6-10-2024

Edge-Connected Microcontroller Security

Divya Syal

Gavin Ryder

Neena Ekanathan

SANTA CLARA UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Date: June 10, 2024

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Divya Syal
Gavin Ryder
Neena Ekanathan

ENTITLED

Edge-Connected Microcontroller Security

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

DocuSigned by:
Behnam Dezfouli
6ED175D6B91A4FF...
Thesis Advisor

DocuSigned by:
Yuhong Liu
DE1F9661B6A44F4...
Thesis Advisor

[illegible]
Department Chair

Edge-Connected Microcontroller Security

by

Divya Syal
Gavin Ryder
Neena Ekanathan

Submitted in partial fulfillment of the requirements
for the degree of
Bachelor of Science in Computer Science and Engineering
School of Engineering
Santa Clara University

Santa Clara, California
June 10, 2024

Edge-Connected Microcontroller Security

Divya Syal
Gavin Ryder
Neena Ekanathan

Department of Computer Science and Engineering
Santa Clara University
June 10, 2024

ABSTRACT

With a wide range of applications and the rise of cyber attacks, securing microcontrollers has become imperative; however, ensuring microcontroller performance is also crucial given how interconnected today's systems are. This project examines the security and performance of next-generation microcontroller units leveraging new security solutions for IoT edge applications. By benchmarking these MCUs against key performance metrics, their viability will be assessed to facilitate the widespread adoption of this latest firmware.

Our research focuses on profiling the power consumption and performance of the new STM32H573 and the integrated Secure Manager, a new technology from ST Microelectronics that allows for privileged execution of code and restricting access to services. The STM32H573 supports various cryptography algorithms (like ECDSA) at the hardware level to enable this. Our work profiles the power consumption and performance of these technologies.

We found that in some cases the Secure Manager is more energy efficient, and can also offer negligible performance overhead. With that said, for the majority of the use cases we evaluated running the application in the Non-Secure mode both increased performance and decreased energy consumption.

Table of Contents

List of Figures

Glossary

CRP Code Readout Protection. 5, 8

DCS Distributed Control System. 4

DDoS Distributed Denial of Service. 4

ECDSA Elliptic Curve Digital Signature Algorithm. iii, 11

ECU Electronic Control Unit. 1

GPIO General Purpose Input/Output. 6, 10, 19

IoT Internet of Things. 1, 2, 4–7

MCU Microcontroller Unit. 1, 2, 4–8, 10, 14

PPK Nordic Semiconductor Power Profiler Kit II. 11, 14, 28

PSA Platform Security Architecture. 6

RSA Rivest–Shamir–Adleman public-key cryptosystem. 11

RTOS Real-time Operating System. 3, 4, 6

SCADA Supervisory Control and Data Acquisition. 4

STiROT ST Immutable Root of Trust. 8

TEE Trusted Execution Environment. 3, 6–8

TF-M ARM Trusted Firmware-M. 2, 7

TI Texas Instruments. 5

UART Universal Asynchronous Receiver-Transmitter. 5

uROT Updatable Root of Trust. 8

UVM Universal Verification Methodology. 5

Chapter 1

Introduction

With the increasing amount of microprocessor IoT devices available on the market today and the roles they play in our everyday lives, securing them has never been more important. As such, companies like STMicroelectronics are working on developing ways to best secure the next generation of microcontrollers behind cutting-edge IoT products. Because these products are used in places like businesses, homes, and vehicles, keeping them safe from attackers is especially important. For example, the Mirai botnet attack in 2016 infected over 145,000 IoT devices at its peak, allowing a flood attack of over 1 Tbps [7]. Network security research company SAM Seamless Network found that of the 1 billion cyber attacks that took place in 2021, over 900 million (90%) of them were performed using IoT devices [14]. Because more IoT devices are coming online every day, more needs to be done to prevent them from being used for malicious activities by threat actors.

STMicroelectronics' new ARM Cortex M33-based MCUs like the STM32H573 represent the next generation of power-efficient, IoT-capable microcontrollers. With modern peripherals such as USB-C, camera interfaces, and sensory I/O, it can be used anywhere from medical devices to robots to smart homes [19]. Because these capabilities make the STM32H573 microcontroller suitable for use in life-critical medical devices, preventing malicious intrusion could mean saving lives. There were 80,000 medical device incidents flagged by the FDA between 2008 and 2018, some of which were fatal [21]. This goes to show that if microcontrollers are insecure, the consequences of being infected with malware or otherwise subject to cyberattack could be fatal. Similar is true in autonomous vehicles, where "attacks on the networks can affect the functionality of all vehicle ECUs and the whole system" [7]. As these devices continue to become more and more connected, the attack surface for such threats only grows larger.

Current research and practice suggest that the use of secure boot schemes and cryptography can be used to secure microcontrollers [1][7]. However, many of these practices may lack security and, crucially, performance. Thus, there is a need for firmware that adopts these practices on new devices like the STM32H573. Our idea is to implement firmware on the STM32H573 that follows these best practices while meeting key benchmarks for performance and power efficiency.

Securing microcontrollers for IoT applications is already an existing area of interest of many device manufacturers, and it has been important from the beginning. With the advances made by ARM Trust Firmware TF-M to turn it into the leading technology to secure microcontrollers, there is work that must be done to bring ST's version of it to the next generation of hardware. However, none of the currently available secure environments have yet been extensively tested on the STM32H573 and may lack optimal power efficiency. We will examine two of the most relevant products and highlight their shortcomings in this section.

ARM's Trust Firmware-M is an open-source Trusted Execution Environment for ARM v7-M and ARM v8-M devices [5]. A trusted execution environment runs on a separate kernel and guarantees the authenticity of code, data, and other crucial runtime information [23]. This would be our main competitor for the firmware solution as the solution offers security for low-power microprocessors, with both secure and non-secure zones per single processing unit. ARM TF-M has gained popularity in the context of security for IoT and embedded devices. This secure processing environment runs on Cortex-M devices to protect keys, sensitive data, and assets and supports other Cortex-M-based microcontrollers and Real-Time Operating Systems [19]. While this solution works seamlessly on Cortex-M, it is not ideal for the STM32H57.

NXP Semiconductors has their own line of microcontrollers with their i.MX RT series of crossover MCUs. With their Kinetis and LPC series, NXP provides these microcontrollers with embedded security with hardware-based security features. However, these are primarily ARM Cortex series-based processors with ARM TF-M embedded as the secure processing environment [18]. In addition to the TrustZone technology, this solution also features secure boot and cryptographic accelerators. NXP Semiconductors also offers the EdgeLock Secure Enclave: a hardware-based secure execution environment that can be integrated with various devices; these secure enclaves offer protect keys, cryptographic functions, and isolation for certain applications [20]. Neither of these solutions can be seamlessly adopted by the STM32H57.

As mentioned above, securing IoT microcontrollers is not an immediately novel idea. However, the adoption of ST's implementation of TF-M on the STM32H573 represents a new and important way of securing ST's IoT hardware from attacks. Benchmarking this technology against key metrics such as wake time, power consumption and general efficiency will open the door for new strategies in adopting secure elements for real-world applications. While ARM TF-M has been used for microcontroller security before, it has not yet been adopted by ST for use on their hardware. While existing solutions may be acceptable, with the growth of IoT devices comes the need for continued work to be done to keep them secure against threat advancements. Our proposed secure firmware implementation would be a novel step in doing just this.

Ultimately, our project runs a firmware solution on the provided STM32H573 hardware which demonstrates the viability of implementing ST's implementation of TF-M on both STM32H573 and future products. Ideally, this firmware will be performant when measured against benchmarks for power consumption, wake time, memory use, and overall efficiency. This will allow us to provide an estimate on both the resources required to implement such a project and allow for the determination of strategies for scaling up the firmware and using it on edge devices.

To achieve this, we would use the ST Trust TEE development tool in order to test and enhance our firmware on the STM32H573 hardware provided to us. This would allow us to test, iterate, and debug our code on the physical hardware. Apart from this tool, we hope to use a breakout board for power consumption measurement to have a more exact idea of the power consumption impact of our firmware design and implementation. As a result of our testing, we hope to ultimately provide a firmware solution that can help change the paradigm for all new secure products using ST microcontroller technology.

The thesis will be organized as such: In Chapter 2, we will discuss related work associated with this project, focusing on MCU security and performance; this chapter will also discuss background on crucial terminology such as real-time operating systems (RTOS) and trusted execution environments (TEE). In Chapter 3, we will highlight societal issues that this research affects including its ethical, social, economic, health, and environmental impacts. Chapter 4 outlines the methodology we used to develop and implement the testing applications to benchmark the STM32H573. In Chapter 5, we will discuss the results and analysis of running those testing applications on the board. Lastly, Chapter 6 covers our conclusion and future work followed by any acknowledgments and references.

All the code from this project is available on GitHub under the SIOTLAB organization.

Chapter 2

Background

As a whole, the implementation of a secure and fast microcontroller unit (MCU) is not new. Such a device requires the orchestration of many key components, including the real-time operating system (RTOS), bootloader, security architecture, and power and performance management and testing. A capable MCU combines these components in a manner that is both secure, reliable, and performant.

2.1 Related Work

While the performance of RTOSs on MCUs is undoubtedly important, security is absolutely critical. Fortunately, much work has been done on how to secure MCUs. Existing work around MCU RTOS security focuses on keeping time-critical jobs separate, process stability, and making sure they are secure in a networked environment [31]. In modern applications, RTOSs are often linked together via supervisory control and data acquisition (SCADA) or a distributed control system (DCS) [31]. As mentioned earlier, vulnerabilities in IoT-capable MCU systems provide a common attack surface for distributed denial of service (DDoS) attacks. Due to performance and memory constraints, many MCUs lay out everything in a single memory space and run all processes at the privileged level. This leads to a very large attack surface [12]. Recent research has proposed an efficient memory view-switching mechanism, allowing for memory isolation with little performance overhead [12]. This is achieved by first capturing the minimum memory regions needed for each process, then assembling a per-process memory view based on static program and clustering analysis, and finally enforcing these derived limits using view switching. Notably, the view switcher is the only process that runs in privileged mode and uses hardware memory isolation [12]. This is novel, as many RTOSs provide optional kernel memory isolation but not process memory isolation.

One of the larger considerations of this paper's research involves analyzing MCU power consumption. Researchers at Texas Instruments have outlined four categories to consider when benchmarking microcontroller power consumption [4]. These include standby power when waiting for an event to wake up the CPU, peripheral power, or power that allows communication with analog signals, data logging power when logging data for later transmission and analysis, and active power, or power that is consumed when the CPU is active. In their work, the researchers claim that analyzing standby power when the microcontroller is not operating is more important than active power. Working with the Texas Instruments (TI) "Wolverine"-based MSP-430FR59xx family of microcontrollers, they measure automatic wake-up on time intervals, RAM retention, power source monitoring, and temperature. To estimate active power, the researchers focus on software execution, bus clocks, CPU sleep activity, acceleration, and optimized code. However, while their work assumes that standby power primarily affects battery life in microcontrollers, their research does not address power consumption by security applications or firmware during active power. Instead, the researchers propose that optimizing standby current will have the largest impact on the microcontroller's battery life for most applications [4]. In our research, we will measure the effect of all power categories on the firmware-embedded STM32H573 before narrowing down a category to analyze and optimize that most significantly impacts power consumption.

Researchers at the University of Birmingham in the UK exposed the hardware and software vulnerabilities of various bootloaders through a binary analysis technique [30]. Binary analysis, in this case, involves analyzing the machine code to identify vulnerabilities and risks in the bootloader firmware. In their work, to bypass the CRP mechanisms, the researchers used voltage glitching to induce a hardware-based fault injection and used Return-Oriented Programming (ROP), which involves setting up a chain of gadgets to execute a software attack, as two methods to expose vulnerabilities [30]. In their work on STM8 and NXP chips, they were able to overwrite flash sectors and alter memory during updates using glitching methods [30]. This paper's work and methods emphasize the importance of measuring performance against additional security measures. Modifications such as internal clock changes or software-based solutions are possible methods to test ways to mitigate these vulnerabilities.

Analysis into bootloader design has been conducted by researchers in Beijing in their creation of a new MCU chip [13]. In their implementation, the researchers applied the Universal Verification Methodology (UVM) framework to verify their hardware and UART VIP, a verification hardware component, to verify the Universal Asynchronous Receiver-Transmitter (UART) module, a component within the chip that communicates with external devices [13]. By incorporating these additions, the bootloader becomes more portable because of the use of standardized frameworks. With the STM32H573 in the context of this paper, it is important to consider bootloader verification frameworks that ensure reliability and portability in addition to performance.

Lastly, intensive security research has been done by Microsoft in their development of Azure Sphere, their IoT device security platform. In their work, they identify hardware-based roots of trust, compartmentalization, and certification-based authentication (among others) as hallmarks of highly secure devices and cornerstones of their platform [28]. This motivates us to ensure that the STM32H573 has similar implementations and reaches the same level of security. Furthermore, the Pluton Security Engine focuses on hardware-based security operations. This way, there can be no "rogue software cannot be used to read out keys" [28] and performance is improved.

2.2 Real Time Operating Systems

In the current market, RTOSs are plentiful and MCU manufacturers have many options to choose from. Examples include FreeRTOS, RT-Thread, eCOS, LynxOS, QNX, VxWorks, OSEK, and many others [29]. Based on the Rhealstone standard proposed by ARM in 1990, task switching time, preemption time, interrupt latency, semaphore shuffling time, deadlock breaking time, and inter-task messaging latency are common metrics for measuring the performance of such systems [22]. While many benchmarking methods focus on the Rhealstone benchmark, [Anh] argues that it is unsuitable because RTOSs cannot break deadlocks and take different messaging approaches [2]. Instead, [Anh] evaluates RTOSs using criteria such as capabilities, performance, API richness, and debugging support. More recently, researchers have compared modern RTOSs (Keil RTX5 and FreeRTOS v10.2.0) [15]. Unlike traditional Rhealstone methods, their approach utilizes DWT counting on ARM Cortex CPUs. Crucially, as noted by the authors, this removes the possibility of discrepancies that could be present if using the GPIO pin [15]. While timer-based methods can be delayed when interrupts are disabled, this method is not subject to that constraint. This approach takes the cycle count before and after a critical section of code, taking the difference between them and dividing it by the clock frequency.

A crucial component of the security of any modern MCU is the secure communication between secure and non-secure parts of the firmware and hardware. Many boards, including the STM32H573, achieve this by implementing the ARM PSA API standard. This standard for platform security architecture ensures "execution isolation to safely manage and protect the computing resources of low-end IoT devices" [10]. Points of emphasis for the PSA standard include an enforced security lifecycle, attestation, device binding of stored data, and the use of trusted cryptographic services [26]. In order to implement the PSA functional API, a device must pass the test suites concerning cryptography, storage, and attestation. Notably, the API does not allow an application to access key values directly [26].

2.3 Trusted Execution Environment

As IoT systems become more complex and networks become increasingly connected, traditional security measures cannot guarantee protection and privacy for users. The original response to trusted computing, or the concept of attaining secure computation and data, involved a separate hardware module, the trusted platform module (TPM), to prevent tampering of sensitive data like cryptographic keys [23]. However, to overcome the limitations of TPM, which only allows strict APIs and no separate environment for execution, a new approach called a trusted execution environment (TEE) was established. A TEE is a secure and isolated environment that allows third-party code to be executed without outside tampering to its applications. A TEE's capabilities also extend to memory and storage protection [23]. While this environment guarantees the confidentiality and authenticity of the code executed on it, a TEE could not run without a separation kernel. This type of kernel partitions the system to ensure that different sections of the system can maintain varying levels of security. For instance, the separation kernel may control and secure information between the TEE environment and the normal operating system [23].

The building blocks of TEE include secure boot, secure scheduling, inter-environment communication, secure storage, and a trusted I/O path [23]. While the secure boot verifies code that can be loaded and executed, secure scheduling makes sure that the CPU executes processes from the TEE and regular system with balance and efficiency. Inter-environment communication ensures that TEE can communicate with the rest of the system and secure storage only allows authorized processes to access sensitive and protected data. Communication between the trusted execution environment and peripheral devices such as keyboard or mouse inputs and outputs is taken care of by the trusted I/O path to prevent attacks. Ultimately, the TEE ensures that software and hardware attacks do not compromise its runtime states or securely stored data. [23]

The Trusted Firmware-M (TF-M) is an open-source project that models how to implement a Trusted Execution Environment (TEE) for Arm-based processors. The TF-M works in parallel with the TEE to facilitate a secure software execution and isolation environment to secure embedded and Internet of Things (IoT) devices [3]. The TEE secures an environment within the main OS where critical operations can be performed; this includes cryptographic functions, secure key storage, secure boot, and more. No matter the status of the main operating system (secure, insecure), the TEE will always be secure. The TF-M complements the TEE with a concrete example implementation of these security features, including the secure boot, cryptographic libraries, secure vs. non-secure code regions, and more [3]. The open-source nature of this project ensures this implementation remains updated with security and software advancements for a wide range of use cases. The TF-M is critical in such a connected world full of sensitive data.

The use of MCUs in a healthcare setting is a perfect example of this. Connected pumps provide access over a remote or local network, changing access based on the role of the user. For instance, nurses have limited access while doctors have full control over dosages and other vital information. Such pumps have been attacked in the past, mostly with voltage and clock glitch attacks as well as MCU fault attacks [11]. One solution that has been proposed is a software platform, chiefly compatible with STM32 MCUs, that allows developers to evaluate their design against hardware threats by simulating common attack vectors [11]. These include the revealing of the secret key by analyzing voltage traces during the execution of cryptographic operations, something that can be mitigated with a high-quality power trace [11].

2.4 Bootloaders

In embedded microcontrollers, the bootloader is one of the first pieces of code to execute and is vital in starting system hardware. It provides protection for the device's flash memory, verifies user programs, monitors firmware updates, and often handles firmware errors [13]. The bootloader also ensures that protected actions such as reads and writes to memory are secure and hardware or software attacks are dealt with [30]. Since the bootloader has access to the MCU's RAM and flash memory, new security concerns arise. The general solution to patch this vulnerability involves integrating a generic safeguard into the bootloader, often called Code Readout Protection (CRP) mechanisms [30]. However, security concerns in the bootloader can compromise the entire system.

2.5 Root of Trust

The ST Immutable Root of Trust (STiROT) is integrated into the hardware of the Arm TrustZone STM32H573xx microcontrollers to enhance their security from a variety of external attacks presented by the untrusted environments surrounding these microcontrollers. Via an immutable "secure boot," we are able to ensure that only authentic firmware is run on our microcontrollers, which is often updated by an external device when connected to our microcontrollers [17]. Therefore, the STiROT aims to enhance security in a tri-fold manner: security of sensitive data, essential operations, and firmware. Physically, the STiROT is embedded into an immutable area of the system flash memory and it launches post-reset the "secure boot" before all executions; this boot launches the runtime protections as well as confirms application authenticity and integrity (first downloading and decrypting, as needed) whenever it is reset (or a voltage is introduced). Therefore there are two types of boots: one boot stage versus the two boot stages. In the first boot stage, the STiROT directly manages the user application, but in the two boot stages, the STiROT indirectly manages the user application via the uROT (the "updatable root stage") – which serves a second boot stage; before checking the authenticity of the user application code and data images in this mode, the uROT's code and data images must be confirmed. The uROT lives in the user's flash memory. If both the application code and data images' authenticity can be confirmed, the user application is run securely. In either scenario, if verification fails, the bootloader is launched and offers the opportunity to restart (download a new user application code or uROT depending on the boot stage) if such external access is permitted. Upon jumping to the uROT or user application code, the STiROT prevents further attacks via strategic configuration of the MPU (memory protection unit); this configuration specifies what code area can be run next [17]. In the two boot cases, the uROT configuration directly influences the security of the user application.

2.6 System Architecture

ST's security firmware version of TEE, known as the Secure Manager, implements ARM TF-M specifically for STM32 microcontrollers. Figure 1 outlines the structure of the Secure Manager and Figure 2 provides a detailed description of the architecture.

ST also provides a suite of software tools to configure and develop projects for their STM32-embedded devices. This toolchain includes the STM32CubeMX for project configuration, STM32CubeIDE for application development, modification, and debugging, and the STM32CubeProgrammer for facilitating flashing to the board.

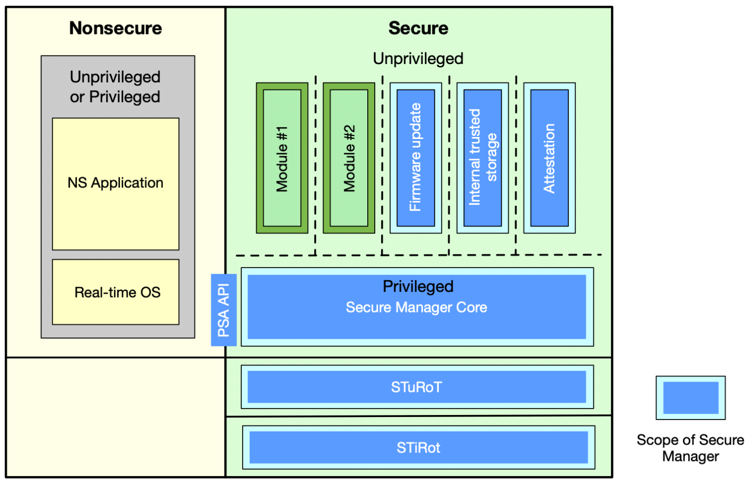


Figure 2.1: The Partition of the MCU's Non-Secure and Secure Modes

Secure applications and secure services sit on top of the Secure Manager Core [24]. In addition to providing Secure Boot and Secure Firmware Update, the firmware provides PSA standard services such as cryptography, trusted storage in internal flash memory, and initial attestation to authenticate devices. Acting as an operating system, the Secure Manager Core isolates secure services and non secure applications using the memory protection unit. The Core also inter-process communication between non-secure applications and secure services, isolation for secure applications, interrupt handling and scheduling for secure applications. Interaction with the Secure Manager Core occurs through the ARM PSA API. Ultimately, the Core is protected by full sandboxing [24].

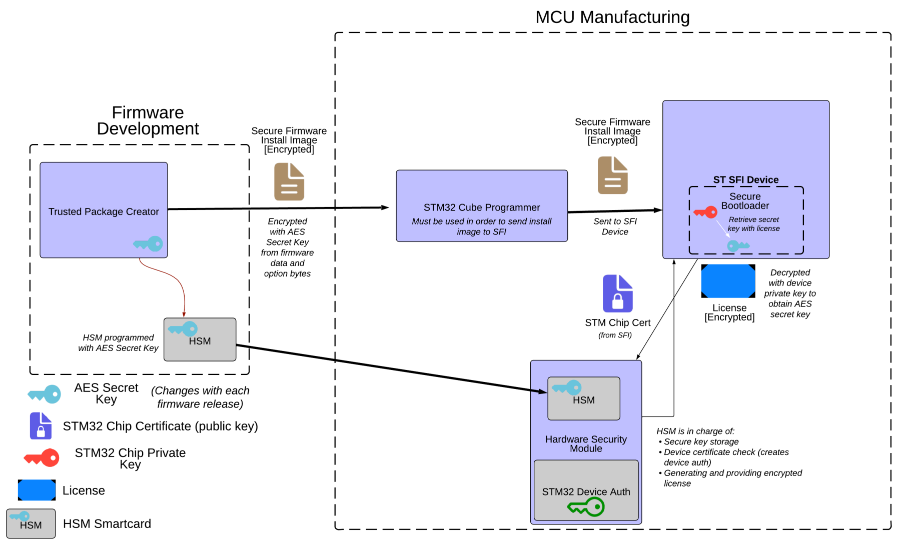

MCU Manufacturing
Firmware Development
Trusted Package Creator
HSM programmed with AES Secret Key
HSM
Secure Firmware Install Image [Encrypted]
Encrypted with AES Secret Key from firmware data and option bytes
STM32 Cube Programmer
Must be used in order to send install image to SFI
Secure Firmware Install Image [Encrypted]
Sent to SFI Device
ST SFI Device
Secure Bootloader
Retrieve secret key with license
STM Chip Cert
(from SFI)
License [Encrypted]
Decrypted with device private key to obtain AES secret key
HSM
Hardware Security Module
STM32 Device Auth
HSM is in charge of:
• Secure key storage
• Device certificate check (creates device auth)
• Generating and providing encrypted license
AES Secret Key
(Changes with each firmware release)
STM32 Chip Certificate (public key)
STM32 Chip Private Key
License
HSM Smartcard

Figure 2.2: Secure Firmware Interface Loading Process

ST provides developers with the Secure Manager Access Kit (SMAK) to create non-secure applications that use services provided by the Secure Manager [27]. Figure 4 illustrates the flow for this development. The first step involves installing the Secure Manager with the help of the STM32CubeProgrammer. Once the Secure Manager is installed, users can develop non-secure applications, that use the PSA API to call secure services, through STM32CubeMX and the STM32Cube IDE. Lastly, the Trusted Package Creator packages the finalized application code and creates an image to flash to the board [27].

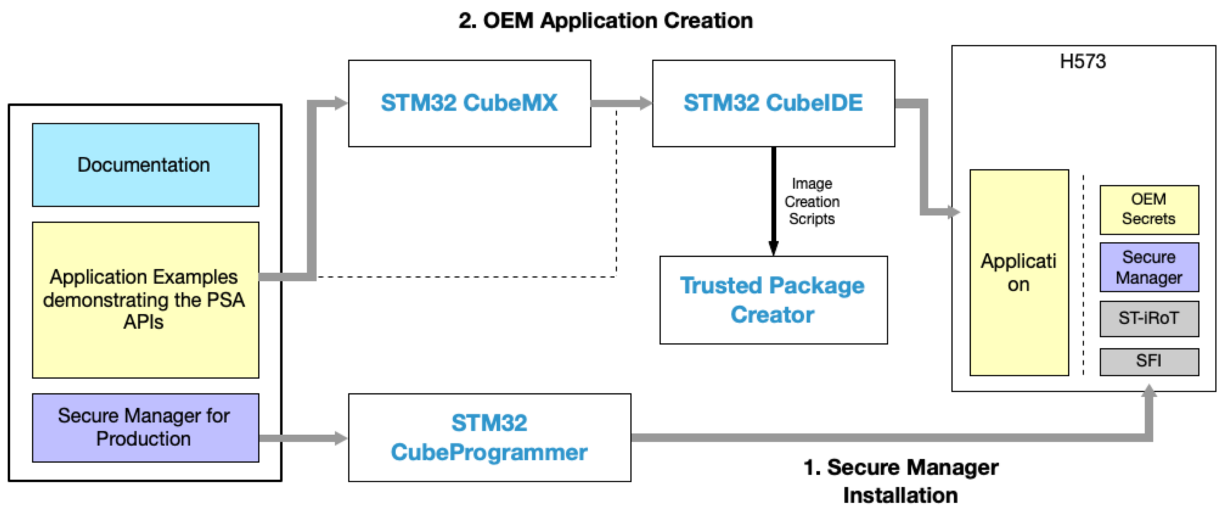


Figure 2.3: Creation of a Non-Secure Application Using Secure Services

2.7 Standards and Constraints

In our test plan for our MCU, we aim to benchmark the impact of the Secure Manager on the performance and energy consumption of industry-standard applications and functionality. To do this, we chose GPIO, I^2C , and ECDSA cryptography as the 3 applications to test. We also evaluated the boot time, which we defined as the beginning of power delivery to the start of application execution, and energy when booting into secure and non-secure modes. For all applications, we used analog write signposting (described in Section 4) for the best timing accuracy. We were constrained by the fact that the ST Secure Manager is extremely opaque and we have no way to monitor exactly what operations it is performing during boot time. To best mitigate this, we had to make use of the fact that we had precise control over when power was provided to the board and when analog writes occurred. While we could not profile the overhead and performance of the inner workings of the Secure Manager, our technique allowed us to measure its impact on the power consumption and performance of the board, particularly during boot time and application execution.

We started with GPIO, as it's commonly used to test basic board functionality and performance. Not only does it represent the most basic functionality with the lowest overhead for MCUs, but it is also integrated into the testing of all of our other applications that use signposting to assess runtime application performance. Our board provides multiple GPIO pins, which are individually configurable for use with TrustZone. Notably, these pins are all in analog and secure mode after reset (if TrustZone is enabled) [16]. To overcome these constraints, we had to explicitly configure each pin we wanted to use exactly as we wanted it, with the maximum output rate, so that fair comparisons could be made between secure and non-secure applications in terms of both power and performance.

Next, we tested the performance and energy of an I^2C application. We chose this as I^2C is the industry standard in communicating with sensors and other peripheral devices that interface with MCUs. I^2C was developed in 1982 by Philips Semiconductor, and the specification supports communication rates up to 100 kHz and 7-bit addresses [8]. This meant that our data had to be relatively small and that we couldn't use a lot of sensors on the bus. We approached this by using an Arduino UNO as our I^2C slave device with an ultrasonic sensor attached. With this setup, we could program our Arduino to ensure we were not overwhelming the bus. Also, we chose a distance sensor as it provided very small data samples that could fit into just 4 bytes. This allowed us to use a high sampling rate of 1000 Hz to collect 1000 data samples from the sensor via the Arduino, for a total data transfer of 4000 bytes per trial. On our STM32, this interface was provided by the D15/SCL and D14/SDA hardware pins on the underside of the board. Equivalent SCL and SDA pins could be found on the Arduino UNO which we used as a mock sensor (slave) device. Additionally, running identical I^2C workloads in secure and non-secure modes proved trivial, and signposting once again allowed for accurate performance timing.

Alongside I^2C , we tested the boot time and energy when the STM32 was booted into fully non-secure and secure applications. We defined the boot time as the time from which the MCU begins drawing power to when the application launches (indicated by a signpost) as mentioned above. Because of the signposting method described in Section 4, and the ability to toggle power to the STM32 from the PPK software when the PPK is operating in Source mode, this proved to be easily measurable using the power trace. Our only constraint here was that the PPK had a maximum sampling rate of 100 kHz [25], but this provided more than enough data and precision for our work.

Lastly, because of the focus on security in the STM32H573, we profiled the performance of the ECDSA algorithm. ECDSA is an industry standard for securely computing the signature of messages and other sensitive information, having been accepted by the International Standards Organization in 1998 [9]. We chose to use a 256-bit key that enables our board to use secure SHA-256 hashing instead of insecure SHA-1 hashing [6] and is as accepted as secure, with an example being its use in large-key AES [9]. While our board supported many other cryptographic features and algorithms alongside ECDSA, we found it to be the easiest to fairly compare across secure and non-secure contexts as other algorithms such as RSA were not supported in both secure and non-secure modes. Furthermore, while large keys can be computationally expensive to perform cryptographic operations with, the STM32H573 has a public key accelerator (PKA) which allows these operations to be both performant and secure [16].

Chapter 3

Societal Issues

There are many societal issues associated with securing MCUs, as they are increasingly used in a variety of applications as mentioned in Chapter 1.

3.1 Ethical

As attacks on IoT devices increase, MCU security is critical to ensuring the safe and proper execution of a wide range of systems we interact with on a daily basis. Some of these systems, like medical devices, are safety-critical and system failures or intrusions could lead to accidents or even deaths. The use of security in MCU devices is critical for protecting both consumer information and preventing IP theft of inventions. If MCUs cannot be secured, then consumer trust is eroded and there is little motivation to innovate on the platform if IP can be easily stolen by malicious actors.

3.2 Social

As our society continues to become increasingly connected and dependent on technology, the impact of their security and performance only continues to increase. Many MCUs, especially those made by ST, have applications in the automotive field where they control safety-critical functions like braking, steering, and airbag deployment. They also play a critical role in national functions like communication networks and transportation services. As such, the security and performance of MCUs is essential to the function of our society.

3.3 Economic

These results can provide a benchmark for the resources required by the STM32H573, which can be allocated in advance, and leveraged to reduce energy consumption when used in products and other practical applications. Increased security also reduces the chance and cost of data breaches and provides innovation incentives for companies to develop novel uses and products on top of MCUs that are both highly performant and highly secure.

3.4 Health and Safety

These MCUs are typically always active, therefore, they must adhere to the required and safe power/energy consumption levels. For applications like medical devices or car airbags, failures could be the difference between life and death. They also play a key role in the functioning of emergency systems like fire alarms or emergency lighting, and potentially dangerous industrial equipment. Their malfunction could jeopardize health and safety in emergencies or put workers in danger.

3.5 Environmental Impact

By allowing MCU designers, developers, and customers to better understand the energy impact of their products and applications, we hope our analysis allows for some energy to be saved. In turn, this can reduce the need to produce batteries or use polluting power sources. Also, MCUs that are more energy-efficient are more durable and last longer, preventing e-waste.

Chapter 4

Methodology

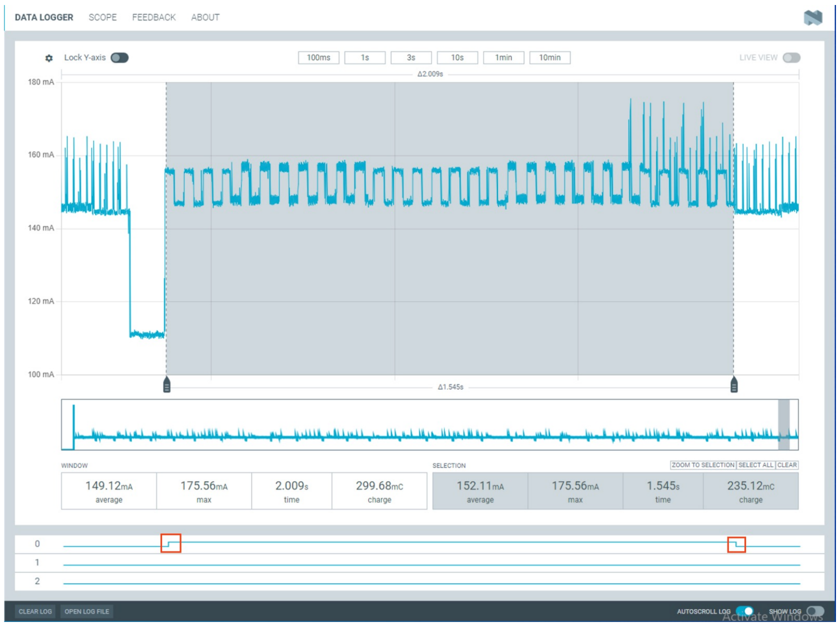
This study aims to isolate and identify the overhead caused by using the Secure Manager and its services in various use cases. By measuring the boot time into secure mode versus non-secure mode and the power consumption of applications flashed to the STM32H573, the results will exhibit how the Secure Manager affects current and execution time. The applications that will be tested include purely secure and purely non-secure applications as well as non-secure applications that use secure services such as cryptography and secure storage through the PSA API. An additional secure and non-secure test using the board's I^{2}C port, which is a communication protocol between devices, will also provide insight into the effects of the Secure Manager.

To most accurately analyze the performance of our MCU, we used analog writes to signpost the starting and ending points of our code's execution. These signposts can be seen highlighted in red boxes in Figure 4.1. Note how these signposts correspond with the measurement window in the trace shown in Figure 4.1. As such, the window represents the period in which the signal on line 0 of the logic analyzer is high (1), corresponding to when the application code of interest is executing. This technique is foundational to accurately measuring execution time in all applications that we tested. This window also provides us with the average current, maximum current, and charge used during the selected time frame. From these values, we can conclude the performance of the STM32H573 during the execution of any section of code we wish to analyze.

There were two physical testbench set-up configurations for these tests: for Non-Secure and for Secure (or using the Secure Manager). Figure 4.2 shows the PPK acting as a power source for a Non-Secure mode set-up. On the other hand, figure 4.3 shows the PPK acting as an ammeter source with the ST-link connection for using the Secure Manager on the board. Figure 4.4 outlines a photo of the real-world testbench set-up used for this project.

4.1 GPIO

Like many industry-standard MCUs, the STM32H573 supports general-purpose I/O in the form of analog pin control and controlling built-in LEDs on the board. These are useful for debugging applications and testing the basic func-



WINDOW

149.12mA average	175.56mA max	2.009s time	299.68mC charge

SELECTION

152.11mA average	175.56mA max	1.545s time	235.12mC charge

Figure 4.1: Code Signposting in Power Trace

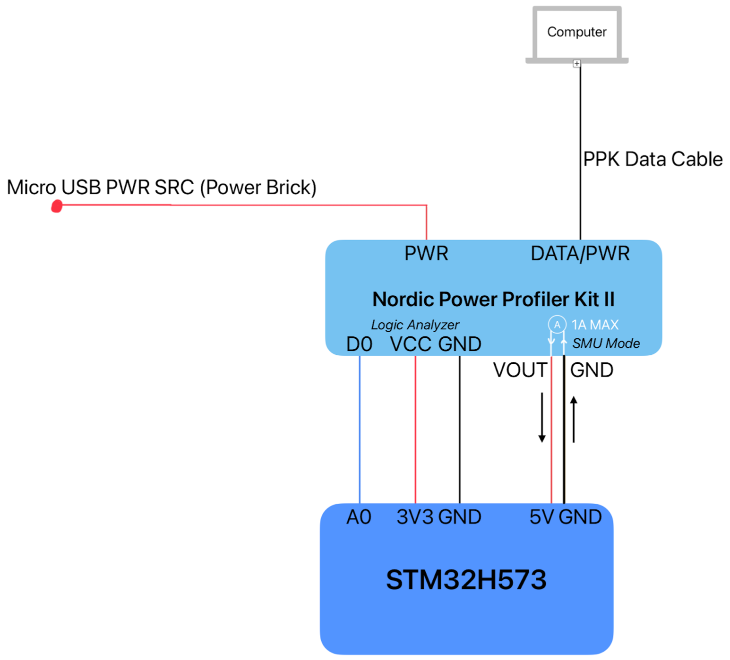


Figure 4.2: Testbench Diagram for PPK Acting as Power Source

tionality of the MCU. For our test applications, we chose to blink all user LEDs on the board 30 times, with a 25 ms delay between on and off for a total delay of 50 ms per blink cycle. This app was easily comparable across secure and non-secure modes due to its simplistic nature and configuration. In both configurations, all LEDs were set to

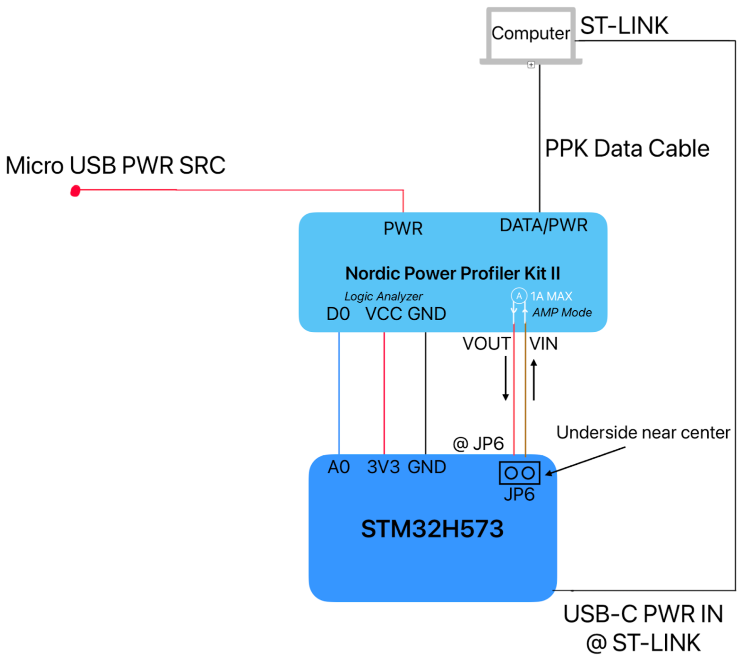


Figure 4.3: Testbench Diagram for PPK Acting as Ammeter Source with ST-Link

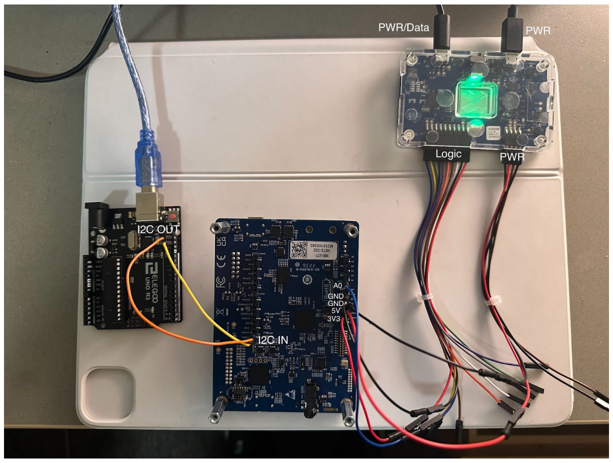


Figure 4.4: Example Testbench Setup for I^{2}C

Low maximum output speed (Very High was not supported by all LEDs) and in the Secure configuration, all LEDs were reserved to the secure part of the CPU (pin reservation). We then measured the exact runtime performance using signposting as described above.

4.2 Switching Between Purely Secure and Purely Non Secure Modes

Since the STM32H573 supports both secure and non-secure code execution and pin reservation, it is possible to switch from the secure mode to the non-secure mode. However, the reverse is not possible. Switching between these states

Table 4.1: I^2C Timing Metrics

	Non-Secure	**Secure**
Average	1.97680	1.95993
Median	1.97250	1.96500
St. Dev.	0.01148	0.02138

is done via the interrupt vector table, where the secure application retrieves the non-secure reset handler from the start of the non-secure portion of the table. It then calls this handler to switch the application to the non-secure mode.

4.3 I^2C Secure and Non-Secure Applications

In the project setup, the board's internal clock has been set to the maximum of 250 MHz, in order to minimize inconsistencies in data that may come from variations in clock frequency. Both a secure and non-secure application was implemented to use the I^2C port to collect data from a sensor. The port from the STM32H573 was directly connected to an Arduino UNO board with jumper wires, which was then connected to an ultrasonic sensor that measured object proximity. Although the Arduino is essentially the I^2C sensor in this case, the ultrasonic sensor was used to put dummy data on the wire. To standardize the results, 20 trials were run on this application using a 1000Hz polling rate, to measure the average, median, and standard deviation of how long it takes to receive 1000 I^2C data points. Figure 4.5 outlines the code used to test and signpost the I^2C applications. The statistical summary of the results is shown in Table 4.1.

```
long sampleCount = 0;
long sampleThreshold = 1000; //1k samples
startTiming();
while(1); {
      float distance; //written via memory
      uint8_t distanceBytes[4]; //32 bits total
      // sent 1000 times → 4000 bytes

      // Request 4 bytes of data from slave device (Arduino)
      if (HAL_I2C_Master_Receive(&hi2c1,
                (uint16_t) (ARDUINO_I2C_ADDRESS << 1, distanceBytes,
                sizeof(distanceBytes), HAL_MAX_DELAY) == HAL_OK) {
            // Convert received bytes back to float
            memcpy(&distance, distanceBytes, sizeof(distance));
            sampleCount++;
      } else {
            Error_Handler();
      }

      if (sampleCount > sampleThreshold) {
            stopTiming();
            break;
      }

      HAL_Delay(1); // Poll every 1 ms
}
```

Figure 4.5: I^2C Code Snippet with Signposting Function Calls

4.4 Cryptography Applications

One of the key features that distinguishes the STM32H573 from other MCUs is its wide-ranging support for cryptographic operations and functionality. The performance and energy use of cryptographic operations is a key part of evaluating the power and performance of security mechanisms. To evaluate this functionality, we chose to evaluate the ECDSA functionality, as it was supported in the non-secure mode and via the ARM PSA API. This let us compare the performance of the algorithm between non-secure and secure (PSA) modes.

Just as we did in other trials, we ”signposted” our code with digital writes to a pin on the STM32, which was connected to the built-in logic analyzer on our Power Profiler. You can see the code and resulting power trace in Figures 4.5 and **??** in Section 7. This allowed us to accurately mark and measure the execution of our code for power and performance purposes. This method was repeated for 30 trials for each mode. Our code performed one execution of the ECDSA algorithm for both the secure (PSA) and non-secure modes.

We found that the non-secure operation was much faster (12x faster than ECDSA via PSA), although it used 75% more current on average when compared to the PSA API. However, even with the much larger current draw, the performance advantage in non-secure mode led to less energy being consumed overall. We found that, on average, 1 run of the ECDSA signature algorithm in non-secure mode used 2.3x less power than a run of the algorithm in secure mode via the PSA API.

Chapter 5

Results and Analysis

Overall, this study developed, tested and analyzed purely non-secure, purely secure, and secure service approaches by implementing I^{2}C , GPIO, and ECDSA-based applications on the SMT32H573. Results and analysis from these tests are shown and described in this section.

5.1 Boot Results

Any device preparing to execute an application or perform an operation must first undergo a boot process. Because of this, our research also evaluates the energy consumption and duration of running a non-secure and secure boot. We defined this time frame as the time from which power is supplied to the board to when our application code begins to execute. As described in 4, we measured when this way by immediately signposting via the Logic Analyzer. Since we could control and quantitatively identify when power delivery to the board began in the power trace generated, this allowed us to accurately measure the boot time in each mode using the criteria we defined.

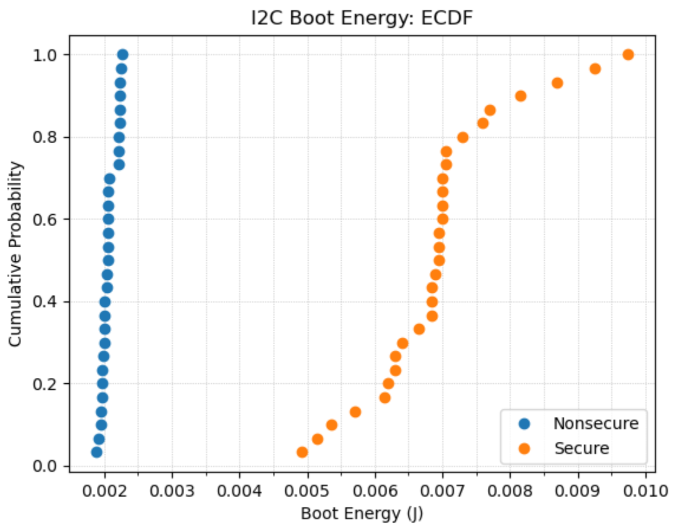


Figure 5.1: Energy Consumption of Boot Up Process

As shown in Figure 5.1, a secure boot consumed double the energy as opposed to a non-secure boot. In addition, there was 40% difference in current drawn and a nearly 500% increase in execution time longer for the secure boot process.

5.2 Boot Analysis

Diving deeper into the mechanics of why we may have seen these results for the two different boot processes, Figure 5.2 [24] shows the several boot paths that can be followed.

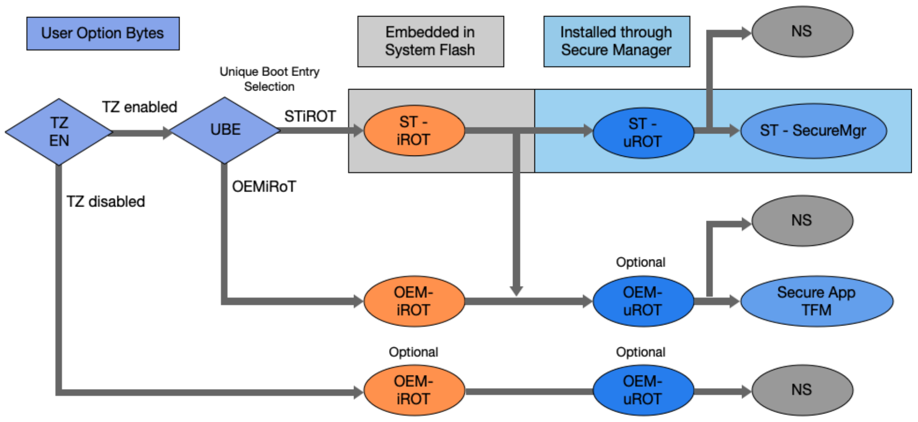


Figure 5.2: The details of the different paths that can be taken in the boot process

When a developer creates an application, they must configure the ’User Option Bytes’ in the project configuration before flashing to the board. Within this configuration, they must choose whether to enable or disable TrustZone. With TrustZone disabled, the boot process follows an optional Root of Trust (RoT) step, which this study did not include, and boots into the strictly non-secure domain. On the other hand, if TrustZone is enabled, the developer can configure the ’Unique Boot Entry’ selection to either take the third-party OEM Root of Trust or follow ST’s built-in Root of Trust embedded within the system’s flash memory. This project’s secure boot process followed ST’s immutable RoT, installed the Secure Manager, went through the updatable RoT and finally allowed us to either run a secure application or use the secure services provided by the Secure Manager. Ultimately, these extra steps involved in the secure boot, that are not involved in the non-secure boot process, could account for that significant execution time and energy consumption increase that was seen in the results.

5.3 GPIO Results

GPIO application results comparing the purely secure application to the purely non-secure application are shown in Figure 5.3.

As shown, there was about a 6% increase in energy for the non-secure application as opposed to the secure appli-

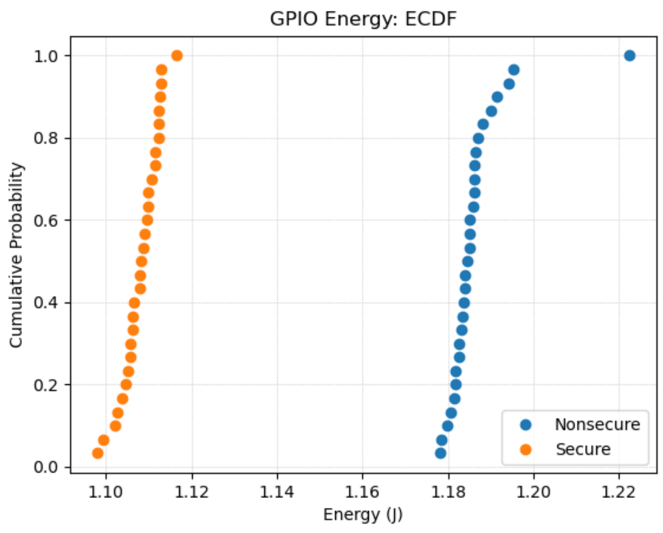


Figure 5.3: Energy Consumed by the MCU for the GPIO Applications

cation. Additionally, the runtime between the two applications had a nearly negligible difference at about 0.1% and a difference in current drawn by about 6%.

5.4 I^2C Results

I^2C application results comparing the purely secure application to the purely non-secure application are shown in Figure 5.4.

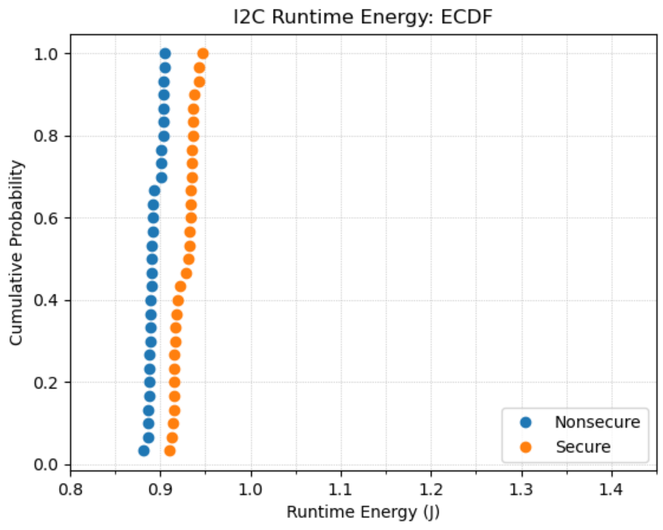


Figure 5.4: Energy Consumed by the MCU for the I^2C Applications

As shown, there was about a 64% increase in energy for the secure application as opposed to the non-secure application. Similar to the GPIO application tests, the runtime for I^{2}C between the two applications had a nearly negligible difference at about 1% and a difference in current drawn by about 5%.

5.5 Purely Non-Secure and Purely Secure Analysis

Both the GPIO and I^{2}C applications tested the performance and compared a purely secure and purely non-secure application, in which the only differences between the two were what domain they were running in. This analysis provides some insight into the background processes that may contribute to the energy and runtime differences seen in the results described above previously.

For a purely secure application, it boots in secure mode. As a result, this means that the Secure Manager is installed. The Secure Manager runs in the background constantly checking for security violations. For instance, it checks if a program or process is trying to access pins or other resources that are already allocated to a different program or process. Additionally, on the secure side, the applications are fully sandboxed, meaning control is restricted and programs can run in an isolation or virtual environment. Figure 5.5 [24] displays the temporal isolation set-up within the secure domain. A secure application with TrustZone can access or control the code or secrets of itself and anything above it. However, a non-secure application can only access or control its own resources. This temporal isolation set-up is only offered for a secure application in the context of the comparison in this study.

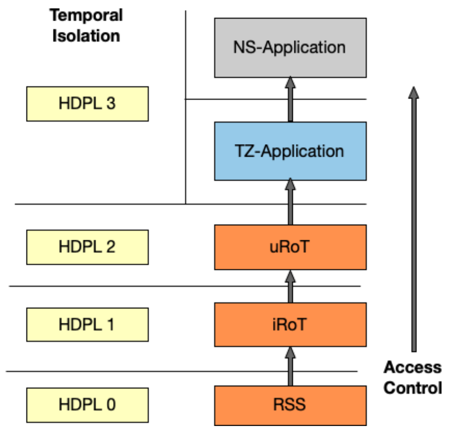


Figure 5.5: HDPL Temporal Isolation Levels with the Secure Manager

For a purely non-secure application, the board boots into non-secure mode. As a result, there is no temporal isolation set-up or split of resources between the domains, so the non-secure application has access to all resources. Additionally, since there is no installation of the Secure Manager, there are no underlying checks happening. Ultimately, the background processes such as checking by the Secure Manager and temporal isolation requirement on the secure side should result in a noticeable difference in energy consumption. However, the difference in runtime and energy consumption between the two applications in both I^2C and GPIO were very small. Although it isn't negligible, it is a very minimal difference especially compared to the differences seen with the boot results. Overall, this small difference paves the way for interesting discussions about Secure Manager efficiency and opens the door to running purely secure applications on the STM32H573.

5.6 ECDSA Results

As opposed to a purely non-secure versus purely secure comparison, the ECDSA applications in this study compared a purely non-secure application with a non-secure application that uses a secure service offered by the Secure Manager for the cryptography operation. Results are shown in Figure 5.6.

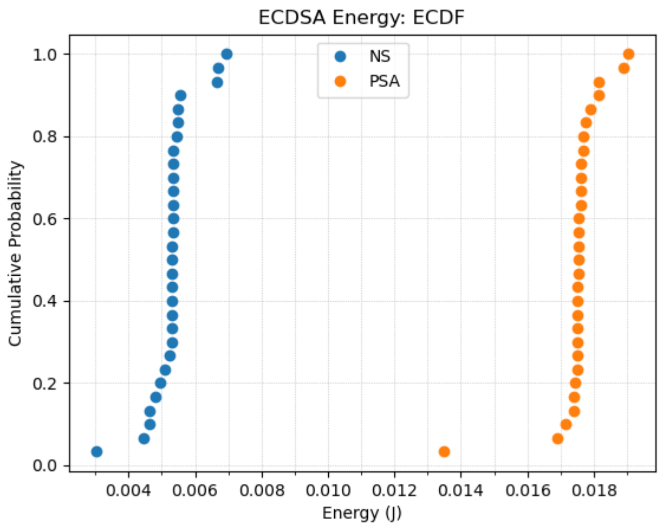


Figure 5.6: Energy Consumed by the MCU for the ECDSA Applications

As shown, there was about a 230% increase in energy for the secure application as opposed to the non-secure application. Additionally, the PSA application took about 10 times longer to run and drew about 75% more current than the purely non-secure application.

5.7 Secure Service Analysis

This study's ECDSA application tested and bench-marked the energy consumption from using the PSA secure services provided by the Secure Manager. This analysis aims to provide insight into why the results may have shown a significant energy consumption increase for the PSA application as opposed to the purely non-secure application.

When a PSA API call happens, in this case, to perform the ECDSA Sign operation, several actions are taken. An interrupt is triggered to switch control to the Secure Manager, which then takes over to perform the requested cryptography operation on the data. Once this operation is complete, another interrupt is triggered to switch control back to the non-secure domain to then continue executing the rest of the non-secure application. Additionally, the way that this data is shared also has a specific configuration. As shown in Figure 5.7 [24], there is SRAM space reserved for a non-secure application as well as for the Secure Manager itself. However, the orange shaded area indicates the NS/S shared buffer space. This space is where data must be stored for the Secure Manager to access it, so in this scenario, the non-secure application must store the fixed data to be encrypted in that specific buffer.

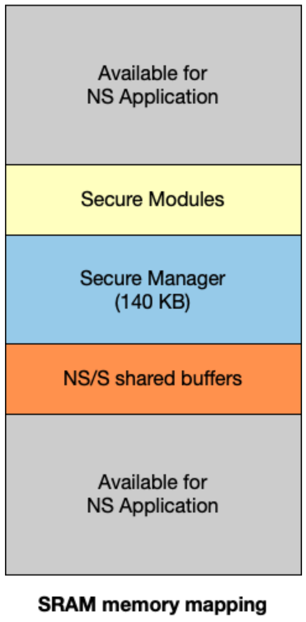


Figure 5.7: SRAM Mapping for Secure Services

On the purely non-secure side, there is no specificity on how data must be stored in the SRAM since there is no sharing between domains involved. Additionally, there are also no interrupts or context-switching happening either. Ultimately, stricter steps like having the shared buffer and background processes like interrupts for secure services could account for that significant runtime and energy consumption increase that we found in the performance results.

5.8 Summary of the Results

Table 5.1 displays a summary of the results discussed above. The asterisk by equivalent is to indicate a very minimal but not fully negligible difference in measurement between applications.

5.9 Use Case Recommendations

One of the biggest applications of this research is to be able to provide use-case recommendations to embedded systems designers and developers on what types of applications would be best for their products based on what metrics are most

Table 5.1: Summary of Results

	Best Performance	Lowest Current	Least Energy
Boot	Non-Secure	Non-Secure	Non-Secure
GPIO	Equivalent*	Secure	Secure
I2C	Equivalent*	Non-Secure	Non-Secure
ECDSA	Non-Secure	Secure	Non-Secure

*Minimal but not negligible difference

Table 5.2: Use Case Recommendations

Most Important Metric	**Recommendation**
Execution Time	Either*
Energy	Either*
Boot Time	Purely Non-Secure
Security	Purely Secure/PSA Services

*Purely non-secure and purely secure, Minimal but not negligible difference

important to them. Table 5.2 summarizes these recommendations.

For execution time and energy, both purely non-secure and purely secure applications had very minimal differences in performance metrics as discussed earlier. For boot time, for instance, if some device needs to be restarted constantly in its real-world operation, based on this study's results, a non-secure boot consumes significantly less energy and takes much less time to execute. Lastly, for security, based on our observations, the board itself in a secure domain allows a program to be temporally isolated, provides hardware cryptography services, and closes the board from outside influence. Since these security services are only offered in the secure domain, it would require a secure boot. Thus, if security is the most important metric for a use-case, the recommendation would be to use a purely secure application or secure services as the best approach.

Chapter 6

Conclusion

6.1 Summary

In sum, this study investigates the performance and security of the STM32H573 by measuring critical indicators (runtime, power consumed, etc.) between the nonsecure and secure modes of the MCU via a variety of applications (I2C, Boot Time, and ECDSA) and boot time. To measure these metrics, the Nordic Power Profiler Kit was leveraged which provides execution time, average current, maximum current, and charge (which are used to compute power). In terms of boot time, the secure boot consumed double the energy compared to the non-secure boot which may be explained by the fact that the secure boot has many more steps. In terms of the GPIO application, there was a 6% increase in energy for the non-secure application as opposed to the secure application. Regarding the I2C results, there was a 64% increase in energy for the secure application as opposed to the non-secure application. While the previous two differences are noticeable, they are minimal compared to the boot results and the small range of values for the GPIO and I2C. In terms of the ECDSA results, there was a 230% increase in energy for the secure application compared to the non-secure application which may be explained by the fact that many interrupts and steps are taken to execute the secure services.

Therefore, the recommendations are as follows: either secure/non-secure for execution time, non-secure/secure for energy, non-secure for boot time, and purely secure/PSA services for security. These results provide a window into what modes are best for what applications embedded systems developers can leverage to enhance the efficiency of their overall system.

6.2 Future Work

While this study provides meaningful insights into the Secure Manager and its services' impact on performance and security, there is additional work that can be done to build a comprehensive profile of the new STM32H573. The present environment indicates that the performance and security of MCUs is a large issue currently, and such a profile may help mitigate this issue.

Firstly, the performance of the remaining secure services may be tested, including attestation, internal trusted storage, and isolation. This study only leveraged one secure service, cryptography. Different secure services may exhibit different performance metrics based on a range of factors, including resource utilization and inter-service dependencies.

Secondly, this study could be expanded on to include multi-threaded applications. This study utilized single-threaded applications only that relied on one thread to execute items one at a time. Multi-threaded applications are responsible for executing multiple threads of execution concurrently. This approach would involve context-switching (between the different threads) and scheduling (algorithm determining when the different threads are each executed). Therefore measuring the energy and power consumption could reveal how this MCU would perform within more complex real-world embedded systems that require the MCU to perform multiple tasks rapidly and reliably. Such work may also illuminate how to best utilize the STM32H573's hardware acceleration to optimize the multi-threading and concurrent execution of threads.

Finally, optimizations for the Secure Manager and its performance may be explored. This may include exploring the internal mechanisms to determine areas where performance may be improved according to the results of the performance from various secure services, single-threaded applications, multi-threaded applications, and beyond.

6.3 Learning Outcomes

As none of us had any exposure to MCU development or benchmarking prior to this project, we learned a substantial amount about MCUs and testbed configuration. In particular, we identified three main areas of learning.

6.3.1 MCU Development

While we had all programmed in C before in classes, none of us had actually configured an MCU to run C code, much less one that was brand new to the market and had minimal documentation. While the code itself proved to be trivial, configuring the MCU so that code could be flashed, especially in the secure mode, proved to be very challenging. While ST's CubeMX tool provided a helpful GUI for configuring pins and services, it was overwhelming at first being presented with an array of pin options and services we knew little to nothing about. By the end of the project, we were very comfortable and experienced with setting up our project to use the pins and services in a way we understood.

6.3.2 Testbench Development

As CSEN majors, most of our background is in software so developing a hardware testbench proved to be something that was novel and uniquely challenging. To do this, we combined our software skills to configure the board to communicate with the logic analyzer via one of the analog pins. Then we abstracted the logic behind this into 2 functions we used to create the start and stop signposts. We then used our knowledge of the pins on the underside of the board to ensure it both had power from the PPK and could deliver power and logic signal to the analyzer so it would show on the software.

6.3.3 Project Management

Pacing and managing a year-long, research-intensive project was also challenging at times. We had a few hard deadlines and we worked hard to ensure that we had a strong background of research and understanding before we moved into implementing our project. Even when implementation proved challenging as it took us a couple of months to truly feel comfortable with running code on our board to accomplish our objectives, we had it done by the beginning of the Spring quarter as we had planned. This left us plenty of time to work on presenting our work and conducting additional trials as needed.

References

[1] Salah Adly et al. “Prevention of Controller Area Network (CAN) Attacks on Electric Autonomous Vehicles”. In: *Applied Sciences* 13.16 (2023), p. 9374.

[2] Tran Nguyen Bao Anh and Su-Lim Tan. “Real-time operating systems for small microcontrollers”. In: *IEEE micro* 29.5 (2009), pp. 30–45.

[3] ARM. *Trusted Firmware-M Documentation — Trusted Firmware-M v1.8.1 documentation*. ci-builds.trustedfirmware.org, 2022.

[4] Jacob Borgeson, Stefan Schauer, and Horst Diewald. “Benchmarking MCU power consumption for ultra-low-power applications”. In: *Texas Instruments* (2012).

[5] ARM Corporation. *Trusted Firmware-M*. Arm.com, 2023.

[6] Shay Gueron, Simon Johnson, and Jesse Walker. “SHA-512/256”. In: *2011 Eighth International Conference on Information Technology: New Generations*. IEEE. 2011, pp. 354–358.

[7] Center for Internet Security. *Blog — The Mirai Botnet - Tips to Defend Your Organization*. CIS, July 2021.

[8] Alvin Jacob, Wan Nurshazwani Wan Zakaria, and M Tomari MRB. “Evaluation of I2C communication protocol in development of modular controller boards”. In: *ARPN Journal of Engineering and Applied Science* 11.8 (2016), pp. 4991–4996.

[9] Don Johnson, Alfred Menezes, and Scott Vanstone. “The elliptic curve digital signature algorithm (ECDSA)”. In: *International journal of information security* 1 (2001), pp. 36–63.

[10] Junyoung Jung et al. “A secure platform model based on ARM platform security architecture for IoT devices”. In: *IEEE Internet of Things Journal* 9.7 (2021), pp. 5548–5560.

[11] Zahra Kazemi et al. “Hardware security evaluation platform for MCU-based connected devices: application to healthcare IoT”. In: *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*. IEEE. 2018, pp. 87–92.

[12] Chung Hwan Kim et al. “Securing Real-Time Microcontroller Systems through Customized Memory View Switching.” In: *NDSS*. 2018.

[13] Xiaoning Li et al. “Design and verification of mcu chip bootloader”. In: *2020 IEEE 9th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*. Vol. 9. IEEE. 2020, pp. 395–402.

[14] N Liebermann. “2021 IoT Security Landscape”. In: *SAM Seamless Network, Tel Aviv-Yafo, available at https://securingsam.com/wp-content/uploads/2022/04/SAM_IOT-Security-Report. pdf (accessed 26th June, 2022)* (2022).

[15] Yahia Mazzi, Ahmed Gaga, and Fatima Errahimi. “Benchmarking and Comparison of Two Open-source RTOSs for Embedded Systems Based on ARM Cortex-M4 MCU”. In: *Indian Journal of Science and Technology* 14 (Apr. 2021), pp. 1261–1273.

[16] ST Microelectronics. *Datasheet - STM32H573xx*. ST Microelectronics, Jan. 2024.

[17] ST Microelectronics. *Getting started with STiROT (ST immutable Root Of Trust) for STM32H5 MCUs*. ST Microelectronics, Sept. 2023.

[18] ST Microelectronics. *Secure Manager for STM32H5 - STMicroelectronics*. STMicroelectronics, 2023.

[19] ST Microelectronics. *STM32H563/573*. ST Microelectronics, 2023.

[20] NXP. *General Purpose Microcontrollers — NXP Semiconductors*. www.nxp.com.

[21] Associated Press. *Medical devices for pain, other conditions have caused more than 80,000 deaths since 2008*. STAT, Nov. 2018.

[22] Douglas Paulo Bertrand Renaux. “Comparative performance evaluation of CMSIS-RTOS”. In: *2014 Brazilian Symposium on Computing Systems Engineering*. IEEE. 2014, pp. 126–131.

[23] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. “Trusted execution environment: What it is, and what it is not”. In: *2015 IEEE Trustcom/BigDataSE/Ispa*. Vol. 1. IEEE. 2015, pp. 57–64.

[24] *Secure Manager*.

[25] Nordic Semiconductor. *Power Profiler Kit II*. www.nordicsemi.com, 2021.

[26] SeongHan Shin et al. “An Investigation of PSA Certified”. In: *Proceedings of the 17th International Conference on Availability, Reliability and Security*. 2022, pp. 1–8.

[27] *SMAK for STM32H5*.

[28] Doug Stiles. “The hardware security behind Azure Sphere”. In: *IEEE Micro* 39.2 (2019), pp. 20–28.

[29] Ioan Ungurean. “Timing comparison of the real-time operating systems for small microcontrollers”. In: *Symmetry* 12.4 (2020), p. 592.

[30] Jan Van den Herrewegen et al. “Fill your boots: Enhanced embedded bootloader exploits via fault injection and binary analysis”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), pp. 56–81.

[31] D Yu Weider. *Real-time operating system security*. Tech. rep. Technical Report. San Jose State University, 2010.

Chapter 7

Appendices

	Nonsecure: Boot Energy	Secure: Boot Energy
Mean	0.002073	0.006932
Median	0.002052	0.006950
St. Dev.	0.000115	0.001062

Table 7.1: Summary of Boot Results

	Nonsecure: Energy	Secure: Energy
Mean	1.186150	1.108098
Median	1.184800	1.108500
St. Dev.	0.007953	0.004312

Table 7.2: Summary of GPIO Results

	Nonsecure: Runtime Energy	Secure: Runtime Energy
Mean	0.893637	0.927172
Median	0.890725	0.931850
St. Dev.	0.007148	0.010702

Table 7.3: Summary of I^2C Results

	NS: Energy	PSA: Energy
Mean	0.005299	0.017525
Median	0.005300	0.017550
St. Dev.	0.000693	0.000872

Table 7.4: Summary of ECDSA Results