ENABLING EMBEDDED SOFTWARE SECURITY VIA INTROSPECTION THROUGH HARDWARE PERFORMANCE COUNTERS

A Dissertation

by

KARL OTT

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Rabinarayan N. Mahapatra
Committee Members,	Duncan M. Walker
	Riccardo Bettati
	Jiang Hu
Head of Department,	Scott Schaefer

August 2023

Major Subject: Computer Science & Engineering

ABSTRACT

Embedded systems are ubiquitous, and more systems are continuing to be deployed through the recent development of the Internet of Things. Embedded systems are specialized computer systems that are designed to perform specific tasks, typically with a dedicated function or limited application domain. They are engineered to be small, efficient, and low-powered, and can be found in a wide range of devices and applications, including cars, medical equipment, home appliances, and industrial machines. Embedded systems play a critical role in many of the devices and systems that we rely on every day, and their importance is only expected to grow as the world becomes more connected and reliant on technology.

Security has often been a secondary consideration in the design and implementation of embedded systems, with a focus on functionality and performance taking priority. However, this has changed in recent years as the risks associated with insecure embedded systems have become more apparent. One of the main challenges is that many embedded systems have limited resources, such as memory and processing power, which can make it difficult to implement strong security measures. Additionally, these systems may have a long lifespan and may not receive security updates, which can leave them vulnerable to attacks. A newer concern is that many embedded systems are designed to be connected to the internet, the Internet of Things, or other networks, which can increase their attack surface. Attackers can exploit vulnerabilities in these systems to gain access to sensitive data or disrupt their operation. Hence, securing these systems is becoming increasingly important. The aim of this dissertation is to increase the security of software in embedded systems.

We propose the use of existing hardware primitives commonly found on modern CPUs in embedded systems, specifically hardware performance counters and watchdog timers to enhance the security of embedded system software. The first method is to train a model that can detect anomalous execution of software. The second method we propose in this work is a way to have the embedded software continuously authenticate to ensure proper execution. The final piece of

the work is to enhance a traditional watchdog timer with hardware performance counters to make it more robust against malicious modifications.

DEDICATION

To my parents, brother, sister, and rest of my family.

ACKNOWLEDGMENTS

Like most contemporary works this dissertation is the result of many people and even more minds.

As I sit down to write this acknowledgment chapter, I am filled with an overwhelming sense of gratitude towards the individuals and institutions that have played a significant role in the completion of my dissertation. The journey to obtaining a doctoral degree has been a long and challenging one, but it would not have been possible without the help and support of those around me. Therefore, I would like to express my sincere appreciation to everyone who has contributed to my academic success.

First and foremost, I would like to express my gratitude to my supervisor, Dr. Rabi Mahapatra, whose unwavering patience, guidance, and support have been invaluable throughout the entire process of writing this dissertation. Your commitment to excellence, your attention to detail, and your passion for the subject matter have truly inspired me to become a better scholar. Your insightful feedback and constructive criticism have been instrumental in shaping my research and honing my skills as a researcher. I am forever grateful for the knowledge and wisdom you have imparted to me, and I will always cherish the time and effort you have invested in my academic growth.

I would also like to extend my gratitude to the members of my dissertation committee, Dr. Hank Walker, Dr. Riccardo Bettati, and Dr. Jiang Hu, for their invaluable input, edifying courses, constructive feedback, and critical evaluation of my research. Your thoughtful comments, insightful suggestions, and constructive criticism have helped me to refine my ideas and sharpen my arguments. I am grateful for your encouragement, support, and guidance throughout the process, and I am honored to have had the opportunity to work with such distinguished scholars in my field.

I would also like to thank the faculty and staff at Texas A&M University, who have provided me with a stimulating intellectual environment in which to pursue my studies. Some of the many are Bruce, Karrie, Dave, Sara, Andrea, and many others. The resources, facilities, and services offered by the university have been instrumental in supporting my research endeavors.

In addition to my supervisor and the university, I would like to extend my appreciation to my colleagues and fellow students, whose intellectual curiosity and camaraderie have made the journey more enjoyable and rewarding. All the members of my lab: Ankit, Jerry, DD, JD, Divyesh, Burak, and many other fellow students: Adam, Pulakesh, and many others.

Moreover, I would like to express my appreciation to my family and friends, who have been a constant source of support and encouragement throughout my academic journey. Your love, patience, and understanding have been a source of comfort during times of stress and anxiety. Your unwavering belief in me has given me the strength and confidence to persevere, even when the road seemed insurmountable. I am particularly grateful to my parents, who have always encouraged me to pursue my dreams and supported me in every step of the way. Your sacrifice, hard work, and dedication have been an inspiration to me, and I could not have done it without your love and support.

I am also grateful to the individuals who have participated in spirit in my research. Without their willingness to share their experiences and insights, this dissertation would not have been possible. Those of which are Maryella, Ben, Chris, Drew, Robin, Curtis, Ed, Kevin. I am especially grateful to the participants who have generously given their time and allowed me to discuss ideas with them. Their contributions have been invaluable in shedding light on the research questions and providing a rich and nuanced understanding of the topic.

I am deeply indebted to the individuals and institutions that have contributed to my academic success. Your support and guidance have been instrumental in helping me reach this milestone in my academic journey. I am honored and humbled by your generosity and kindness, and I will always treasure the memories and experiences we have shared. Thank you from the bottom of my heart.

Finally, I would like to thank God for His infinite grace and mercy. Without His blessings and guidance, I would not have had the opportunity to pursue my academic dreams. I am forever grateful for His divine providence and His steadfast love, which have sustained me during times of doubt and uncertainty. For the countless minds I have had the privilege of interacting with greater

than my own, I can know truly there is no mind greater than God's.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Rabi Mahapatra [advisor] of the Department of Computer Science and Engineering, Professor Hank Walker of the Department of Computer Science and Engineering, Professor Riccardo Bettati of the Department of Computer Science and Engineering, and Professor Jiang Hu of the Department of Electrical Engineering.

All work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by Teaching Assistant-ships from the Texas A&M University Department of Computer Science & Engineering.

NOMENCLATURE

HPC	Hardware Performance Counter
HMM	Hidden Markov Model
LSTM	Long Short Term Memory
IoT	Internet of Things
ROP	Return Oriented Programming
RNN	Recurrent Neural Network
CFI	Control Flow Integrity
CA	Continuous Authentication
EKG	Electrocardiograms
NDFA	Non-deterministic Finite Automaton
STFT	Short time Fourier transform
WDT	Watchdog Timer
FF	Flipflops
LUT	Look up table

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

1.1 Embedded System Software Security Lags Behind

It has often been quipped that the 'S' in IoT stands for security.

With the Internet of Things (IoT) boom billions of internet enabled embedded systems are being deployed yearly. Over the past 10 years, the number of IoT devices deployed worldwide has grown exponentially. While data for all IoT devices is not available, we can examine the trend from 2011 to 2021 based on available statistics. In 2011, the number of connected IoT devices was estimated to be around 1.84 billion, according to IoT Analytics [1]. By 2015, this number had risen to 15.41 billion [2], and by 2020, it reached 30.73 billion [3]. Looking ahead, projections suggest that the growth in IoT deployment will continue over the next decade. According to Statista, the number of connected IoT devices is expected to reach 75.44 billion by 2025 and 125 billion by 2030 [3]. This growth is being driven by the increasing adoption of IoT devices across various industries and the widespread use of smart homes and smart cities. As the technology continues to evolve and become more sophisticated, it is likely that we will see even greater numbers of IoT devices being deployed in the future. However, software security for these systems has only recently been designated as a core issue [4, 5, 6].

The Miria botnet [7, 8] and Satori [9] and Masuta [10] botnets have shown the severity that embedded system software insecurity poses. The Mirai botnet first emerged in 2016 and quickly gained notoriety for its size and power. It was responsible for several large-scale DDoS attacks, including the one that targeted DNS provider Dyn in October 2016, which caused major disruptions to a number of popular websites, including Twitter, Netflix, and PayPal. The Mirai botnet works by scanning the internet for vulnerable IoT devices that are protected by weak or default passwords. Once it finds a device, it infects it with malware and adds it to the botnet. The botnet can then be used to launch DDoS attacks on targeted websites or networks. Since its emergence, the Mirai botnet has evolved and spawned several variants, including the Satori and Masuta botnets. These

variants use similar techniques to infect IoT devices and create botnets for the purpose of launching DDoS attacks.

Security is a broad field that covers many aspects in digital systems. Commonly, there are three main pillars of security in digital systems. The three pillars are confidentiality, integrity, and assurance (CIA triad).

The CIA triad is a widely recognized model used to describe the fundamental principles of information security [11, 12, 13]. The acronym CIA stands for confidentiality, integrity, and availability, which are considered the three core pillars of information security.

Confidentiality: Confidentiality refers to the protection of sensitive information from unauthorized access or disclosure. It ensures that only authorized individuals or systems have access to sensitive data and that the data is not exposed to unauthorized users or systems.

Integrity: Integrity refers to the protection of the accuracy, completeness, and reliability of data. It ensures that data remains accurate and unaltered and that any changes made to data are authorized and legitimate.

Availability: Availability refers to the accessibility of data and resources to authorized users. It ensures that authorized users can access the data they need when they need it and that the system is available to them without interruption.

The CIA triad provides a framework for designing, implementing, and evaluating information security measures in organizations to ensure that information is protected against unauthorized access, manipulation, or destruction. The core of the work in this dissertation focuses on maintaining the integrity of embedded software. If a higher assurance of integrity can be established, the other two pillars are also elevated as if there is confidence that the embedded software is executing as expected then there can also be higher confidence that confidentiality has not been breached. Availability also increases with the higher confidence in the integrity of the embedded software since there is more confidence that the software is running as designed.

The internet at large has trended increasingly hostile since it obtained wide-scale adoption. This new class of embedded systems, IoT devices, pose a unique and novel security challenges. In

some ways it has echos of the security crisis of the early Internet in the 1990s. However, many of these embedded devices are not directly exposed to the Internet and are deployed behind a firewall or NATtted network.

IoT devices are often resource constrained which increases the challenge for adding in security after the initial main design. Making security a core design component often time increases to total time to market [14]. Time to market is often a critical factor in IoT systems. Being the first established vendor in the market can increase the initial inertia for the products [15].

Due to the heavy constraints of embedded system in terms of limited hardware capabilities and rapid time-to-market constraints many of the established security practices for servers, desktops, and mobile devices can not be effectively deployed [16, 17]. It would either increase the total cost of the end device in a very cost sensitive market, increase the development time to an unacceptably long time for a quick to market product area, or it is simply not suitable for these more constrained environments.

The challenges for securing embedded systems has driven a broad set of research both academically and industrially. Beyond the hardware and marketing constraints, a major challenge also lies in the sheer diversity of embedded systems and IoT devices [18]. Embedded systems are be incredibly diverse, as they are used in a wide range of industries and applications [19]. Some examples of embedded systems include: Consumer electronics: Embedded systems are found in everyday devices such as smartphones, smart home devices, and home appliances [20]. Automotive industry: Embedded systems are used extensively in modern cars for everything from engine management to safety systems [21, 22]. Industrial control systems: Embedded systems are used to control machinery and processes in factories and industrial settings. Aerospace and defense: Embedded systems are used in aircraft, spacecraft, and military equipment for navigation, communication, and control. Medical devices: Embedded systems are used in a variety of medical devices, including pacemakers, insulin pumps, and blood glucose monitors. Internet of Things (IoT): Embedded systems are at the heart of IoT devices, enabling them to connect to the internet and communicate with other devices [23, 24]. Robotics: Embedded systems are used extensively

in robotics for control, sensing, and actuation. Given this diversity, embedded systems vary widely in their size, complexity, functionality, and performance requirements. They may be designed for different operating conditions, power sources, and environmental factors, which can all impact their design and performance. Therefore, the design of an embedded system is typically tailored to the specific requirements of the application it is intended for.

However, many of the proposed solutions in research would require almost a starting over from the ground up to build these systems with security in mind from the start.

1.2 Basics Of Embedded Systems And IoT

Embedded systems are computer systems that are designed to perform a specific task, often with very specific requirements. They are typically small, low-power devices that are integrated into larger systems or products. Examples of embedded systems include everything from the micro-controller in your microwave oven to the sensors and controllers in a modern car.

Embedded systems and Internet of Things (IoT) devices are two related concepts that involve small, specialized computer systems that are designed to perform specific tasks.

Embedded systems: An embedded system is a computer system that is designed to perform a specific function within a larger system. These systems are often integrated into other devices such as automobiles, medical equipment, and home appliances. Embedded systems are typically small, low-power devices with limited computing resources.

The key components of embedded systems include the following.

Hardware: Embedded systems are built using hardware components such as micro-controllers, sensors, actuators, memory devices, and other peripherals.

Software: Embedded systems are programmed using software that is designed to run on the hardware components. This software is often written in low-level languages such as C or assembly language.

Real-time operations: Embedded systems are designed to respond quickly to external events or inputs, often in real-time. This requires the system to be optimized for speed and efficiency.

Power consumption: Embedded systems are often battery-powered or have limited power

sources, so they need to be designed to be energy-efficient.

Connectivity: Embedded systems may need to communicate with other systems or devices, so they often include connectivity options such as Wi-Fi, Bluetooth, or Ethernet.

Security: Embedded systems may need to protect sensitive data or control critical systems, so security features such as encryption and authentication may be necessary.

Testing and verification: Embedded systems must be thoroughly tested and verified to ensure they meet their specifications and perform as expected. This often requires specialized testing equipment and techniques.

Internet of Things (IoT) devices: IoT devices are a type of embedded system that are designed to connect to the internet and communicate with other devices. These devices are typically sensors, actuators, or other small devices that are integrated into larger systems. IoT devices often have wireless connectivity and can be controlled remotely.

Sensors and actuators: Sensors are devices that detect changes in their environment and provide data to other devices. Actuators are devices that perform actions in response to data received from other devices. Both sensors and actuators are commonly used in embedded systems and IoT devices.

Micro-controllers: Micro-controllers are small computer chips that are designed specifically for use in embedded systems. They typically have limited processing power and memory but are optimized for low power consumption.

Programming languages: The programming languages used for embedded systems and IoT devices are typically lower-level languages such as C and assembly language. These languages allow developers to write code that directly controls the hardware of the device.

Connectivity: IoT devices are designed to be connected to the internet, which allows them to communicate with other devices and exchange data. Common connectivity protocols for IoT devices include Wi-Fi, Bluetooth, and cellular networks.

Security: Because IoT devices are connected to the internet, they can be vulnerable to security threats such as hacking and data breaches. It is important to design IoT devices with security in

mind, including strong authentication and encryption protocols.

1.3 Research Focus

The work presented in this dissertation aims to be addressable to the here and now of the embedded software security crisis without requiring a massive rearchitecting of established design, programming, and engineering principles. However, it also aims to be amenable to any future developments in the software security space that will happen. The main goal set out to be achieved within this work is to characterize what the "normal" or "known good" intrinsic characteristics of embedded software under execution is during runtime. Ideally, this must be done in a way that fits all the usual constraints for embedded software. As such it must have low to no overheads as to not increase the requirements for the base software or overall power requirements for the device, and it must also not significantly increase development time. This is accomplished by looking at the properties that are intrinsic to the embedded software that is being executed by way of hardware performance counters.

1.4 Contributions

The contributions of this dissertation are as follows:

1. This research investigates using the signals of HPCs to enable on device anomaly detection for embedded system software. The approach used is to stream the HPC data signals to a predictor. The predictor is trained on previous known good executions of the software and it takes a small snapshot of HPC data and makes a prediction for the next value of the counters to come. If the prediction performance falls outside of a certain threshold then it concludes the software is not operating as expected and issues a corrective action.

2. This research propose the use of a continuous authentication technique for the embedded software to pass. It uses frequency analysis on slices of the time series signals generated by the HPCs. The frequency response is encoded into a finite state machine on known good runs of the software. When deployed the same encoding scheme of the frequency response happens, but this time it is checked against the previously defined state machine. If too

many invalid transitions in the state machine occur then the software fails the authentication checks and is considered to be compromised from the original intention.

3. This research enhances an ordinary watchdog timer, which are commonly used in embedded applications, with HPC data. This provides a multi-factor security approach. Not only does the software have to routinely "pet" the watchdog timer, but it must also have a matching HPC response. If either the timer expires, or the HPCs does not match then the software fails the check and corrective action is taken.

1.5 Organization

The dissertation is organized as follows. Chapter 2 gives a design for securing embedded software using hardware performance counters with anomaly detection. Chapter 3 approaches the problem with a continuous authentication scheme based on hardware performance counters. Chapter 4 proposing enhancing a watchdog timer with hardware performance counters to increase robustness and security of embedded software. Finally, Chapter 5 concludes the dissertation as well as giving some avenues for future work.

2. HARDWARE PERFORMANCE COUNTERS FOR EMBEDDED SOFTWARE ANOMALY DETECTION[1]

A recent trend in software security has utilized hardware performance counters as a security mechanism for integrity checks as well as malware detection. In this work we have developed two methods to check and validate the runtime integrity of a program to protect against malicious intrusions. The two methods developed utilize Hidden Markov Models and Long Short Term Memory neural networks trained on traces of a program's performance counter data which allows for classification, offline anomaly detection, and online anomaly detection. In our benchmark of embedded software the HMMs achieved a classification accuracy of 100%, while offline anomaly detection achieved an average 98% accuracy with only 1% false positives, and online detection with a heuristic achieved 95% with only 0.38% false positives. On the same embedded software benchmark LSTMs neural networks achieved an offline anomaly detection rate of 100% with no false positives, and an online anomaly accuracy was 98% on average with no false positives.

2.1 Introduction

Software security vulnerabilities are a widespread unsolved problem. Software security vulnerabilities range from simple configuration issues to serious fundamental design flaws. Each day potentially vulnerable devices are deployed which may contain serious security vulnerabilities that could allow for an unauthorized user to access sensitive information. Part of this trend is a result of The Internet of Things (IoT) and the demand for smart devices. IoT devices are typically resource constrained and designed to have a quick time to market at low cost. This results in devices that may have little to no consideration for security concerns [26]. As a result of resource constraints these devices usually have to write all the world facing software in a low level, potentially dangerous, language such as C [4, 6, 5]. The use of an unsafe low level language exposes another entire

[1]Reprinted with permission from [25]. © 2018 IEEE. Reprinted, with permission, from Karl Ott, Rabi Mahapatra, Hardware Performance Counters for Embedded Software Anomaly Detection, 16th Intl Conf on Dependable, Autonomic and Secure Computing, August 2018.

class of vulnerabilities, most notably memory safety violations. Violations in memory safety can allow for an adversary to gain control over a piece of software. Traditionally memory safety violations are achieved by means of a buffer overflow. At a high level buffer overflows allow users to write past the end of a buffer dedicated for an operation. The result of a buffer overflow can lead to corruption of data, crashing programs, or unauthorized arbitrary code execution [27]. Arbitrary code execution is a violation in the vulnerable program's integrity which allows for the execution of malicious code.

Security vulnerabilities can lead to but are not limited to: data leaks, automated remote code execution, and denial of service attacks. Ideally vulnerabilities would be removed entirely, but due to real world constraints they must be minimized through mitigation techniques. Mitigation techniques aim to raise the bar for an adversary to successfully exploit a security vulnerability. Many mitigation techniques have been deployed or proposed in the literature ranging from fundamental changes in the software stack, to simple changes at an operating system level that are seamless in nature. Techniques that require fundamental changes are typically more thorough, complete, and have a much lower associated performance overhead. These deep-level changes are slower to penetrate commercial products due to changes in the base level software stack [28, 29].

Therefore, rapid changes in the resource constrained, quick to market segments, such as embedded or IoT devices, require a lightweight, non breaking mitigation technique to be introduced. As such, we propose a security mechanism which has minimal hardware requirements, low performance overhead, raises the bar for adversaries to successfully exploit, and is layerable with other vulnerability mitigations. Specifically, the methods proposed are layerable with other security mitigations, to offer defense in depth, such as but not limited to: address space layout randomization, W xor X memory permissions, stack canaries, and control flow integrity. The methods are also readily layerable with static integrity checking solutions provided in the Trusted Platform Module [30].

In this work we are proposing two models that recognize and enforce a program's expected execution trajectory which is obtained from hardware performance counters. Given an accurate

model of the program's expected trajectory, differences between the expected model output and calculated model output signify a fault in the software which may be malicious. The proposed method is particularly suited for embedded systems which often have a well defined set of functionality they must provide according to the system's specifications. The method is also flexible in implementation. The proposed models are then compared against each other, as well as a method similar to one that has previously been proposed in the literature [31].

The contributions of this paper are two new novel techniques using HMMs and LSTMs, which have varying costs, that can be used to help mitigate against a certain classes of software vulnerabilities and defects. Further contributions are that these techniques are implemented in a black box fashion requiring no source or binary level changes to the software it is protecting. Additionally, while demonstrated in software here the implementation for these techniques is flexible and could be moved to a lower hardware level implementation. A hardware implementation would lower the associated overheads across the board. Moreover, online versions of the HMM and LSTM techniques achieve a high accuracy with an average of 95% accuracy with only 0.38% false positives and 98% accuracy and 0 false positives, respectively.

The paper is structured with Section 2.2 detailing the background information of hardware performance counters, Hidden Markov Models, anomaly detection, and related work. Section 2.3 describes the setup used to carry out the experiment. Section 2.4 reports the results obtained from the HMMs from the experiment and Section 2.5 reports the results for the LSTMs. Section 2.6 discusses the findings reported and compares against other techniques. Finally, Section 2.7 is the conclusion and additionally outlines potential future work.

2.2 Background

2.2.1 Hardware Performance Counters

Hardware performance counters have been widely available on microprocessors since the early 1990s. Traditionally, performance counters were few and those that existed were limited to what events they would count [32]. However, in modern microprocessors there are upwards of 8 counters

per core, depending on the processor, which in turn each count hundreds of different events. Events range from instruction mixture, such as: number of loads, number of stores, number of branch instructions; to microarchitectural events such as the hit count of the cache at each level, number of successfully predicted branches, etc. Since these counters are implemented in hardware they offer a low overhead method to empirically measure a program's performance characteristics [33, 32]. In addition it can be shown that one natural output of hardware performance counters is a time series representation of a program's performance characteristics. This time series representation can then be compared against future executions of the program to identify malicious attacks or program contamination.

The time series hardware performance counters form is the basis that this work is founded on. Since counter values are cheap to measure, accurate, and repeatable we can use them to build a model to describe what values we expect to see over time.

2.2.2 Anomaly Detection

Anomaly detection, also known as outlier detection, is the identification of: items, events or observations which do not conform to an expected pattern or other items in a dataset. The importance of anomaly detection is due to the fact that anomalies in data translate to significant (and often critical) actionable information in a wide variety of application domains. Anomaly detection differs from misuse detection in that only "good" or "expected" behavior is defined. That is, there is no "bad" or "unexpected" behavior to compare against, unexpected behavior is only detected if it falls outside the expected behavior. Whereas misuse detection only succeeds when the observed behavior matches that which is known to be "bad" behavior [34]. Alternatively, anomaly detection can be viewed as a classification problem in which there is only one class, "good" or "expected".

Our methods presented in this work make use of the fact that we only know what the "good" or "expected" performance counter behavior is for each program and any deviations or outliers from the expected values are flagged as anomalous.

2.2.3 Hidden Markov Models

Hidden Markov Models (HMM) are defined by a model λ, where λ is defined to be the set of matrices A, B, and π. The matrix A represents the probabilities of a state transition. The B matrix is known as the observation probabilities, or emission probabilities. The columns represent the observation seen and the rows represents the hidden states. Lastly, π is the starting state probabilities. The "hidden" portion in the HMM is the state, which is to say that the state is not directly observable but can be reasoned from the observations. From the Markovian Assumption, the values of the future state depend only on the current state. That means the state at an arbitrary point in time contains all that is needed to be known in order to predict the future of the process. These properties make HMMs well suited for application in time series analysis.

Traditionally there are three fundamental problems with HMMs. The problems are:

1. Given a model λ calculate the probability of the observation sequence O.

2. Given the model λ calculate the most probable state sequence of an observation sequence.

3. Lastly given a set of observation sequences calculate a model λ that best fits the given observation sequences.

The first problem is also known as the filtering problem and can be efficiently solved by the Forward-Backward algorithm. The second problem to calculate the most probable state path is solved via the Viterbi algorithm. The third problem is commonly referred to as training an HMM which is typically accomplished with Baum-Welch algorithm. Additional information regarding HMMs and the algorithms to operate on them can be found in [35].

In this work we make use of trained HMMs' ability to calculate a probability of a given observation sequence. The calculated probabilities in the training set are then used to create a normal distribution on which we base our anomaly detection. That is, if the output probability of the HMM is an outlier on the constructed normal distributions gets flagged as anomalous.

2.2.4 Long Short Term Memory Neural Networks

Long Short Term Memory (LSTM) neural networks are a kind of Recurrent Neural Network (RNN) that enables the neural network to be capable of learning long term dependencies that may be present in the input. As a result LSTMs are effective and scalable for problems relating to sequential data [36]. As mentioned previously hardware performance counter samples naturally form a time series, which is a form of sequential data, which LSTM neural networks naturally excel in dealing with.

We use trained LSTM neural networks in this work to implement a form of anomaly detection. Specifically, we train the LSTMs on a windows of history of hardware performance values and have the LSTM network make a prediction of what the next value following end of the window is. If this predicted value falls outside of a threshold of what the actual measured value is, then it is flagged as anomalous.

2.2.5 Threat Model

The threat model allows for an attacker to exploit vulnerabilities that enable arbitrary code execution, such as a traditional stack overflow that hijacks the program's control flow into attacker controlled memory. Since the proposed method is flexible in implementation the requirements for the boundary an adversary can penetrate are also flexible. If the detection mechanism is implemented in kernel space then the only requirement is an adversary cannot gain unfettered kernel space access. If the detection mechanism is implemented in hardware then this requirement is removed, however other changes might be required.

2.2.6 Related Work

There has been some previous work that investigated the use of hardware performance counters for security purposes. Demme et al. used hardware performance counters to detect known malware on the Android platform and rootkits on a Linux host via standard classification algorithms [37]. Wang et al. used hardware performance counters to detect a kernel modifying rootkit in virtualized systems [38]. Ozsoy et al. modified a basic x86 compatible processor to enable malware detection

on chip with a hardware subsystem [39]. Hardware performance counters have been used to create a "golden signature" of linear relations which the system firmware is periodically checked against to see if it has been modified [40]. In [41] performance counters are used in a power transform and one class SVM to build an unsupervised anomaly detection scheme to try and detect return oriented programming (ROP) exploits at different stages of execution. CFIMon is a method utilizing performance counters as part of a system to enforce control flow integrity (CFI) by way of static analysis to build a table of all known good branch and jump target addresses and comparing against run time target addresses [42]. [31] used performance counters to verify integrity of device firmware by modifying and manually inserting code detours to check against a "golden signature" database.

Unlike the previous work our work does not rely on a golden signature to check against, modifying of source or binary code. Additionally, our work does not suffer from cross contamination noise in the samples due to sampling hardware performance counters in a system wide mode, and is readily layerable with existing mitigation techniques. For golden signature methods to work effectively the program must be extensively profiled to create a signature which can verify the program for any given valid input. This is additionally problematic when a program contains loops which iterate a number of times that can only be determined at runtime or contain an unprofiled early exit condition. If golden signatures techniques do not account for such cases programs with these constructs will be incorrectly identified as malicious.

2.3 Experimental Setup

2.3.1 Hardware/Software Configuration

The method for collecting performance counter data is the built in subsystem in the Linux kernel called perf_events. The subsystem allows for a different configurations for collection. For our setup we use an available frontend that allows for the collection of performance counter data without modifying or recompiling the code. The perf_events interface also has other advantages in that it will automatically save and restore the performance counter values when the monitored

program is scheduled out. The results of this is that we can get accurate traces of the values of the performance counters over time since other program's values will not be falsely attributed to the monitored one. The method used to collect performance counter samples was to read the values every N amount of retired instructions. In this experiment we set the value of N to 262144 retired instructions. This value was chosen as it did not create substantial overhead for the monitored process (less than 5%) and still provided accurate sample values as seen in Figure 2.1. The figure shows the increase in relative overhead to an unmonitored process as well as the amount of samples collected with an increasing sampling rate.

The data collection was carried out on the 4.10-32 Linux kernel, on a Intel Core i7-6600U processor. This particular processor has 4 configurable counters with 3 fixed functionality counters per core. We profiled the benchmark programs from the CoremarkPro suite from EEBMC [43], these are industry standard benchmarks for embedded systems. Additionally, the kernel's frequency scaling governor was adjusted so as to not have the processor's frequency scale down when there is a light system load running. The benchmark programs were compiled statically with musl [44], a minimal C library. These measures are done in an effort to reduce noise, help increase repeatability between program runs, and more accurately simulate an embedded software environment.

2.3.2 Choice Of Event Type

Weaver et al. show that the current implementation of performance counters are inherently noisy [33]. Moreover, many of the performance events can capture data of shared resources such as: caches, branch predictors, memory prefetchers, etc. These performance events are susceptible to interference from other programs that run alongside the monitored program. Furthermore, it is trivial to demonstrate that performance counters that track shared hardware resources, such as caching performance, are unreliable outside of a controlled environment. Figure 2.2 shows the average repeated runs of the same program with the same inputs and different levels of background interference.

There are hundreds of performance events available but only a small subset of them can be cap-

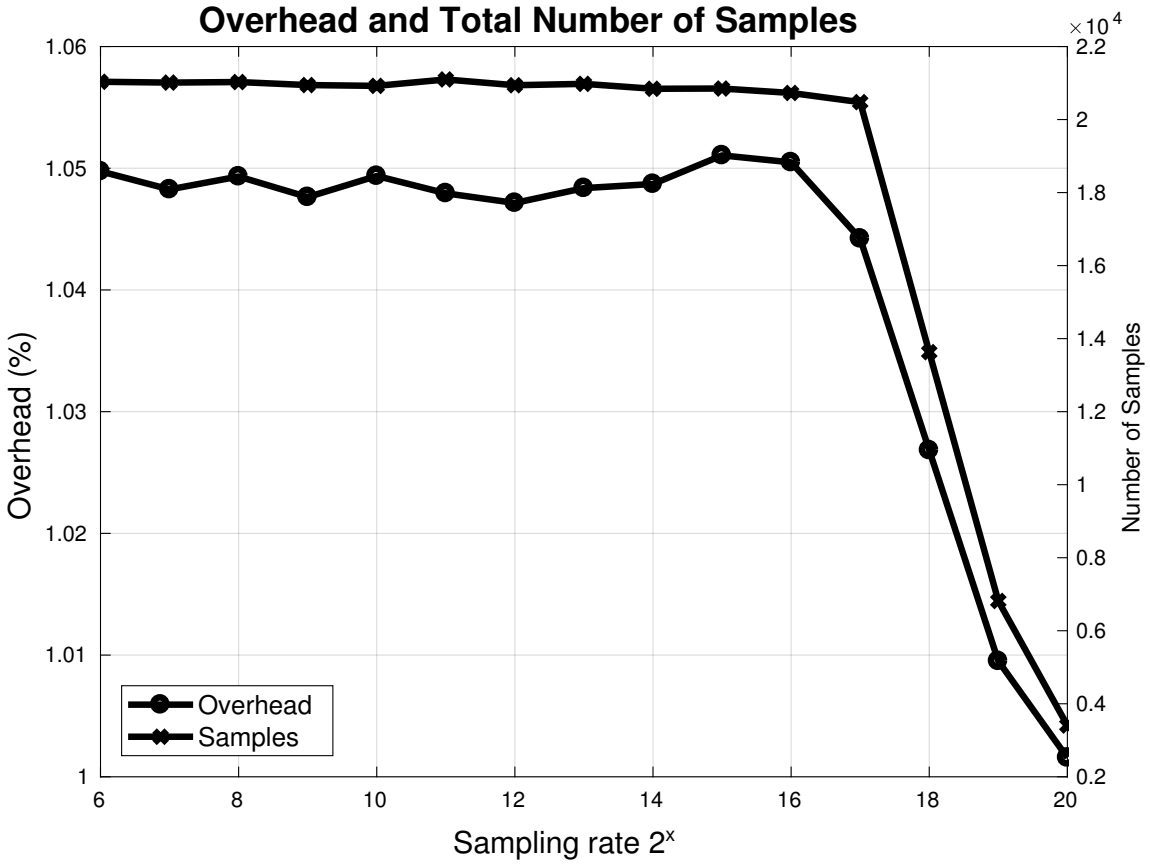

Figure 2.1: The average overhead and number of samples collected at different sampling rates of retired instructions. Around 2^{17} the number of samples collected becomes unstable as it does not increase as the sampling rate increases.

tured at once. To overcome these challenges the performance counters selected in this experiment are ones that tend to be more reliable and repeatable as demonstrated in [33]. Specifically, the four events used in this experiment were:

1. Retired branches
2. Retired conditional branches
3. Retired loads
4. Retired stores

Other events might contain more discriminating information, such as floating point operations, however there is no guarantee that a program will use floating point instructions which limits the usefulness of tracking such an event. It should be noted that after 262144 retired instructions the 4 dimensional data point collected will only report counts for the events listed above.

2.3.3 Hidden Markov Model Design

After the data was collected it was further discretized to reduce the number of observation symbols for the HMM. The method to reduce the number of observation symbols was to divide the range of each event into 14 equally sized bins and 2 larger bins on each side of the equally spaced bins. The edge bins are extended to go from 0 to the first bin edge and the last bin edge to 262144. The edge bins are defined this way so that there is always an observation symbol for every possible value. These 16 quantized observations were then used as the observations in the HMM. We used a 2 hidden state system for the state transition matrix in the HMM. The Markov model is then trained using the Baum-Welch algorithm on the quantized observations using known good runs of the benchmark programs. The state transition matrix, A, was initialized to be roughly $\frac{1}{2}$ for each transition and the emission matrix, B, was initialized to be roughly $\frac{1}{16}$ for each element. Each of the benchmark programs used had their own trained Markov model for each event that was recorded.

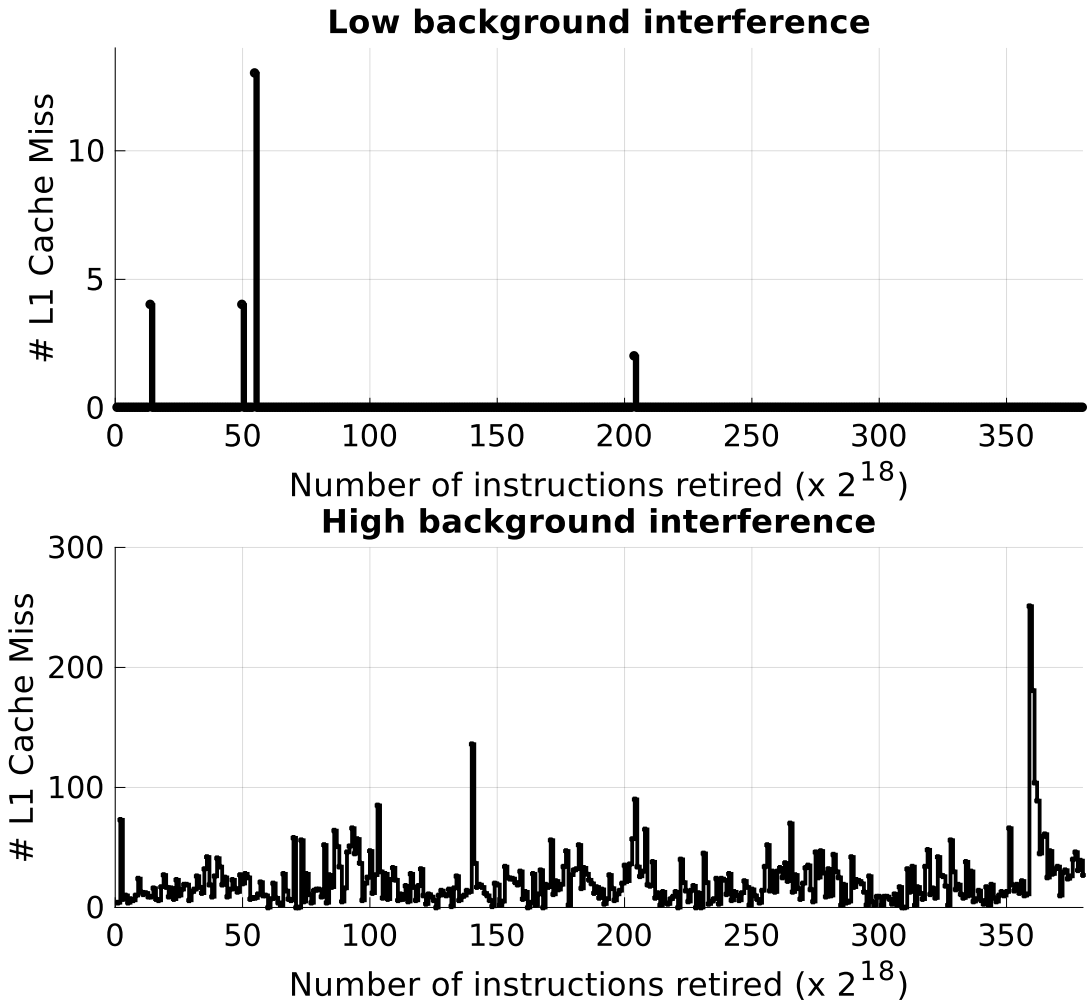

Figure 2.2: The average number of level 1 instruction cache misses over time for the same program with the same inputs with different levels of background interference between multiple runs.

HMMs allow for powerful inference of an observation sequence which we can use the models to classify an unknown observation sequence to the most likely corresponding program. Additionally, the trained models can be used for anomaly detection. There are multiple methods for anomaly detection with HMMs, in this work we examine the following methods:

1. Using the forward algorithm to determine the probability of the current observation sequence and then flagging it as anomalous if it fails to change for a number of observation points in an online manner.

2. Calculating the likelihood of a given observation sequence corresponding to a known program and flagging it as anomalous doesn't pass a defined threshold.

Figure 2.3 shows the high level architecture and data flow for the proposed system.

2.3.4 LSTM Neural Network Design

The LSTM network's training set is the same training data set that was used for the HMM. Each application's run in the dataset is windowed and used as an input to the network and the target output value is the value immediately following the window. Once trained the LSTM's output is used as an expected value and is compared against the actual measured value, and if the value falls outside of a defined threshold it is flagged as anomalous. See Figure 2.4 for a high level diagram of the setup for the LSTM neural network.

The specific setup is to train a 4 layer LSTM network on each of the performance counters that are being monitored. The first 2 layers of the LSTM are LSTM cells of 50 and 100 nodes, respectively. The third layer is a fully connected dense layer of 100 nodes, and the last layer is a single fully connected node. The raw performance counter values are mapped down into a space range of -1 to 1 and used as input to the LSTM network. The output of the network is a value in the -1 to 1 range, which is then mapped back into full performance counter range. With this setup the highest value in the data set is 1 and the lowest value is -1, with intermediate values falling in between. The result is that each program has four different LSTM networks that are making a prediction for the next value to follow.

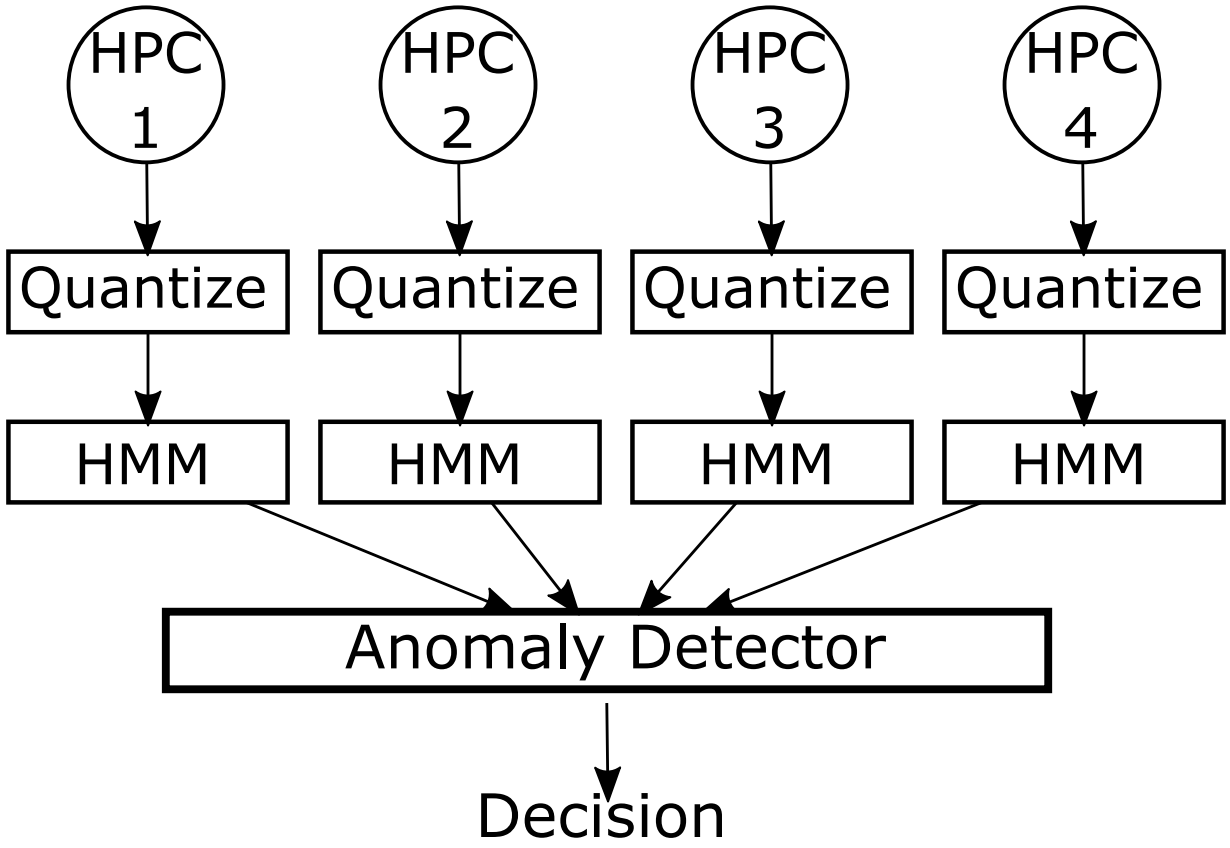

Figure 2.3: The architecture and dataflow for anomaly detection using HMMs. Each protected program would have its own quantization edges, HMMs, and anomaly detector values. This is operating under the case that we know a certain program is executing and the model is used to verify correct, or expected, operation.

We use two different methods for online and offline anomaly detection:

1. If one of the 4 LSTM network mispredicts at a rate outside of a defined threshold the program is flagged as anomalous.

2. The total count of mispredictions is stored and if it passes a defined threshold the program is flagged as anomalous.

2.3.5 Validation Data Set

The validation set of data was constructed using the unused portion of the training set, known good runs that were included in the training set, and runs of the program that were modified at runtime to produce a modified version of that program. The types of modifications to the programs are done to test different conditions. Specifically, the modifications include:

1. Replacing the monitored program with another randomly selected program in the benchmark at a random starting place roughly between the first and last quarter of expected execution, referred to as Mixed.

2. Replacing the monitored program with constrained randomly generated instructions at a starting point between the first and last quarter intervals, referred to as Random. The instructions are constrained in order to prevent infinite loops and segmentation faults from occurring.

3. Identical to the first method but the output of the monitor program's run is truncated to be the same length as the original sequence length, which is referred to as Limited. The output run is truncated in order to test the performance

2.4 HMM Results

In this section three different methods utilizing HMMs are evaluated. Specifically we look at using HMMs for classification as well as a two different techniques for anomaly detection.

2.4.1 Classification

Once the HMMs have been trained for each event for their respective program they can readily be used to classify an unknown observation sequence. In this case the unknown observation sequence is run through the binning step for each program's specific performance event and the probability of the newly quantized unknown observation sequence can be calculated. For example, let $O = \{o_1, o_2, o_3, o_4\}$ be a set of unknown observation sequences belonging to a specific program where o_n is an sequence corresponding to one of the performance events being tracked. Each o_n is quantized using each program's bin set for each event. Using the defined HMM for each event of a known program a probability is calculated for the newly quantized observation sequence. The model for each event that produces the highest probability for the unknown sequence defines which class the sequence belongs to. In the case where a sequence contains multiple classes a decision is made by selecting the mode class if possible. If a mode selection is not possible then the class that corresponds to the highest probability is selected. Figure 2.5 shows the classification process for an unknown observation sequence.

To further test the classification scheme a set of HMMs was trained for each program using a holdout validation method. The holdout method set aside 25% of the dataset to use as a testing set, while the remaining 75% was used as the training set. Using the procedure previously outlined the successful classification rate on the testing set was 100%.

2.4.2 Offline Anomaly Detection

The HMMs can be utilized to also support anomaly detection. This case is similar to classification but differs because the sequence is known to belong to a particular program. Since the sequence is already known to belong to a specific program we can use the HMM to calculate a probability and determine a threshold for anomalous behavior. This method is referred to as offline detection because it is assuming that the program will have executed to completion. However, if a program's control flow is hijacked and it starts to run arbitrary code there is no guarantee that execution will halt. Offline detection can be accomplished by using a training set to estimate a

distribution of probabilities of normal behavior and outliers to the constructed distribution can be classified as anomalous.

The offline method was used to construct a distribution based on the probabilities of known good samples from the data set and classify outliers as anomalous achieved the following results. By calculating the mean and standard deviation and experimentally determining the bounds based on the standard deviation which offers a high accuracy rate we were able to achieve an average 98% successful anomaly detection rate with only 1.0% false positives on our validation benchmarks described earlier.

2.4.3 Online Anomaly Detection

Online anomaly detection aims to overcome the shortcomings of the offline variant by detecting anomalies closer to when they have occurred, ideally as soon as they occur at runtime. The implementation we have chosen is to use the difference in the probabilities of the first T and $T-1$ observations. This approach is particularly appealing because it does not add any additional computational overhead in calculating a sequence's probability, only extra space is required. The approach is used to build a heuristic which counts how many consecutive repeated probabilities have occurred during known good runs, L a tunable parameter. This approach has an additional benefit that if the probability transitions fall within L, but the anomalous execution path runs for longer it has a greater chance of being detected. With this tunable heuristic in place observation sequences can be tested against and checked if anomalous.

The results for online anomaly detection on the same validation benchmark for each program is shown in Figure 2.6. Averaging these results in 2.6 we get a 95.3% successful detection rate with 0.38% false positives.

2.5 LSTM Results

2.5.1 LSTM Classification

The current structure of the LSTM is not well suited for a traditional classification task. An alternative approach would be to train a neural network to do multiclass classification instead of

prediction. This task is beyond the scope of this paper.

2.5.2 Offline Anomaly Detection

Offline detection can be accomplished this setup of LSTM neural networks in a similar fashion to HMMs. The difference is instead of deciding based on the probability output from the HMMs we look at the rate of mispredictions from the LSTM to determine if it should be flagged as anomalous. If the number of mispredictions fall above a defined threshold the entire run can be flagged as anomalous.

The offline detection with LSTMs on our validation benchmark achieved 100% accuracy with 0 false positives.

2.5.3 Online Anomaly Detection

Online Figure 2.7 shows the results for online detection utilizing LSTM neural networks. The methodology for detecting anomalous runs is to define a threshold for an allowable number of mispredicted values. Once the threshold is passed the program is flagged as anomalous. The results shown in Figure 2.7 demonstrate that LSTMs are suitable for applications with performance counter data.

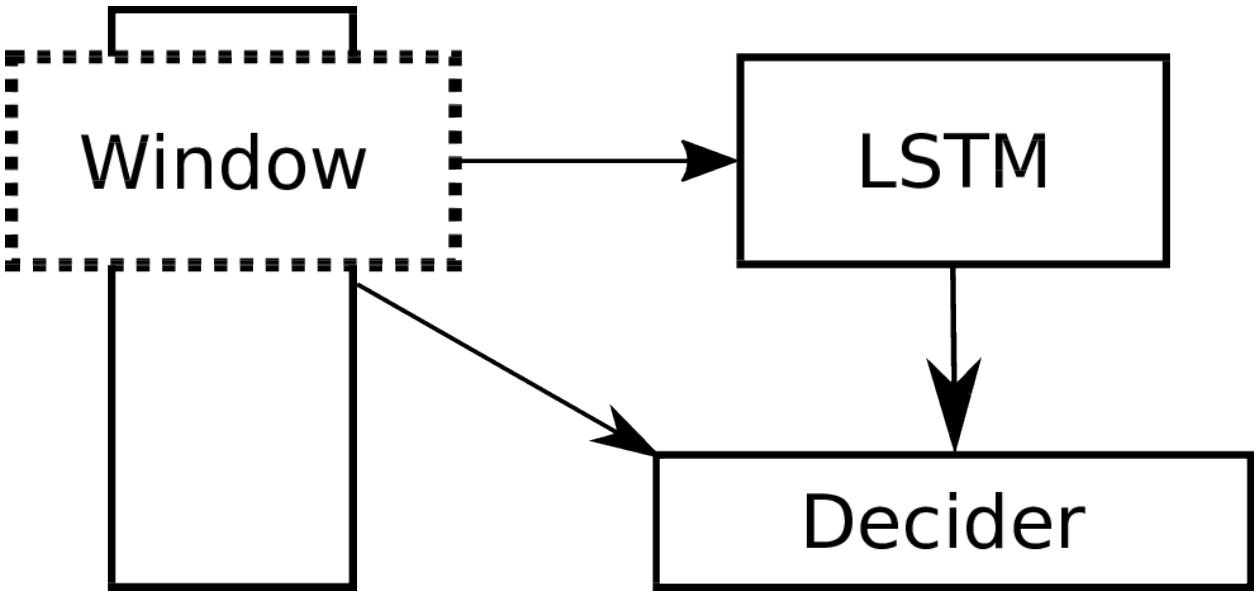

Figure 2.4: The architecture and dataflow for online anomaly detection using LSTMs. Each protected program would have its own LSTMs for each event monitored, in the case of this experiment 4. The LSTMs are given an input window starting at $N_{t-winlen}$ to N_t which is used to predict the value at N_{t+1}. Here t denotes the current time and $winlen$ refers to the length of the window.

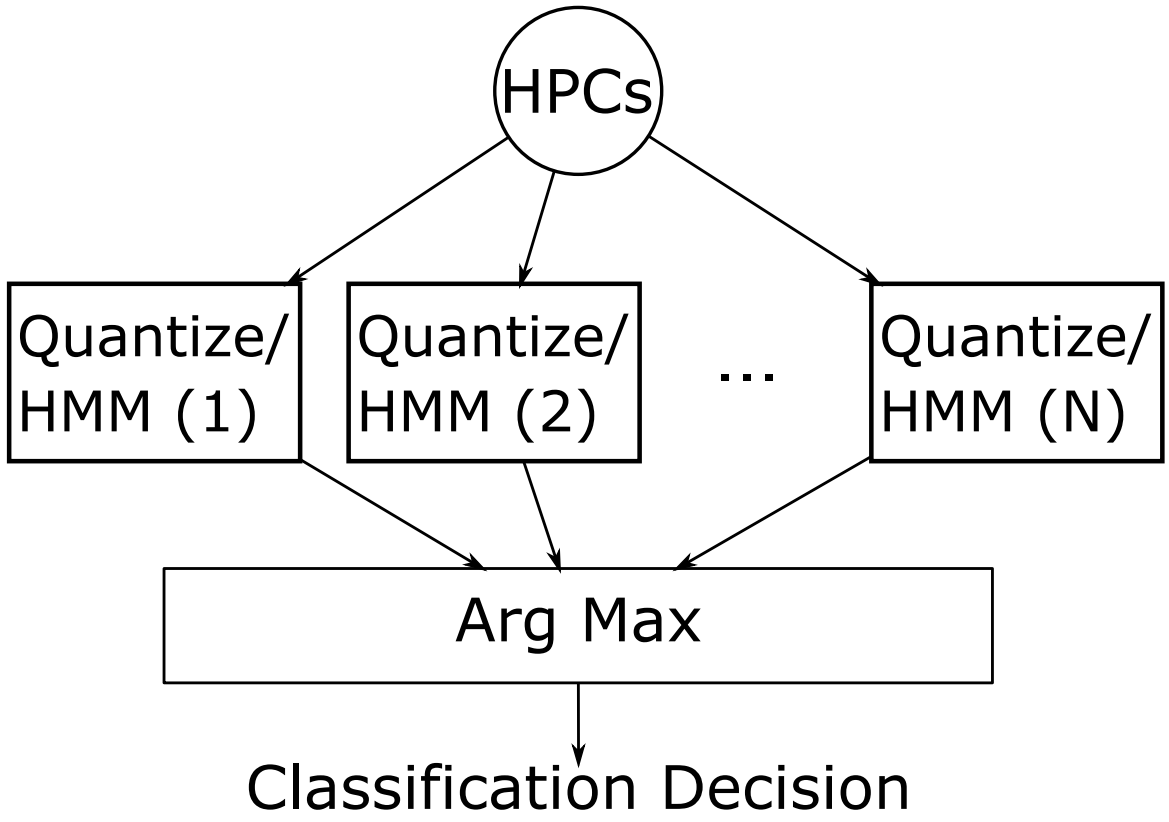

Figure 2.5: The architecture for using HMMs with hardware performance counters to classify an unknown observation sequence. For each program to be monitored there would exist a corresponding quantization and HMM probability calculation. The classification is determined by selecting the corresponding model with the highest probability.

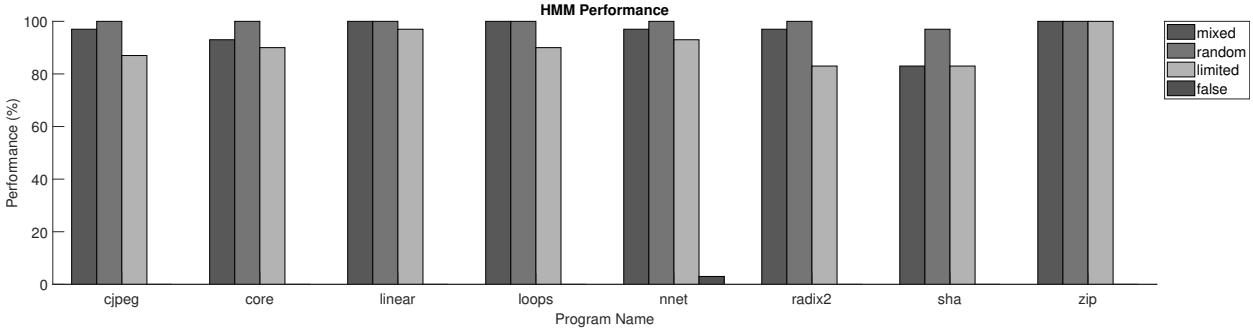

Figure 2.6: The results from using HMMs to detect anomalies. The results for each program shown here are using the best parameters for the L and how many standard deviations away from the mean. They were specifically selected in order to maximize detection and minimize false positives. Each bar corresponds to one of the previously defined validation sets, going from left to right: Mixed, Random, Limited, and false positive rates respectively.

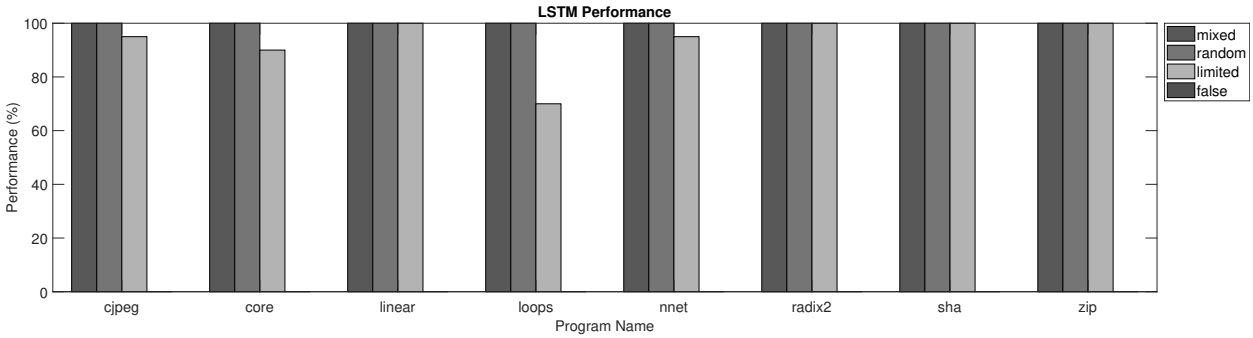

Figure 2.7: The detection performance of the LSTM neural networks. In this experiment if one of the four LSTMs mispredicted an amount of times over a previously defined threshold the program is flagged as anomalous. Again, the bars going from left to right represent the: Mixed, Random, Limited, and false positive rates respectively.

Type	Len	cjpeg				core				linear				loops				nnet				radix2				sha				zip			
Std Dev		1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Mixed	1	97	97	97	87	93	93	93	90	100	100	100	100	100	100	80	70	100	97	97	97	100	97	90	70	83	83	83	83	100	100	100	100
	2	97	97	97	77	90	90	90	90	100	100	100	100	100	100	83	70	100	97	97	97	100	97	93	87	83	83	83	83	100	100	100	100
	3	97	97	97	80	90	90	90	90	100	100	100	100	100	100	83	70	100	97	97	97	100	90	87	83	83	83	83	83	100	100	100	100
	4	97	97	97	87	90	90	90	90	100	100	100	100	97	97	80	60	100	97	97	97	100	93	80	73	83	83	83	83	100	100	100	100
	5	97	97	97	93	90	90	90	90	97	97	97	97	97	97	80	60	100	97	87	77	100	97	80	70	83	83	83	83	100	100	100	100
	6	77	77	77	77	90	90	90	90	97	97	97	97	97	97	80	60	100	97	87	77	97	93	77	67	83	83	83	83	100	100	100	100
	7	77	77	77	77	90	90	90	90	97	97	97	97	97	97	80	60	100	97	87	77	97	93	80	67	83	83	83	83	100	100	100	100
	8	77	77	77	77	90	90	90	90	97	97	97	97	97	97	80	60	100	97	87	77	97	93	80	67	83	83	83	83	87	87	87	87
Random	1	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	87	53	97	97	97	97	100	100	100	100
	2	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	90	90	90	90	100	100	100	100
	3	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	77	83	83	83	83	100	100	100	100
	4	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	93	67	77	77	77	77	100	100	100	100
	5	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	87	47	67	67	67	67	100	100	100	100
	6	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	77	43	63	63	63	63	100	100	100	100
	7	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	93	73	40	50	50	50	50	100	100	100	100
	8	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	90	70	40	40	40	40	40	100	100	100	100
Limited	1	87	77	77	70	90	90	90	90	100	97	97	97	97	90	67	57	100	93	93	93	87	83	57	30	83	83	83	83	100	100	100	100
	2	87	77	77	67	90	90	90	90	100	97	97	97	97	93	70	57	97	93	93	93	87	83	63	57	83	83	83	83	100	100	100	100
	3	87	77	77	67	90	90	90	90	100	97	97	97	97	93	70	53	97	93	93	93	87	87	73	67	83	83	83	83	100	100	97	97
	4	87	77	77	77	90	90	90	90	100	97	97	97	93	90	73	50	97	93	93	93	83	83	63	57	83	83	83	83	100	100	100	100
	5	87	80	77	77	90	90	90	90	97	97	97	97	93	90	73	50	97	93	83	73	83	83	67	57	83	83	83	83	100	100	100	100
	6	77	77	77	77	90	90	90	90	97	97	97	97	93	90	73	50	97	93	83	73	83	83	70	57	83	83	83	83	100	100	100	100
	7	77	77	77	77	90	90	90	90	97	97	97	97	93	90	73	53	97	93	83	73	83	83	70	57	83	83	83	83	100	100	100	100
	8	77	77	77	77	90	90	90	90	97	97	97	97	93	90	73	53	97	93	83	73	80	63	53	43	83	83	83	83	87	87	87	87
False Positives	1	37	0	0	0	10	10	0	0	63	7	0	0	33	0	0	0	60	7	3	0	43	0	0	0	0	0	0	0	50	0	0	0
	2	40	0	0	0	0	0	0	0	37	0	0	0	37	0	0	0	47	10	3	3	40	0	0	0	0	0	0	0	50	0	0	0
	3	40	0	0	0	0	0	0	0	40	0	0	0	37	0	0	0	53	3	0	0	40	0	0	0	0	0	0	0	50	0	0	0
	4	40	0	0	0	0	0	0	0	43	3	0	0	37	0	0	0	50	17	0	0	40	0	0	0	0	0	0	0	50	0	0	0
	5	47	0	0	0	0	0	0	0	0	0	0	0	37	0	0	0	43	3	0	0	40	0	0	0	0	0	0	0	50	0	0	0
	6	0	0	0	0	0	0	0	0	0	0	0	0	33	0	0	0	43	3	0	0	40	0	0	0	0	0	0	0	47	0	0	0
	7	0	0	0	0	0	0	0	0	0	0	0	0	33	0	0	0	43	3	0	0	40	0	0	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0	0	0	0	37	0	0	0	43	3	0	0	40	0	0	0	0	0	0	0	0	0	0	0

Table 2.1: Anomaly Detection Results reported in %. The performance of online anomaly detection for each monitored program using different lengths (column Len) and standard deviations for detection bounds (row Std Dev) for each type of anomolous dataset: mixed, random, and limited. The false positive percentage is also shown.

2.6 Discussion

Figures 2.6 and 2.7 show the complete detection and false positive performance for all the programs in the data set. It is shown that the HMM and LSTM detection method performs well when presented with constrained random data.

The HMM classifier performed well which allows for unidentified observation sequences to be classified to a particular program, then that program's anomaly detector mechanism can be utilized to ensure proper execution.

A further investigation on some of the failed detections reveal that the randomly selected program was the same program as the one currently running. In essence this results in the program basically restarting. This is still anomalous behavior that is not detected. In both cases LSTMs and HMMs fail to identify it as anomalous behavior. However, in other cases when a different program starts to run in place of the intended monitored program it has a high chance of being detected on average, 97% and 99% for HMMs and LSTMs respectively.

2.6.1 Comparison

We compare our approaches against each other and another technique. The other technique is a "golden signature" database approach that is similar to what has been proposed in the literature [31]. It has been modified to remove the explicit need to modify the program at the binary level to add in explicit checkpoints. Instead we use the sampling rate of 262144 retired instructions as an implicit checkpoint. The trade off is that this increases the size of the database needed to validate the program, but no knowledge is needed to be known about it beforehand to determine where the explicit checkpoints should be inserted.

2.6.1.1 Database Of Expected Values

Specifically, a database is built from the same training data set the HMM and LSTM used, a 75%/25% split for training and testing. The database stores a range of performance counters over time to verify that expected value matches the actual measured value. If the measured value does not fall within the previously recorded value that was stored in the database the program is flagged

Program	DB(MB)	HMM(Bytes)	LSTM(KB)
cjpeg	6.6	212	68
core	10.8		
linear	11.8		
loops	7.1		
nnet	9.6		
radix2	4.9		
sha	13.5		
zip	10.0		

Table 2.2: The amount of required storage for each of the techniques.

as modified.

The approach is straightforward and simple to implement and trades computation for storage. The amount of required storage depends on the complexity and length of the program. Simple calculations can be done to determine an estimate for required amount of storage for a database approach. For example: consider a program that samples four performance counters ever 262144 instructions. Sampling at this granularity requires at least 3 bytes to fully store the value. However, more storage is needed to express the range of acceptable expected values. The value to represent the range depends heavily on the noise that is inherent to performance counters, for this example we assume that it can be expressed in 1 byte. The result is that at each sample requires 16 bytes of space to store all four counter values and range. For example, an application in our benchmark has an average of 5 hundred thousand samples which would result in 8 megabytes of storage for this single application.

Table 2.2 shows the amount of storage required to implement the "golden signature" database, HMMs, and LSTMs, for each program in the benchmark. Since the same hyper parameters and topology were used across all the programs the storage required for the HMMs and LSTMs is the same across all programs.

2.6.2 HMM vs LSTM

On average the LSTMs outperform HMMs in terms of anomalous detection accuracy. Part of this is due to the Markovian assumption that HMMs have. The assumption means that HMMs do

not account any history in the calculations. The assumption directly contrasts with LSTMs which require to have a window of history, in this experiment 32 values, in order to make an accurate prediction for what the next value is.

Comparing the accuracy results of HMMs against LSTMs it is clear that LSTMs outperform HMMs. However, it comes at a cost of higher storage requirements as well as computational complexity. Table 2.2 shows that the LSTM neural network requires about 320 times more storage for each program. Additionally, LSTMs computational cost is higher when compared against HMMs. On average evaluating one window of history to make a prediction for one performance counter takes 4 milliseconds on a GPU and double that on a CPU. On the other hand, the HMM only takes on average 13 microseconds per sampled performance counter data point. Using these numbers it can be seen that, on average, the LSTM takes approximately 300 times longer to compute than an HMM.

2.6.3 Implementations

As previously stated the proposed detection scheme using HMMs and LSTMs allow for flexible implementations for anomaly detection. A software implementation in kernel space isolates the detection process from unprivileged programs. This has the advantage that the only assumption that needs to be made is that the kernel is not compromised. An alternative hardware implementation removes that last assumption. A hardware implementation would only need a non configurable set of performance counters which would stream data to a hardware based HMM or LSTM anomaly detector that can be trained in an offline manner. A hardware implementation would also allow for finer grained sampling, which in turn increases the difficulty for an adversary to evade detection [41].

2.7 Conclusion And Future Work

2.7.1 Conclusion

HMMs and LSTMs are shown to be an effective low overhead methods for analysis of hardware performance counter data in anomaly detection applications. For online anomaly detection they

achieve a correct detection rate of 95% and 98% on average, respectively. They also have the benefit that the approach allows for flexible implementations and low false positive rates, 0.38% for HMMs and 0% for LSTMs. The methods achieve these results by utilizing the inherent repeatable performance characteristics of programs to detect anomalies during execution. The techniques described here can complement any static time integrity methods, such as those provided by a Trusted Platform Module or similar. Additionally, the approaches are also easily layerable with existing mitigation techniques such as: stack canaries, address space layout randomization, W xor X memory permissions, and control flow integrity techniques.

2.7.2 Future Work

Immediate future work would involve collecting data from real world programs with security vulnerabilities and exploit them. Additional investigations could be done to see if there exists a better set of performance events, per program, that yield higher discriminatory information between programs while remaining deterministic and repeatable between runs. An investigation to see if the Viterbi path could also potentially be used in the anomaly detection process. Another aspect that could be investigated is the frequency domain of the performance counter data.

3. CONTINUOUS AUTHENTICATION OF EMBEDDED SOFTWARE[1]

This work explores the use of frequency information of hardware performance counter data to enable continuous authentication (CA) in embedded software. Current approaches for CA are not targeted for use with embedded software. In order to be more suitable for embedded scenarios the proposed method uses short time Fourier transforms on streams of hardware performance counters. The frequency information is used to build an encoding to a state transition. These transitions are used to build a corresponding state machine which recognizes and authenticates the protected program continuously at run time. The method achieves an average successful authentication true positive rate of 97% with a 1.5% false positive rate.

3.1 Introduction

More internet connected embedded systems are developed and deployed each day. Part of this networked enabled trend is due to the recent boom relating to the Internet of Things (IoT). Despite the boom, the security on these systems is lacking or disregarded as a design parameter [4, 6, 5]. The lack of consideration for security in these systems leads them to be compromised by a wide array of existing known, and unknown security vulnerabilities. The problem is further compounded due to the constraints on resources that embedded systems have. The resource constraints hamper the effectiveness or ease of implementation of the modern ubiquitous mitigation techniques to be deployed, such as: address space layout randomization, memory permissions, control flow integrity, etc. [46]. These mitigation techniques are not a perfect solution to solve all issues related to software security and they do not aim to be. However, they do increase the difficulty for an malicious actor to accomplish successful exploitation.

An unrelated and different approach for helping ensure correct execution is known as integrity checking. Integrity checking can happen statically, before runtime, or dynamically, at runtime. The

[1]Reprinted with permission from [45]. © 2019 IEEE. Reprinted, with permission, from Karl Ott and Rabi Mahapatra, Continuous Authentication of Embedded Software, 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications, August 2019.

dynamic version of integrity checking either: has an external measure to check against to see if it differs from what is expected, or leverages the software to perform introspection and determine if the integrity has been compromised. These techniques differ from continuous authentication techniques as they deal with ensuring the software has not been modified. One example of such an integrity check is to check an attribute about the program, such as a hash value on the basic blocks the program is composed of. That is, if the hash values of the basic blocks are different from what is expected then the program has been modified. This technique has some overlap and some contrast with continuous authentication (CA). In the context of CA the goal is to continuously monitor and pass an authentication check throughout runtime. For example, a new trend within the wearable IoT field is exploring using bio-metric signals, such as electrocardiograms (EKG), as a method to continuously authenticate the wearer of the sensors [47]. These signals are unique from person to person, are not easily leaked or recordable in a format that can be used to falsely authenticate as someone else, and are low cost to capture and process.

The notion of continuous authentication is closely related to dynamic integrity checking for software. Dynamic integrity checking for software are run time methods to validate that the software has not been modified. Typically the dynamic integrity checks do not operate in a black box fashion. The integrity checks are incorporated within the program's source code or they are added later through binary analysis and modification [48]. Since our proposed method operates without any prior knowledge of the program and instead analyzes intrinsic hardware performance counter (HPC) signals we categorize it as a method for continuous authentication.

However, none of the previously proposed CA, and only a few dynamic integrity check techniques, specifically target embedded software. Additionally, previous techniques do not have sufficiently low overheads, do not operate in a black box fashion, require detailed information about the software, or are inflexible in implementation (hardware or software level). The method we propose is well suited for embedded software, has a low runtime execution cost, operates as a bolt on monitor which treats the monitored software as a black box, and is flexible in implementation.

In this paper, we propose the use of HPCs as a basis to implement a method for continuous

authentication to be checked and enforced at runtime. Our proposed method uses a short time Fourier transform (STFT) that is applied on a window of the HPC signal. The STFT computes the frequency of the signals in the window while retaining temporal information. The frequency output window from the STFT is encoded to a space bounded state transition. The encoding results in a sequence of state transitions over time which we use to construct a spaced bounded state machine is then built which accepts, recognizes, and continuously authenticates the monitored program. This raises the bar for a malicious actor to compromise or take over vulnerable software as well as providing assurance that the program is operating as expected as it is continuously being authenticated. Additionally, we show the cost to implement such a method in hardware via high level synthesis.

On our datasets the proposed method achieved an true positive rate of 97% for successful authentication on EEBMC Coremark Pro benchmark suite. Additionally, the proposed method had a low false positive rate of only 1.5%.

The paper is structured with Section 3.2 covering background information and relevant related work. Section 3.3 covers the specific methodology used in our approach. Section 3.4 explains the specific details in our experimental setup. Section 3.5 evaluates and reports the results of the proposed method within the confines of our experimental setup. Section 3.6 discusses the results obtained from the experiment. Lastly, Section 3.7 gives the conclusion for the work.

3.2 Background

3.2.1 Hardware Performance Counters

HPCs were originally intended for performance tuning and provide an easy, accurate, and low overhead way to gather information such as caching performance, branch prediction performance, number of memory accesses, etc. On modern CPUs there are hundreds to thousands of different events that can be captured by HPCs. Figure 3.1 shows different HPC profile windows for one event across the benchmark program. However, the number of counters physically available on a CPU is less than 8 per core, for most CPUs. In the context of security or other non-performance

related analysis the configuration of these counter events is critical.

The data the HPCs provide is vital for the proposed technique for continuous authentication. The HPC data provides the base inherent signal which the short time Fourier transforms operates on and which is used to construct the state machines that provide the authentication.

3.2.2 Short Time Fourier Transforms

Short time Fourier transforms (STFTs) are utilized to calculate how the frequency of the HPCs signal change over time. Specifically, STFTs address the problem of maintaining temporal information (typically lost in standard Fourier transforms) by only taking the Fourier transform over a smaller interval of the signal. The result is that there is less resolution in the output of the frequency domain in order to preserve temporal information. Figure 3.2 shows the STFT output of the raw values that are shown in Figure 3.1. Temporal resolution is required for our proposed CA method to operate.

We utilize STFTs for extracting the frequency information from a window of the programs normal execution. The frequency information makes up the basis of the proposed continuous authentication method. That is, we are using the calculated frequency information to authenticate our program. Figure 3.1 shows an example of what the raw time series looks like for one of the programs in the benchmark set, while 3.2 shows the output of the Fourier transform on the same raw time series signal.

3.2.3 Continuous Authentication

Continuous authentication schemes have typically been used to analyze bio-metric data. These schemes are typically grouped into two distinct categories, either physiological or behavioral. The physiological bio-metric traits tend to be more static in nature. These traits typically include bio-metric data relating to fingerprints, facial features, or with the recent trend in wearable devices, EKG signals. Those grouped into the behavioral bio-metric category tend to be features of human behavior that are observed or measured and vary from person to person. Behavioral bio-metrics tend to deal with typing, talking, or other learned activities that have subtle differences between

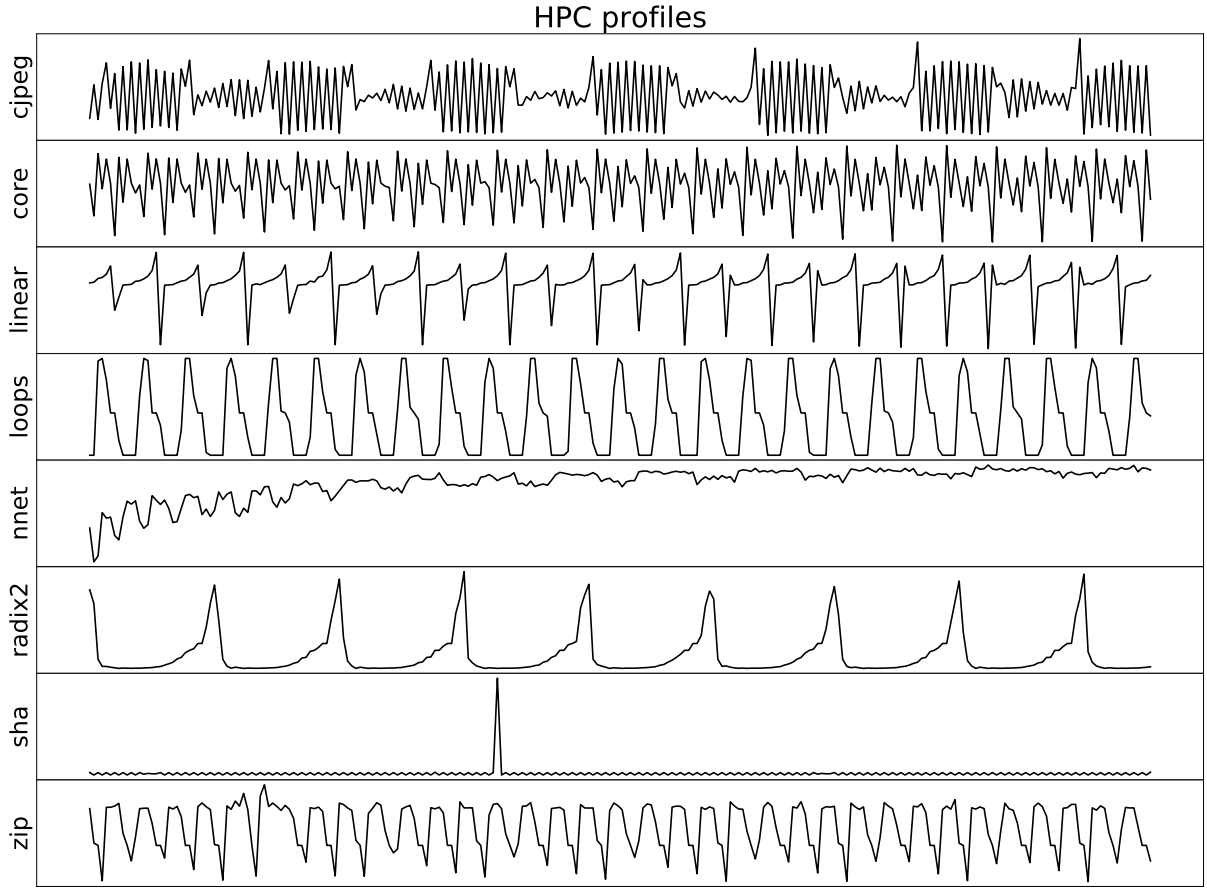

Figure 3.1: A small time slice of HPC time series collected in the data set. Each point along the X axis corresponds to 2^{18} retired instructions for one of the selected HPC values. The Y axis is the value that was recorded at sampling time.

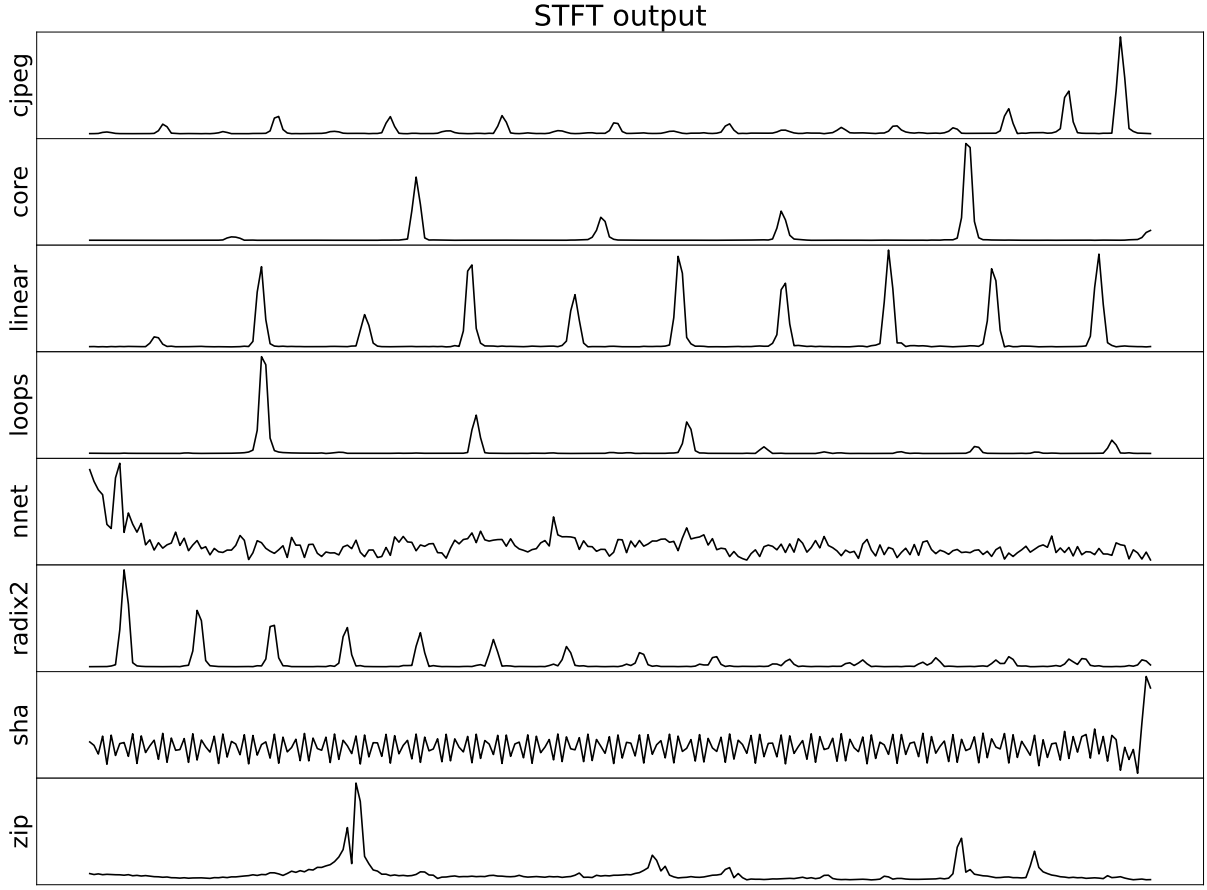

Figure 3.2: The STFT output from a 512 slice of data points similar to that which is shown in Figure 3.1. The spikes in the graph here correspond to frequencies that the raw time series contains.

data sets. Some of the early behavioral features utilized keystroke dynamics to continuously authenticate the current user keystrokes match the authenticated user. [49].

Similar to the bio-metric signals historically utilized for continuous authentication, we make use of a program's HPCs signals, which could be thought of as a "physiological" signal inherent to programs, to enable continuous authentication during runtime. Like most of the other continuous authentication methods proposed in the literature our proposed method is intended to act as a second level or factor of authentication. The proposed scheme is not intended to replace first factor authentication methods such as digital signatures that can be checked before runtime, but instead offers higher assurance that the program does not change at runtime.

3.2.4 Threat Model

The threat model for the proposed method varies depending on the level of implementation. If the implementation is a pure hardware scheme then only the silicon itself needs to be trusted. However, if implemented in software the underlying operating system, or any other components that run in the same privileged space, as well as the hardware must be trusted.

In this method we allow for any code injection or code reuse attacks that aim to ultimately take over the process. Note, this threat model does not include cases where an attacker changes a single or similarly small amount of instructions as this change is unlikely to be reflected in the corresponding frequency information that is calculated from the STFT. The proposed method is readily layerable with existing mitigations like address space randomization, control flow integrity, memory permissions, etc. However, many embedded systems are unable to support these techniques effectively due to tight resource constraints [46]. Our method aims to help raise the bar against successful exploitation especially in light of the lack of embedded system support for other mitigation techniques. The main rationale being that an adversary who wanted to run arbitrary code would need to forge their code to pass the CA checks.

3.2.5 Related Work

New mitigation techniques are developed and proposed routinely. The techniques typically focus on the mitigation of certain classes of threats. For example, memory permissions eliminate the impact of a simpler class of stack based buffer overflow vulnerabilities by specifying that the stack cannot be treated as executable code. Recently, hardware performance counters (HPCs) have been proposed for uses in software security. The uses of HPCs in a software security context include but are not limited to: malware detection [37, 39], rootkit detection [38], integrity checking [40, 31], anomaly detection [41, 25], and specific exploitation technique detection (such as return oriented programming attacks). In this paper we focus only on the recent works that investigate and propose the use of HPCs for to integrity checking. One of the earliest works to explore the use of re-purposing HPCs was done by a system called CFIMon [42]. CFIMon was specifically built to try and enforce a program's control flow integrity (CFI). It used static analysis on the programs binary to build a table of all known good branch and jump target addresses. The address table was compared against the addresses generated at runtime and stored in the last branch register. Wang et al. used hardware performance counters to detect a kernel modifying rootkit in virtualized systems [38]. Their proposed system was called NumChecker and operated by obtaining a database of HPC values for known good runs of system calls. Once the database was built the VM hypervisor measures the system calls the guest VM makes and compares the HPC values against those stored in the known good database.

Additionally, HPCs have been used for integrity checking by creating a golden signature of linear relations which the system firmware is periodically checked against to see if it has been modified [40]. A method called ConFirm [31] was developed for integrity checking in firmware for embedded systems. The ConFirm used performance counters to verify integrity of device firmware by modifying and manually inserting code detours to check against a golden signature database. Recently, Das et al. [50] have done a study of HPCs and their use in security applications. The outcome of the study is that HPCs must be used cautiously and carefully in security contexts otherwise they run the risk of being unreliable and untrustworthy. Using this as a guideline we

ensure that our approach falls within the recommendations.

of threats. For example, memory permissions eliminate the impact of a simpler class of stack based buffer overflow vulnerabilities by specifying that the stack cannot be treated as executable code. There has been previous work that investigated the use of HPCs for security applications, however in this paper only focus on works that are more related to integrity checking using HPCs.

techniques focus on mitigation certain classes of threats. For example, memory permissions significantly reduce the impact of a simpler class of stack based buffer overflow vulnerabilities by specifying that the stack cannot be treated as executable code. More recently, hardware performance counters (HPCs) have been proposed for uses in security. The uses of HPCs in a software security context have ranged from malware detection [37, 39], rootkit detection [38], integrity checking [40, 31], anomaly detection [41], and specific exploitation technique detection (such as return oriented programming attacks).

3.3 Methodology

There are 5 large challenges and design choices to be made for implementing a system which offers continuous authentication via HPCs. Things that must be considered are: the type of events chosen, how to deal with the small amount of noise present in the HPCs, how frequently the HPCs are sampled, methods for making a continuous authentication check, and how often to run an authentication check. As previously stated the proposed technique is flexible in implementation, therefore we consider two different implementations. The first is done within a software based approach, while the second is implemented as a hardware prototype.

3.3.1 Event Choice

The recent hardware security vulnerabilities Spectre [51] and Meltdown [52] show that shared hardware on CPUs, such as branch predictors, cache performance, TLBs, etc, is susceptible to external, potentially malicious, interference. The same is true for HPCs which track events of shared resources, and thus makes them unsuitable for security purposes in their current implementation.

To overcome the problem of event choice we turn to the literature [32, 33] to determine which

Event Name	Description
Retired Loads	The number of retired load instructions since the last sample.
Retired Stores	The number of retired store instructions since the last sample.
Retired Branches	Number of retired branches (conditional and unconditional).
Retired Conditional Branches	Number of conditional branches since the last sample.

Table 3.1: The chosen set of events to monitor based on a survey from the literature.

Sample #	Event	Run 1	Run 2	Run 3	Run 4
1	Branches	52775	52775	52777	52776
	CBranches	47467	47467	47469	47467
	Loads	43174	43172	43171	43173
	Stores	13158	13158	13158	13158
2	Branches	54452	54448	54438	54452
	CBranches	48942	48938	48928	48942
	Loads	48525	48524	48522	48525
	Stores	14258	14258	14258	14258

Table 3.2: The HPC values of the same program with the same input at the same sample points between repeated runs. Some of the events shown here appear to be deterministic while others show some slight noise.

event types are appropriate. As a result, we pick events that show good determinism, repeatability, and do not suffer from external interference or influence between different runs and different programs. We specifically use the events shown in Table 3.1.

3.3.2 HPCs Noise

The current implementation of HPCs, while fast and accurate, have some inherent noise due to modern CPU optimizations such as pipelining and out of order execution. The noise manifests itself in different ways: typically instruction skid and miscounts due to the previously mentioned CPU optimizations. Table 3.2 shows how the HPCs give different values between consecutive runs at the same sample points.

3.3.3 Sampling Frequency

To get an idea of how much overhead is incurred from software sampling the HPCs via the *perf_events* interface we measure the total execution time of the benchmark programs and take the average execution time while running without sampling, and then repeat the process with sampling. Figure 3.3 shows the overhead at different sampling rates. Ideally the sampling rate would be as high as possible if the overhead amount is still acceptable. However, the rate is a tunable parameter which allows the amount of overhead incurred to be adjusted.

Algorithm 1 Proposed continuous authentication method

$cont \leftarrow true$
repeat
  $data\_window \leftarrow HPC\_values$
  $data\_window \leftarrow data\_window * Hanning\_window$
  $fft\_output \leftarrow STFT(data\_window)$
  $state\_trans \leftarrow ToBits(fft\_output \geq threshold)$
  $state\_trans \leftarrow CRC(state\_trans)$
  if $state\_trans$ *in* $statemachine\_nextstates$ **then**
    $statemachine\_nextstates \leftarrow state\_trans$
  else
    $cont \leftarrow false$
  end if
until $cont = false$ *or* END_STATE

3.3.4 Proposed Continuous Authentication Method

The literature shows a number of methods that have been developed for continuous authentication. However, these are typically implemented with heavy weight classifiers, such as neural networks, which are not suitable for use on embedded system software due to computational constraints. Therefore, we propose a new method which has low resource and computational complexity requirements. Algorithm 1 shows a high level algorithmic implementation for the CA check. The method used for building the state machine before deployment is explained below.

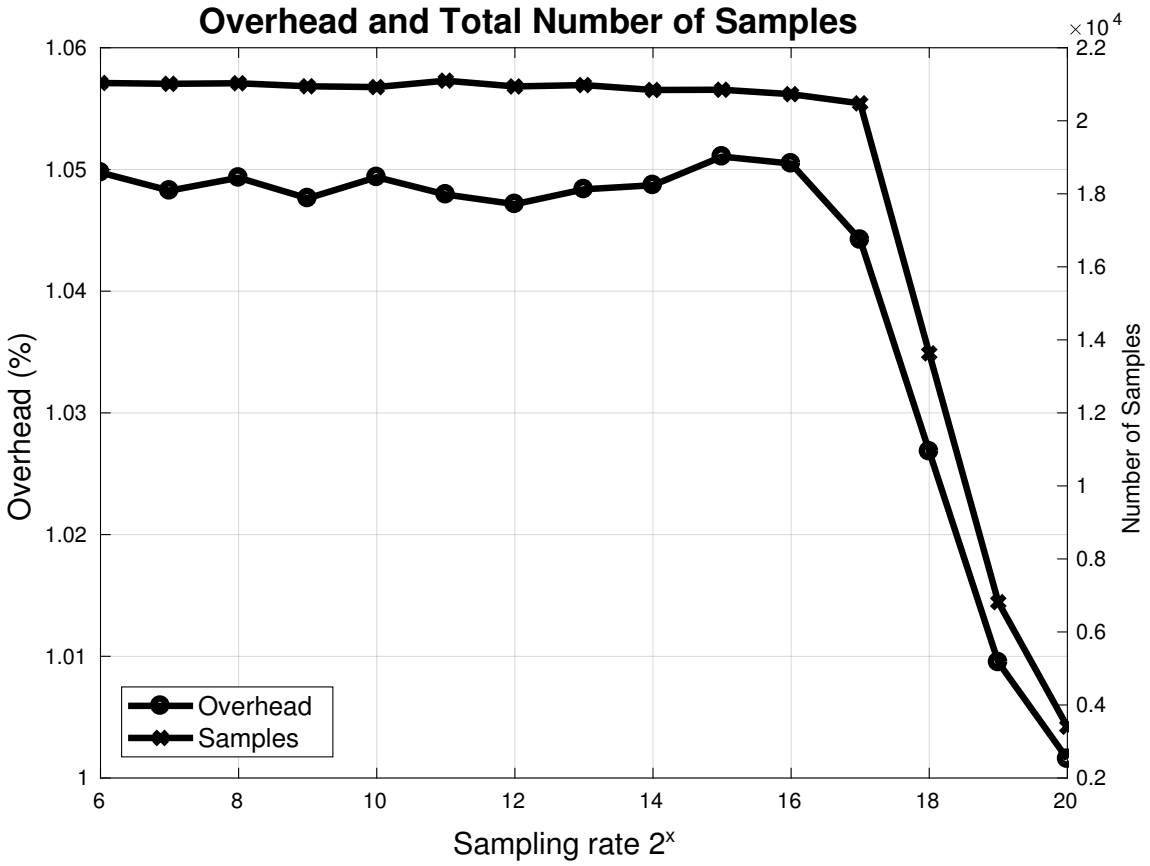

Figure 3.3: The overhead incurred and total number of samples captured at different sampling frequencies. From the line it can be seen there is a trend that the overhead increases as the sampling frequency increases until around 2^{17}. At this point the overhead starts to act erratic. To verify that the *perf_events* interface isn't working properly we also check the number of samples that are being recorded. It is clear that despite the increase in the sampling frequency the total number of samples does not increase.

In our specific experiment we use a window of 512 data points with a 50% overlap between. Each window is run through a Hanning window function to help reduce the amount of spectral leakage incurred from the STFT. The newly windowed values are then passed into the STFTs. The output of the transform is 257 data points of which the first one holds the average value of the signal processed by the STFT, which leaves 256 data points containing frequency information. From here a threshold value is determined by scaling the mean of the output frequencies of the transform across all windows for each program. The scaling factor is determined on a per program basis. The threshold is used to further reduce the 256 complex floating point frequency values to a single bit for each, which represents whether the corresponding frequency bin's value was above the previously computed threshold or not. The 256 bits are further reduced to 8 bits by way of the CRC-8 function.The 8 bit value is stored and represents that window's state transition value. The process is repeated for each subsequent window, which results in an 8 bit value that changes over time as new windows are processed. The last window is zero padded if it is not 512 sample points in length. A state machine is constructed that uses these 8 bit values as transitions paths. This makes authentication relatively simple, as invalid transitions that are calculated from this process are rejected. As long as the transitions through the state machine remain valid the software is considered authenticated.

The 256 value is hashed down to 8 bits in order to bound the size of the state machine. By hashing to an 8 bit value that puts an upper bound of 256 different state transitions. The rationale for using CRC-8 as a hash function is that it is computationally cheap but also maintains a decent level of transition fidelity. Additionally, due to the 8 bit size constraint of the state machine using anything more computationally expensive, such as SHA-256, is wasteful as the output will not have any of the strong cryptographic properties that the full 256 bits would.

However, due to the inherent noise in the HPC signals some valid program runs may fail to authenticate due to a non-matching 8 bit value from the CRC-8 function. To overcome this we introduce the notion of a ϵ transition, which is closely related to ϵ transitions from formal automata theory, specifically non deterministic finite state machines (NDFA). The specifics of

implementing ϵ transitions is to have a non-valid transition path move to a special state which contains all previously defined transition paths back to the valid state machine. With the notion of an ϵ transition we can allow for a defined amount of invalid transitions to occur before we consider the program as failing to authenticate. This approach has an inherent trade off that if too many ϵ transitions are allowed then the state machine may falsely authenticate an invalid program. Figure 3.4 shows a high level diagram of the data flow and processing required to implement the proposed method. Figure 3.5 shows a small portion of one of the state machines generated and used to authenticate. Note that each state has an incoming and outgoing connection to the ϵ transition state.

It should be noted that the parameters that are per program are: the HPC sampling rate, the HPC window size, the window function (e.g. Hanning), the threshold values, the hash function (e.g. CRC-8), and the number of ϵ transitions allowed. The output results in a corresponding state machine that is used later to authenticate the program. All other functions and parameters, such as the STFTs, are the same across the entire data set.

3.3.5 Authentication Intervals

Once the CA method is defined the last thing to consider is how often to authenticate using the proposed method.Specifically, the amount of overlap between windows can be tuned to determine how often authentication occurs. The rate at which authentication occurs can be increased if the amount of overlap is increased and can be decreased if the amount of overlap is decreased. The only limit on the authentication interval is that it must occur at a constant rate otherwise it will incur the cost of extra storage to track when authentication checks should occur.

3.4 Experimental Setup

We consider two different implementations. The first we evaluate the proposed method in a software context. This has all the associated sampling overheads as shown in Figure 3.3, but requires less integration and fewer changes. The second is a hardware prototype to estimate the associated cost in terms of space and speed. The level of integration varies depending on the

complexity of the embedded system.

3.4.1 Benchmark Software

Since the focus of the methodology is on embedded software we use a benchmark that is specifically made for embedded systems. Specifically we use the EEBMC Coremark Pro benchmark. The EEBMC Coremark Pro benchmark is an industry standard benchmark primarily for performance analysis of embedded systems.

The proposed method is constructed and evaluated on normal successful runs to test the success rate. Additionally, the proposed method is evaluated on abnormal runs, that should not successfully authenticate, to measure the rate at which the method erroneously authenticates.

3.4.2 Software Implementation

For the software implementation we use a low power Intel CPU. Specifically, we use a Intel i7 6600U, which is a dual core CPU with hyper threads and has 7 HPCs per core. Four of the counters are flexible and can be set to track many of the events. The remaining three counters are fixed function. The fixed function counters are set to count retired instructions, count unhalted clock cycles on the core, and reference count of unhalted clock cycles. We specifically use the fixed function counter for retired instructions to generate an interrupt on overflow after it has counted 262144, or 2^{18} retired instructions. The flexible counters are set to count the events listed in Section 3.3.1. Each time the retired instruction counter interrupts we take a sample of the current count of the flexible counters. The sample results in a 4 dimensional vector for each sample point.

The samples are gathered on Linux kernel version 4.10-32 using the *perf_events* interface. *Perf_events* provides an easy to use interface, at low overheads, to access HPCs with varying granularity. In this experiment we set the samples to be collected on a thread level basis. On a per thread level granularity the HPCs are saved and restored each time the thread is scheduled out of execution. The result is that other concurrently executing threads and processes do not effect or influence the data values sampled through *perf_events*. These sampled data points naturally create a time series which will be used as the basis which the authentication.

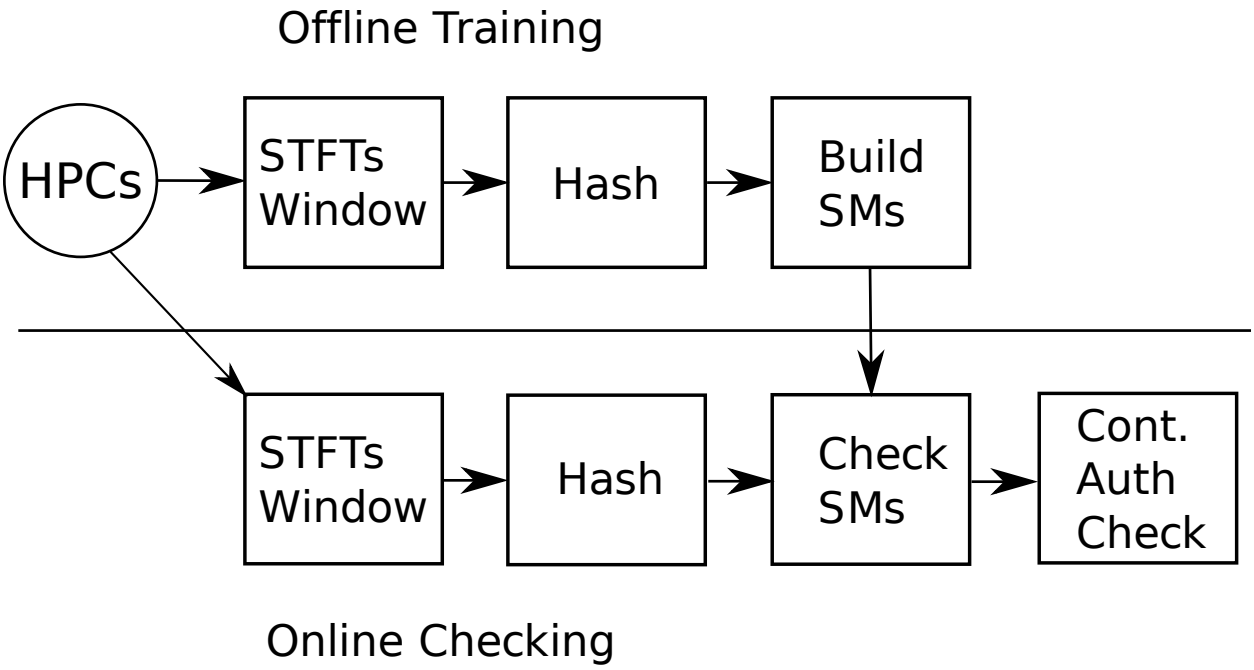

Figure 3.4: The data flow and computations required of the proposed continuous authentication method. The method is 2 phase in that they are constructed in an offline manner. They are then deployed and are used to continuously authenticate the software that was used in the offline phase. The HPC signals are streamed and windowed to be forwarded to the STFT. The window size is set at 512 data points which is then passed to the STFT to extract frequency information. With the newly extracted frequency information is quantized to a signal 256 bit value. The hash function, CRC-8 in this case, is used to further reduce the size of the 256 bit frequency information to a single 8 bit value. With these single 8 bit values a state machine is constructed and is used as the basis for continuously authenticating.

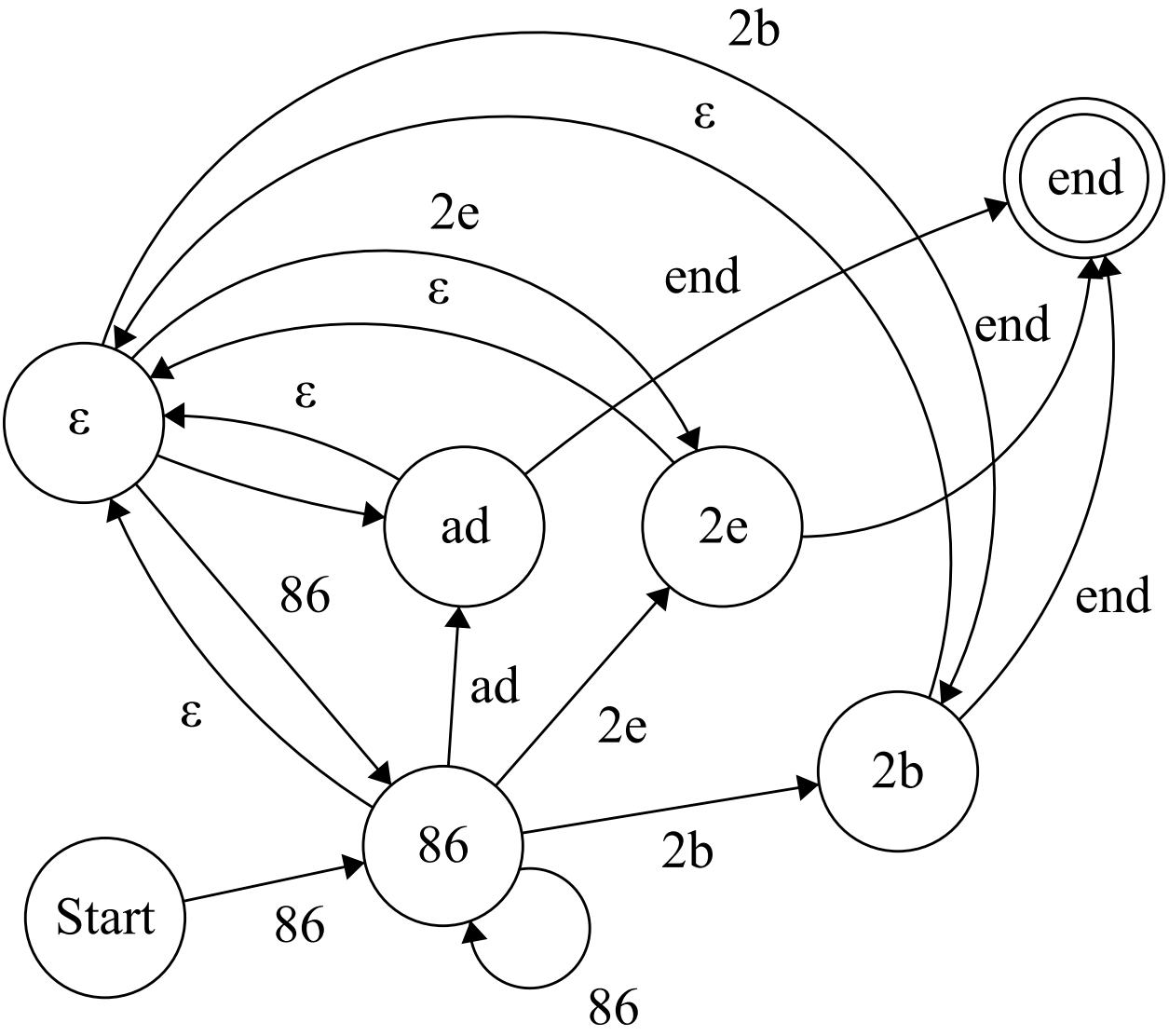

Figure 3.5: A partial example of a constructed state machine, with ϵ transitions, used to continuously authenticate the monitored software. The overall operation is similar to a traditional state machine, however the ϵ state records how many times it has been visited. If the defined threshold is passed on the number of transition to the ϵ state, then the software fails to authenticate. Otherwise, the software will continue to traverse the state machine until reaching the final accepting state.

We set aside 15% of each program's dataset to be used as a validation set for the proposed method and the remaining 85% is used as a construction set that is used to build the state machines for each program.

3.4.3 Hardware Implementation Of CA Module

The hardware implementation we a Xilinx Zynq 7000 SoC XC7Z020. The Zynq platform has a dual core ARM A9 CPU with a Xilinx FPGA attached via AXI interfaces. A prototype hardware block is developed with Vivado HLS to estimate the resource utilization required to help provide a hardware level component to support continuous authentication of software. For the most part the process for authentication is the same as the software version. The main differences are in the ways to sample the HPCs that are a result from tighter hardware level integration that is possible due to the tight coupling between the CPU and FPGA. Otherwise, the rest of the process of calculating frequencies of the windows via STFTs, quantization down to 256 bits, a type of hash function to further reduce to 8 bits (CRC-8 in this case), and traversing the constructed state machine remain the same.

The hardware design is configured to forward the HPCs overflow interrupt to the FPGA. Once the FPGA receives the interrupt it sends an AXI request to read the values of the HPCs, which it then stores in a buffer until it has a full window. Once a window buffer reaches capacity it is passed through the STFT block. The frequency output from the STFT block is passed through to compare against the previously computed threshold value to reduce the frequency output to a single 256 bit number. The CRC-8 operates on the 256 bit number to further reduce it down to a single 8 bit value. The 8 bit value represents the state transition that is to be taken on the previously computed state machines. If there is no valid transition available from the current state an interrupt is generated and sent to the CPU to halt execution.

The hardware block is designed in such a way that it has multiple buffers for receiving the HPC data. This is done so the CPU does not have to wait for the hardware block to finish the computation, as the hardware sampler can continue to store the samples in a different buffer. Additionally, the hardware design allows for context switches of programs. This is accomplished by saving and

Program #	True Pos (%)	False Pos	Avg # States
cjpeg	70	0.0	12
core	99	0.0	12
linear	84	2.3	10
loops	79	0.0	97
nnet	84	0.0	14
radix2	92	0.0	9
sha	83	0.0	11
zip	80	0.0	20

Table 3.3: The numbers reported are using the proposed method without the ϵ transitions.

restoring the current place in the state machine when a context switch is detected. Furthermore, the hardware design is constructed in such a way that the verifying state machine is able to be fetched from BRAM for the corresponding program that is scheduled in.

3.5 Results

The results presented for the software implementation show the successful rate of authentication for a normal executions of the protected software. The results for the hardware implementation focus on the implementation cost of space, power, and speed.

3.5.1 Software

Some experiments were carried out in order to evaluate the true positive rate and effectiveness of the proposed method for continuous authentication. Specifically, we randomly select 15% of the data set to be used as the testing set, which is used to evaluate the true positive rate of successful authentications. The processes is repeated 30 times. Each time a new 15% of the data set is held out and is used as the testing set. Additionally, another data set is used to measure how well the proposed method will reject HPC profiles that are from different programs and those that were collected from a normal execution of the monitored program that was modified at runtime to run other code. The average of the 30 repeated runs is reported.

Table 3.3 shows the rates at which the proposed method correctly authenticated each program in the testing set before allowing for ϵ transitions. With this fixed type of approach the average

Program #	True Pos (%)	False Pos	Avg # States	# ϵ
cjpeg	100	0.0	12	1
core	100	0.0	6	3
linear	92	0.0	20	5
loops	92	3.3	97	1
nnet	100	4.6	13	1
radix2	100	0.0	9	2
sha	100	0.0	11	1
zip	94	4.3	20	1

Table 3.4: The results for allowing the use of ϵ transitions while constructing the state machines.

true positive rate across all programs is only 83%. When we allow for more flexibility, for instance by allowing for ϵ transitions, higher rate of true positive rate can be achieved as shown in Table 3.4. By allowing for ϵ transitions the successful authentication true positive rate is improved to an average of 97% with only 1.5% false positives. We choose the number of allowable ϵ transitions by finding which values allow for the highest amount of true positives and the lowest amount of false positives.

With our proof of concept software implementation each authentication attempt takes on average about 85 microseconds, most of which is due to the STFT calculation. Using a window size of 512 values and 4 bytes per sample value, we estimate that 2 kilobytes of storage is required to buffer the required data frame for each HPC. The storage requirements for the state machine vary per program, however since there is an upper bound of 2^8 states the storage cost is low. However, a worst case size requirement where every state is connected to every other state, using one hot encoding, requires 32 bytes per state for a total size requirement of 8 kilobytes.

3.5.2 Hardware

We implement the proposed method into a hardware block using Vivado HLS. The synthesized result allows for a clock period of 2.9 nanoseconds for the particular hardware target. It also consumed 6 out of 280 BRAM blocks, or 1% utilization. Additionally, it utilized 24 out of the 220, or about 5% available DSP blocks. Lastly, it needed 19862 out of 106400, or about 9%, and

16346 of 53200, about 15%, for FF and LUT respectively. Lastly, the resulting hardware module needs 8 clock cycles to run the STFT on the HPC window, convert the result to a state transition, and verify the transition in the state machine. The proof of concept hardware block requires 24 nanoseconds per HPC window to decide to authenticate or reject the software.

3.6 Discussion

3.6.1 Software

The initial performance without allowing for ϵ transitions is not very good. The reason behind this is due to the inherent noise in the HPCs. By allowing for at least one ϵ transition the true positive dramatically rises. Further tuning each individual program's parameters a rate of correct authentications of 97% across all benchmark programs can be achieved.

It should be noted that we believe if the HPCs can provide deterministic measurements then the use of ϵ transitions would not be needed. Furthermore, the proposed method is flexible in nature, meaning that if the use of the CRC-8 function was not an ideal fit any other function which deterministically maps data from a larger space to a smaller space could be used in its place.

3.6.2 Hardware

For the hardware implementation to be efficiently utilized and integrated there would need to be some interaction from the base software. This interaction can take place in the kernel, which therefore does not require any changes to userspace programs. There is the potential for less interaction needed with the software if certain modifications are made to exposed the needed information at the hardware level, such as which process is running. Ideally, the proposed technique could operate independently from the CPU and simply observe the behavior of the software via HPCs.

3.7 Conclusion

We have devised a new novel technique that allows for continuous authentication of embedded software at runtime, in both software and hardware. The proposed technique utilizes the intrinsic properties of the monitored software as reported from the HPCs. When monitored over time the HPCs signal can be constructed as a time series. From this time series we extract the frequency

information by using Fourier transforms on windows of the time series. The frequency information is encoded to a single 8 bit value. Once encoded, the 8 bit value is used as a state transition value. Using the series of state transition values a state machine is constructed. The embedded software is authenticated by checking if valid transitions are made in the state machine at runtime. If invalid transitions are made the continuous authentication has failed, and the program is rejected. Additionally, no source or binary level changes are needed to apply this technique to the monitored software since it operates on the HPC data streams.

When evaluated, the proposed technique achieved an true positive rate, correctly authenticating the correct software, of 97% on our benchmarks and a 1.5% false positive rate, incorrectly authenticating invalid software.

4. HARDWARE PERFORMANCE COUNTER ENHANCED WATCHDOG FOR EMBEDDED SOFTWARE SECURITY[1]

This work proposes a novel use of long-short term memory autoencoders coupled with a hardware watchdog timer to the enhance robustness and security of embedded software. With more and more embedded systems being rapidly deployed due to the Internet of Things boom security for embedded systems is becoming a crucial factor. The proposed technique in this paper aims to create a mechanism that can be trained in an unsupervised fashion and detect anomalous execution of embedded software. This is done through the use of long-short term memory autoencoders and a hardware watchdog timer. The proposed technique is evaluated in two scenarios: the first is for detecting generic arbitrary code execution. It can accomplish this with an average accuracy of 91%. The second scenario detecting when there is a malfunction and the program starts executing instructions randomly. It can detect this with an average of accuracy of 88%.

4.1 Introduction

The continuing boom of Internet of Things (IoT) devices has cemented the popularity and longevity of embedded systems. With more and more internet enabled embedded systems deployed each day a possible danger grows as shown with the relatively recent Miria botnet attacks [54]. However, the Miria botnet was largely due to misconfiguration of the devices instead of a more complicated attacks. Regardless, the botnet showcases a danger that the widespread deployment of IoT devices can unleash. IoT devices are being deployed in more security and safety critical applications, yet the security of the devices tends to be disregarded as an important design parameter [55, 26, 4, 6, 5].

The past two decades has been a vibrant time for security research and application in more traditional computing environments such as desktops and mobile devices. However, the best stan-

[1]Republished with permission from [53]. © 2023. Reprinted, with permission, from Karl Ott and Rabi Mahapatra, Hardware Performance Counter Enhanced Watchdog for Embedded Software Security, 24th International Symposium on Quality Electronic Design (ISQED), April 2023.

dards and practices are not always suitable for embedded devices as even mobile devices typically have orders of magnitude more computing capability when compared to an embedded system. Historically, embedded systems were not connected to the larger internet so the necessity for security primitives was much lower. As these systems were not connected to the internet they could not be attacked broadly and at low cost like traditional computing devices were. This picture is rapidly changing as the surge of IoT devices hits the market.

Other outcomes from the IoT boom have resulted in a renewed research interest and solutions aimed to address the security challenges posed by these resource limited internet enabled embedded systems. Many of the proposed solutions require large changes to the entire development stack such as using formal methods [56], different programming ecosystems or languages [57], or additional security hardware [58]. These approaches have differing impacts on the design and deployment of embedded systems. For example, using formal verification could drastically increase the security of embedded systems, however it may also drastically increase the development time and thus the time to market of the system. Additionally, our proposed system can be used in conjunction with formal methods to provide more assurance at runtime that the embedded software is running as expected. Additional security hardware would increase the cost as more components or IP cores are being brought into the design.

A more recent trend has been utilizing previously unused existing resources to provide additional security primitives. The most notable instance of this is re-purposing hardware performance counters (HPCs) for security purposes. The benefit is that these techniques can be easily deployed if the system already has support for HPCs. Additionally, many of these systems already contain a watchdog timer which may not be properly utilized which adds more risk to the systems [59]. A watchdog timer provides a cheap method for timeouts. If the watchdog is not reset before the timer ends then corrective action is taken, which typically involves resetting the software to a known good state. The result is a watchdog timer can provide marginal security benefits. However, a watchdog timer alone is not a security primitive as a sufficiently advanced attack could issue the commands to reset the timer and thus avoid the restorative action.

The technique proposed in this paper is to augment a traditional watchdog timer based approach with a LSTM autoencoder based on HPC information to make it more suitable as a security primitive. The main goal is to provide additional barriers that an adversary or software malfunction would have to overcome to allow for unintended execution of the software in an embedded system. Additionally, the technique is evaluated on a standard set of compiled embedded system benchmark programs. With this done the proposed method does not require any modification of the source code for the program, and the program can be treated in a black box fashion.

The proposed technique utilizes fine grained HPC data along with a watchdog timer. The fine grained nature of the HPC data provides information about which section of code was executed. The data collected from the HPC information is used to build a model based on LSTM autoencoders that is checked during the runtime of the program to ensure that it matches what is expected. The watchdog timer is deployed to provide a timeout feature. When the watchdog timer is reset, the current data in the HPCs buffer is sent to the LSTM model to be verified. If the HPC data does not match as to what is expected then it can be assumed the program is not executing correctly. Additionally, if the watchdog timer fails to get reset, and times out, then it can be assumed that the software is not executing as expected. This fusion of HPC data and a watchdog timer aims to provide a novel and lightweight mechanism to ensure the correct execution of embedded software against adversaries.

The paper is organized as follows. Section 4.2 explains the relevant required background information. Section 4.3 introduces our proposed technique and details how the experiment to evaluate the technique was setup. The results are presented in section 4.4. The results are presented in section 4.4. A discussion relating to the results and experiment is in section 4.5. Lastly, the conclusion is in section 4.6.

4.2 Background

4.2.1 Hardware Performance Counters

HPCs are implemented as a set of configurable counters. As the name states they are implemented in hardware. The number and type of events that can be tracked vary depending on the specific processor. Since these are implemented in hardware they offer a low overhead way to get performance insights into the software running on top. The events that can be monitored can vary from memory accesses, such as loads and stores, to caching performance, like number of level 1 data cache hits, to branch prediction performance, for example the number of correct predictions, and so and so on.

However, current implementations and interfaces to HPCs are easy to use incorrectly which will result in subtly incorrect results. This is doubly important when trying to use HPCs in the context of trying to provide additional security. A recent paper from Das et. al. [50] highlights the challenges of trying to work with HPCs in a security context. The approach used in this work is not susceptible to the issues described in [50].

4.2.2 Watchdog Timer

Watchdog timers are commonly used in embedded systems. They can offer a cheap and simple way to enhance reliability. The simplest implementation is a timer that must be periodically reset by the software before it times out, otherwise a corrective action will be taken. Corrective actions aim to put the software back into a known safe state. Commonly in embedded systems this might simply be to reset the software in order to return to a known safe state.

The benefit watchdog timers offer is a simple way to robustly protect against accidental or random modification of software. However, watchdog timers do not offer much protection against active malicious modification of software. If an adversary has knowledge that a watchdog timer is in place then it is not too difficult to incorporate the required instruction or code to reset the timer before a time out occurs. Additionally, using watchdog timers correctly and effectively can be a difficult task. One of the difficulties is partition code tasks into proper blocks that fall within the

defined timeout, or timeout window. The technique proposed in this work also aims to simplify the usage of watchdog timers.

4.2.3 Threat Model

Trying to evaluate the effectiveness of proposed security solutions ranges from difficult to impossible unless the scope is precisely defined. A threat model is what provides that definition of scope. The threat model we consider in this work aims to be as broad as possible. The threat model will also vary on the implementation level for our proposed method. If it is implemented in software the threat model will be different if the method is implemented in hardware. Specifically, the software implementation will only require that any code running at a higher privilege level can be trusted and will not be modified. A hardware implementation would not have this restriction. In a practical sense this means that the technique described here will need the kernel to be trusted.

The threat model also requires that the HPCs are not writable or modifiable. In an ideal case the HPCs would be completely deterministic. Which means that given the same code and the same input for the code the same HPC values are collected on repeated independent runs of the software. However, in the current real implementations of HPCs they are not fully deterministic. The sources of nondeterminism have been investigated and discussed previously in the literature [50, 32, 33]. Our proposed method tries to deal with the noisy and nondeterministic nature of the current implementations of HPCs.

The defined threat model allows for a wide range of attacks that we aim to protect against. Specifically, we aim to protect against attacks that try to take over control of an already running process.

4.2.4 Related Work

In the past decade along with the IoT boom there has been a boom in research trying to utilize HPCs for software security purposes.

Many of the previous works relate to using HPCs for malware detection. One of the earliest works by Demme et al. [37] investigates using HPCs for malware detection in Android systems,

and rootkit detection in Linux systems. Demme et al. used a classification based approached, that is known malware was profiled to collect HPC traces and various classifiers were trained on the traces to detect malware. In [41] Tang et al. use an unsupervised one-class SVM for malware detection. [38] user HPCs for rootkit detection by comparing HPC traces of Linux kernel system calls against a "golden signature".

Other approaches try to use HPCs as a way to implement existing mitigation techniques with a hardware backed implementation such as control flow integrity (CFI) [60]. CFIMon uses HPCs as part of a system to enable CFI. It makes uses of a table of valid address created from static analysis. It then uses HPCs at runtime to check branch targets against the table of known good jump targets [42].

HPCs use in anomaly detection and integrity checking for detecting changes in software has also been proposed in [40, 31, 25]. Malone et al. [40] use a system of linear relations based on HPCs to try and verify the integrity of software at runtime. A "golden signature" database is used by Wang et al. in [31] as a way to verify program integrity. Alternative approaches for ensuring integrity has been done by way of continuous authentication [45].

More recently, there has been some newer works which highlight some of the dangers and pitfalls in current implementations of HPCs. A study by Das et al. [50] investigate many recent papers that propose the use HPCs in a security context. The paper also summarizes some of the pitfalls of HPCs, such as non determinism [32, 33]. Using these recommendations, as well as our own investigation we ensure our use of HPCs picks counter events that are deterministic or have a high level of determinism.

Our proposed work is most similar to [31] in the end goal. We aim to create a system that can be used to ensure software deployed in embedded systems is operating normally and as expected. However, we address aim to address some of the shortcomings that are present in previous works.

4.3 Experimental Setup

To validate the proposed method we build a software prototype simulator on top of Linux. The simulator runs the natively compiled software on the CPU and makes use of the provided hardware

features, specifically the CPU's hardware performance counters.

4.3.1 HPC Event Choice

Previous works in this space [25] demonstrate how HPCs of shared events such as caching performance can be influenced from any programming running on the system. Other recent hardware vulnerabilities such as Spectre [51] and Meltdown [52] also show that shared resources can be influenced by other software running on the system. This makes any HPC event that can be externally influenced a poor choice for security applications.

Modern CPUs have hundreds of events that can be tracked. Additionally, they also allow for several events to be tracked simultaneously. As a result there are millions of possible combinations of events that can be tracked. This makes a brute force enumeration of all possible combinations intractable. Instead, we use the knowledge that HPC events of shared resources are not a good choice. Additionally, we use the literature [32, 50, 33, 25, 45] to inform our HPC selection. We pick events a set of events that demonstrate a high level of determinism and give repeatable numbers between different runs of a program. The chosen events, to our knowledge, can not be influence or interfered with externally.

In this work we have chosen the five following events. The first event is retired instructions. This event provides a count of how many instructions have been retired after completing execution. The second and third events are retired loads and stores, respectively. These HPC events provide information for how many memory operations in the form of loads and stores have been executed. The fourth and fifth events relate to control flow. Specifically, these events are retired branches and retired conditional branches. In this case, retired conditional branches is a subset of retired branches. These HPC events correspond to control flow events in code, e.g. if statements, looping constructs, and function calls. More succinctly we make use of the following events: retired instructions, retired loads, retired stores, retired branches, retired conditional branches.

4.3.2 Static Binary Rewriting

We deploy DynInst [61] in order to get static binary rewriting functionality. This allows us to change an already compiled program to support the proposed HPC and watchdog timer method. A small library is written which serves to hold the code that will be written into the compiled program in order to facilitate extracting HPC information in specific locations of the program. The library makes use of the native Linux system call perf_event_open to setup and read the performance counters.

Using perf_event_open is advantageous as it provides a nice interface, that if configured correctly will handle many of the difficulties that can arise from trying to collect HPC information, easily. Using this interfaces HPCs can be configured to only run and monitor when the specific task to be monitored are scheduled to run, in Linux this means either a process or a thread. That means when the monitored task is scheduled out of execution the HPC values will not be perturbed by non monitored tasks that are scheduled to run before or after the monitored tasks. This gives isolation to the HPC values that are collected.

4.3.3 Fine Grained HPCs

Previously mentioned was the notion fine grained HPC information that is extracted from a target program. Under Linux HPCs can operate in 3 different modes. The first mode is a simple counting mode. In the counting mode the counters are enabled before the start of the program. They are then disabled and read at the end of the program. This operation has very low runtime overheads but offers virtually no granularity of where the HPC events occurred during the program's execution. The second way HPCs can operate in is a sampling mode. In the sampling mode the counters are configured to either be sampled at specific time intervals (typically some number of times per second) or to sample when an overflow interrupt on an event is triggered (typically after some configured number of instructions have been retired). This mode has a higher runtime overhead as there is more machinery in place as the overflow interrupts and reading of HPCs must be handled. Additionally, it is possible to have the Linux kernel also extract more contextual in-

formation (register states, etc). This contextual information allows for a the ability to determine where the extracted HPCs events occurred during the execution of the program. The last mode manual use of the HPCs.

This is done by modifying the source code of the program with explicit calls to setup, start, stop, and read the counters at specific points in the program. This mode allows for the highest granularity of HPC information, as it has been explicitly annotated in the source code. However, it has the obvious drawback that the source code must be modified in order to support this mode. If the source code is not easily available or modifiable then enabling this finer granularity of HPC collect will be difficult.

To overcome the downsides of the explicit HPC mode we develop a tool, referred to as *fgperf* in this text, which can statically rewrite compiled programs to automatically insert the reading of HPCs at specific locations. By doing this we can have fine grained HPC information without any source code modifications at low runtime overhead costs since the modification is done statically to the program stored on disk before execution. Figure 4.1 shows the shape of a time series trace for HPC event for one of the programs in the benchmark corpus.

4.3.4 Inserting Instrumentation Points

Now with *fgperf* in place a method is required to automatically determine where in the flow of the program the instrumentation points should be inserted. Previous work in the literature [31] did not address this point, which would mean that the location for the instrumentation points must be manually determined. This requires a working knowledge of the layout of the program, which might be difficult to come by if the source code is not available.

In this work we automatically insert instrumentation points at each prologue and epilogues in each function. Figure 4.2 shows how the instrumentation points are inserted into the program. However, a small amount of post processing must be done in order to properly attribute the HPC values to the correct calling function. This is done to avoid incorrectly accounting the HPC to the wrong function. With this scheme in place the accounting can be done with a simple state machine for determining the action and a stack for storage. The state machine allows for HPC values to be

processed in a streaming manner, while only requiring a limited amount of storage and processing capability.

The specific purpose of the state machine is to properly account HPC sample values to the correct function. As previously stated, when a function is entered the process is to take a sample of the HPC values and store them on a stack. When a function is exiting the HPCs are sampled again and the new current value is subtracted from the all the values on the stack. In effect this gives the functionality we need to account for individual functions HPC output, but not any of the subfunctions that they may call. With this approach the outcome is that every function can be considered in total isolation, regardless if it is a standalone function call, or part of a long call chain of functions.

4.3.5 Evaluating HPCs And Watchdog Timer Parameters

With *fgperf* and a method to automatically determine HPC collection placement points in place the last needed pieces are component to act as a watchdog timer (WDT) and a component to evaluate if the HPC values that are collected are within acceptable ranges. In this work we refer to this module as the checker. The checker does as the name implies, it takes the signals from the WDT and the LSTM autoencoder and checks to make sure they are within a tolerable bounds. Figure 4.3 shows the high level overview of the proposed system.

We deploy a fairly standard configuration for a WDT. That is, the watchdog must be "pet" at a specified frequency or it will raise the alarm. Specifically, the WDT has been set to expect to get a HPC reading within a defined set of time ranges. If the WDT does not get "pet" within the boundary of the set of the previously determined time ranges then the program is considered to be operating out of the normally specified behavior, and is therefore considered anomalous.

To evaluate to see if the HPC values are within in acceptable range we make use of and long-short-term memory (LSTM) autoencoder. Autoencoders are a type of neural network. Auto encoders are different from other forms of neural networks as they are generally not used to predict or classify, but instead their goal is to reconstruct the input with minimal loss. They are typically used in applications where the inputs have some noisy characteristics, and ideally, the output will

have most of the noise removed leaving only the signal or in applications to reduce the dimensionality of the data. Autoencoders operate in an unsupervised manner. In our proposed system the LSTM autoencoder acts similar to an intrusion detection system. Our system will calculate the reconstruction error of the LSTM autoencoder and use that as a signal to try and determine If the software is behaving as expected. Figure 4.4 hows the high level architecture for the proposed LSTM autoencoder topology.

4.3.6 Testing Corpus

We use a common benchmark for embedded systems, EEBMC [43], in order to evaluate the characteristics of the proposed system. The benchmark programs are adapted in order to test the effectiveness of the proposed system.

The EEBMC benchmark programs are adapted to allow for the current address space and thread of execution to be taken over. The transfer of the thread of execution to a different stream of instructions would ideally be detected under the proposed enhanced watchdog system. This test corpus represents an arbitrary code execution attack. The mechanism for how it is accomplished (buffer overflow, return oriented programming, etc.) is not the focus of this work and can be viewed as an implementation detail for the ultimate goal of arbitrary code execution.

The other modification to the testing benchmark corpus is to determine if the proposed system can detect random malfunctions. This differs from the previous modification in that the thread of execution will run through an unmodified address space and start executing a stream of pseudo random instructions. The instructions executed are setup so they will not cause a segmentation fault, or any other issue that could cause premature or abnormal termination of the programs.

4.4 Experimental Results

4.4.1 Specific Parameters

In section 4.3 we lay out the broad setup for the experiment. However, there are many specific parameters that can be adjusted in the proposed method. For this specific experiment we use an LSTM autoencoder. The topology of the autoencoder varies depending on the program. This is

done as an optimization technique. If the autoencoder model performs the same with fewer nodes then that topology is selected. The timeslice that the autoencoders use is also variable depending on the program. The variable timeslice is deployed as programs have different temporal behavior from one another, and thus the autoencoder model's performance will be impacted depending on the size of the timeslice. There is no one size fits all model.

We did not do an exhaustive search of the hyperparameters for the autoencoder, but instead focused on trying to keep the size of the network low. A basic grid search was deployed to find optimal hyperparameters for autoencoder topology and timeslice length. Additionally, the other hyperparameters were the number of epochs the model was allowed to train. If this is to be deployed in an embedded system scenario then the amount of computation required should be minimized in order to help keep the space and power requirements down as well. The parameters that influence the amount of computation and power in this approach are the model topology and time slice length. Table 4.1 shows the chosen and selected various hyperparameters for each of the programs in the benchmark.

A 30/70 split of the data set is used. That is 30% of the data is used for training and the remaining 70% of the data is used for testing. The rationale for this split is that the data set collected with *fgperf* is large. Additionally, since this is an anomaly detection approach none of the non normal data is used in the training. The testing set utilized a 50/50 split of good executions and executions that were faulty.

Another parameter in this specific experiment is to flush the buffer of the autoencoder after it evaluates that input. A different configuration would allow for some amount of overlap, and therefore keep some of the previous data points for the next evaluation. Such and approach would cause more evaluations of the autoencoder, and thus there would be more computations happening thereby increasing the amount of power that the system would use. However, this has a potential to decrease the latency for the system to detect anomalous behavior. This, is an inherent parameter in the proposed system.

We scale the HPC input values via a power transform so as to have an acceptable input range of

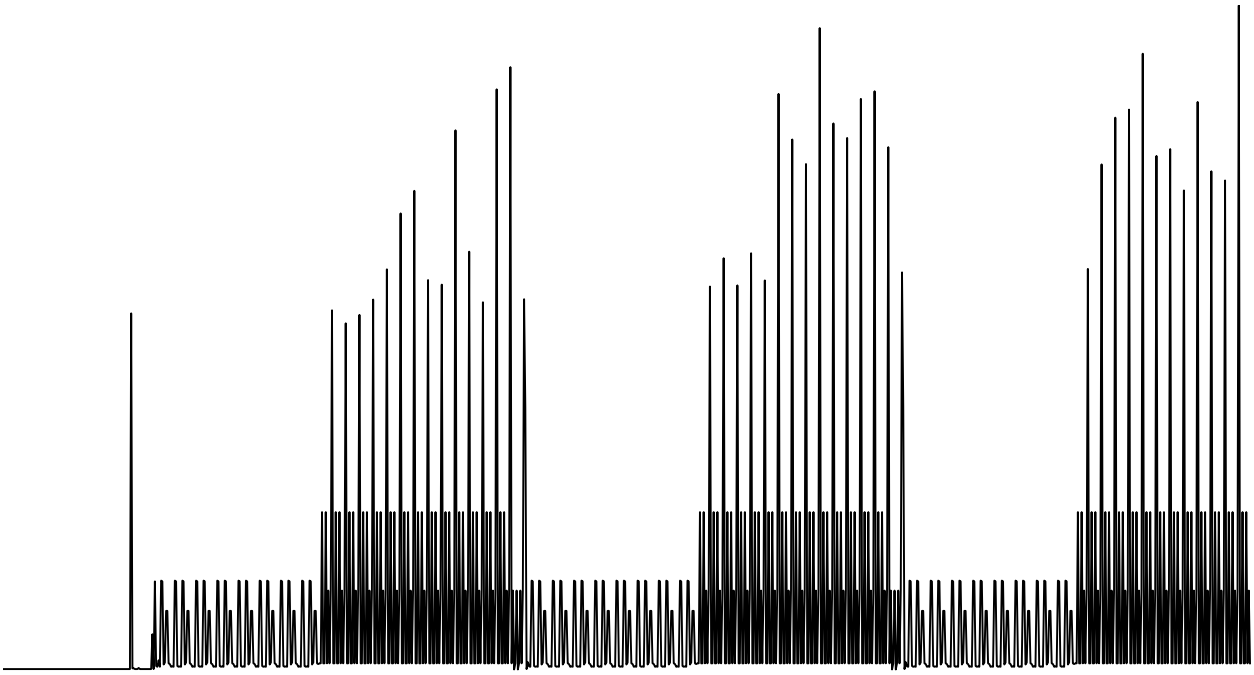

Figure 4.1: This shows the shape of the time series from a data sample collected by *fgperf*. The units for the axis are not important for showing the shape of the time series for this particular sample.

Program name	Topology	Timeslice Length
cjpeg	32-16-16-32	32
linear	32-16-16-32	32
loops	32-16-16-32	32
nnet	32-16-16-32	4
parser	32-16-16-32	32
radix	32-16-16-32	4
sha	32-8-8-32	16
zip	32-16-16-32	4

Table 4.1: The programs used in the benchmark with their corresponding hyperparameter values of the topology configuration and timeslice length.

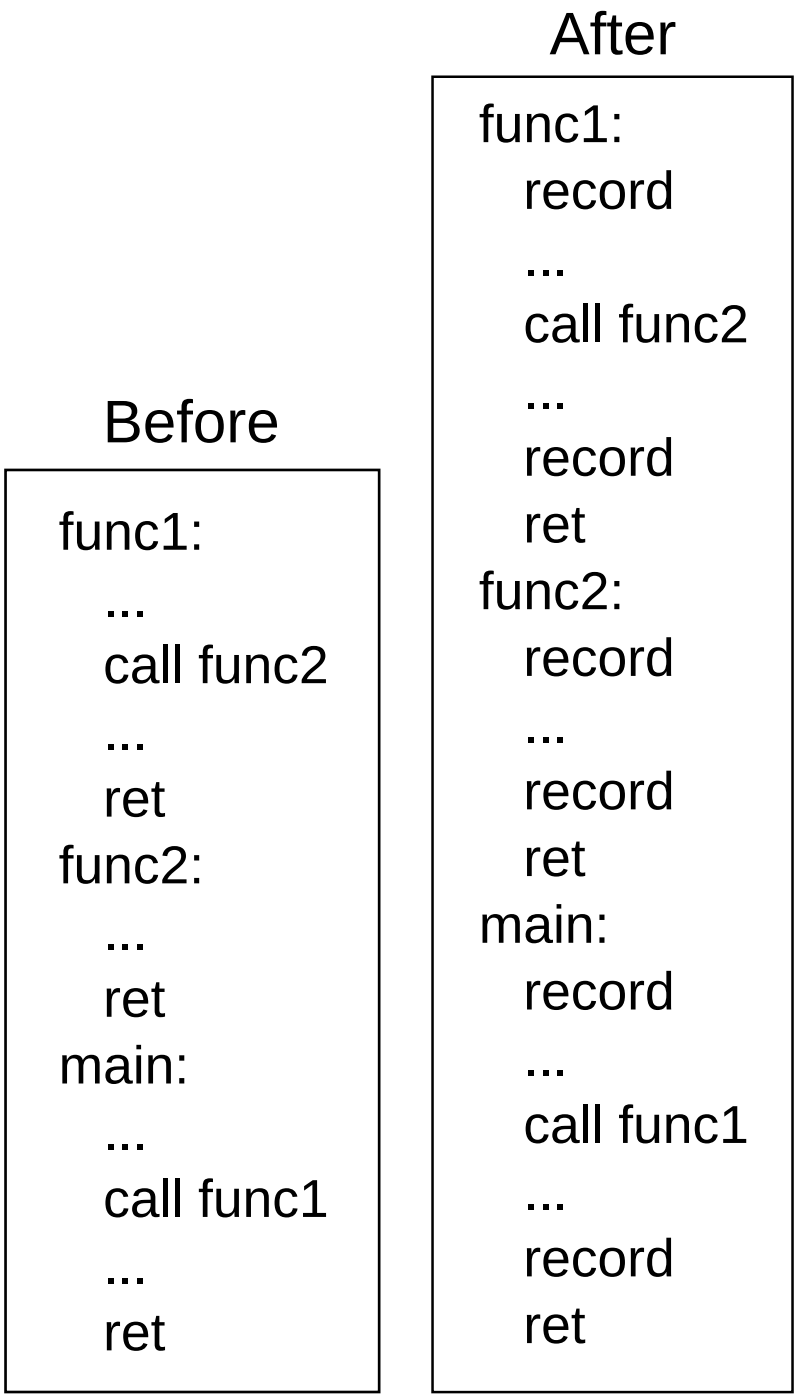

Figure 4.2: This is a simplified and abstract diagram that shows what the transformation *fgperf* does to a compiled program. The ... indicate that there are arbitrary instructions there. The *call* and *ret* instructions there to illustrate some arbitrary function level control flow. The "record" instruction is an abstract instruction that means to take a sample of the HPCs when it is executed. Additionally, the "record" instruction also has a timing component that is used with the watchdog timer. This modification is only necessary for the software simulated version.

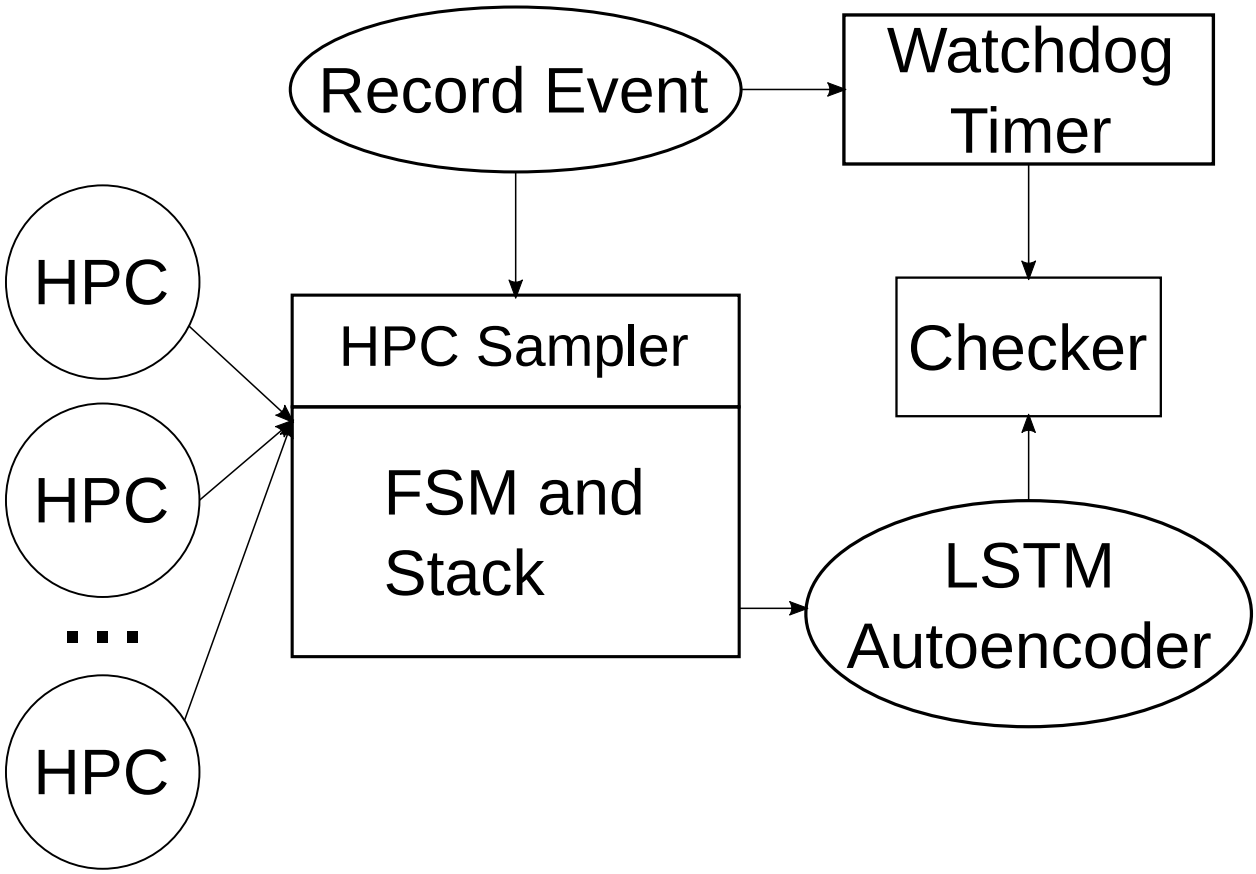

Figure 4.3: A simplified overview of the proposed architecture's data path is shown here. In this case the HPCs are already configured and are counting. When a "record" event is hit the sampler will take a snapshot of the current value of the HPCs and put them onto the stack. The sampler will also inform the FSM of any accounting computation that should be done. When an accounted for data point leaves the stack it enters into the LSTM autoencoder. The autoencoder processes a window of data points and computes the mean absolute error (MAE) of the reconstructed output. This error value along with a signal from the WDT is sent to a module called the checker. The checker ultimately decides if the software is operating as expected.

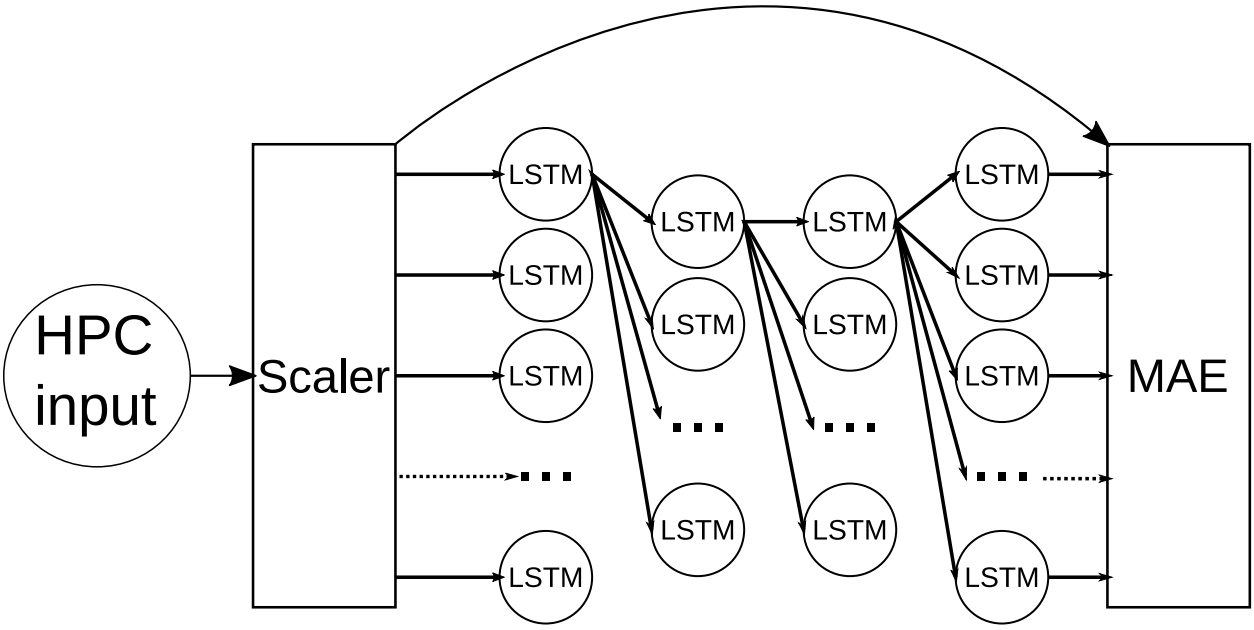

Figure 4.4: The high level architecture of the LSTM autoencoder. The first 2 layers of nodes represent the encoder part of the autoencoder, while the last 2 layers are the decoder portion. The first and last layer are the input and output layers respectively. The middle 2 layers are the encoder and decoder layers respectively. The scaler block represents a power scaler. The MAE block takes the output of the autoencoder and output of the scaler to calculate the MAE between the two. This calculated MAE value represents the reconstruction error introduced by the autoencoder. The MAE value is forwarded to the checker.

the autoencoder to a standard scale of approximately -1 to 1 scale. The output of the autoencoder is not unscaled into the original HPC scale, but left in the scaled form. It is used against the scaled input to calculate the mean absolute error of the reconstruction error of the autoencoder. The lower MAE the better the quality of the reconstruction. The calculated MAE value is what is sent to the checker to see if it is within an acceptable range.

Lastly, we implement a high level synthesis (HLS) design to provide an estimate for hardware cost. The HLS is not optimized for space, however LSTM type nodes have a lot of state and can therefore use a lot of space. For brevity, the biggest LSTM autoencoder used 384 BRAM cells, 7229 DSP slices, 893989 flip flops (FF), and 554323 look up tables (LUTs). The smallest used: 85 BRAM, 5095 DSP slices, 587647 FF, 310285 LUTs. Both could achieve a critical path timing of 5 nanoseconds. These resource usages only include the cost for the LSTM autoencoder, not the WDT.

4.4.2 Checker Design And Parameters

The design of the checker, which takes the final output signals of the LSTM autoencoder and WDT to make the decision to determine if the software is operating correctly. In this particular work we deploy a simple checker that will flag the software as anomalous if it passes a threshold for error or if it fails an acceptable timer interval.

4.4.3 Arbitrary Execution Results

Table 4.2 shows the proposed technique's performance across the set of the benchmark programs. The table shows the classification rates for each program. Additionally, it breaks down the individual contribution of the LSTM autoencoder and the WDT and their combined performance. Going from right to left, where a major column is denoted by the double vertical bar, the first major column shows the program's name that is being evaluated. The second major column shows the performance of just the autoencoder alone. The third major column shows the rates for the WDT when trying to determine normal runtime alone. The last major column shows the combined performance of the LSTM autoencoder and the WDT.

The accuracy of each of the techniques: the autoencoder, the WDT, and the combination of them is reported in Table 4.4.

This scenario is for when the software is maliciously under attack from an adversary. That is, once a vulnerability has been exploited the adversary can now exercise arbitrary code execution. The approach does not aim to halt or significantly impede the exploitation itself, but it aims to significantly raise the bar in what kind of arbitrary code that can be executed. Ideally, the only way this proposed system can be defeated would be to be running code that passes both the autoencoder and WDT. This means that an attacker would need to have knowledge of the program's instruction composition (% of loads, % of stores, etc.) within a given timeslice. Additionally, an attacker would ideally need have to know timing information to bypass the WDT.

4.4.4 Random Execution Results

Table 4.3 shows the results the scenario for when the program starts to randomly execute function within its own address space. This case aims to represent when embedded software goes wrong due to outside influence or programming error. An example here is if the program has some bit flips in memory which cause the program to start acting erratically.

We present the accuracy of the two components, autoencoder and WDT, and when combined in Table 4.5.

This scenario is synthetic compared to the previous scenario. This is because the frequency which faults that induce this sort of behavior outside of a malicious or contrived scenario is exceedingly rare.

	Autoencoder				WDT				Combined			
Program	TP	FP	TN	FN	TP	FP	TN	FN	TP	FP	TN	FN
cjpeg	100.00%	0.00%	100.00%	0.00%	27.27%	0.00%	100.00%	72.73%	100.00%	0.00%	100.00%	0.00%
linear	90.91%	0.00%	100.00%	9.09%	36.36%	0.00%	100.00%	63.64%	100.00%	0.00%	100.00%	0.00%
loops	77.27%	0.00%	100.00%	22.73%	0.00%	0.00%	100.00%	100.00%	77.27%	0.00%	100.00%	22.73%
nnet	86.36%	27.27%	72.73%	13.64%	63.64%	0.00%	100.00%	36.36%	86.36%	27.27%	72.73%	13.64%
parser	100.00%	4.55%	95.45%	0.00%	9.09%	0.00%	100.00%	90.91%	100.00%	4.55%	95.45%	0.00%
radix	100.00%	4.55%	95.45%	0.00%	22.73%	0.00%	100.00%	77.27%	100.00%	4.55%	95.45%	0.00%
sha	27.27%	0.00%	100.00%	72.73%	0.00%	0.00%	100.00%	100.00%	27.27%	0.00%	100.00%	72.73%
zip	100.00%	0.00%	100.00%	0.00%	27.27%	0.00%	100.00%	72.73%	100.00%	0.00%	100.00%	0.00%

Table 4.2: Results for the arbitrary code execution evaluation. The program name of the program in the EEBMC benchmark. TP stands for true positive. False positive is shortened to (FP). Lastly, true negative and false negative are abbreviated as TN and FN respectively.

	Autoencoder				WDT				Combined			
Program	TP	FP	TN	FN	TP	FP	TN	FN	TP	FP	TN	FN
cjpeg	100.00%	0.00%	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%	100.00%	0.00%	100.00%	0.00%
linear	9.09%	0.00%	100.00%	90.91%	0.00%	0.00%	100.00%	100.00%	9.09%	0.00%	100.00%	90.91%
loops	90.91%	0.00%	100.00%	9.09%	0.00%	0.00%	100.00%	100.00%	90.91%	0.00%	100.00%	9.09%
nnet	36.36%	27.27%	72.73%	63.64%	0.00%	0.00%	100.00%	100.00%	36.36%	27.27%	72.73%	63.64%
parser	100.00%	4.55%	95.45%	0.00%	0.00%	0.00%	100.00%	100.00%	100.00%	4.55%	95.45%	0.00
radix	100.00%	4.55%	95.45%	0.00%	0.00%	0.00%	100.00%	100.00%	100.00%	4.55%	95.45%	0.00%
sha	68.18%	0.00%	100.00%	31.82%	0.00%	0.00%	100.00%	100.00%	68.18%	0.00%	100.00%	31.82%
zip	100.00%	0.00%	100.00%	0.00%	0.00%	0.00%	100.00%	100.00%	100.00%	0.00%	100.00%	0.00%

Table 4.3: Results for random code execution in the same address space. The program name of the program in the EEBMC benchmark. TP stands for true positive. False positive is shortened to (FP). Lastly, true negative and false negative are abbreviated as TN and FN respectively.

4.5 Discussion

The proposed approach aims to provide a lightweight technique for raising the bar for exploiting and increasing the robustness of embedded software. Since these are embedded environments there are constraints on space, power, how much computing power is available, etc. With these constraints in mind we tried to develop the proposed method to only require a minimal amount of state space. The work here is similar to that in [31]. The approach in [31] has three shortcomings. The first is that the placement of detour points is manual. The second is that the detour points can be bypassed depending on what the attacker does. The third shortcoming is that it uses a database of "golden values" to check against to ensure the software is still operating correctly.

Our approach solves the first shortcoming by inserting a detour point at every major control flow change, e.g. calling and returning from functions. We address the second short coming by incorporating a WDT. While it is possible an attacker could insert and start executing code that never activates the sampling event, the watchdog prevents this. This is due to the WDT reset being tied to the sample event. That way, if the watchdog timer is not "pet" within the determined interval then the program will be flagged as anomalous. The last point is extensive state space. Our method has four points for state space. The first is the timeslice buffer where the HPC events are stored for processing. The second is the LSTM autoencoder itself. The third is the parameters for the scaler, power scaler in this case. The fourth is the thresholds and timing intervals held in the checker. This is only counting the extra storage space required, not any additional hardware space required for computation.

4.5.1 Arbitrary Code Execution

Constructing a benchmark to measure the effectiveness for the proposed technique is difficult. To the best of our knowledge there are no existing available benchmarks for security research like this described in the literature. Previous works that use similar HPC approaches are either focused on malware detection [39, 37, 41] or they build their own custom environment to evaluate the effectiveness [42, 31].

Program Name	Acc. Autoencoder	Acc. WDT	Acc. Combined
cjpeg	100.00%	63.64%	100.00%
linear	95.45%	68.18%	100.00%
loops	88.64%	50.00%	88.64%
nnet	79.55%	81.82%	79.55%
parser	97.73%	54.55%	97.73%
radix	97.73%	61.36%	97.73%
sha	63.64%	50.00%	63.64%
zip	100.00%	63.64%	100.00%

Table 4.4: Calculated accuracy from the results for arbitrary execution.

Program Name	Acc. Autoencoder	Acc. WDT	Acc. Combined
cjpeg	100.00%	50.00%	100.00%
linear	54.55%	50.00%	54.55%
loops	95.45%	50.00%	95.45%
nnet	54.55%	50.00%	54.55%
parser	97.73%	50.00%	97.73%
radix	97.73%	50.00%	97.73%
sha	84.09%	50.00%	84.09%
zip	100.00%	50.00%	100.00%

Table 4.5: Calculated accuracy from the results for random execution.

From Table 4.2 we can see that the LSTM autoencoder is doing the majority of the work in regards to detecting anomalous behavior. This is not unexpected as we are retrofitting a WDT onto existing software that was built without using a WDT. However, we can see that the WDT still plays a role in the *linear* program. The *TP* percentage increases from 90.91% to 100% after the WDT has been incorporated. The WDT also has the added benefit of a cheap and effective way of enforcing that a record event gets executed.

4.5.2 Random Execution

An interesting, but unsurprising outcome is the Acc. WDT column in Table 4.5. The WDT is 50% for every program in the test corpus. While this shows that the WDT does not contribute anything meaningful to the detection of catching programs executing random instructions. The result here shows an obvious place for improvement in future work. However, the only way to accomplish this might be to allocate more state space for the WDT compared to the simple set of valid intervals used in this work. Even though the WDT does not contribute to the performance in detection it is still a valuable resource as it places a hard, strict, and simple bound on the behavior of anomalous execution.

4.6 Conclusion

We propose a new technique and design a simulator to evaluate our proposed technique of combining autoencoders and watchdog timers for embedded software security. We aim to minimize the amount of state space required for proposed method as it is to be deployed in an embedded environment. The technique achieves an average of 91% accuracy in detecting anomalous behavior of software against arbitrary code execution. When detecting if the program has started to execute random instructions it has reached an 88% accuracy on the test corpus.

The technique has similar goals, protecting embedded software, to those previously proposed in the literature. However, it overcomes some of the shortcomings that exist in the existing literature that utilize HPCs for embedded software security.

5. CONCLUSION & FUTURE DIRECTIONS

5.1 Conclusions

This dissertation focuses on enhancing the current state of embedded system software security. It starts with a method to detect anomolous execution of embedded system software. Further, the dissertation proposes a method of continuous authentication for embedded software. In addition to anomaly detection and continuous authentication the dissertation also proposes enhancing a traditional watchdog timer with hardware performance counters to increase robustness against both random transient errors, as well as malicious attacks. The objectives through this dissertation are to:

1. Enhance the current state of embedded system software security
2. Aim to keep the overhead of the proposed methods low and suitable for embedded systems
3. Flexible in implementation across the diverse landscape of embedded systems
4. Layerable with other existing security methods and vulnerability mitigations
5. Not become immediately obsolete with future developments in the security space

To accomplish this in Chapter 2 a hardware performance counter based anomaly detection methods are proposed. Hidden Markov models and long short term memory neural networks are used for anomaly detection. Chapter 3 explored the use of short time Fourier transforms as a way of continuous authentication for embedded software. Chapter 4 investigates augmenting a watchdog timer an autoencoder based in HPCs to enhance security of embedded system software.

5.2 Dissertation Summary

This dissertation proposes three separate ways to increase the security of embedded system software. The first way presents two separate approaches to using anomaly detection. The first approach for anomaly detection makes use of hidden Markov models based on the stream of HPC

data for the embedded software. It trains an HMM model on known good HPC data streams of executions of the embedded software. The model is then deployed and it calculates a likelihood of the streamed HPC data so far. If the likelihood drops below a specific threshold then the software's execution is flagged as anomolous. The second approach trains on the same stream of HPC data from known good executions of embedded software. However, this time it trains an LSTM model to predict what the next HPC data point should be. If the prediction results falls outside of a specific threshold the software is flagged as anomolous.

The second approach works by way of utilizing the knowledge of program phases. The frequency information of the HPC data stream is calculated by way of short time Fourier transforms, then the frequency information is encoded into a finite state machine. This way the state machine encodes the valid transitions between every other state. Once the state machine is built and deployed it is simple to check that the software is executing as intended. To verify the embedded software is running as intended the frequency information of the HPC data stream is computed and transformed in to a state. The computed state is the passed into the state machine which checks to see if it is a valid state transition from the current state it is in. If it is not a valid state the software is flagged as anomolous and corrective action is taken. Additionally, epsilon transitions can be added which allow for a free state transition between any two states.

The final approach makes use of watchdog timers. Specifically, a watchdog timer is enhanced with an autoencoder model trained on the HPC data stream. Every time the watchdog timer is reset, the HPC data is passed along into the autoencoder. The data is transformed through the autoencoder and is compared to the original HPC value. The auto encoder is trained on known good executions of the embedded software's HPC data stream. If there is a large difference between the output of the autoencoder and the original HPC value this indicates a high error amount, which is indicative of unexpected execution of the embedded software. Corrective action can be issued at this point. The watchdog also serves as another mechanism that must be satisfied by normal operation of the embedded software. If the watchdog timer is not reset before it expires that indicates the embedded software is not executing as intended and corrective action can be taken.

5.3 Future Directions

With the boom and rapid expansion of embedded systems, by way of the the Internet of Things, embedded system software security's importance will only become more paramount. With more systems deployed the overall threat level that insecure systems pose is rising, as demonstrated by the Mirai and similar botnets. To help defuse this threat future work in the space of embedded system software security is needed.

5.3.1 Custom Hardware Performance Counters

The work throughout this dissertation was limited to already available HPCs on contemporary CPUs. Newly proposed, that currently to not exist, HPCs may give much better insight into the execution characteristics of embedded software. Such HPCs could simplify the approaches taken and increase the state of embedded software security. Proposing such general counters would be a difficult, as ideally they would need to work across the broad and diverse landscape of embedded software.

5.3.2 Purpose Fit Lightweight Artificial Intelligence or Machine Learning

The techniques used in this dissertation are adapted from full scale AI and ML techniques. With more AI and ML research focused on edge computing these approaches and techniques could be adapted into this particular application, specifically in deploying custom hardware on chip in embedded systems.

5.3.3 Runtime Randomization Of HPCs During Embedded Software Execution

To further increase the security of embedded software, the techniques in this dissertation could be adapted and enhanced to deal with a random selection of the subset of all available HPCs. This would have the effect of making it harder for an adversary hit a moving target of the subset of available HPCs thereby increasing the difficulty of defeating the system.

REFERENCES

[1] I. Analytics, "State of the iot 2018: Number of iot devices now at 7b - market accelerates as companies look for competitive advantage." https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/, 2018.

[2] I. Markit, "Iot trend watch 2016: The rise of the internet of things." https://ihsmarkit.com/research-analysis/iot-trend-watch-2016-the-rise-of-the-internet-of-things.html, 2016.

[3] Statista, "Internet of things (iot) connected devices installed base worldwide from 2015 to 2025 (in billions)." https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/, 2021.

[4] "2017 embedded markets study." https://staff.emu.edu.tr/mehmetbodur/Documents/courses/CMPE423/A21T1/lec-00/embedded-market-study-2017.pdf. Accessed: 2017-5-4.

[5] "2019 embedded markets study." https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf. Accessed: 2019-6-4.

[6] "2018 embedded systems safety & security survey." https://barrgroup.com/sites/default/files/downloads/barr_group_2018_embedded_systems_safety_security_survey.pdf. Accessed: 2019-5-12.

[7] B. Krebs, "Hacker group that took down twitter, facebook, krebsonsecurity and more comes forward." https://krebsonsecurity.com/2016/10/hacker-group-that-took-down-twitter-facebook-krebsonsecurity-and-more-comes-forward/, 2016.

[8] E. Newcomer, "How the mirai botnet knocked major websites offline and disrupted the internet," *IEEE Spectrum*, December 2017.

[9] CERT, "Satori botnet: A new variant of mirai," tech. rep., Carnegie Mellon University, April 2019.

[10] Symantec, "Masuta botnet targets iot devices with obsolete routers." `https://www.symantec.com/blogs/threat-intelligence/masuta-botnet-obsolete-router-exploits`, 2018.

[11] N. S. 800-53, "Security and privacy controls for federal information systems and organizations," 2020.

[12] I. 27001, "Information technology security techniques information security management systems requirements," 2013.

[13] W. Stallings, "Cryptography and network security: principles and practice," *Pearson Education*, 2014.

[14] A. K. Sharma and D. P. Sanghi, "Internet of things: architectures, protocols, and applications," *Journal of Electrical and Computer Engineering Innovations*, vol. 6, no. 1, pp. 13–22, 2018.

[15] R. Meena, A. Anand, and V. Goyal, "A survey of iot architectures, protocols, applications, and enabling technologies," *Journal of Network and Computer Applications*, vol. 98, pp. 27–48, 2017.

[16] S.-T. Yoon, S. G. Hong, and S. H. Kim, "Design and implementation of an iot device for smart factory," *Journal of Ambient Intelligence and Humanized Computing*, vol. 9, no. 6, pp. 1775–1786, 2018.

[17] C. Gomez, J. Oller, and J. Paradells, "Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology," *Sensors*, vol. 12, no. 9, pp. 11734–11753, 2012.

[18] J. W. Valvano, *Embedded Systems: Introduction to Arm® Cortex™-M Microcontrollers*. CreateSpace Independent Publishing Platform, 2019.

[19] R. Zurawski, *Embedded Systems Handbook*. CRC Press, 2005.

[20] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2018.

[21] K. Iniewski, *Embedded Systems: Design, Analysis and Verification*. CRC Press, 2018.

[22] X. Fan and J. Wang, *Real-Time Embedded Systems*. Springer International Publishing, 2017.

[23] S. S. M. Chowdhury, K. Ahmed, N. Kumar, and A. Kundu, "A review on internet of things (iot): architecture, applications, security, and challenges," *IEEE Internet of Things Journal*, vol. 5, no. 5, pp. 3949–3973, 2018.

[24] A. D. Zamora and R. P. Jover, *Internet of things: principles and paradigms*. CRC Press, 2016.

[25] K. Ott and R. Mahapatra, "Hardware performance counters for embedded software anomaly detection," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, pp. 528–535, IEEE, 2018.

[26] R. H. Weber, "Internet of things–new security and privacy challenges," *Computer law & security review*, vol. 26, no. 1, pp. 23–30, 2010.

[27] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, vol. 2, pp. 119–129, IEEE, 2000.

[28] T. Saito, R. Watanabe, S. Kondo, S. Sugawara, and M. Yokoyama, "A survey of prevention/mitigation against memory corruption attacks," in *Network-Based Information Systems (NBiS), 2016 19th International Conference on*, pp. 500–505, IEEE, 2016.

[29] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries.," in *NDSS*, 2015.

[30] T. Morris, "Trusted platform module," in *Encyclopedia of cryptography and security*, pp. 1332–1335, Springer, 2011.

[31] X. Wang, C. Konstantinou, M. Maniatakos, and R. Karri, "Confirm: Detecting firmware modifications in embedded systems using hardware performance counters," in *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*, pp. 544–551, IEEE, 2015.

[32] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pp. 141–150, IEEE, 2008.

[33] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pp. 215–224, IEEE, 2013.

[34] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.

[35] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[36] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2017.

[37] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 559–570, ACM, 2013.

[38] X. Wang and R. Karri, "Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters," in *Proceedings of the 50th Annual Design Automation Conference*, p. 79, ACM, 2013.

[39] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 651–661, IEEE, 2015.

[40] C. Malone, M. Zahran, and R. Karri, "Are hardware performance counters a cost effective way for integrity checking of programs," in *Proceedings of the sixth ACM workshop on Scalable trusted computing*, pp. 71–76, ACM, 2011.

[41] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 109–129, Springer, 2014.

[42] Y. Xia, Y. Liu, H. Chen, and B. Zang, "Cfimon: Detecting violation of control flow integrity using performance counters," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pp. 1–12, IEEE, 2012.

[43] "Eembc - coremark - processor benchmark." `http://www.eembc.org/coremark/index.php?b=pro`. Accessed: 2017-1-20.

[44] "musl libc." `https://www.musl-libc.org/`. Accessed: 2017-01-20.

[45] K. Ott and R. Mahapatra, "Continuous authentication of embedded software," in *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom)*, pp. 128–135, IEEE, 2019.

[46] S. Etalle and A. Abbasi, "Identifying & addressing challenges in embedded binary security," 2017.

[47] W. Louis, M. Komeili, and D. Hatzinakos, "Continuous authentication using one-dimensional multi-resolution local binary patterns (1dmrlbp) in ecg biometrics," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 12, pp. 2818–2832, 2016.

[48] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks," in *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pp. 49–54, ACM, 2009.

[49] M. Frank, R. Biedert, E. Ma, I. Martinovic, and D. Song, "Touchalytics: On the applicability of touchscreen input as a behavioral biometric for continuous authentication," *IEEE transactions on information forensics and security*, vol. 8, no. 1, pp. 136–148, 2013.

[50] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," in *IEEE Symposium on Security and Privacy (SP)*, pp. 345–363, IEEE, 2019.

[51] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[52] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. H. an d Stefan Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[53] K. Ott and R. Mahapatra, "Hardware performance counter enhanced watchdog for embedded software security," in *24th International Symposium on Quality Electronic Design (ISQED)*, IEEE, 2023.

[54] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, *et al.*, "Understanding the mirai botnet," in *26th {USENIX} security symposium ({USENIX} Security 17)*, pp. 1093–1110, 2017.

[55] T. Xu, J. B. Wendt, and M. Potkonjak, "Security of iot systems: Design challenges and opportunities," in *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 417–423, IEEE, 2014.

[56] T. Lecomte, D. Déharbe, É. Prun, and E. Mottin, "Applying a formal method in industry: a 25-year trajectory," in *Brazilian Symposium on Formal Methods*, pp. 70–87, Springer, 2017.

[57] R. Bagnara, A. Bagnara, and P. M. Hill, "The misra c coding standard and its role in the development and analysis of safety-and security-critical embedded software," in *International Static Analysis Symposium*, pp. 5–23, Springer, 2018.

[58] B. Davis, R. N. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka, *et al.*, "Cheriabi: Enforcing valid pointer provenance and minimizing pointer privilege in the posix c run-time environment," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 379–393, 2019.

[59] P. Koopman, "Risk areas in embedded software industry projects," in *Proceedings of the 2010 Workshop on Embedded Systems Education*, pp. 1–8, 2010.

[60] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[61] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pp. 9–16, 2011.