CAUID: Chip Against Unintended Information Disclosure

Sunday 2nd June, 2024 - 21:34

Georgi Bozhkov
University of Luxembourg
Email: georgi.bozhkov.001@student.uni.lu

This report has been produced under the supervision of:
Bernard Steenis
University of Luxembourg
Email: bernard.steenis@uni.lu

Abstract—**This Bachelor Semester Project develops the concept of a security chip named CAUID (Chip Against Unintended Information Disclosure).**

The scientific deliverable investigates the question of whether hardware-based or software-based antivirus systems are superior. By establishing specific evaluation criteria and detailing various antivirus features, the results are thoroughly analyzed to draw a well-supported conclusion.

In the technical deliverable, an in-depth description of the components comprising CAUID is provided, along with a detailed explanation of the functionality of the chip. Additionally, a simulation is developed to further illustrate the operational aspects of the antivirus system, showcasing its potential effectiveness and practical application.

1. Introduction

The story of antivirus systems goes back to 1971, when Ray Tomlinson, an American computer programmer, developed a program called ”Reaper” with the intention of protecting the first wide-scale packet-switched network and the predecessor of the Internet, ARPANET, from a detected virus called ”Creeper”. Even though the created program was essentially also a virus, made with the goal of finding files corrupted by ”Creeper” and erasing them, this could be considered the first concept of an antivirus system. Since that point on, they have become a necessity for every device that is connected to the Internet.

A method of penetrating the security system of a computer, more commonly referred to as ”hacking”, is an act that allows the gaining of personal information without the consent of an individual. Despite numerous advancements made to various security systems throughout the years, people with technological expertise and malicious intent are still able to frequently manage to get past the barrier-like layers and access confidential information. Additionally, it is inevitable for the occurrence of such unethical events to rise due to the globalisation of technology over time.

Due to the aforementioned increase in danger on the Internet, people have been searching for the most reliable and secure antivirus systems for their devices in order to maximise their safety online. The Chip Against Unintended Information Disclosure, or CAUID for short, aims to be a dependable security system that minimally affects the performance of the device while reducing its chances of being compromised by malicious attacks targeting it.

The task of this BSP is to develop and proof the concept of a convenient and responsive antivirus system that does not interfere with the experience of the user but also enhances the security of the device it is implemented on. The task in the scientific deliverable section is to justify the implementation choice of CAUID, which is done by answering the scientific question, ”Are hardware-based antivirus systems better than software-based ones?” In order to find an answer to the question, a number of features that the security systems acquire are assessed based on given and predefined criteria. Additionally, a table is created, containing the final results, which are then analyzed and provide assistance with effectively reaching a conclusion.

The goal of the technical deliverable of the BSP is to prove that the concept of a CAUID antivirus system is achievable, which is done in two parts. The first part contains a description of the implementation of the chip. It includes an explanation of all the components CAUID is comprised of and what their purpose is, as well as a comprehensive explanation of its working. For the second part of the proof, a Python simulation is developed with the intention of showing how the chip functions under its intended circumstances. Important sections of the code are further interpreted.

2. Project description

2.1. Domains

2.1.1. Scientific. The domain of this scientific deliverable is centered around the knowledge sphere of cybersecurity. As the goal of the deliverable is to compare two types of antivirus systems, it is essential for this section to provide an elementary level of knowledge regarding their working processes. That includes various algorithms and monitoring methods utilized by them.

2.1.2. Technical . The technical deliverable of the BSP focuses on the hardware components of an ordinary computer. In order for CAUID to work as intended, it needs to be able to operate in conjunction with the CPU. For successful communication process, instructions and data buses are to be utilized

2.2. Targeted Deliverables

2.2.1. Scientific deliverables. The goal of the scientific deliverable is to find an answer to the question, ”Are hardware-based antivirus systems better than software-based ones?” This is achieved by providing a definition of what an antivirus is. Subsequently a comprehensive analysis of both types of security systems across various criteria is done. Afterwards, the comparisons are shown in a structured table format, showing a clear comparison between the two types of antivirus systems. Through an assessment of the results, a conclusion is reached.

2.2.2. Technical deliverables. The goal of the technical deliverable is to justify the existence of CAUID as a concept. This is to be done by providing a detailed explanation of the structure of CAUID, including its components and working procedure, as well as mentioning the additional features it acquires. Additionally, a simulation of the working of the chip with the Python programming language is developed and all important sections of the code are to be explained.

3. Pre-requisites

3.1. Scientific pre-requisites

From a scientific standpoint, before beginning with the BSP, a solid understanding of the procedures of an antivirus system is required.

3.2. Technical pre-requisites

From a technical standpoint, before beginning with the BSP, one must have an understanding of how a computer is structured on a physical basis and knowledge about how communication between parts of the computer works. It is important for one to know what the purpose of caches is, as the security chip bears much resemblance to it. The production of this section contains a simulation created in the Python programming language; therefore, an intermediate on the topic is necessary.

4. Scientific Deliverable

Question: Are hardware-based antivirus systems better than software-based ones?

4.1. Requirements

The main objective of this section of the BSP is to arrive at a validated answer to the previously mentioned scientific question. The justified conclusion is the result of an in-depth look at what an antivirus is providing us with the needed information to continue with the project. Afterwards, a thoroughly examined comparison between hardware-based and software-based antivirus systems is done. The observation is theoretically based and analyses the differences between both security systems by individually assessing their performance based on features that they are both accustomed to.

Before beginning the comparison, the upcoming design section is to contain a subsection that includes a set of thoroughly defined criteria that are based on characteristics that outline what a desired antivirus system is. The procedure for the juxtaposition between hardware-based antivirus (HBA) and software-based antivirus (SBA) is as follows:

- **Features and their Integration:** In terms of an antivirus, a feature is a set of algorithms and/or data recognized by the security system with the purpose of increasing the protection of a device against potential threats. The features listed in this comparison are necessary for antivirus systems to be acquainted with in order to establish a secure space against cyberattacks. Before the comparison between the performances, an in-depth definition of each feature is given with the purpose of showing its importance for the creation of such systems. Additionally, after the explanation, a paragraph is provided explaining how the said feature is integrated into both types of antivirus systems.
- **Comparison between HBA and SBA:** The information about how features are integrated into both types of antivirus systems is to be organized into a table within the production section with the goal of easing the analysis procedure. After thoroughly examining the information and by using the predetermined criteria and detailed definitions of each feature, we can accurately compare HBA and SBA. We will highlight the positive and negative traits of both systems and compare their performances, with the aim of reaching a validated answer to the scientific question.

4.2. Design

4.2.1. What is an Antivirus. Before proceeding with the project, it is important to understand what an antivirus is and how it functions, specifically focusing on the pattern matching of executable instruction sequences.

An antivirus is a system designed to detect, prevent, and remove malicious software, also known as malware, from computers and other electronic devices. These systems work by scanning files, programs, and network traffic to identify and eliminate threats that could compromise the security and functionality of a device. One of the primary methods an antivirus uses to identify malware is pattern matching, which involves analyzing executable instruction sequences to find known signatures of malicious code. [1]

There are various types of antivirus systems, however this project focuses on software-based and hardware-based ones. Software-based antivirus systems are installed on the operating system and continuously monitor activities to detect suspicious behavior. Hardware-based antivirus systems, on the other hand, are integrated directly into the computer's hardware. Regardless of the antivirus system, they all follow the same or similar steps:

- **Scanning:** The antivirus scans files, programs, and network traffic for known malware signatures and suspicious patterns.
- **Detection:** When a match is found, the antivirus identifies the file or program as potentially malicious.
- **Quarantine:** The suspicious file or program is isolated to prevent it from causing harm to the system.
- **Removal:** The antivirus attempts to remove the malware from the system.
- **Monitoring:** The antivirus continuously monitors the system to detect and respond to new threats. [2] [3]

Understanding the basic functioning of antivirus systems, is important for appreciating the differences between software-based and hardware-based antivirus systems. This foundational knowledge sets the stage for comparing the two approaches and determining whether hardware-based antivirus systems offer superior protection and efficiency.

4.2.2. Criteria.

In order to give a justification for the results of the comparison between the HBA and SBA systems, it is essential to establish certain criteria for evaluation. These criteria will serve as the foundation with which it is possible to assess the capabilities of both types of antivirus.

Usability The usability of an antivirus system focuses on the scale of difficulty that users experience when interacting with it. Various tasks and factors, such as installing, ease of interface navigation, updating, and configuring, are what this criterion encompasses. The desired security system is user-friendly and does not require proficiency in the sphere of technology in order for one to interact with it. Its installation on a device should be straightforward, as should the user interface. The available features need to be quickly accessible and understandable to the average person in order to avoid confusion and minimize complexity. The usability of an antivirus should also include the possibility of self-repairing whenever an error within the system occurs. Conclusively, the intention of this criterion is to evaluate features of the security system that improve the experience of a user.

Performance Performance refers to the amount of system resources an antivirus software needs to efficiently operate. The utilization includes the consumption of the power of the CPU, RAM, and network bandwidth. For the ideal security system, it is vital to minimize the usage of resources while, in parallel, preserving its capabilities. By optimising the utilization, smooth working and a lack of significant effects on the running of a device are ensured.

Effectiveness In the context of antivirus systems, effectiveness relates to the precision they acquire to detect and eliminate malicious attacks. Making use of various detection methods against malware, such as real-time detection and heuristic analysis, increases the chances of success for the antivirus. The rate of false-positives is another factor covered by the effectiveness of the system. Instances of identifying non-malicious files as a potential threat must be minimal to non-existent. In addition to accuracy, the effectiveness of a cybersecurity system is also based on its responsiveness. Time is of the utmost importance when an attempt of security penetration is initiated; therefore, the antivirus must apply the appropriate countermeasures in a nimble manner.

4.3. Features and their Integration

4.3.1. Real-Time Detection.

Real-time threat detection allows for the immediate identification and response to malicious activities. Software-based antiviruses continuously monitor system activities and incoming files, comparing them against a database of known malware signatures to detect suspicious behavior. [4]

While SBA provides effective real-time protection, its reliance on the operating system can introduce delays in threat detection and response. On the other hand, hardware-based antivirus systems operate independently of the OS, embedded directly into the hardware architecture of the device. This integration enables HBA to perform real-time threat detection with minimal latency, offering immediate identification and mitigation of threats.

4.3.2. Resource Consumption.

System resource consumption directly impacts the overall performance of the computer system. [5]

SBA systems typically require a significant amount of system resources, including CPU processing power and RAM memory, especially during scanning operations. This resource consumption can lead to system slowdowns, increased boot times, and reduced performance, particularly on older or less powerful computers. Additionally, frequent updates and background scanning processes further contribute to resource usage, affecting the usability of SBA. In contrast, HBA generally has low resource consumption because it utilizes dedicated hardware resources for threat detection and mitigation. HBAs have minimal impact on system performance, allowing for smoother operation and an improved user experience, even during intensive tasks or high-demand scenarios.

4.3.3. Resistance to Malware.

Malware developers continuously create sophisticated techniques to evade detection;

therefore, it is important for antivirus software to be able to apply countermeasures as fast as possible. [**?**]

SBA systems primarily rely on methods such as signature-based detection and heuristic analysis to identify malware patterns and behaviors. However, these methods can be easily bypassed by encrypted malware variants that alter their code to evade detection. Additionally, fileless malware can hide from traditional antivirus scanners by exploiting vulnerabilities in the operating system. In contrast, HBA systems offer enhanced resistance to malware evasion techniques. They operate at a lower level of the system, monitoring CPU instructions and system behavior. This enables them to detect and block malware at the hardware level, making it more difficult for malware to evade detection. [6] [7]

4.3.4. Updating. With the aim of ensuring their reliability, antivirus systems require frequent updates with information about newly developed versions of viruses.

In both HBA and SBA, these updates are essential for keeping the antivirus protection current and capable of detecting and neutralizing the latest threats. However, there are differences in their implementation and impact. HBA systems often rely on firmware updates provided by the manufacturer. These updates, while less frequent, are important for the enhancement of the malware detection capabilities of the device. On the other hand, SBA systems receive regular software updates from the vendor, which are more frequent and seamless but require an internet connection in order to be updated.

4.3.5. Behaviour-based Detection. Behavior-based detection in antivirus systems involves monitoring the behavior of programs and processes to identify suspicious or malicious activity that may indicate the presence of malware. [9]

In SBA systems, behavior-based detection is typically implemented through heuristic analysis and machine learning algorithms. These systems analyze the behavior of programs in real-time, looking for anomalies or patterns indicative of malicious behavior. SBA can quickly adapt to new threats by updating their detection algorithms based on the latest behavioral patterns observed. HBA systems approach behavior-based detection differently, as they do not recognize software directly but follow sequences of instructions. Instead, they focus on monitoring the execution of instructions and detecting deviations from expected behavior at the hardware level.

4.3.6. Scheduled Scanning. Scheduled scanning is a feature in antivirus systems that enables users to set specific times for automatic virus scans. [13]

In SBA systems, users can schedule scans to run at designated times, providing convenience and allowing for regular scans without manual intervention. Conversely, HBA systems do not typically support scheduled scanning due to their real-time monitoring approach and lack of software support. HBA continuously monitors system activities, offering immediate threat detection without the need for scheduled scans.

4.3.7. Heuristic Analysis. This approach is used by antivirus systems to detect previously unknown or new malware based on behavioral patterns. [11] [12]

In SBA systems, heuristic analysis involves analyzing the behavior of programs and files to identify potential threats based on their attributes and actions. They use complex algorithms to assess the risk level of files and programs, allowing them to detect and block suspicious activity before it can cause harm. On the other hand, HBA systems approach heuristic analysis differently, as they do not utilize algorithms but rather compare the received instructions with the ones they store in order to detect potential threats.

4.4. Production

This section contains an assessment of the results from section 4.2. Upon the completion of analysis of the data, a solution to the scientific question is reached.

4.4.1. Creation of a Table. In order to adequately address the scientific question posed in this BSP, a table has been created to summarize the results of the comparison between HBA and SBA systems. This table aims to provide a simplified overview of the capabilities and performance of each type of antivirus system, facilitating a comprehensive analysis to answer the scientific question question.

TABLE 1: Comparison of Features between HBA and SBA

Features	HBA	SBA
Real-Time Detection	Yes (OS independent)	Yes (OS dependent)
Resource Consumption	Yes (Minimal)	Yes (Noticeable)
Resistance to Malware	Yes (Strong)	Yes (Dependence on Updates)
Updating	Yes (Manual)	Yes (Automatic)
Behaviour-based Detection	Yes (Instruction Monitoring)	Yes (Software-based Monitoring)
Scheduled Scanning	No (Real-Time Monitoring)	Yes (Convenient, Customizable by User)
Heuristic Analysis	Yes (Instruction Comparison)	Yes (Algorithm Usage)

4.4.2. Comparison between HBA and SBA. As shown in Table 1, HBA systems excel in real-time threat detection due to their integration into the CPU architecture, which enables immediate identification and response to malicious activities. Unlike software-based antivirus, HBA operates at the hardware level, allowing it to monitor instructions from the processor directly. This deep integration allows the hardware-based systems to detect and block malware in real-time, significantly reducing the window of opportunity for malicious activities. By analyzing CPU instructions, HBA systems can detect deviations from expected behavior, making them highly effective in identifying and mitigating threats. Additionally, the hardware-level monitoring provided by HBA systems offers enhanced resistance to malware evasion techniques, such fileless malware. This proactive approach to security makes it more difficult for malware to evade detection, thereby providing a higher level of protection for the system and its users.

One of the strengths shown by HBA systems, as highlighted in Table 1, is their ability to maintain minimal impact on system performance. By utilizing dedicated hardware components for threat detection and mitigation, HBA systems effectively reduce resource consumption compared to their software-based counterparts. This reduction translates into smoother system operation and an overall improved user experience. Furthermore, the integration of HBA systems into the CPU architecture enables them to excel in real-time threat detection and resistance to malware evasion. These hardware-based solutions provide immediate identification and response to malicious activities, making it easier for it to capture malicious attacks.

While HBA systems display superiority in several aspects, including real-time threat detection and malware evasion resistance, they also exhibit limitations. Notably, HBA systems lack certain features such as scheduled scanning and automatic updates. These limitations arise from their real-time monitoring approach and the inherent constraints of their hardware-based design, as indicated in Table 1. Despite these drawbacks, the robust performance and enhanced security offered by HBA systems make them a compelling choice for users seeking advanced protection against malware threats.

In terms of behavior-based detection, Table 1 shows that SBA systems utilize Programs and algorithms to monitor and analyze program behavior in real-time. These systems can quickly adapt to new threats by updating their detection algorithms based on the latest behavioral patterns observed. However, the reliance on software-based algorithms in SBA systems can sometimes lead to false positives or missed detections, especially when dealing with sophisticated malware variants. In contrast, HBA systems, as given in Table 1, follow sequences of CPU instructions and detect deviations from expected behavior at the hardware level. This method of monitoring enables the antivirus system to efficiently detect and respond to suspicious activity without relying on software-based analysis. However, implementing behavior-based detection in hardware can be challenging and may require specialized hardware components, making it less flexible than software-based approaches. Despite these challenges, HBA systems offer robust protection against malware threats by leveraging real-time hardware monitoring and detection capabilities.

4.4.3. HBA vs SBA. When evaluating HBA and SBA systems, it becomes evident that each approach offers distinct advantages and drawbacks. SBA systems excel in their flexibility and ease of updating, allowing for seamless integration with existing operating systems and frequent updates to combat emerging threats. However, they also often suffer from higher resource consumption, leading to system slowdowns and reduced performance, particularly on older or less powerful computers. Moreover, their reliance on software-based algorithms for behavior-based detection may result in false positives or missed detections, compromising their effectiveness in detecting sophisticated malware variants.

On the other hand, HBA systems, boast real-time threat detection capabilities and minimal impact on system performance. By integrating directly into the CPU architecture and monitoring system activities at the hardware level, HBA solutions offer immediate identification and response to malicious activities, enhancing overall system security. Additionally, their lower resource consumption ensures more efficient system operation and improved user experience, even during intensive tasks. While HBA systems may lack certain features, such as scheduled scanning and automatic updates, their robust protection against malware evasion techniques and efficient behavior-based detection mechanisms outweigh these limitations.

With its real-time threat detection, minimal impact on system performance, and efficient behavior-based detection mechanisms, hardware-based antivirus provides robust security solutions unmatched by software-based alternatives. Given these factors, we can reach a solution to the scientific question, which is that HBA is superior against cyber threats.

4.5. Assessment

This section evaluates the achievements made within the scientific deliverable and assesses whether they meet the conditions outlined in the requirement section.

The evaluation section of the BSP successfully provides an appropriate answer to the scientific question. The conclusion is drawn from a thorough analysis based on fundamental criteria essential for evaluation. The features discussed are crucial for antivirus systems, and their integration into both HBA and SBA is comprehensively explained.

However, there are some negative aspects of the scientific deliverable. One significant drawback is that the comparison between the two antivirus systems is entirely theoretical. Although the performance of the systems is described based on justified information, there is no empirical evidence or experiments conducted to support these claims. Various unmentioned improvements regarding both types of antivirus systems could alter their effectiveness, usability, and performance. Additionally, the scientific section only mentions a limited number of features. While these features are important, there are many others that either only one type of antivirus possesses or that both types have, which are not discussed.

5. Technical Deliverable

5.1. Requirements

The goal of the technical section of the BSP is to provide thoroughly justified proof that the concept of CAUID can be effectively implemented. To achieve this, the design section is divided into three parts. Each subsection delves into distinct aspects of the security chip, necessary for a comprehensive understanding of its design and functionality. The architecture, operational mechanisms, and potential applications of the security chip are assessed, providing a detailed overview of its workings and capabilities. The subsections are the following:

- **Components** This section highlights the key components of CAUID that are engaged during its operation. Each part of the chip will be thoroughly described, explaining its purpose and role within the framework of the security chip.

- **Functionality** This section outlines the theoretical working cycle of CAUID, detailing its entire process. It describes how the chip collaborates with the device it is integrated into, how its components work together, the methods they use to initiate tasks, and the steps involved in detecting and responding to potential threats.
  The subsections within this part contain two segments each. The first one elucidates the rationale behind various implementation decisions. The second one, which is the last numberated paragraph, outlines the steps of the working procedure of the HBA from its start of a cycle to its end. It shows what role the aforementioned decision from the first segment plays in the operation of the security chip.

- **Additional Features** This section focuses on design aspects, which are mostly unrelated to the previously described functionality of the chip, but do however increase the quality of the chip. It highlights specific features and implementation choices that enhance the effectiveness of the HBA.

To further validate the concept of CAUID, the production part contains a Python program that demonstrate the functionality of the security chip in action. This program simulates the behavior of the HBA system, and the section provides a detailed explanation of the critical parts of the code, showing how each part contributes to the overall operation of CAUID.

It is important to note that the primary task of this section is solely to show the feasibility of realizing the CAUID concept; therefore, the design and production sections must accurately depict the structure of malware signatures and the functionality of a CPU. However, said depiction is simplified to a certain level, sufficient for ensuring that the chosen components and decisions are valid for correctly establishing the proof of concept.

5.2. Design

Before continuing with the design section of the technical deliverable, it is essential to introduce the concept of signature-based malware, as the functionality of CAUID is centered around detecting and mitigating this type of threat.

Signature Malware is defined by specific patterns in the data and memory addresses that interact with the memory of the CPU and its caches. These patterns consist of unique sequences or behaviors that set malware apart from legitimate data. For instance, malware might access certain memory locations, execute instructions in an unusual sequence, or manipulate data in ways that are atypical for benign software.

5.2.1. Components.

This subsection contains information regarding the most important parts of the security chip used during the working cycle of the HBA.

Comparators The comparators are tasked with juxtaposing the data provided by the CPU with the repository of malware signatures housed within the data storage of CAUID. This process enables CAUID to identify any potential threats or malicious activities, allowing for effective mitigation measures to be implemented.

Data Storage The data storage component is responsible for archiving various predetermined methods of security penetration. It maintains an extensive database of known malware signatures that are used for identifying and mitigating threats. By storing this information, the data storage ensures that CAUID has immediate access to the necessary data to analyse, thereby enhancing the overall effectiveness and responsiveness of the antivirus system.

Control Unit The control unit (CU) is responsible for organizing the operations within CAUID, ensuring efficient threat detection and response. It manages the interaction between the data storage and the comparator, coordinating the analysis of incoming data. When data arrives, the component directs it to the appropriate storage location and then ensures that the comparators receive the necessary data blocks for analysis against known malware signatures stored in the data storage. The CU handles unexpected errors, such as false positives, communication problems, and data corruption, by implementing corrective measures and rerouting processes as needed. This includes rechecking data or instructing the system to re-evaluate specific segments to confirm the presence of threats. Additionally, the CU keeps track of the similarity rate between the result of the comparators and the content of the Data storage and based on the outcome is able to initiate countermeasures. By managing these issues and optimizing the workflow between data storage and comparators, the control unit ensures the stability of the antivirus system and enhances protection against malicious activities, maintaining the operation of CAUID within the CPU.

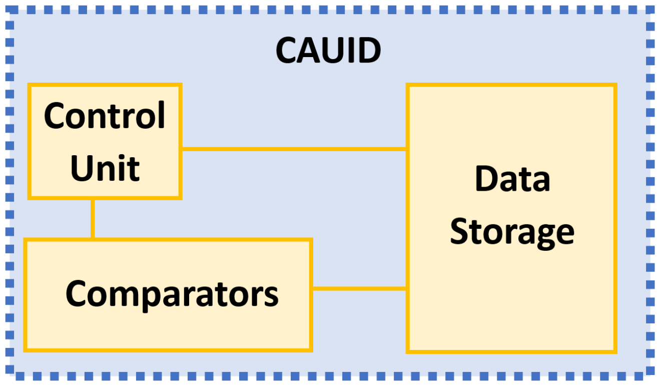


Figure 1: Main components of CAUID

5.3. Functionality

Access to RAM For CAUID to function as intended, it requires transferring content from its data storage into the memory of the device. The two viable options for this are flash memory or random access memory (RAM). Flash memory, a type of non-volatile storage, retains data without power but has slower performance compared to RAM. In contrast, RAM, a volatile memory type, offers significantly faster data access times, which is necessary for the real-time processing demands of CAUID. Due to these superior access speeds and lower latency, RAM is selected as the preferred memory solution for the operations of the security chip.

1. Once the device has been powered on, its CPU is activated and the boot-up sequence has initiated. During this process, the Processor loads the operating system into the RAM. Simultaneously, the security chip executes a similar operation by transferring its stored malware signature data into the volatile memory. This concurrent data transfer process ensures that CAUID has immediate access to the most current system information. By maintaining synchronized data storage between the operating system and the security chip, CAUID can perform real-time comparisons, leveraging techniques to effectively protect the device against potential security threats.

Placement of CAUID The decision to integrate CAUID within the CPU through architectural modifications of the processor ensures a balance between performance and security. This integration provides the chip with direct hardware access to CPU data, disregarding the need for communication via bus interfaces. This direct access facilitates rapid and efficient data analysis, which is essential for real-time monitoring. Furthermore, the implementation of the HBA within the processor includes prioritizing access to the Level 3 (L3) cache, enhancing its ability to analyze CPU operations immediately upon boot-up. The L3 cache contains data shared across all cores, ensuring that all necessary information for comparison is readily available, thereby improving the overall efficiency and effectiveness of the security measures.

2. Following initialization, when the CPU begins to execute instructions, CAUID retrieves data from the L3 cache for analysis without disrupting the operation of the processor.

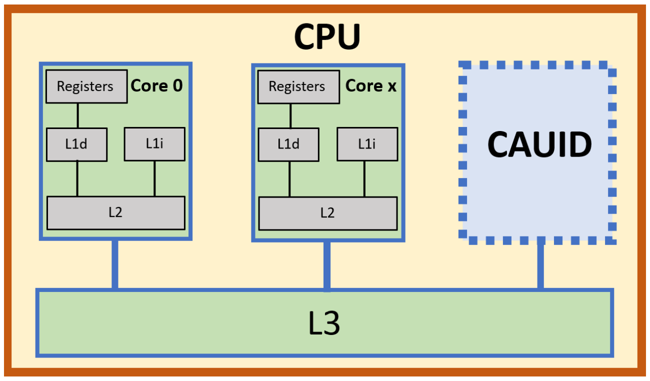


Figure 2: Placement of CAUID within the CPU

Number of Comparators CAUID is equipped with 1000 comparators to efficiently handle and analyze malware signatures. Each comparator has a capacity of 1 KB, making the maximum amount of data analyzed per cycle 1000 KB.

The malware signatures are divided into equal-sized blocks, ensuring that each comparator processes an equal portion of data.

A concern with the operation of CAUID is its ability to match the operation speed of the CPU, which is essential for effective malware detection and overall device performance. If CAUID cannot keep up with the processor, it could lead to delays in malware detection and hinder the device's operation. To address this issue, several features have been incorporated into CAUID's design to optimize processing speed and maintain antivirus efficiency.

While the comparators within CAUID are effective for analyzing malware signatures individually, having all comparators work on a single instruction simultaneously can be inefficient and slow. To overcome this, parallelization techniques are implemented, allowing multiple instructions to be compared simultaneously. The HBA leverages the parallel processing capabilities of these comparators, enabling concurrent comparisons and thereby enhancing detection speed and accuracy. This approach allows CAUID to perform a large number of analyses in real-time, facilitating the rapid identification and mitigation of potential threats. Additionally, the malware data is divided into equally sized data blocks upon receipt, making the comparison process more manageable and enhancing the overall speed and efficiency of CAUID.

This implementation increases resource consumption; however, the impact on system performance is minimal. The operations performed by CAUID are not highly demanding, ensuring that resource consumption remains within acceptable limits, making it nearly unnoticeable during regular operation.

3. After receiving the data from the L3 cache of the CPU he control unit of the chip segments the incoming data into blocks and allocates these blocks to the available comparators. Once the data distribution is completed, the comparators begin with the analysis. They compare the incoming data blocks against the stored malware signatures of the HBA. These comparators operate in parallel, processing multiple instructions concurrently. They continuously scan and compare the data blocks. The CU also manages the flow of data and coordinates the parallel operations to ensure synchronized and efficient processing.

Detection Method The detection method used in CAUID requires a 75% resemblance between analyzed data sequences and stored data to initiate countermeasures. This threshold enhances the security of the chip and its ability to capture malicious attacks by avoiding the need for exact matches, which can lead to false negatives due to minor data variations or errors. In dynamic computing environments, where data transmission can be affected by noise or interference, expecting perfect matches is unrealistic and often results in missed detections of malicious activity. By allowing for some variance while ensuring substantial resemblance, CAUID effectively identifies malicious behavior, minimizes false positives, and maintains operational stability.

4. While the comparators are processing, CU actively monitors the resemblance between incoming data and stored malware signatures. The CU determines the 75% resemblance threshold by tracking the sequence in which data blocks enter and exit the comparators, as well as the specific comparators they were assigned to. If the component identifies that the similarity percentage meets or exceeds the 75% threshold, it flags the data as potentially malicious sends a signal to the CPU to initiate a shutdown. If it does not detect a 75% resemblance, CAUID continues its comparison process with the next instruction, without interruption.

5.4. Additional Features

Updating The update method for CAUID involves using SD cards developed specifically for providing the security chip with new data. This approach enhances security by minimizing the risk of unauthorized access or tampering with the firmware of the HBA. During the bootup sequence of the computer, the PC scans for the presence for an inserted SD card containing the update files. Once detected, the update process begins automatically, integrating new data on malware variants into the database of the chip. After the process is completed, the SD card must be removed, after which the OS of the device loads. This method, if updated, not only ensures that CAUID stays up-to-date with emerging threats but also minimizes the possibility of its compromisation.

5.5. Producton

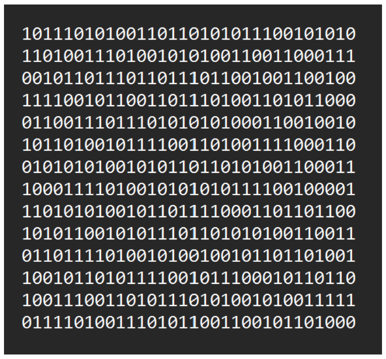
The objective of this section of the technical deliverable is to develop a simulation of the operation of CAUID. It aims to replicate the functionality of the HBA during its predicted scenarios, providing a detailed understanding of its operations. Additionally, this section describes the important parts of the program code, showing their purpose and functionality. The explanations highlight how each segment of the code corresponds to components and elements of the actual working procedure of CAUID.

5.5.1. .txt files. To fully assimilate the procedure of CAUID, it is essential to create a substitution to malware signatures, non-malicious data sequences, and the data storage of the HBA. A solution is utilizing .txt files containing randomized 32-bit sequences of data that represent instructions, which the program can read and compare, reflecting how CAUID functions in real-world scenarios.

It is important to note that the following .txt files are examples. The way the simulation is to be created is to be able to accept any combinations of sequences and compare them, making it more versatile.

- **data_storage.txt** This file, as the name suggests, serves as a representation of the data storage of CAUID, mirroring the actual storage mechanism of the chip.

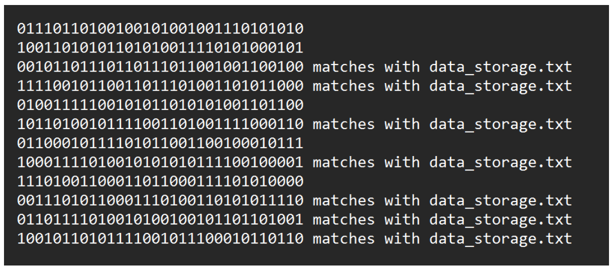
The concept of CAUID has a vast array of malware signatures needed to identify malicious intent. Each signature encapsulates a sequence of instructions needed for detecting various forms of malware. For the creation of the program, the txt file contains a representation of a single malware signature, comprising a sequence of 14 instructions.



```
10111010100110110101011100101010
11010011101001010100110011000111
00101101110110111011001001100100
11110010110011011101001101011000
01100111011101010101000110010010
10110100101111001101001111000110
01010101001010110110101001100011
10001111010010101010111100100001
11010101001011011110001101101100
10101100101011101101010100110011
01101111010010100100101101101001
10010110101111001011100010110110
10011100110101110101001010011111
01111010011101011001100101101000
```

Figure 3: data_storage.txt

- **data1.txt** This file represents a sequence of 12 instructions designed to represent non-malicious data. While the instructions within **data1.txt** may blandly resemble the signature malware data housed in data_storage.txt, the resemblance is not substantial enough to meet the stringent 75% threshold required for classification as potential malware. Consequently, CAUID diligently analyzes the contents of **data1.txt**, discerning its benign nature, and proceeds with its routine operations as the system remains on alert against any potential threats.

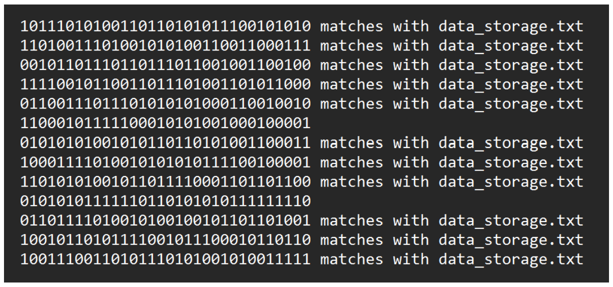


```
01110110100100101001001110101010
10011010101101010011110101000101
00101101110110111011001001100100 matches with data_storage.txt
11110010110011011101001101011000 matches with data_storage.txt
01001111100101011010101001101100
10110100101111001101001111000110 matches with data_storage.txt
01100010111101011001100100010111
10001111010010101010111100100001 matches with data_storage.txt
11101001100011011000111101010000
00111010110001110100110101011110 matches with data_storage.txt
01101111010010100100101101101001 matches with data_storage.txt
10010110101111001011100010110110 matches with data_storage.txt
```

Figure 4: data1.txt

- **data2.txt** This file represents a signature malware sequence comprised of 13 instructions. While slightly divergent from what the malware signature saved within data_storage.txt, the sequence contained within data2.txt surpasses the threshold of 75% resemblance when compared to the data signatures stored in the data storage of CAUID. This provokes the HBA to apply countermeasures so it can neutralize the threat.

To illustrate the possible outcomes of CAUID after a working cycle, two variations of a Python program are



```
10111010100110110101011100101010 matches with data_storage.txt
11010011101001010100110011000111 matches with data_storage.txt
00101101110110111011001001100100 matches with data_storage.txt
11110010110011011101001101011000 matches with data_storage.txt
01100111011101010101000110010010 matches with data_storage.txt
11000101111100010101001000100001
01010101001010110110101001100011 matches with data_storage.txt
10001111010010101010111100100001 matches with data_storage.txt
11010101001011011110001101101100 matches with data_storage.txt
01010101111110110101010111111110
01101111010010100100101101101001 matches with data_storage.txt
10010110101111001011100010110110 matches with data_storage.txt
10011100110101110101001010011111 matches with data_storage.txt
```

Figure 5: data2.txt

created. Both programs are identical in structure, apart from the files they compare. The first version (Outcome1.py) juxtaposes data1.txt with data_storage.txt, the other (Outcome2.py) compares data2.txt with data_storage.txt. All the differences between the two variations are shown in the upcoming Functionality section of the BSP.

5.5.2. Functionality. This section explains the Python simulation designed to demonstrate the operation of CAUID. The program components that interpret the behavior of specific elements or methods in the HBA are organized hierarchically: classes are the main segments representing their respected component, and significant methods are their subsections. Some methods are placed in the ”Informative Methods” subsection within each class segment. These methods are created with the intention of providing information about the procedure of the actual chip but are not needed for the core functionality of the program. Said methods will only be listed. Each class and method is described, with significant methods including an explanation of the corresponding components or methods in the actual CAUID operation.

- **class CPU**

def start The start method represents the booting process of the CPU, where it begins loading the OS into RAM. It then calls the **load_os_to_ram** method to simulate the loading of the operating system into RAM. After this, it triggers CAUID to detect the boot-up by calling **cauid.boot_up_decetcted()**. This simulates the activation of HBA in response to the system powering on and ensures that CAUID loads its malware signatures into RAM, preparing it for real-time monitoring and threat detection.

def execute_instruction(self) The execute_instruction method in the CPU class is designed to simulate the execution of instructions by the CPU. The method begins by checking if the **self.running** attribute is true, indicating whether the CPU is currently operational. If so, the processor proceeds to execute an instruction.

This method demonstrates how the CPU and CAUID work in conjunction to maintain system security. Each time an instruction is executed, CAUID checks it for potential malicious intent.

def shutdown(self) The shutdown method in the CPU class simulates the process of shutting down the processor. The method sets the self.running attribute to False, effectively stopping the CPU from executing any further instructions. By setting self.running to False, the method halts all CPU operations.

In the context of the actual working of the chip, when the CU of CAUID identifies a potential threat with a similarity threshold of 75% or higher, it sends a signal to the CPU to shutdown. This preventive measure is designed to halt the execution of potentially malicious instructions, thereby preserving system integrity and preventing further damage. By stopping the CPU's operations, the shutdown method ensures that detected threats cannot propagate or cause data loss or corruption. This highlights the importance of hardware-based security measures in maintaining system security and preventing malicious activity.

Informative Methods

def load_os_to_ram(self)
def __init__

- **class CAUID**

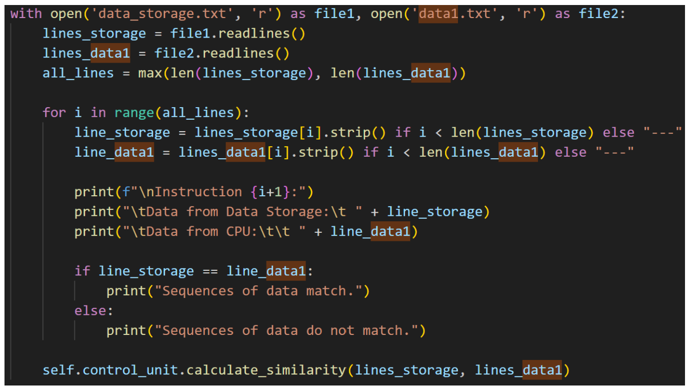
def check_for_malware(self) The method opens two files: data_storage.txt and the respective data, depending on the program. The contents of these files are read into lines_storage and lines_data1 or 2 respectively. The all_lines variable is determined by finding the maximum length of the two lists of lines, ensuring that the comparison can handle files of different lengths. The program then enters a loop that iterates over each line in the files. For each line, it compares the corresponding lines from data_storage.txt and data1.txt or data2.txt. This comparison is repeated for all lines in both files.

What this method represents in terms of the theoretical working of CAUID is the initiation of the comparison process between the data stored in the chip and the instructions from the CPU.

This part of the code contributes to the difference between Outcome1.py and Outcome2.py. The distinctions are highlighted in the figures below.

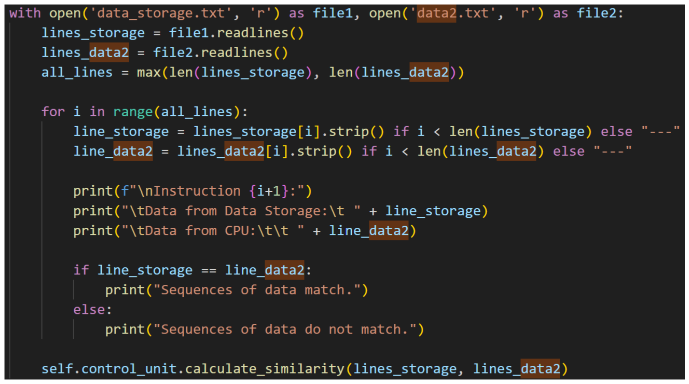
Informative Methods

def __init__
def boot_up_detected(self)
def load_data_to_ram(self)



```python
with open('data_storage.txt', 'r') as file1, open('data1.txt', 'r') as file2:
    lines_storage = file1.readlines()
    lines_data1 = file2.readlines()
    all_lines = max(len(lines_storage), len(lines_data1))

    for i in range(all_lines):
        line_storage = lines_storage[i].strip() if i < len(lines_storage) else "---"
        line_data1 = lines_data1[i].strip() if i < len(lines_data1) else "---"

        print(f"\nInstruction {i+1}:")
        print("\tData from Data Storage:\t " + line_storage)
        print("\tData from CPU:\t\t " + line_data1)

        if line_storage == line_data1:
            print("Sequences of data match.")
        else:
            print("Sequences of data do not match.")

    self.control_unit.calculate_similarity(lines_storage, lines_data1)
```

Figure 6: def check_for_malware of Outcome1.py



```python
with open('data_storage.txt', 'r') as file1, open('data2.txt', 'r') as file2:
    lines_storage = file1.readlines()
    lines_data2 = file2.readlines()
    all_lines = max(len(lines_storage), len(lines_data2))

    for i in range(all_lines):
        line_storage = lines_storage[i].strip() if i < len(lines_storage) else "---"
        line_data2 = lines_data2[i].strip() if i < len(lines_data2) else "---"

        print(f"\nInstruction {i+1}:")
        print("\tData from Data Storage:\t " + line_storage)
        print("\tData from CPU:\t\t " + line_data2)

        if line_storage == line_data2:
            print("Sequences of data match.")
        else:
            print("Sequences of data do not match.")

    self.control_unit.calculate_similarity(lines_storage, lines_data2)
```

Figure 7: def check_for_malware of Outcome2.py

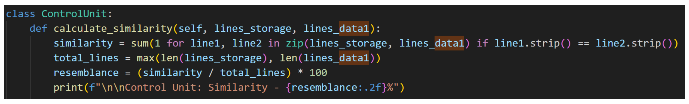
- **class ControlUnit**

def calculate_similarity This method of the program takes the arguments lines_storage and lines_data1 or 2. It calculates the number of matching lines using a generator expression within the sum function. For each pair of lines, it strips any leading or trailing whitespaces and checks for equality. The result, stored in the similarity variable, represents the count of matching lines. Afterwards, the method determines the total number of lines by taking the maximum length of lines_storage and lines_data1 or 2, ensuring that the calculation can handle files of different lengths. This value is stored in the total_lines variable. The resemblance percentage is then calculated by dividing the number of matching lines by the total number of lines and multiplying by 100. For convenience, the percentage is printed to provide feedback on the similarity level. If the resemblance percentage is 75% or higher, the method calls self.initiate_shutdown() to simulate shutting down the CPU. Otherwise, the behaviour of CAUID does not change.

What this method represents from the actual working is the procedure of the CU and how it works in conjunction with the comparators of the chip in order to measure the similarity rate between the data of the HBA and the sequence of processor instructions.

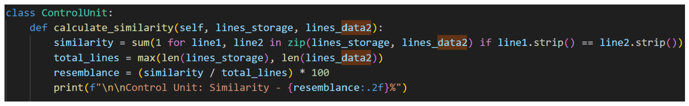
Aside from the distinctions between Outcome1.py and

Outcome2.py in their def check_for_malware(self) methods, def calculate_similarity also sets them apart.



```python
class ControlUnit:
    def calculate_similarity(self, lines_storage, lines_data1):
        similarity = sum(1 for line1, line2 in zip(lines_storage, lines_data1) if line1.strip() == line2.strip())
        total_lines = max(len(lines_storage), len(lines_data1))
        resemblance = (similarity / total_lines) * 100
        print(f"\n\nControl Unit: Similarity - {resemblance:.2f}%")
```

Figure 8: def calculate_similarity of Outcome2.py



```python
class ControlUnit:
    def calculate_similarity(self, lines_storage, lines_data2):
        similarity = sum(1 for line1, line2 in zip(lines_storage, lines_data2) if line1.strip() == line2.strip())
        total_lines = max(len(lines_storage), len(lines_data2))
        resemblance = (similarity / total_lines) * 100
        print(f"\n\nControl Unit: Similarity - {resemblance:.2f}%")
```

Figure 9: def calculate_similarity of Outcome2.py

Informative Methods

def initiate_shutdown(self)

5.6. Assessment

The technical section of this project provides a thorough analysis of the components and functionalities of CAUID. Key components important for its operation, such as the control unit, comparators, and data storage, are described in detail. Techniques utilized by the HBA are also covered, providing a clear understanding of the theoretical operation of CAUID. The theoretical functionality of the chip is well-documented, ensuring a comprehensive understanding of the foundational principles behind the design and operation of CAUID.

In addition to the theoretical description, the simulation includes an explanation of all additional files and their purposes. Essential classes and methods for achieving the functionality of the program are clearly defined. The program demonstrates versatility by being able of comparing two files containing any symbols and any number of lines. Through these efforts, the primary objective of the technical deliverable to prove the feasibility of CAUID as a concept has been successfully achieved.

Despite these positives, some areas could be improved. Performance considerations were mentioned but not explored in depth. Since the main goal was to prove the concept, the interpretation of the program was oversimplified, only comparing single instructions at a time. Additionally, the simulation did not demonstrate parallelization, which is a key feature of CAUID for enhancing performance in real-world applications.

Acknowledgement

The author of this BSP expresses their gratitude to the supervisor, Bernard Steenis, for their provision of support, feedback, and suggestions for improvement during the making of this project.

6. Conclusion

The development of a fully hardware-based antivirus system represents a significant potential improvement in the field of cybersecurity. The minimized resource usage and reduced possibility of malicious compromise are highly desirable factors for users of electronic devices. However, at present, the concept of developing a security chip is still in its nascent stages.

This Bachelor Semester project focused on creating and proving the concept of CAUID. Through a detailed comparison of the characteristics of software-based and hardware-based antivirus systems, it was concluded that hardware-based antiviral systems offer superior performance. The main objective of the scientific deliverable, which aimed to determine the advantages of hardware-based antivirus systems, has been successfully achieved. Following this, the project aimed to prove the feasibility of CAUID as a concept. Through an in-depth explanation of the theoretical functionality of the chip and the successful creation of a Python program simulating the operation of the hardware-based antivirus system, the goal of the technical deliverable has been met.

References

[1] What is antivirus software? By Sophos

[2] How does antivirus software work? By Darren Allan

[3] How Does Antivirus Software Work? By Aliza Vigderman, Gabe Turner

[4] Real-time Antivirus Protection FAQs By ReasonLabs

[5] What is Low resource usage? Optimizing Cybersecurity: The Importance of Low Resource Usage in Antivirus Software and its Impact on Computer Performance by ReasonLabs

[6] Back to Basics: Hardware Security as the Ultimate Defense Against Ransomware Attacks By techspective

[7] An In-depth Look at Hardware-based Cybersecurity By BlackBear Cyber Security

[8] What is Secure Enclave? Safeguarding Sensitive Data: The Role of Secure Enclave in Cybersecurity by RasonLabs

[9] Behavior-based security vs. signature-based security: How they differ By Twain Taylor

[10] Why Choose Hardware-Assisted Security By Donna Beyersdorf

[11] How Does Heuristic Analysis Antivirus Software Work? By Logix

[12] What is Heuristics-based Detection? The Power of Heuristics-based Detection in Cybersecurity: Staying Ahead of Evolving Threats By ReasonLabs

[13] What is Scheduled Scanning? The Importance of Scheduled Scanning for Efficient Cybersecurity: Keeping your System Safe from Malicious Threats By ReasonLabs

Appendix

Source code

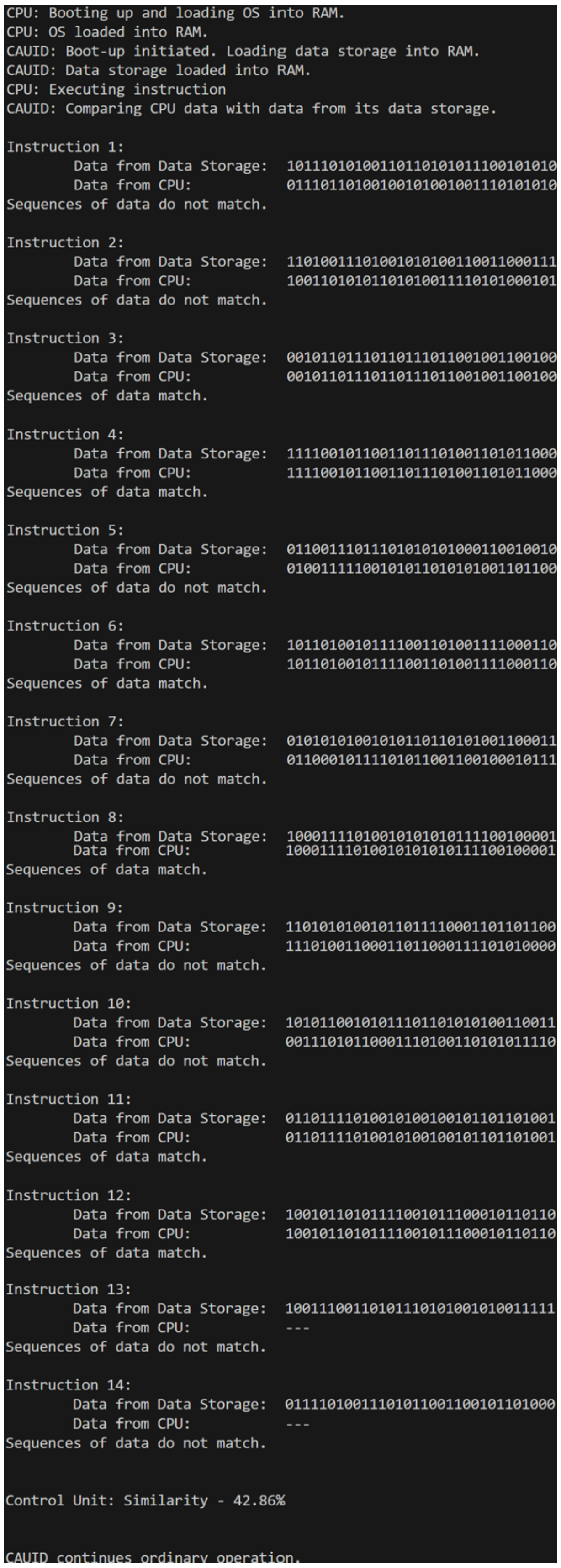
Listing 1: Outcome1.py

```python
class CPU:
    def __init__(self):
        self.running = True

    def start(self):
        print("\nCPU: Booting up and loading OS into
            RAM.")
        self.load_os_to_ram()
        cauid.boot_up_detected()

    def load_os_to_ram(self):
        print("CPU: OS loaded into RAM.")

    def execute_instruction(self):
        if self.running == True:
            print("CPU: Executing instruction")
            cauid.check_for_malware()

    def shutdown(self):
        print("CPU: Shutting down.")
        self.running = False

class CAUID:
    def __init__(self):
        self.control_unit = ControlUnit()

    def boot_up_detected(self):
        print("CAUID: Boot-up initiated. Loading data
            storage into RAM.")
        self.load_data_to_ram()

    def load_data_to_ram(self):
        print("CAUID: Data storage loaded into RAM.")

    def check_for_malware(self):
        print("CAUID: Comparing CPU data with data from
            its data storage.")

        with open('data_storage.txt', 'r') as file1,
            open('data1.txt', 'r') as file2:
            lines_storage = file1.readlines()
            lines_data1 = file2.readlines()
            all_lines = max(len(lines_storage),
                len(lines_data1))

            for i in range(all_lines):
                line_storage = lines_storage[i].strip() if i <
                    len(lines_storage) else "---"
                line_data1 = lines_data1[i].strip() if i <
                    len(lines_data1) else "---"

                print(f"\nInstruction {i+1}:")
                print("\tData from Data Storage:\t " +
                    line_storage)
                print("\tData from CPU:\t\t " + line_data1)

                if line_storage == line_data1:
                    print("Sequences of data match.")
                else:
                    print("Sequences of data do not
                        match.")

            self.control_unit.calculate_similarity(lines_storage,
                lines_data1)

class ControlUnit:
    def calculate_similarity(self, lines_storage, lines_data1):
        similarity = sum(1 for line1, line2 in
            zip(lines_storage, lines_data1) if line1.strip()
            == line2.strip())
        total_lines = max(len(lines_storage),
            len(lines_data1))
        resemblance = (similarity / total_lines) * 100
        print(f"\n\nControl Unit: Similarity -
            {resemblance:.2f}%")

        if resemblance >= 75:
            self.initiate_shutdown()
        else:
            print("\n\nCAUID continues ordinary
                operation.")

    def initiate_shutdown(self):
        print("\n\nControl Unit: Sending shutdown signal to
            CPU.")
        cpu.shutdown()

cauid = CAUID()
cpu = CPU()
cpu.start()
cpu.execute_instruction()
```

Listing 2: Outcome2.py

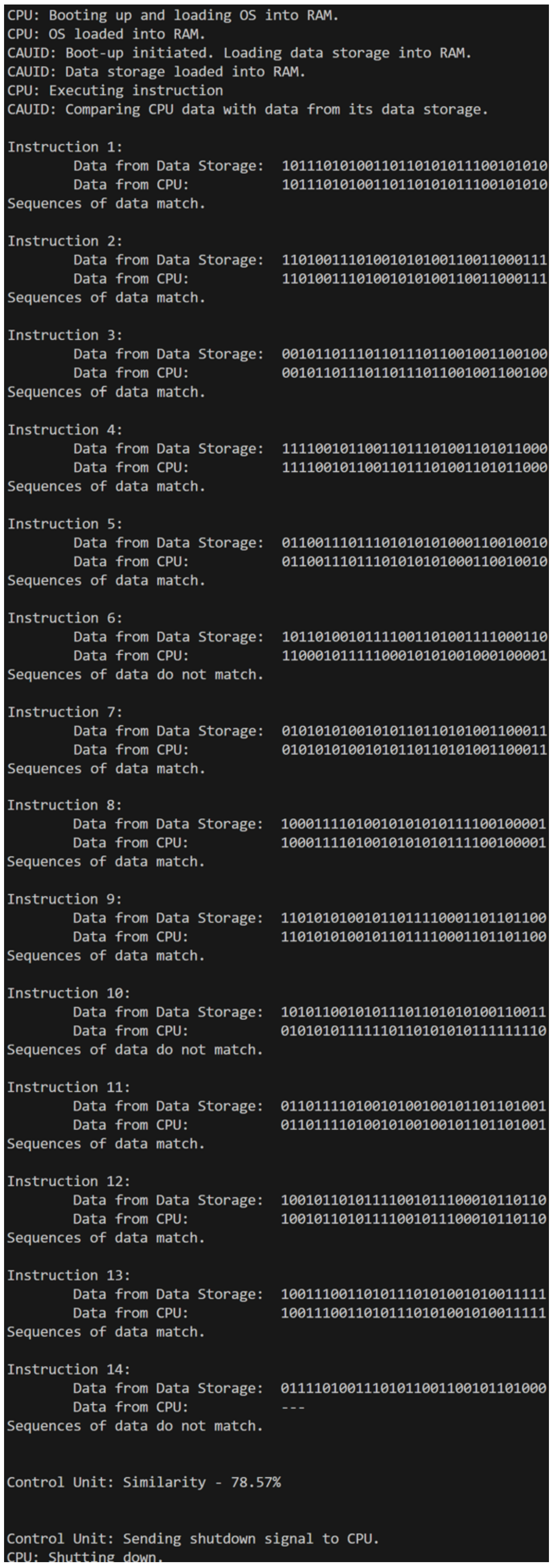
```python
class CPU:
    def __init__(self):
        self.running = True

    def start(self):
        print("\nCPU: Booting up and loading OS into
            RAM.")
        self.load_os_to_ram()
        cauid.boot_up_detected()

    def load_os_to_ram(self):
        print("CPU: OS loaded into RAM.")

    def execute_instruction(self):
        if self.running == True:
            print("CPU: Executing instruction")
            cauid.check_for_malware()

    def shutdown(self):
        print("CPU: Shutting down.")
        self.running = False

class CAUID:
    def __init__(self):
        self.control_unit = ControlUnit()

    def boot_up_detected(self):
        print("CAUID: Boot-up initiated. Loading data
            storage into RAM.")
        self.load_data_to_ram()

    def load_data_to_ram(self):
        print("CAUID: Data storage loaded into RAM.")
    def check_for_malware(self):
        print(”CAUID: Comparing CPU data with data from its data storage.”)

        with open(’data_storage.txt’, ’r’) as file1, open(’data2.txt’, ’r’) as file2:
            lines_storage = file1.readlines()
            lines_data2 = file2.readlines()
            all_lines = max(len(lines_storage), len(lines_data2))

            for i in range(all_lines):
                line_storage = lines_storage[i].strip() if i < len(lines_storage) else ”---”
                line_data2 = lines_data2[i].strip() if i < len(lines_data2) else ”---”

                print(f”\nInstruction {i+1}:”)
                print(”\tData from Data Storage:\t ” + line_storage)
                print(”\tData from CPU:\t\t ” + line_data2)

                if line_storage == line_data2:
                    print(”Sequences of data match.”)
                else:
                    print(”Sequences of data do not match.”)

            self.control_unit.calculate_similarity(lines_storage, lines_data2)

class ControlUnit:
    def calculate_similarity(self, lines_storage, lines_data2):
        similarity = sum(1 for line1, line2 in zip(lines_storage, lines_data2) if line1.strip() == line2.strip())
        total_lines = max(len(lines_storage), len(lines_data2))
        resemblance = (similarity / total_lines) * 100
        print(f”\n\nControl Unit: Similarity – {resemblance:.2f}%”)

        if resemblance >= 75:
            self.initiate_shutdown()
        else:
            print(”\n\nCAUID continues ordinary operation.”)

    def initiate_shutdown(self):
        print(”\n\nControl Unit: Sending shutdown signal to CPU.”)
        cpu.shutdown()

cauid = CAUID()
cpu = CPU()
cpu.start()
cpu.execute_instruction()
```

Program output



```
CPU: Booting up and loading OS into RAM.
CPU: OS loaded into RAM.
CAUID: Boot-up initiated. Loading data storage into RAM.
CAUID: Data storage loaded into RAM.
CPU: Executing instruction
CAUID: Comparing CPU data with data from its data storage.

Instruction 1:
        Data from Data Storage:  10111010100110110101011100101010
        Data from CPU:           01110110100100101001001110101010
Sequences of data do not match.

Instruction 2:
        Data from Data Storage:  11010011101001010100110011000111
        Data from CPU:           10011010101101010011110101000101
Sequences of data do not match.

Instruction 3:
        Data from Data Storage:  00101101110110111011001001100100
        Data from CPU:           00101101110110111011001001100100
Sequences of data match.

Instruction 4:
        Data from Data Storage:  11110010110011011101001101011000
        Data from CPU:           11110010110011011101001101011000
Sequences of data match.

Instruction 5:
        Data from Data Storage:  01100111011101010101000110010010
        Data from CPU:           01001111100101011010101001101100
Sequences of data do not match.

Instruction 6:
        Data from Data Storage:  10110100101111001101001111000110
        Data from CPU:           10110100101111001101001111000110
Sequences of data match.

Instruction 7:
        Data from Data Storage:  01010101001010110110101001100011
        Data from CPU:           01100010111101011001100100010111
Sequences of data do not match.

Instruction 8:
        Data from Data Storage:  10001111010010101010111100100001
        Data from CPU:           10001111010010101010111100100001
Sequences of data match.

Instruction 9:
        Data from Data Storage:  11010101001011011110001101101100
        Data from CPU:           11101001100011011000111101010000
Sequences of data do not match.

Instruction 10:
        Data from Data Storage:  10101100101011101101010100110011
        Data from CPU:           00111010110001110100110101011110
Sequences of data do not match.

Instruction 11:
        Data from Data Storage:  01101111010010100100101101101001
        Data from CPU:           01101111010010100100101101101001
Sequences of data match.

Instruction 12:
        Data from Data Storage:  10010110101111001011100010110110
        Data from CPU:           10010110101111001011100010110110
Sequences of data match.

Instruction 13:
        Data from Data Storage:  10011100110101110101001010011111
        Data from CPU:           ---
Sequences of data do not match.

Instruction 14:
        Data from Data Storage:  01111010011101011001100101101000
        Data from CPU:           ---
Sequences of data do not match.


Control Unit: Similarity - 42.86%


CAUID continues ordinary operation.
```

Figure 10: Output of Outcome1.py



```
CPU: Booting up and loading OS into RAM.
CPU: OS loaded into RAM.
CAUID: Boot-up initiated. Loading data storage into RAM.
CAUID: Data storage loaded into RAM.
CPU: Executing instruction
CAUID: Comparing CPU data with data from its data storage.

Instruction 1:
        Data from Data Storage:  10111010100110110101011100101010
        Data from CPU:           10111010100110110101011100101010
Sequences of data match.

Instruction 2:
        Data from Data Storage:  11010011101001010100110011000111
        Data from CPU:           11010011101001010100110011000111
Sequences of data match.

Instruction 3:
        Data from Data Storage:  00101101110110111011001001100100
        Data from CPU:           00101101110110111011001001100100
Sequences of data match.

Instruction 4:
        Data from Data Storage:  11110010110011011101001101011000
        Data from CPU:           11110010110011011101001101011000
Sequences of data match.

Instruction 5:
        Data from Data Storage:  01100111011101010101000110010010
        Data from CPU:           01100111011101010101000110010010
Sequences of data match.

Instruction 6:
        Data from Data Storage:  10110100101111001101001111000110
        Data from CPU:           11000101111100010101001000100001
Sequences of data do not match.

Instruction 7:
        Data from Data Storage:  01010101001010110110101001100011
        Data from CPU:           01010101001010110110101001100011
Sequences of data match.

Instruction 8:
        Data from Data Storage:  10001111010010101010111100100001
        Data from CPU:           10001111010010101010111100100001
Sequences of data match.

Instruction 9:
        Data from Data Storage:  11010101001011011110001101101100
        Data from CPU:           11010101001011011110001101101100
Sequences of data match.

Instruction 10:
        Data from Data Storage:  10101100101011101101010100110011
        Data from CPU:           01010101111110110101010111111110
Sequences of data do not match.

Instruction 11:
        Data from Data Storage:  01101111010010100100101101101001
        Data from CPU:           01101111010010100100101101101001
Sequences of data match.

Instruction 12:
        Data from Data Storage:  10010110101111001011100010110110
        Data from CPU:           10010110101111001011100010110110
Sequences of data match.

Instruction 13:
        Data from Data Storage:  10011100110101110101001010011111
        Data from CPU:           10011100110101110101001010011111
Sequences of data match.

Instruction 14:
        Data from Data Storage:  01111010011101011001100101101000
        Data from CPU:           ---
Sequences of data do not match.


Control Unit: Similarity - 78.57%


Control Unit: Sending shutdown signal to CPU.
CPU: Shutting down.
```

Figure 11: Output of Outcome2.py

7. Plagiarism Statement

I declare that I am aware of the following facts:

- I understand that in the following statement the term ”person” represents a human or **ANY AUTOMATIC GENERATION SYTEM**.
- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a pre-check by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one’s own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:

  1) Not putting quotation marks around a quote from another person’s work
  2) Pretending to paraphrase while in fact quoting
  3) Citing incorrectly or incompletely
  4) Failing to cite the source of a quoted or paraphrased work
  5) Copying/reproducing sections of another person’s work without acknowledging the source
  6) Paraphrasing another person’s work without acknowledging the source
  7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one’s own name (‘ghost-writing’)
  8) Using another person’s unpublished work without attribution and permission (‘stealing’)
  9) Presenting a piece of work as one’s own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

  Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.