = 0.5 * assignment_223_0
    assignment_225_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_226_0 = assignment_225_0 + -0.0046
    assignment_227_0 = assignment_226_0 / 0.9876000000000001
    assignment_228_0 = 1000.0 * assignment_227_0
    assignment_229_0 = assignment_228_0 + 2000.0
    assignment_230_0 = -assignment_229_0
    assignment_231_0 = mem_198a8_14_32 + assignment_230_0
    assignment_232_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_233_0 = assignment_232_0 + -0.0046
    assignment_234_0 = assignment_233_0 / 0.9876000000000001
    assignment_235_0 = 1000.0 * assignment_234_0
    assignment_236_0 = assignment_235_0 + 2000.0
    assignment_237_0 = -assignment_236_0
    assignment_238_0 = mem_198a8_14_32 + assignment_237_0
    assignment_239_0 = mem_198cc_20_32 * 1000.0
    assignment_240_0 = assignment_238_0 * assignment_239_0
    assignment_241_0 = assignment_240_0 / 1000.0
    assignment_242_0 = FP_integral_out_0_27_32 + assignment_241_0
    assignment_243_0 = assignment_242_0 / mem_198b0_16_32
    assignment_244_0 = assignment_231_0 + assignment_243_0
    assignment_245_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_246_0 = assignment_245_0 + -0.0046
    assignment_247_0 = assignment_246_0 / 0.9876000000000001
    assignment_248_0 = 1000.0 * assignment_247_0
    assignment_249_0 = assignment_248_0 + 2000.0
    assignment_250_0 = -assignment_249_0
    assignment_251_0 = mem_198a8_14_32 + assignment_250_0
    assignment_252_0 = -FP_derivative_x3_0_24_32
    assignment_253_0 = assignment_251_0 + assignment_252_0
    assignment_254_0 = 3.0 * assignment_253_0
    assignment_255_0 = assignment_254_0 + FP_derivative_x1_0_22_32
    assignment_256_0 = -FP_derivative_x2_0_23_32
    assignment_257_0 = assignment_255_0 + assignment_256_0
    assignment_258_0 = 3.0 * FP_derivative_t2_0_25_32
    assignment_259_0 = 4.0 * FP_derivative_t1_0_26_32
    assignment_260_0 = assignment_258_0 + assignment_259_0
    assignment_261_0 = mem_198cc_20_32 * 1000.0
    assignment_262_0 = 3.0 * assignment_261_0
    assignment_263_0 = assignment_260_0 + assignment_262_0
    assignment_264_0 = assignment_257_0 / assignment_263_0
    assignment_265_0 = assignment_264_0 * 1000.0
    assignment_266_0 = assignment_265_0 * 0.0
    assignment_267_0 = assignment_244_0 + assignment_266_0
    assignment_268_0 = mem_198ac_15_32 * assignment_267_0
    assignment_269_0 = 0.0 + assignment_268_0
    assignment_270_0 = FP_pid_fixcycle_y_0_21_32 + -0.0
    assignment_271_0 = ((0x0 << 32) | mem_19aad_6_16) / mem_19898_7_32
    assignment_272_0 = assignment_271_0 + -0.0046
    assignment_273_0 = assignment_272_0 / 0.9876000000000001
    assignment_274_0 = 1000.0 * assignment_273_0
    assignment_275_0 = assignment_274_0 + 2000.0
    assignment_276_0 = -assignment_275_0
    assignment_277_0 = mem_198a8_14_32 + assignment_276_0
    assignment_278_0 = mem_198ac_15_32 * assignment_277_0
    assignment_279_0 = -assignment_278_0
    assignment_280_0 = assignment_270_0 + assignment_279_0
    assignment_281_0 = assignment_269_0 + assignment_280_0
    assignment_282_0 = assignment_281_0 * mem_198a4_37_32
    assignment_283_0 = assignment_224_0 - assignment_282_0
    assignment_284_0 = mem_198bc_38_32 * assignment_283_0
    assignment_285_0 = assignment_218_0 - assignment_284_0
    assignment_286_0 = assignment_217_0 + assignment_285_0
    FP_pid_fixcycle_y_1_43_32 = assignment_286_0

#-----------------------------------------------------------
#-----------------------------------------------------------
#-----------------------------------------------------------

#Gen_output:  pressure
def generate_pressure():
    assignment_1_0 = mem_199fb_50_32 * mem_198a4_51_32
    pressure = assignment_1_0
```

A.4 Drone Differential Equations Output

```
#!/usr/bin/env python3
import dadra
import math


def modelRec(state, FPstate_deltat_57_32, FPreg_ax_r0_0_32
   =0.1321582976668071, FPreg_ay_r1_1_32=0.32993030595956074,
   FPreg_az_r2_2_32=0.6754967833384182, FPstate_beta_56_32
   =0.6864729397366336, FPreg_gx_r3_3_32=0.5257138543345415,
   FPstack_gy_0_45_32=0.23623629042347727, FPstack_gz_4_46_32
   =0.0016262269322375733, FPstack_mx_8_47_32=0.207115455996444,
   FPstack_my_12_48_32=0.2141901117143904, FPstack_mz_16_49_32
   =0.5742019759212731):

    FPstate_q0_58_32 = state[0]
    FPstate_q1_59_32 = state[1]
    FPstate_q2_60_32 = state[2]
    FPstate_q3_61_32 = state[3]

    FP_0_sfloat_63_32 = FPreg_ax_r0_0_32 * FPreg_ax_r0_0_32
    FP_1_sfloat_64_32 = FPreg_ay_r1_1_32 * FPreg_ay_r1_1_32
    FP_2_sfloat_65_32 = FP_0_sfloat_63_32 + FP_1_sfloat_64_32
    FP_3_sfloat_66_32 = FPreg_az_r2_2_32 * FPreg_az_r2_2_32
    FP_4_sfloat_67_32 = FP_2_sfloat_65_32 + FP_3_sfloat_66_32
    FP_5_sfloat_68_64 = FP_4_sfloat_67_32
    FP_8_sfloat_71_64 = (FP_5_sfloat_68_64) ** (1.0/2.0)
    FP_11_sfloat_74_32 = FP_8_sfloat_71_64
    FP_14_sfloat_77_32 = FPstack_mx_8_47_32 * FPstack_mx_8_47_32
    FP_15_sfloat_78_32 = FPstack_my_12_48_32 * FPstack_my_12_48_32
    FP_16_sfloat_79_32 = FP_14_sfloat_77_32 + FP_15_sfloat_78_32
    FP_17_sfloat_80_32 = FPstack_mz_16_49_32 * FPstack_mz_16_49_32
    FP_18_sfloat_81_32 = FP_16_sfloat_79_32 + FP_17_sfloat_80_32
    FP_19_sfloat_82_64 = FP_18_sfloat_81_32
    FP_22_sfloat_85_64 = (FP_19_sfloat_82_64) ** (1.0/2.0)
    FP_25_sfloat_88_32 = FP_22_sfloat_85_64
    FP_28_sfloat_91_64 = FPreg_gx_r3_3_32
    FP_31_sfloat_94_64 = FP_28_sfloat_91_64 * 0.017453292519943295
    FP_34_sfloat_97_32 = FP_31_sfloat_94_64
    FP_36_sfloat_99_64 = FPstack_gy_0_45_32
    FP_39_sfloat_102_64 = FP_36_sfloat_99_64 * 0.017453292519943295
    FP_42_sfloat_105_32 = FP_39_sfloat_102_64
    FP_44_sfloat_107_64 = FPstack_gz_4_46_32
    FP_47_sfloat_110_64 = FP_44_sfloat_107_64 * 0.017453292519943295
    FP_50_sfloat_113_32 = FP_47_sfloat_110_64
    FP_52_sfloat_115_32 = FPstate_q0_58_32 + FPstate_q0_58_32
    FP_53_sfloat_116_32 = FPstate_q1_59_32 + FPstate_q1_59_32
    FP_54_sfloat_117_32 = FPstate_q2_60_32 + FPstate_q2_60_32
    FP_55_sfloat_118_32 = FPstate_q3_61_32 + FPstate_q3_61_32
    FP_56_sfloat_119_32 = FPstate_q0_58_32 * FPstate_q0_58_32
    FP_57_sfloat_120_32 = FPstate_q0_58_32 * FPstate_q1_59_32
    FP_58_sfloat_121_32 = FPstate_q0_58_32 * FPstate_q2_60_32
    FP_59_sfloat_122_32 = FPstate_q1_59_32 * FPstate_q1_59_32
    FP_60_sfloat_123_32 = FPstate_q1_59_32 * FPstate_q3_61_32
    FP_61_sfloat_124_32 = FPstate_q2_60_32 * FPstate_q2_60_32
    FP_62_sfloat_125_32 = FPstate_q3_61_32 * FPstate_q3_61_32
    FP_63_sfloat_126_32 = 1.0 / FP_11_sfloat_74_32
    FP_64_sfloat_127_32 = 1.0 / FP_25_sfloat_88_32
    FP_65_sfloat_128_32 = FPstack_mx_8_47_32 * FP_64_sfloat_127_32
    FP_66_sfloat_129_32 = FPstack_my_12_48_32 * FP_64_sfloat_127_32
    FP_67_sfloat_130_32 = FPstack_mz_16_49_32 * FP_64_sfloat_127_32
    FP_68_sfloat_131_32 = FP_52_sfloat_115_32 * FP_65_sfloat_128_32
    FP_69_sfloat_132_32 = FP_52_sfloat_115_32 * FP_66_sfloat_129_32
    FP_70_sfloat_133_32 = FP_52_sfloat_115_32 * FP_67_sfloat_130_32
    FP_71_sfloat_134_32 = FP_53_sfloat_116_32 * FP_65_sfloat_128_32
    FP_72_sfloat_135_32 = FP_56_sfloat_119_32 * FP_65_sfloat_128_32
    FP_73_sfloat_136_32 = FPstate_q3_61_32 * FP_69_sfloat_132_32
    FP_74_sfloat_137_32 = FP_72_sfloat_135_32 - FP_73_sfloat_136_32
    FP_75_sfloat_138_32 = FPstate_q2_60_32 * FP_70_sfloat_133_32
    FP_76_sfloat_139_32 = FP_74_sfloat_137_32 + FP_75_sfloat_138_32
    FP_77_sfloat_140_32 = FP_59_sfloat_122_32 * FP_65_sfloat_128_32
    FP_78_sfloat_141_32 = FP_76_sfloat_139_32 + FP_77_sfloat_140_32
    FP_79_sfloat_142_32 = FP_53_sfloat_116_32 * FP_66_sfloat_129_32
    FP_80_sfloat_143_32 = FP_79_sfloat_142_32 * FPstate_q2_60_32
    FP_81_sfloat_144_32 = FP_78_sfloat_141_32 + FP_80_sfloat_143_32
    FP_82_sfloat_145_32 = FP_53_sfloat_116_32 * FP_67_sfloat_130_32
    FP_83_sfloat_146_32 = FP_82_sfloat_145_32 * FPstate_q3_61_32
    FP_84_sfloat_147_32 = FP_81_sfloat_144_32 + FP_83_sfloat_146_32
    FP_85_sfloat_148_32 = FP_61_sfloat_124_32 * FP_65_sfloat_128_32
    FP_86_sfloat_149_32 = FP_84_sfloat_147_32 - FP_85_sfloat_148_32
    FP_87_sfloat_150_32 = FP_62_sfloat_125_32 * FP_65_sfloat_128_32
    FP_88_sfloat_151_32 = FP_86_sfloat_149_32 - FP_87_sfloat_150_32
    FP_89_sfloat_152_32 = FPstate_q3_61_32 * FP_68_sfloat_131_32
    FP_90_sfloat_153_32 = FP_56_sfloat_119_32 * FP_66_sfloat_129_32
    FP_91_sfloat_154_32 = FP_89_sfloat_152_32 + FP_90_sfloat_153_32
    FP_92_sfloat_155_32 = FPstate_q1_59_32 * FP_70_sfloat_133_32
    FP_93_sfloat_156_32 = FP_91_sfloat_154_32 - FP_92_sfloat_155_32
    FP_94_sfloat_157_32 = FPstate_q2_60_32 * FP_71_sfloat_134_32
    FP_95_sfloat_158_32 = FP_93_sfloat_156_32 + FP_94_sfloat_157_32
    FP_96_sfloat_159_32 = FP_59_sfloat_122_32 * FP_66_sfloat_129_32
    FP_97_sfloat_160_32 = FP_95_sfloat_158_32 - FP_96_sfloat_159_32
    FP_98_sfloat_161_32 = FP_61_sfloat_124_32 * FP_66_sfloat_129_32
    FP_99_sfloat_162_32 = FP_97_sfloat_160_32 + FP_98_sfloat_161_32
    FP_100_sfloat_163_32 = FP_54_sfloat_117_32 * FP_67_sfloat_130_32
    FP_101_sfloat_164_32 = FP_100_sfloat_163_32 * FPstate_q3_61_32
    FP_102_sfloat_165_32 = FP_99_sfloat_162_32 + FP_101_sfloat_164_32
    FP_103_sfloat_166_32 = FP_62_sfloat_125_32 * FP_66_sfloat_129_32
    FP_104_sfloat_167_32 = FP_102_sfloat_165_32 - FP_103_sfloat_166_32
    FP_105_sfloat_168_32 = FP_88_sfloat_151_32 * FP_88_sfloat_151_32
    FP_106_sfloat_169_32 = FP_104_sfloat_167_32 * FP_104_sfloat_167_32
    FP_107_sfloat_170_32 = FP_105_sfloat_168_32 + FP_106_sfloat_169_32
    FP_108_sfloat_171_64 = FP_107_sfloat_170_32
    FP_111_sfloat_174_64 = (FP_108_sfloat_171_64) ** (1.0/2.0)
    FP_114_sfloat_177_32 = FP_111_sfloat_174_64
    FP_116_sfloat_179_32 = (-FP_68_sfloat_131_32) * FPstate_q2_60_32
    FP_117_sfloat_180_32 = FPstate_q1_59_32 * FP_69_sfloat_132_32
    FP_118_sfloat_181_32 = FP_116_sfloat_179_32 + FP_117_sfloat_180_32
    FP_119_sfloat_182_32 = FP_56_sfloat_119_32 * FP_67_sfloat_130_32
    FP_120_sfloat_183_32 = FP_118_sfloat_181_32 + FP_119_sfloat_182_32
    FP_121_sfloat_184_32 = FPstate_q3_61_32 * FP_71_sfloat_134_32
    FP_122_sfloat_185_32 = FP_120_sfloat_183_32 + FP_121_sfloat_184_32
    FP_123_sfloat_186_32 = FP_59_sfloat_122_32 * FP_67_sfloat_130_32
    FP_124_sfloat_187_32 = FP_122_sfloat_185_32 - FP_123_sfloat_186_32
    FP_125_sfloat_188_32 = FP_54_sfloat_117_32 * FP_66_sfloat_129_32
    FP_126_sfloat_189_32 = FP_125_sfloat_188_32 * FPstate_q3_61_32
    FP_127_sfloat_190_32 = FP_124_sfloat_187_32 + FP_126_sfloat_189_32
    FP_128_sfloat_191_32 = FP_61_sfloat_124_32 * FP_67_sfloat_130_32
    FP_129_sfloat_192_32 = FP_127_sfloat_190_32 - FP_128_sfloat_191_32
    FP_130_sfloat_193_32 = FP_62_sfloat_125_32 * FP_67_sfloat_130_32
    FP_131_sfloat_194_32 = FP_129_sfloat_192_32 + FP_130_sfloat_193_32
    FP_132_sfloat_195_32 = FP_131_sfloat_194_32 + FP_131_sfloat_194_32
    FP_133_sfloat_196_32 = FP_60_sfloat_123_32 + FP_60_sfloat_123_32
    FP_134_sfloat_197_32 = FPstate_q2_60_32 * FP_52_sfloat_115_32
    FP_135_sfloat_198_32 = FP_133_sfloat_196_32 - FP_134_sfloat_197_32
    FP_136_sfloat_199_32 = FPreg_ax_r0_0_32 * FP_63_sfloat_126_32
    FP_137_sfloat_200_32 = FP_135_sfloat_198_32 - FP_136_sfloat_199_32
    FP_138_sfloat_201_32 = FPstate_q3_61_32 * FP_54_sfloat_117_32
    FP_139_sfloat_202_32 = FP_57_sfloat_120_32 + FP_57_sfloat_120_32
    FP_140_sfloat_203_32 = FP_138_sfloat_201_32 + FP_139_sfloat_202_32
    FP_141_sfloat_204_32 = FPreg_ay_r1_1_32 * FP_63_sfloat_126_32
    FP_142_sfloat_205_32 = FP_140_sfloat_203_32 - FP_141_sfloat_204_32
    FP_143_sfloat_206_32 = FPstate_q2_60_32 * FP_131_sfloat_194_32
    FP_144_sfloat_207_32 = 0.5 - FP_61_sfloat_124_32
    FP_145_sfloat_208_32 = FP_144_sfloat_207_32 - FP_62_sfloat_125_32
    FP_146_sfloat_209_32 = FP_145_sfloat_208_32 * FP_114_sfloat_177_32
    FP_147_sfloat_210_32 = FP_60_sfloat_123_32 - FP_58_sfloat_121_32
    FP_148_sfloat_211_32 = FP_147_sfloat_210_32 * FP_131_sfloat_194_32
    FP_149_sfloat_212_32 = FP_146_sfloat_209_32 + FP_148_sfloat_211_32
    FP_150_sfloat_213_32 = FP_149_sfloat_212_32 - FP_65_sfloat_128_32
    FP_151_sfloat_214_32 = FPstate_q1_59_32 * FP_131_sfloat_194_32
    FP_152_sfloat_215_32 = FPstate_q1_59_32 * FPstate_q2_60_32
    FP_153_sfloat_216_32 = FPstate_q0_58_32 * FPstate_q3_61_32
    FP_154_sfloat_217_32 = FP_152_sfloat_215_32 - FP_153_sfloat_216_32
    FP_155_sfloat_218_32 = FP_154_sfloat_217_32 * FP_114_sfloat_177_32
    FP_156_sfloat_219_32 = FPstate_q2_60_32 * FPstate_q3_61_32
    FP_157_sfloat_220_32 = FP_156_sfloat_219_32 + FP_57_sfloat_120_32
    FP_158_sfloat_221_32 = FP_157_sfloat_220_32 * FP_131_sfloat_194_32
    FP_159_sfloat_222_32 = FP_155_sfloat_218_32 + FP_158_sfloat_221_32
    FP_160_sfloat_223_32 = FP_159_sfloat_222_32 - FP_66_sfloat_129_32
    FP_161_sfloat_224_32 = FPstate_q2_60_32 * FP_114_sfloat_177_32
    FP_162_sfloat_225_32 = 0.5 - FP_59_sfloat_122_32
    FP_163_sfloat_226_32 = FP_162_sfloat_225_32 - FP_61_sfloat_124_32
    FP_164_sfloat_227_32 = FP_163_sfloat_226_32 * FP_131_sfloat_194_32
    FP_165_sfloat_228_32 = FP_58_sfloat_121_32 + FP_60_sfloat_123_32
    FP_166_sfloat_229_32 = FP_165_sfloat_228_32 * FP_114_sfloat_177_32
    FP_167_sfloat_230_32 = FP_164_sfloat_227_32 + FP_166_sfloat_229_32
    FP_168_sfloat_231_32 = FP_167_sfloat_230_32 - FP_67_sfloat_130_32
    FP_169_sfloat_232_32 = (-FP_54_sfloat_117_32) * FP_137_sfloat_200_32
    FP_170_sfloat_233_32 = FP_53_sfloat_116_32 * FP_142_sfloat_205_32
    FP_171_sfloat_234_32 = FP_169_sfloat_232_32 + FP_170_sfloat_233_32
    FP_172_sfloat_235_32 = FP_143_sfloat_206_32 * FP_150_sfloat_213_32
    FP_173_sfloat_236_32 = FP_171_sfloat_234_32 - FP_172_sfloat_235_32
    FP_174_sfloat_237_32 = FPstate_q3_61_32 * (-FP_114_sfloat_177_32)
    FP_175_sfloat_238_32 = FP_174_sfloat_237_32 + FP_151_sfloat_214_32
    FP_176_sfloat_239_32 = FP_175_sfloat_238_32 * FP_160_sfloat_223_32
    FP_177_sfloat_240_32 = FP_173_sfloat_236_32 + FP_176_sfloat_239_32
    FP_178_sfloat_241_32 = FP_161_sfloat_224_32 * FP_168_sfloat_231_32
    FP_179_sfloat_242_32 = FP_177_sfloat_240_32 + FP_178_sfloat_241_32
    FP_180_sfloat_243_32 = FP_59_sfloat_122_32 + FP_59_sfloat_122_32
    FP_181_sfloat_244_32 = 1.0 - FP_180_sfloat_243_32
    FP_182_sfloat_245_32 = FP_61_sfloat_124_32 + FP_61_sfloat_124_32
    FP_183_sfloat_246_32 = FP_181_sfloat_244_32 - FP_182_sfloat_245_32
    FP_184_sfloat_247_32 = FPreg_az_r2_2_32 * FP_63_sfloat_126_32
    FP_185_sfloat_248_32 = FP_183_sfloat_246_32 - FP_184_sfloat_247_32
    FP_186_sfloat_249_32 = FPstate_q3_61_32 * FP_131_sfloat_194_32
    FP_187_sfloat_250_32 = FPstate_q0_58_32 * FP_131_sfloat_194_32
    FP_188_sfloat_251_32 = FP_55_sfloat_118_32 * FP_137_sfloat_200_32
    FP_189_sfloat_252_32 = FP_52_sfloat_115_32 * FP_142_sfloat_205_32
    FP_190_sfloat_253_32 = FP_188_sfloat_251_32 + FP_189_sfloat_252_32
    FP_191_sfloat_254_32 = FPstate_q1_59_32 * 4.0
    FP_192_sfloat_255_32 = FP_191_sfloat_254_32 * FP_185_sfloat_248_32
    FP_193_sfloat_256_32 = FP_190_sfloat_253_32 - FP_192_sfloat_255_32
    FP_194_sfloat_257_32 = FP_150_sfloat_213_32 * FP_186_sfloat_249_32
    FP_195_sfloat_258_32 = FP_193_sfloat_256_32 + FP_194_sfloat_257_32
    FP_196_sfloat_259_32 = FP_161_sfloat_224_32 + FP_187_sfloat_250_32
    FP_197_sfloat_260_32 = FP_196_sfloat_259_32 * FP_160_sfloat_223_32
    FP_198_sfloat_261_32 = FP_195_sfloat_258_32 + FP_197_sfloat_260_32
    FP_199_sfloat_262_32 = FPstate_q3_61_32 * FP_114_sfloat_177_32
    FP_200_sfloat_263_32 = FPstate_q1_59_32 * FP_132_sfloat_195_32
    FP_201_sfloat_264_32 = FP_199_sfloat_262_32 - FP_200_sfloat_263_32
    FP_202_sfloat_265_32 = FP_201_sfloat_264_32 * FP_168_sfloat_231_32
    FP_203_sfloat_266_32 = FP_198_sfloat_261_32 + FP_202_sfloat_265_32
    FP_204_sfloat_267_32 = FP_114_sfloat_177_32 + FP_114_sfloat_177_32
    FP_205_sfloat_268_32 = FPstate_q1_59_32 * FP_114_sfloat_177_32
    FP_206_sfloat_269_32 = (-FP_52_sfloat_115_32) * FP_137_sfloat_200_32
    FP_207_sfloat_270_32 = FP_55_sfloat_118_32 * FP_142_sfloat_205_32
    FP_208_sfloat_271_32 = FP_206_sfloat_269_32 + FP_207_sfloat_270_32
    FP_209_sfloat_272_32 = FPstate_q2_60_32 * 4.0
    FP_210_sfloat_273_32 = FP_209_sfloat_272_32 * FP_185_sfloat_248_32
    FP_211_sfloat_274_32 = FP_208_sfloat_271_32 - FP_210_sfloat_273_32
    FP_212_sfloat_275_32 = FPstate_q2_60_32 * (-FP_204_sfloat_267_32)
    FP_213_sfloat_276_32 = FP_212_sfloat_275_32 - FP_187_sfloat_250_32
    FP_214_sfloat_277_32 = FP_213_sfloat_276_32 * FP_150_sfloat_213_32
    FP_215_sfloat_278_32 = FP_211_sfloat_274_32 + FP_214_sfloat_277_32
    FP_216_sfloat_279_32 = FP_186_sfloat_249_32 + FP_205_sfloat_268_32
    FP_217_sfloat_280_32 = FP_216_sfloat_279_32 * FP_160_sfloat_223_32
    FP_218_sfloat_281_32 = FP_215_sfloat_278_32 + FP_217_sfloat_280_32
    FP_219_sfloat_282_32 = FPstate_q0_58_32 * FP_114_sfloat_177_32
    FP_220_sfloat_283_32 = FPstate_q2_60_32 * FP_132_sfloat_195_32
    FP_221_sfloat_284_32 = FP_219_sfloat_282_32 - FP_220_sfloat_283_32
    FP_222_sfloat_285_32 = FP_221_sfloat_284_32 * FP_168_sfloat_231_32
    FP_223_sfloat_286_32 = FP_218_sfloat_281_32 + FP_222_sfloat_285_32
    FP_224_sfloat_287_32 = FP_53_sfloat_116_32 * FP_137_sfloat_200_32
    FP_225_sfloat_288_32 = FP_54_sfloat_117_32 * FP_142_sfloat_205_32
    FP_226_sfloat_289_32 = FP_224_sfloat_287_32 + FP_225_sfloat_288_32
    FP_227_sfloat_290_32 = FPstate_q3_61_32 * (-FP_204_sfloat_267_32)
    FP_228_sfloat_291_32 = FP_227_sfloat_290_32 + FP_151_sfloat_214_32
    FP_229_sfloat_292_32 = FP_228_sfloat_291_32 * FP_150_sfloat_213_32
    FP_230_sfloat_293_32 = FP_226_sfloat_289_32 + FP_229_sfloat_292_32
    FP_231_sfloat_294_32 = FPstate_q0_58_32 * (-FP_114_sfloat_177_32)
    FP_232_sfloat_295_32 = FP_231_sfloat_294_32 + FP_143_sfloat_206_32
    FP_233_sfloat_296_32 = FP_232_sfloat_295_32 * FP_160_sfloat_223_32
    FP_234_sfloat_297_32 = FP_230_sfloat_293_32 + FP_233_sfloat_296_32
    FP_235_sfloat_298_32 = FP_168_sfloat_231_32 * FP_205_sfloat_268_32
    FP_236_sfloat_299_32 = FP_234_sfloat_297_32 + FP_235_sfloat_298_32
    FP_237_sfloat_300_32 = FP_179_sfloat_242_32 * FP_179_sfloat_242_32
    FP_238_sfloat_301_32 = FP_203_sfloat_266_32 * FP_203_sfloat_266_32
    FP_239_sfloat_302_32 = FP_237_sfloat_300_32 + FP_238_sfloat_301_32
    FP_240_sfloat_303_32 = FP_223_sfloat_286_32 * FP_223_sfloat_286_32
    FP_241_sfloat_304_32 = FP_239_sfloat_302_32 + FP_240_sfloat_303_32
    FP_242_sfloat_305_32 = FP_236_sfloat_299_32 * FP_236_sfloat_299_32
    FP_243_sfloat_306_32 = FP_241_sfloat_304_32 + FP_242_sfloat_305_32
    FP_244_sfloat_307_64 = FP_243_sfloat_306_32
    FP_247_sfloat_310_64 = (FP_244_sfloat_307_64) ** (1.0/2.0)
    FP_250_sfloat_313_32 = FP_247_sfloat_310_64
    FP_252_sfloat_315_32 = 1.0 / FP_250_sfloat_313_32
    FP_253_sfloat_316_32 = (-FPstate_q1_59_32) * FP_34_sfloat_97_32
    FP_254_sfloat_317_32 = FPstate_q2_60_32 * FP_42_sfloat_105_32
    FP_255_sfloat_318_32 = FP_253_sfloat_316_32 - FP_254_sfloat_317_32
    FP_256_sfloat_319_32 = FPstate_q3_61_32 * FP_50_sfloat_113_32
    FP_257_sfloat_320_32 = FP_255_sfloat_318_32 - FP_256_sfloat_319_32
    FP_258_sfloat_321_32 = FP_257_sfloat_320_32 * 0.5
    FP_259_sfloat_322_32 = FP_179_sfloat_242_32 * FP_252_sfloat_315_32
    FP_260_sfloat_323_32 = FP_259_sfloat_322_32 * FPstate_beta_56_32
    FP_261_sfloat_324_32 = FP_258_sfloat_321_32 - FP_260_sfloat_323_32
    FP_262_sfloat_325_32 = FP_261_sfloat_324_32 * FPstate_deltat_57_32
    FP_263_sfloat_326_32 = FP_262_sfloat_325_32 + FPstate_q0_58_32
    FP_264_sfloat_327_32 = FPstate_q0_58_32 * FP_34_sfloat_97_32
    FP_265_sfloat_328_32 = FPstate_q2_60_32 * FP_50_sfloat_113_32
    FP_266_sfloat_329_32 = FP_264_sfloat_327_32 + FP_265_sfloat_328_32
    FP_267_sfloat_330_32 = FPstate_q3_61_32 * FP_42_sfloat_105_32
    FP_268_sfloat_331_32 = FP_266_sfloat_329_32 - FP_267_sfloat_330_32
    FP_269_sfloat_332_32 = FP_268_sfloat_331_32 * 0.5
    FP_270_sfloat_333_32 = FP_203_sfloat_266_32 * FP_252_sfloat_315_32
    FP_271_sfloat_334_32 = FP_270_sfloat_333_32 * FPstate_beta_56_32
    FP_272_sfloat_335_32 = FP_269_sfloat_332_32 - FP_271_sfloat_334_32
    FP_273_sfloat_336_32 = FP_272_sfloat_335_32 * FPstate_deltat_57_32
    FP_274_sfloat_337_32 = FP_273_sfloat_336_32 + FPstate_q1_59_32
    FP_275_sfloat_338_32 = FPstate_q0_58_32 * FP_42_sfloat_105_32
    FP_276_sfloat_339_32 = FPstate_q1_59_32 * FP_50_sfloat_113_32
    FP_277_sfloat_340_32 = FP_275_sfloat_338_32 - FP_276_sfloat_339_32
    FP_278_sfloat_341_32 = FPstate_q3_61_32 * FP_34_sfloat_97_32
    FP_279_sfloat_342_32 = FP_277_sfloat_340_32 + FP_278_sfloat_341_32
    FP_280_sfloat_343_32 = FP_279_sfloat_342_32 * 0.5
    FP_281_sfloat_344_32 = FP_223_sfloat_286_32 * FP_252_sfloat_315_32
    FP_282_sfloat_345_32 = FP_281_sfloat_344_32 * FPstate_beta_56_32
    FP_283_sfloat_346_32 = FP_280_sfloat_343_32 - FP_282_sfloat_345_32
    FP_284_sfloat_347_32 = FP_283_sfloat_346_32 * FPstate_deltat_57_32
    FP_285_sfloat_348_32 = FP_284_sfloat_347_32 + FPstate_q2_60_32
    FP_286_sfloat_349_32 = FPstate_q0_58_32 * FP_50_sfloat_113_32
    FP_287_sfloat_350_32 = FPstate_q1_59_32 * FP_42_sfloat_105_32
    FP_288_sfloat_351_32 = FP_286_sfloat_349_32 + FP_287_sfloat_350_32
    FP_289_sfloat_352_32 = FPstate_q2_60_32 * FP_34_sfloat_97_32
    FP_290_sfloat_353_32 = FP_288_sfloat_351_32 - FP_289_sfloat_352_32
    FP_291_sfloat_354_32 = FP_290_sfloat_353_32 * 0.5
    FP_292_sfloat_355_32 = FP_236_sfloat_299_32 * FP_252_sfloat_315_32
    FP_293_sfloat_356_32 = FP_292_sfloat_355_32 * FPstate_beta_56_32
    FP_294_sfloat_357_32 = FP_291_sfloat_354_32 - FP_293_sfloat_356_32
    FP_295_sfloat_358_32 = FP_294_sfloat_357_32 * FPstate_deltat_57_32
    FP_296_sfloat_359_32 = FP_295_sfloat_358_32 + FPstate_q3_61_32
    FP_297_sfloat_360_32 = FP_263_sfloat_326_32 * FP_263_sfloat_326_32
    FP_298_sfloat_361_32 = FP_274_sfloat_337_32 * FP_274_sfloat_337_32
    FP_299_sfloat_362_32 = FP_297_sfloat_360_32 + FP_298_sfloat_361_32
    FP_300_sfloat_363_32 = FP_285_sfloat_348_32 * FP_285_sfloat_348_32
    FP_301_sfloat_364_32 = FP_299_sfloat_362_32 + FP_300_sfloat_363_32
    FP_302_sfloat_365_32 = FP_296_sfloat_359_32 * FP_296_sfloat_359_32
    FP_303_sfloat_366_32 = FP_301_sfloat_364_32 + FP_302_sfloat_365_32
    FP_304_sfloat_367_64 = FP_303_sfloat_366_32
    FP_307_sfloat_370_64 = (FP_304_sfloat_367_64) ** (1.0/2.0)
    FP_310_sfloat_373_32 = FP_307_sfloat_370_64
    FP_312_sfloat_375_32 = 1.0 / FP_310_sfloat_373_32
    FP_313_sfloat_376_32 = FP_263_sfloat_326_32 * FP_312_sfloat_375_32
    FP_314_sfloat_377_32 = FP_274_sfloat_337_32 * FP_312_sfloat_375_32
    FP_315_sfloat_378_32 = FP_285_sfloat_348_32 * FP_312_sfloat_375_32
    FP_316_sfloat_379_32 = FP_296_sfloat_359_32 * FP_312_sfloat_375_32
    FP_317_sfloat_380_32 = FP_314_sfloat_377_32 * FP_316_sfloat_379_32
    FP_318_sfloat_381_32 = FP_313_sfloat_376_32 * FP_315_sfloat_378_32
    FP_319_sfloat_382_32 = FP_317_sfloat_380_32 - FP_318_sfloat_381_32
    FP_320_sfloat_383_32 = FP_319_sfloat_382_32 + FP_319_sfloat_382_32
    FP_321_sfloat_384_64 = FP_320_sfloat_383_32
    FP_324_sfloat_387_64 = math.asin(FP_321_sfloat_384_64)
    FP_327_sfloat_390_64 = FP_324_sfloat_387_64 * 180.0
    FP_330_sfloat_393_64 = FP_327_sfloat_390_64 / 3.141592653589793
    FP_333_sfloat_396_32 = FP_330_sfloat_393_64
    FP_335_sfloat_398_32 = FP_313_sfloat_376_32 * FP_313_sfloat_376_32
    FP_336_sfloat_399_32 = FP_314_sfloat_377_32 * FP_314_sfloat_377_32
    FP_337_sfloat_400_32 = FP_335_sfloat_398_32 - FP_336_sfloat_399_32
    FP_338_sfloat_401_32 = FP_315_sfloat_378_32 * FP_315_sfloat_378_32
    FP_339_sfloat_402_32 = FP_337_sfloat_400_32 - FP_338_sfloat_401_32
    FP_340_sfloat_403_32 = FP_316_sfloat_379_32 * FP_316_sfloat_379_32
    FP_341_sfloat_404_32 = FP_339_sfloat_402_32 + FP_340_sfloat_403_32
    FP_342_sfloat_405_64 = FP_341_sfloat_404_32
    FP_345_sfloat_408_32 = FP_313_sfloat_376_32 * FP_314_sfloat_377_32
    FP_346_sfloat_409_32 = FP_315_sfloat_378_32 * FP_316_sfloat_379_32
    FP_347_sfloat_410_32 = FP_345_sfloat_408_32 + FP_346_sfloat_409_32
    FP_348_sfloat_411_32 = FP_347_sfloat_410_32 + FP_347_sfloat_410_32
    FP_349_sfloat_412_64 = FP_348_sfloat_411_32
    FP_352_sfloat_415_64 = math.atan2(FP_349_sfloat_412_64,
   FP_342_sfloat_405_64)
    FP_355_sfloat_418_64 = FP_352_sfloat_415_64 * 180.0
    FP_358_sfloat_421_64 = FP_355_sfloat_418_64 / 3.141592653589793
    FP_361_sfloat_424_32 = FP_358_sfloat_421_64
    FP_363_sfloat_426_32 = FP_313_sfloat_376_32 * FP_313_sfloat_376_32
    FP_364_sfloat_427_32 = FP_314_sfloat_377_32 * FP_314_sfloat_377_32
    FP_365_sfloat_428_32 = FP_363_sfloat_426_32 + FP_364_sfloat_427_32
    FP_366_sfloat_429_32 = FP_315_sfloat_378_32 * FP_315_sfloat_378_32
    FP_367_sfloat_430_32 = FP_365_sfloat_428_32 - FP_366_sfloat_429_32
    FP_368_sfloat_431_32 = FP_316_sfloat_379_32 * FP_316_sfloat_379_32
    FP_369_sfloat_432_32 = FP_367_sfloat_430_32 - FP_368_sfloat_431_32
    FP_370_sfloat_433_64 = FP_369_sfloat_432_32
    FP_373_sfloat_436_32 = FP_314_sfloat_377_32 * FP_315_sfloat_378_32
    FP_374_sfloat_437_32 = FP_313_sfloat_376_32 * FP_316_sfloat_379_32
    FP_375_sfloat_438_32 = FP_373_sfloat_436_32 + FP_374_sfloat_437_32
    FP_376_sfloat_439_32 = FP_375_sfloat_438_32 + FP_375_sfloat_438_32
    FP_377_sfloat_440_64 = FP_376_sfloat_439_32
    FP_380_sfloat_443_64 = math.atan2(FP_377_sfloat_440_64,
   FP_370_sfloat_433_64)
    FP_383_sfloat_446_64 = FP_380_sfloat_443_64 * 180.0
    FP_386_sfloat_449_64 = FP_383_sfloat_446_64 / 3.141592653589793
    FP_389_sfloat_452_32 = FP_386_sfloat_449_64

    return [
        FP_313_sfloat_376_32,
        FP_314_sfloat_377_32,
        FP_315_sfloat_378_32,
        FP_316_sfloat_379_32,
        FP_333_sfloat_396_32,
        FP_361_sfloat_424_32,
        FP_389_sfloat_452_32
    ]
```

Listing A.4: Result of the Drone differential equation lifting

Appendix B

Installation Guide

Appendix B has an installation guide for the main tools used in the program. The installation of both Python 2 and 3 will be omitted, but note that they are necessary, and for most program installations a virtual environment is recommended.

B.1 Installation

B.1.1 virtualenvwrapper

For the installation of virtualenvwrapper it is only necessary to install it from the Python package manager. Note that, it will need to be installed both for Python2 and Python3, which may require the use of the command with *pip3*.

```
$ pip install virtualenvwrapper
```

Listing B.1: Virtualenvwrapper installation for Python2/3

Once installed, to create a new environment and change between environments just execute the following commands:

```
$ mkvirtualenv env2       # Creation of a new environment
$ workon env1             # Switch between environments
```

Listing B.2: Virtualenvwrapper environment creation & switch

And once the environment is set, the installation of libraries is as usual.

B.1.2 Jupyter Notebook

To install Jupyter Notebook:

```
$ pip install notebook
```

Listing B.3: Jupyter notebook installation

IPython

IPython should be included with Jupyter Notebook, in case it is not included,the installation procedure is through the Python package manager:

```
$ pip3 install ipython
```

Listing B.4: IPython installation

B.1.3 Ghidra

To install Ghidra, some packages and installations must be installed beforehand:

- **Java Development Kit 11**: Retrieve the latest release from `https://www.oracle.com/java/technologies/javase/jdk11-archive-downloads.html`. Install it and make sure that it is added to the `PATH`.

After installing Java, install Ghidra by following the next steps:

- Dwonload Ghidra's latest release from the release page `https://github.com/NationalSecurityAgency/ghidra/releases`.
- Extract the downloaded program into a known folder.

To run ghidra, access the extracted folder thorugh command-line and execute

```
$ ./ghidraRun
```

Listing B.5: Ghidra installation

Ghidra bridge

To install ghidra_bridge:

- Install the ghidra_bridge package:

```
$ pip install ghidra_bridge
```

Listing B.6: Ghidra_bridge installation

- Install the server scripts to a directory on the Ghidra's script path (e.g., `/ghidra_scripts`), or you can add more directories in the Ghidra Script Manager by clicking the "3 line" button left of the big red "plus" at the top of the Script Manager).

```
$ python3 -m ghidra_bridge.install_server ~/ghidra_scripts
```

Listing B.7: Ghidra_bridge script installation

B.1.4 radare2

Install radare2 from git:

```
$ git clone https://github.com/radareorg/radare2
$ cd radare2
$ sys/install.sh
```

Listing B.8: Radare2 installation

r2pipe

Install r2pipe through the Python manager:

```
$ pip3 install r2pipe
```

Listing B.9: R2pipe installation

r2ghidra

Install r2ghidra using the radare2 package manager:

```
$ r2pm update
$ r2pm -ci r2ghidra
```

Listing B.10: R2ghidra installation

Note: make sure that in the folder `/.local/share/radare2/plugins` there is a sub-folder named `r2ghidra-sleigh`. If not, decompress the file named `r2ghidra_sleigh-x.x.x.zip` and change the resulting unzipped folder to `r2ghidra-sleigh`.

B.1.5 angr

install angr using the Python package manager:

```
$ pip3 install angr
```

Listing B.11: Angr installation

Note: claripy should be installed as a dependency of angr. If it is not available or missing, please reinstall angr.

B.1.6 DaDRA

install DaDRA using the Python package manager:

```
$ pip3 install --upgrade dadra
```

Listing B.12: DaDRA installation

B.1.7 ICSREF

ICSREF is more complicated to install. Make sure that Python 2 is used for the installation of this tool. Note: ICSREF is built on Python 2 and, as such, a virtual environment should be used for the installation (mkvirtualenv is recommended).
The steps are as follows:

- Install the system dependencies:

```
$ sudo apt install git python-pip libcapstone3 python-dev
  libffi-dev build-essential virtualenvwrapper graphviz
  libgraphviz-dev graphviz-dev pkg-config
```

Listing B.13: ICSREF dependency installation

- Download ICSREF from git:

```
$ git clone https://github.com/momalab/ICSREF.git
$ cd ICSREF
```

Listing B.14: ICSREF download

"'

- Install the python package dependencies from wheelhouse:

```
$ pip install --no-index --find-links=wheelhouse -r
  requirements.txt
```

Listing B.15: ICSREF wheelhouse dependency installation

"'

- Create a bash alias:

```
$ echo -e "\n# ICSREF alias\nalias icsref='workon icsref &&
   python `pwd`/icsref/icsref.py'\n" >> ~/.bash_aliases &&
  source ~/.bashrc
```

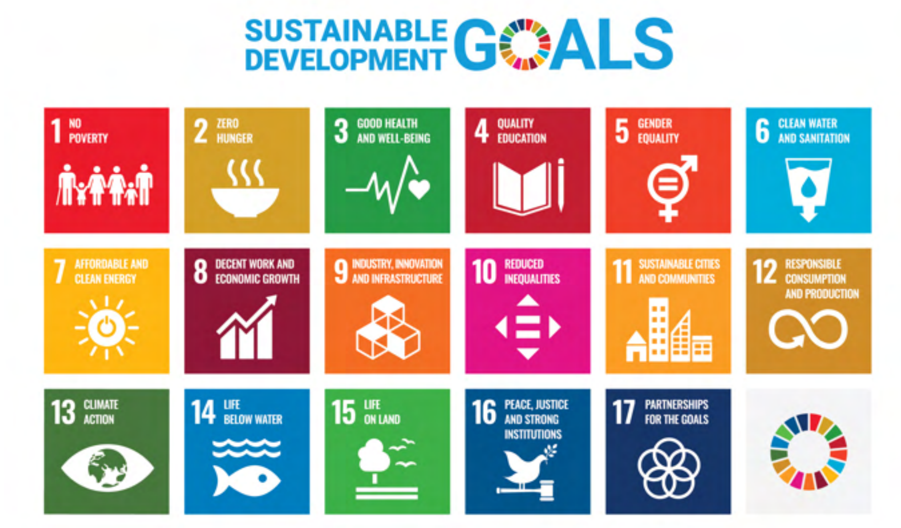
Listing B.16: ICSREF bash alias creation

Appendix C

Sustainable Development Goals

This chapter explains the relationship this final research thesis has with the Sustainable Development goals, approved by the United Nations in 2015, as part of the 20130 Sustainable Agenda [53].

Each of these goals (Figure C.1) consists of a set of objectives that must be achieved in the next 10 years, up to 2030. Adopting these goals by governments and entities should improve poverty in the world, protect the planet, and ensure prosperity.



SUSTAINABLE DEVELOPMENT GOALS
1 NO POVERTY
2 ZERO HUNGER
3 GOOD HEALTH AND WELL-BEING
4 QUALITY EDUCATION
5 GENDER EQUALITY
6 CLEAN WATER AND SANITATION
7 AFFORDABLE AND CLEAN ENERGY
8 DECENT WORK AND ECONOMIC GROWTH
9 INDUSTRY, INNOVATION AND INFRASTRUCTURE
10 REDUCED INEQUALITIES
11 SUSTAINABLE CITIES AND COMMUNITIES
12 RESPONSIBLE CONSUMPTION AND PRODUCTION
13 CLIMATE ACTION
14 LIFE BELOW WATER
15 LIFE ON LAND
16 PEACE, JUSTICE AND STRONG INSTITUTIONS
17 PARTNERSHIPS FOR THE GOALS

Figure C.1: United Nations Sustainable Development Goals

According to the World Commission for Environment and Development of the United Nations, “sustainable development is development that meets the needs of the present without compromising the ability of future generations to meet their own needs, guaranteeing the balance between economic growth, the environment and social wellness" [54].

The UN Sustainable Development Goals could be classified into three distinct categories, depending on their scope, economic, social, and ecological. (Figure C.2). This classification aims to give a new dimension to the SDGs, as all in all, the social and economic SDGs are integrated into the environment. Although their classification does not imply preference of one over the other, it acknowledges that some of the goals are basic and could not exist if the environmental are not followed.

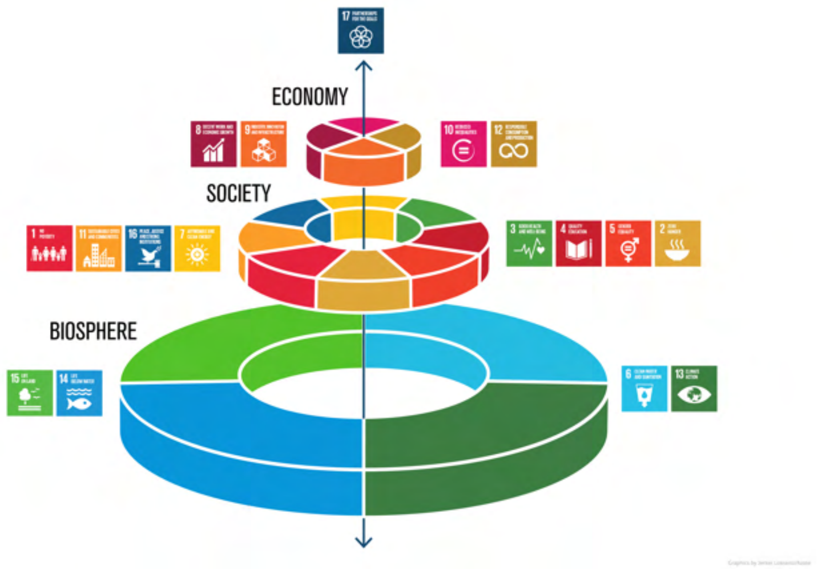


Figure C.2: Sustainable Development Goals categories

Source: Courtesy of Stockholm University - Azote Images [55]

Although at first it would seem that computer applications and software development may not have at first any SDG that matches it, upon further analysis, it is integrated into both SDG 9 and SDG 11.

- **9 - Industry, Innovation, and Infrastructure**.

  This project will innovate in new techniques for the safe implementation and verification of cyber-physical systems. With this development, both industry and infrastructure will be safer. Apart from that, it is part of innovation in the systems verification and security of systems fields. The widespread use of safer embedded systems will avoid unexpected failures and the use of these systems maliciously and, as such, more reliably industry and development.

- **11 - Sustainable Cities and Communities**.

  With the recent development of embedded systems in multiple areas, such as smart cities, building sensorization, and traffic control, it is vital to have the assurance that the implementation is correct. The devices are expected to avoid any harm to people, function properly, and provide safe environments. If any of these three characteristics is violated, the consequences for human development and life, and even the environment, could be catastrophic. With this, the advantage of this tool is that it provides new safeguards in the development and security of embedded systems. Checking and ensuring that these devices do not pose harm will make sustainable environments for human development.

All in all, even if software projects and technologies do not appear to be part of any Sustainable Development Goal, after further analysis, it is possible to see how technology development is both beneficial to society and how it can provide to the goals set by the United Nations.