UNIVERSITY OF CALIFORNIA

Los Angeles

Quality of Time: A New Perspective in Designing Cyber-Physical Systems

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Electrical and Computer Engineering

by

Fatima Muhammad Anwar

2019

© Copyright by

Fatima Muhammad Anwar

2019

ABSTRACT OF THE DISSERTATION

Quality of Time: A New Perspective in Designing Cyber-Physical Systems

by

Fatima Muhammad Anwar
Doctor of Philosophy in Electrical and Computer Engineering
University of California, Los Angeles, 2019
Professor Mani B. Srivastava, Chair

Unprecedented Cyber-Physical Systems (CPS) and Internet of Things (IoT) applications such as health care, connected vehicles, and augmented/virtual reality are revolutionizing smart spaces and change how we build and manage our systems. These applications span the cloud and the edge devices and give birth to new system designs with critical dependence on temporal use cases. As such, cloud services are expected to provide `timely responses` and `schedulable demands`, while edge devices are required to `synchronize observations` and `choreograph actions` across distributed entities. Both cloud and edge demand time awareness in general, and time-indexed queries, precise timestamping, and dynamic clock synchronization in particular. However, contemporary distributed system designs are inherently "clockless" and becoming increasingly complex. They fail to meet consistency, causality, and scheduling demands of underlying applications yet enabling time awareness for various applications running on commodity platforms and operating systems (OS) is a challenge in itself.

In this dissertation, we devise a new way of acquiring time information by introducing the notion of `Quality of Time (QoT)` that collectively captures various time metrics such as resolution, accuracy, stability, and integrity. Analogous to Quality of Service (QoS) in networking, QoT treats time as a controllable OS primitive with observable performance. To provide QoT to applications, we proposed the first OS abstraction – `timeline` – that reacts to application timing demands and exposes QoT to applications in an easy-to-use, secure, and scalable way. This degree of richness of information had never been available to coordinated applications whose activities are choreographed across time and space. This flow of information was immediately relevant to the broader field of IoT addressing the emerging temporal use cases for applications at the cloud and the edge in a secure fashion. As such, QoT expanded distributed applications to global scale with no significant overhead and no performance compromise.

This dissertation focuses on covering various aspects of QoT. In the first part of this thesis, we design extensible abstractions to *characterize timing uncertainty* in the presence of timing variations. In an effort to reduce complexity and overhead of current distributed database designs, our abstractions and systems enable globally replicated lockless transactions with simplified design, low overhead and no loss in performance. The second part exposes timing vulnerabilities in trusted execution technologies and network security mechanisms and provides *timing integrity* by designing secure time architectures in the presence of vulnerabilities. Thus enabling trusted timestamping in commodity systems to preserve one's digital rights and digital signals. The third part focuses on redesigning the hardware, OS and network interfaces that help time information flow between applications and systems, and enable *timing precision*. This precision boosts high speed measurements at large-scale distributed entities. The final section addresses the inefficacy of testing mechanisms for time synchronization protocols deployed in safety-critical environments. We then propose a customized testbed for *testing timing robustness* under failures and adversarial attacks.

Current designs in distributed systems rely on message-passing based protocols and come at a huge energy and bandwidth cost along with high system complexity. In contrast, our designed systems based on QoT support new temporal use cases of globally distributed applications with low computation and communication overhead. We released our system designs to support various time management and clock synchronization use cases in emerging distributed applications.

The dissertation of Fatima Muhammad Anwar is approved.

Ankur M. Mehta

Miryung Kim

Gregory J. Pottie

Mani B. Srivastava, Committee Chair

University of California, Los Angeles

2019

To my dedicated parents *Robina & Anwar* who guided me every step of the way,

my beloved husband *Taqi* who motivated me to go beyond my way,

and

my beautiful daughter *Huda* who unearthed the new side of me along the way.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

In the Name of Allah, the Most Beneficent, the Most Merciful. All praises and thanks be to Allah, “Then which of the blessings of your Lord will you deny”, Quran[55:13]. Indeed I say what Prophet Mohammad PBUH said, “My Lord, increase me in knowledge”, Quran[20:114], and what Prophet Musa said, “My Lord, expand for me my chest, ease for me my task, and untie the knot from my tongue that they may understand my speech” Quran[20:25-28].

I am highly indebted to my academic advisor Mani Srivastava for his extensive support, and for providing me countless opportunities to excel in my career. Thanks to him, the intellectual growth and professional maturity I attained during the last five years is unparalleled. Above all, the collaborative working environment he has maintained at Networked and Embedded Systems Lab (NESL) has been instrumental in my productivity and continuity of my degree. I cherish our working relationship based on mutual trust.

My collaborators across different universities also contributed to my success. I want to thank my faculty collaborators Rajesh Gupta (UCSD) and Raj Rajkumar (CMU) for their valuable support. I also wish to express my gratitude to Anthony Rowe (CMU) for the great ideas and the opportunities he provided to interact with fellow researchers. My student collaborators Sandeep D’souza (CMU) and Adwait Dongare (CMU) have been very much involved in my work. Above all, Andrew Symington (NASA) has been a great mentor during the early years of my PhD. My dissertation would not be completed without their help and support, and I am thankful for that. I am also grateful to my dissertation committee at UCLA: Greg Pottie, Miryung Kim, and Ankur Mehta for their invaluable feedback on my thesis proposal that helped shape this dissertation.

I am fortunate to spend the last five years at NESL. It is a working space filled with friendly and collaborative people. I want to thank all former and current NESLites who directly or indirectly contributed to my success: Dr. Luis Garcia, Dr. Yasser Shoukry, Dr. Chenguang Shen, Dr. Salma Elmaliki, Dr. Bharathan Balaji, Dr. Bo-Jhang Ho, Moustafa Alzantot, Sandeep Singh, Renju Liu, Joseph Noor, Tianwei Xing, Nathaniel Snyder, Xi

Han, Vikranth Jeyakumar, Leon Kozinakov, Brian Wang, Akash Deep, and Ziqi Wang. Thank you to all my seniors and mentors at NESL for your valuable advice, to my peers for your support, and my juniors for the fun moments and laughs. It has indeed been a fantastic experience.

Special thanks to my mother Robina Shaheen who unconventionally raised me and taught me never to give up and aim high, and to my father Muhammad Anwar who has provided for me in every possible way and showed me how proud he is of who I am today. Thank you Ammi Abbu for your love, support, and encouragement. I am forever indebted to you. My siblings deserve a special mention too: Sahar, Ali, Fakhra, and Amirah, you all have been amazing and my support group. Thank you for your frequent visits, moral support and unconditional love. You all made California home away from home.

A big thank you to my beloved husband Taqi and my lovely princess Huda. Only the two of you know how it has been like for the last five years. It was not easy in any way, but when I look back, all I remember is my husband's motivating words and my daughter's lovely smile that pushed me forward to where I am today. Thank you for always being there for me.

VITA

2004 – 2008	B.S. in Electrical Engineering, University of Engineering and Technology, Lahore (UET), Graduation with Honors
2009 – 2011	M.S. Computer Engineering, Ajou University Korean Govt. Scholarship, Brain Korea (BK21)
2011 – 2013	Software Engineer at Samsung Electronics, Mobile Communication Division Promoted to Associate Engineer
2014	Electrical Engineering Departmental Fellowship, UCLA
2014 – 2019	Graduate Student Researcher, Electrical and Computer Engineering Department, UCLA
2016	Teaching Assistant, Electrical and Computer Engineering Department, UCLA
2017	Grace Hopper Scholar
2018	Qualcomm Innovation Fellowship (QInF) Finalist

PUBLICATIONS

Fatima M. Anwar, Luis Garcia, Xi Han, Mani Srivastava. "Securing Time in Untrusted Operating Systems with TimeSeal". (under review)

Fatima M. Anwar, Mani Srivastava. "A Case for Feedforward Control with Feedback Trim to Mitigate Time Transfer Attacks". (under review)

Fatima M. Anwar, Amr Alanwar, Mani Srivastava. “OpenClock: A Testbed for Clock Synchronization Research”. IEEE Symposium on Precision Clock Synchronization for measurement, control, and communication (ISPCS), Oct 2018, CERN, the European Organization for Nuclear Research.

Fatima M. Anwar, Mani Srivastava. “A trust in time saves millions”. USENIX Summit on Hot Topics in Security, Aug 2018, Baltimore, USA.

Muhammad Taqi Raza, Fatima M. Anwar, Songwu Lu. “Exposing LTE Security Weaknesses at Protocol Inter-Layer, and Inter-Radio Interactions”. 13th International Conference on Security and Privacy in Communication Networks (SecureComm), Oct 2017, Niagra Falls, Canada.

Fatima M. Anwar, Mani Srivastava. “Precision Time Protocol over LR-WPAN & 6LoWPAN”. IEEE Symposium on Precision Clock Synchronization for measurement, control, and communication (ISPCS), Aug 2017, California, USA.

Amr Alanwar, Fatima M. Anwar, Yi-Fan Zhang, Justin Pearson, Joao Hespanha, Mani Srivastava. “Cyclops: PRU Programming Framework for Precise Timing Applications”. IEEE Symposium on Precision Clock Synchronization for measurement, control, and communication (ISPCS), Aug 2017, California, USA.

Fatima M. Anwar, Sandeep D’souza, Adwait Dongare, Anthony Rowe, Raj Rajkumar, Mani Srivastava. “Timeline: An Operating System Abstraction for Time-Aware Applications”. IEEE Real Time Systems Symposium (RTSS), Dec 2016, Porto, Portugal.

Amr Alanwar, Fatima M. Anwar, Joao Hespenha, Mani Srivastava. “Realizing Uncertainty-Aware Timing Stack in Embedded Operating Systems”. ACM Embedded Operating Systems Work- shop in conjunction with ESWEEK, Oct 2016, Pittsburgh USA.

CHAPTER 1

Introduction

Technology trends such as Cyber-Physical Systems (CPS) and Internet of Things (IoT) drive us towards a world where a large number of end-point, intermediary devices, and servers share, store and process real-time data leading to the emergence of time-aware applications with extremely diverse timing requirements. The timing requirements of applications in one domain of connected systems may be substantially different for others, and may also change over time. For instance, application requirements in distributed and autonomous robotics lie in the range of seconds to milliseconds. Autonomous driving use cases rely on the millisecond to microsecond precision. While monitoring grid health requires at most nanosecond accuracy. These applications with diverse timing needs run on commodity platforms and operating systems that were never designed to handle these precise and adaptive requirements: these devices undergo variations that cause uncertainty in the notion of time. As such, numerous vulnerabilities compromise timing integrity. Access to precise time on these devices is quite complicated. Above all, there are limited mechanisms to test time robustness against faults and malicious attacks.

Enabling time awareness for various applications in CPS running on commodity platforms and operating systems (OS) in the presence of timing variations and vulnerabilities is a big challenge. Given a broad range of CPS applications that benefit from time awareness, this dissertation puts forward a new concept: Quality of Time (QoT) that collectively captures the state of various timing primitives, and provides the opportunity to applications to observe and control their time quality. QoT is a new paradigm that characterizes *uncertainty*, maintains *integrity*, provide required *precision*, and test *robustness* under failures and adversarial conditions. Hence QoT serves the needs of all time-aware applications.

1.1 Challenges

Awareness of Time has never been more important than now because there has been increased reliance on temporal use cases in unprecedented CPS and IoT applications. For instance, critical services rely on time awareness across the whole autonomous driving ecosystem. Distributed infrastructure constructs a common environmental picture through a shared sense of time with respect to each other. A temporal understanding between infrastructure and autonomous vehicles enables efficient traffic management. Time-critical use cases emerge among vehicles and pedestrians, and in-vehicle sensors and controllers work on a common time scale to infer system states such as traffic sign recognition or assisting brake system. We are surrounded by time-aware applications spread across the spectrum of time and space. However, the clockless assumption underlying the design of these distributed applications increases complexity and inhibits extensibility giving rise to consistency, causality, and scheduling issues. Below, we highlight several challenges that arise in the context of enabling time-awareness over commodity platforms and operating systems in the presence of timing variations and vulnerabilities.

1.1.1 Timing Uncertainty

Maintaining a shared notion of time is critical to the performance of many distributed systems such as swarm robotics [G 13], high-frequency trading [EB12], telesurgery [NG09], Big Science [LWS11] and global-scale databases [CDE13a]. Technologies such as GPS, Precision Time Protocol [LEW05] and chip-scale atomic clocks have made it possible to provide systems with accurate, stable, and a common notion of time across a network. However, other technology trends have made it harder for applications to benefit from these advances in timing technologies. For example, asymmetric medium delays degrade time transfer [LL84], imperfect oscillators cause timing jitter [ZNK08], multi-core systems have timing inconsistencies [KR11], and abstractions like virtual machines introduce more significant timing uncertainty [BCR10]. In multiple domains ranging from database consistency [CDE13a] to interactive cloud gaming [LCC15], the knowledge of timing uncertainty has proven to be useful. However, application-level visibility into timing uncertainty remains mostly unexplored in current systems.

1.1.2 Timing Integrity

Emerging temporal use cases in IoT applications have a critical dependence on high precision and accurate *relative* time. For instance, distance and speed calculations rely on precise round trip times [MPP07] [TSS16], schedulers build upon elapsed and remaining times [RPM13] [MW13], network telemetry depends on residency delays [LPJ15], code profiling requires execution times [CZR17], and data sampling needs timestamps [HSG18]. Attacking a system's sense of time has ramifications such as location theft, network outages, higher delays, and data inconsistencies. Furthermore, critical functionalities for securing applications in shielded execution environments such as Haven [BPH15], SCONE [ATG16], Panoply [STT17] and Graphene-SGX [TPV17] have no access to secure time. Instead, they rely on untrusted operating system (OS) time. A compromised OS may lie about the time or signal early timeouts, and although traditional cryptographic techniques, trusted execution technologies, and network security mechanisms may guarantee data security, they do not cater to *time security.*

Network elements that assist time transfer can also be malicious. They can develop various Man in the middle (Mitm) capabilities such as dropping, replaying, pre-playing and delaying packets. Previous works [MDA16] [YAY13] [DSD18] talk about attacks on time transfer packets and mitigating these attacks using cryptographic [AFZ17a] [IW17] and network security [sgx16] [AFZ17b] mechanisms. Unfortunately, Unfortunately, delay attacks are considered exceedingly hard to protect against and immune to proposed mitigations [UV09] [MVV17]. The ultimate goal of a Mitm attacker is to move the client's clock away from true time towards its malicious time for illegal activities in high-frequency trading [PH16], digital rights violation [CRS14], and spoofing location [SP05], etc.

1.1.3 Timing Precision

Modern distributed systems comprise of many different devices. These devices need to share a common notion of time, which may or may not be pinned to some global time coordinate like UTC, to choreograph their actions. Various hardware and software solutions exist for providing standalone and networked devices with precise knowledge of time such as GPS and atomic clocks, network adapters that perform hardware timestamping [LEW05], and synchronization algorithms that achieve low-nanosecond time synchronization between devices [LWS11]. These precise time solutions are mostly available for wired interfaces, preferably ethernet. We have yet to see precise time solutions for applications in Low Range Wireless Personal Area Networks (LR-WPAN) even though numerous applications in LR-WPAN require precise time-awareness such as real-time positioning systems, formation flying, distributed sound systems, and foraging applications. Various sensors and actuators also demand precise timestamping and scheduling capabilities.

1.1.4 Timing Robustness

Hardware capabilities required for clock synchronization have developed significantly in the past decade; hardware timestamping feature is introduced for many processors, co-processors [AAZ17], and network interface cards [AS17] While various systems have extensively made use of time-based technology developments to push for better performance [ADS16], there are limited testing mechanisms available for comparing performance of synchronization algorithms in a comprehensive manner. Industrial and automotive applications heavily rely on these clock synchronization techniques, yet these applications operate under uncertain environments and prone to hardware faults, network failures, or Mitm attacks.

Comprehensive testing of a clock synchronization algorithm that is robust to faults, failures, and attacks is necessary before practical deployments in safety-critical applications. Unfortunately, many algorithms are not tested for faults and attacks as it is hard to reproduce them on distributed devices. Due to hardware characteristics of a clock, no two clocks are the same; Cho et al. [CS16] have used unique clock characteristics for fingerprinting electronic control units in cars because a clock model is affected by short and long term variations in jitter, wander, and skew due to physical characteristics of oscillators and environmental variations. For a fair comparison of multiple synchronization algorithms, their disciplinable clock models should be derived from the same hardware clock.

1.2 Our Contributions

The contribution of this dissertation to address the challenges mentioned earlier is multifold. In particular, we design systems that characterize uncertainty in time, establish the integrity of time, provide precise time, and test robustness of time.

The notion of time is not perfect; it has an uncertainty that is indicative of hardware and network variations and vulnerabilities. While researchers have built timing solutions tailored to specific domains, they act as a black box with unobservable uncertainty leading to overdesign, complex precise time access leading to hardware-specific solutions, and inefficient timing services oblivious to applications' timing needs leading to high complexity and overhead. We argue that timing quality should be made visible to the application and should be controllable by the user so that the application can adapt to system variations in a hardware-agnostic manner, and degrade gracefully in the presence of failures. In this regard, this work seeks to close the timing loop between systems and applications by proposing `Quality of Time (QoT)` that provides a new way of thinking for time management. Application requirements are written in terms of a *timing contract* specifying the desired timing metrics such as accuracy, stability, and integrity. QoT is the ability to guarantee a certain level of performance listed in a timing contract. This notion of QoT is inspired by the networking Quality of Service (QoS) requirement, where the system delivers the required QoS specified in terms of a traffic contract.

To provide QoT to applications, we design the first abstraction, `timeline`, to assist coordinating applications with their diverse timing requirements. Timeline is a virtual time base with respect to an epoch. It is a platform-independent OS abstraction that greatly simplifies the development of QoT-aware choreographed applications. QoT architecture is centered around the timeline abstraction and provides an expressive application programming interface that treats time as a controllable, verifiable and observable primitive, and enables developers to quickly write coordinated applications whose activities are choreographed across time and space. We evaluate the timeline abstraction and QoT architecture for a Time Division Multiple Access (TDMA) protocol and for a control application. Both applications are compared in terms of application performance, resource consumption, and developer effort with and without QoT support. Our results indicate that QoT significantly improves resource consumption and decreases developer effort for the same application performance. This work has also motivated follow-up research in geo-distributed IoT, virtualization, and coordinated manufacturing and driving.

Motivated by QoT, this thesis also seeks to provide precision timing to heterogeneous sensors, actuators, and radios by designing a generalized Precise Hardware Clock gPHC that abstracts away from hardware type as long as the desired timing properties are satisfied. System support around gPHC abstraction lays the foundation for IoT applications that enjoy high-level features with low-level determinism for time-critical operations. It provides automatic configuration, a high-level programming language, and a library to interface a real-time unit with the OS. A real-time unit paired with an OS should provide the best of both worlds: the real-time unit handles time-sensitive aspects, and the OS provides the filesystem, scheduling, and networking services. Note that gPHC does not require modification to current clock synchronization protocols or hardware platforms. In practice, we used this high precision system to develop a Phasor Measurement Unit (PMU) for smart grid research. A PMU samples and timestamps phasor measurements from the grid at a very high rate. This collected data can only be correlated if distributed PMU are synchronized in time with high precision. As a result, high precision enables PMUs to detect and isolate grid faults promptly.

Another aspect of QoT is to validate timing robustness especially in the context of safety-critical environments prone to faults and attacks. This dissertation provides an

`OpenClock` testbed. It provides multiple disciplinable clocks on a single platform for fair algorithmic comparison under failures and adversarial attacks. It features a rich set of clock abstractions that categorizes clocks based on their functionalities. These set of clocks are managed by a clock management engine that helps virtualize time-related resources on a platform for modular and extensible design. We also provide an attack simulator for testing algorithmic resilience. We evaluate the performance of a number of protocols and show hardware and network biases in the results when these protocols are tested on different devices, whereas biases are alleviated from results when these protocols are tested on OpenClock testbed. We also added a new attack capability for researchers in the testbed to find vulnerabilities and test the resilience of synchronization algorithms. Architectural support for these abstractions is provided in the Linux kernel over an embedded and x86 platform, creating an opportunity for many distributed applications and services to be time-aware.

In addition, this dissertation establishes the necessary and sufficient conditions to provide timing integrity for QoT. Accurate time is essential for the safe and correct operation of all hardware, software, and networked systems. Unfortunately, an adversary can manipulate clocks and cause hardware faults in logic circuits, measurement errors in sampling, and inconsistencies in distributed systems. Cryptographic techniques do not suffice for time security if the "timeliness" property is violated. An adversary is capable of disrupting temporal characteristics of applications even within the protection of shielded execution such as Intel SGX enclaves and ARM TrustZone. We find that time provided by these trusted execution technologies is also not secure against attacks. We successfully launch delay attacks and scheduling attacks on time provided in trusted technologies. Taking insights from our studies and analysis, we holistically designed a secure time architecture encompassing hardware security, trusted execution environments, and network-based protections.

Time has been manipulated for a wide range of incentives such as location, cryptographic keys, and copyright theft. An attacker can move time backward for digital rights management and movie rentals, disrupt temporal forensic analysis by violating causality, increase user perceived delays, and prove to be physically present at a place where it is not. On the other hand, shielded execution environments are emerging such as Haven, SCONE, Panoply and Graphene-SGX that rely on untrusted OS time for critical functionalities. A compromised OS may lie about the time or signal early timeouts. Traditional approaches have focused on designing cryptographic techniques and fail to secure time. This dissertation points out timing vulnerabilities in trusted execution technologies and network security mechanisms; puts forth challenges surrounding secure time, and provides extensive work to address each challenge. In this regard, `TimeSeal` develops the first trustworthy clock that cannot be manipulated by a privileged attacker – the OS. We address three key challenges to secure time, i.e. (i) we find a trusted timer that no adversary can manipulate (defeat *timer attacks*), (ii) we provide a secure path to that trusted timer (overcome *delay attacks*), and (iii) we maintain timekeeping software that is unaffected by attacks (thwart *scheduling attacks*). TimeSeal leverages TEE for hardware timer protection and exploits the structure of TEE to construct high-resolution counters that detect attacks when they occur. TimeSeal secures time using only software based changes on TEE. As a result, we provide a high-resolution secure clock that is able to timestamp and schedule events with an accuracy of 10's of milliseconds in the presence of time attacks. This work also serves as a reminder for hardware vendors that have traditionally focused on secure computation that there is a great need to design secure peripherals such as secure clocks for CPS applications.

Our approach for global clock synchronization – `Feedforward control with feedback trim` – introduces a new clock disciplining mechanism that protects time transfer packets from Mitm attacks. Time Transfer packets share global time among physically or geographically separated entities. Malicious network elements can replay, pre-play, and delay these packets. Cryptographic mechanisms mitigate most of these attacks, but delay attacks violate packet timeliness and are considered too strong to protect against, especially if the powerful network attacker is capable of attacking all packets in the network. In contrast to approaches that focus on mitigating malicious network delays, our approach uses these malicious delays to its advantage. The key enabler is relying on equally-delayed one-way time transfer packets for frequency synchronization, and replacing traditional feedback controllers with feedforward controllers, with occasional feedback trim for time synchronization. Our system design was able to preserve high precision under all kinds of delay attacks in the network. It is well established that critical infrastructure relies on GPS for global time. However, GPS spoofing is a pressing concern. One of the motivations for this work is to find alternate global time synchronization solutions to GPS.

1.3 Organization

This dissertation highlights four aspects of Quality of Time.

Part I: Systems for Characterizing Timing Uncertainty. This part presents the `timeline` abstraction that enables QoT over commodity platforms and OS.

Part II: Systems for Timing Integrity. This part describes `TimeSeal` that secures the relative time and `feedforward control with feedback trim` for securing global time.

Part III: Systems for Timing Precision. This part presents a new abstraction `gPHC` along with its system support to enable high precision over heterogeneous processors, co-processors, and peripherals.

Part IV: Systems for Testing Timing Robustness. This part presents `OpenClock`, a testbed to compare the performance of different clock synchronization algorithms under simulated failures and attacks.

Part I

Systems for Characterizing Timing Uncertainty

CHAPTER 2

Timeline: An Operating System Abstraction for Time-Aware Applications

2.1 Introduction

Time-aware applications range from small-scale wireless sensor networks to large-scale Cyber-Physical Systems. These applications often involve distributed coordination, temporal ordering of events, or reconciled observations, which require a common notion of time. The taxonomy of these applications exhibits diverse timing requirements. While some applications require fine-grained timing [EB12], others can work with relatively coarse-grained timing. On the other hand, some applications have statically-defined timing requirements, while others have context-dependent dynamic timing requirements. From the standpoint of choosing a time reference, some applications require clock synchronization among coordinating peers [WYM02], while others synchronize their clocks to absolute global time (like UTC) [CDE13a].

Nanosecond-level clock synchronization is now achievable using specialized hardware such as GPS, chip-scale atomic clocks, network adapters that perform hardware timestamping, and clock-synchronization protocols. These technologies have enabled a wide range of applications such as, cellular-telephone backhaul networks, which use specialized hardware for synchronizing transmissions to maximize channel utilization, and distributed databases like Google Spanner [CDE13a] which order database transactions with the aid of GPS synchronized clocks. However, it is not feasible to equip all end-point devices with specialized hardware. At the same time, the timing accuracy of applications running on networked edge devices is degraded by asymmetric medium delays and imperfect oscillators. Other technology trends such as timing inconsistency in multi-core platforms and virtualization-induced timing uncertainty further degrade the timing accuracy provided to applications.

Operating systems play a key role in how time is managed and delivered to applications. In most operating systems, the notion of time is generally derived from the highest-quality system timer. Take the case of Linux: multiple clocks are derived from this timer and exposed to user-space via the standardized POSIX-clock interface. One important clock is CLOCK_REALTIME, which is disciplined by best-effort synchronization techniques such as NTP [Mil91] and PTP [LEW05]. However, existing OS timing abstractions continue to manage time in an application-agnostic fashion, i.e., independent of application timing requirements, and without providing any information about the delivered timing uncertainty.

This status quo in time management is highlighted in Figure 2.1 (Left), where clock synchronization is completely oblivious to timing requirements of App 1 and App 2, and always provides best-effort synchronization performance. The operating system returns the time estimate to applications with no information on how *off* this time estimate is from the true time, i.e., the uncertainty in time. We advocate for a new way of thinking in how time is managed in an OS. We close the loop – as shown in Figure 2.1 (Right) – through applications specifying their timing requirements, while the OS orchestrates the underlying platform to meet these requirements, and exposes the achieved timing uncertainty back to the applications. Thus providing applications the ability to adapt to changes in timing uncertainty.

Therefore, we introduce the notion of *Quality of Time (QoT)* as the end-to-end uncertainty in time desired by an application. Application requirements are written in terms of a *timing contract* specifying the desired timing metrics such as accuracy, stability, and integrity. QoT is the ability to guarantee a certain level of performance listed in a timing contract. This notion of QoT is inspired by the networking Quality of Service (QoS) requirement, where the system delivers the required QoS specified in terms of a traffic contract.

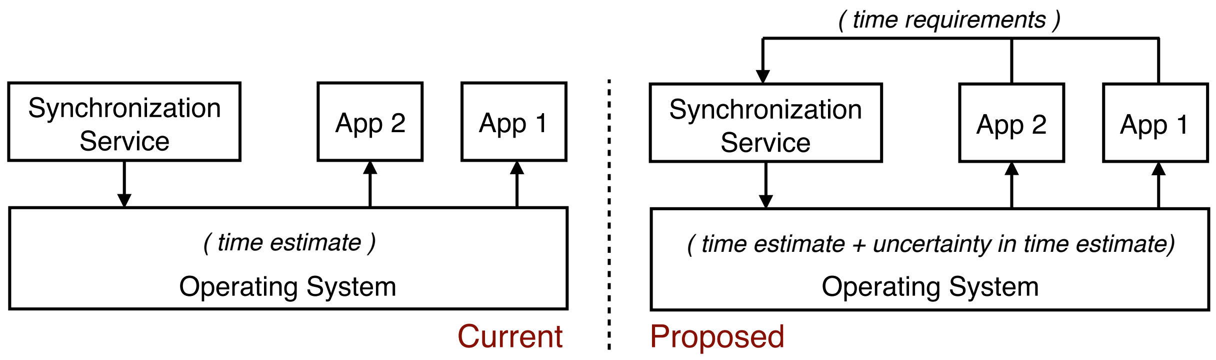


Figure 2.1: Contrast between time management in current operating systems and our proposed way for time management

With the emergence of the Internet of Things (IoT), scarce resources like energy, bandwidth, and processor cycles must be balanced with application QoT requirements. Existing clock-synchronization protocols synchronize the entire network to a common time reference. This wastes hardware and network resources on unwanted synchronization. We instead argue for *factored coordination*. Only those nodes which need to coordinate their tasks, synchronize their clocks. Factored Coordination enables subsets of coordinating nodes to synchronize their clocks to the desired QoT. To enable QoT as well as factored coordination in an OS, we propose a new OS abstraction, called a *timeline*. The timeline abstraction helps virtualize time-related resources in a system and plays a role analogous to sockets in network stacks. Just as QoS-aware applications can read, write, open and close sockets, and specify QoS parameters; QoT-aware applications can bind and unbind from timelines, read and schedule events on the timeline reference, and specify QoT requirements.

The timeline-driven QoT architecture closes the loop between the timing requirements of applications, and how well the system is able to meet their needs by propagating timing uncertainty back to the applications. It characterizes the timekeeping hardware capabilities e.g. oscillators and timestamping mechanisms and exposes them as controllable and disciplinable clocks. The system can adjust these clocks and/or switch between them to balance application needs with the system resources. For example, a system can switch from a high energy rubidium oscillator to an unstable low energy quartz crystal oscillator, from hardware timestamping to software timestamping, from a high synchronization rate to a low rate to balance system resources with application QoT requirements.

The primary contributions of our work are as follows:

- We propose the notion of Quality of Time based on timing uncertainty.
- We investigate how time-aware applications and the OS should exchange information about time and develop a model that describes how (i) applications interact with a shared notion of time, (ii) applications register their timing requirements with the OS and (iii) how timing uncertainty is conveyed from the OS to applications.
- We propose a platform-independent OS abstraction called a *timeline*, and present an application programming interface (API) that greatly simplifies the development of QoT-aware choreographed applications.
- We provide an end-to-end timeline-driven QoT architecture and its corresponding implementation for Linux.
- We conduct a series of micro-benchmarks to verify the performance of our QoT architecture on a Linux-based embedded platform, the Beaglebone Black [Bla].

2.2 Time Fundamentals

2.2.1 Clock Model

A hardware clock measures time by counting the number of cycles of a periodic signal. This periodic signal is typically a sinusoidal obtained from some oscillator, which is then multiplied or divided in hardware to oscillate close to some nominal frequency f_0. Consequently, a hardware clock's perception of time is inherently a discrete quantity.

Every free-running oscillator's output signal is perturbed by a frequency bias, which varies with time or environmental conditions, such as temperature, vibration or supply voltage. This error is described as the *frequency bias* of the oscillator, and quoted in *parts* *per million*, or $ppm = f_e/f_0 * 10^6$, where f_e is the worst-case upper-bound on frequency bias with respect to the nominal frequency. An advantage of this representation is that the stability of two oscillators can be compared independently of nominal frequency. For example, under well-specified operating conditions an oscillator with a 1ppm error accumulates no more than $1\mu s$ over a one second period, irrespective of its nominal frequency. Frequency stability is a combination of three key components: *warm-up*, *short term stability* and *aging*. The warm-up only applies to the initial period where the internal temperature of the crystal is rising asymptotically, and normally stabilizes within milliseconds for crystal oscillators, or minutes for cesium oscillators. Short term stability is a function of independent noise that and manifests as *jitter* in the time domain and accumulates to cause drift. *Aging* describes a long-term gradual shift in the oscillator's output frequency due to its electromechanical properties.

A consequence of frequency instability is that two free-running hardware counters – and hence the devices' relative clocks – will drift with respect to each other unless one of them is periodically *disciplined*. The process of disciplining clocks across a network is referred to as *time synchronization*.

2.2.2 Time Synchronization Basics

The error (ϵ) in an application's time at any instant is the difference between its local perception of time (t_{local}) and the true reference time (t_{global}), or equivalently $\epsilon = t_{local} - t_{global}$. In order to minimize this error, the local clock can be synchronized to a reference clock through message passing. However, the Round Trip Time (RTT) of a message varies over every message exchange as a result of current network conditions and how deterministically each participating device is able to time stamp incoming and outgoing messages. In addition, since each device's local clock is driven by an independent oscillator, its perception of time drifts with respect to the reference between synchronization rounds and as a function of both time and environmental conditions. If the frequency bias with respect to the reference – expressed in parts per million (*ppm*) – is known, then the reference time may be approximated at some future local time, say $(t^{'}_{local})$, using the following linear relationship: $\epsilon^{'} = \epsilon + (t^{'}_{local} - t_{local}) * (1 + ppm/10^6)$. The accuracy of projection of future time is both a function of how well $\epsilon^{'}$ corresponds to the actual offset of future local time from the global reference, and how well the *ppm* value approximates the actual frequency stability over the period. By maintaining a variance in the error estimate, synchronization uncertainty is captured.

2.2.3 Time in Linux

Operating systems also play a key role in how time is managed and delivered to applications. In most commodity operating systems, the notion of time is generally derived from the highest quality timer available on a system. Take the case of Linux: multiple virtual clocks are derived from such a single timer. These virtual clocks are implemented in a layered fashion using multiple abstractions such as **cyclecounters**, **timecounters** and **clocksources**, and expose themselves to userspace via the standardized POSIX clock [IEE] interface. This interface allows clocks to be disciplined, using synchronization algorithms such as NTP [Mil91] PTP [LEW05], which run as userspace daemons. The Linux kernel also allows users to precisely schedule events on any of these clocks' notion of time, using both absolute and relative blocking waits by use of the underlying High Resolution Timer [GN06] subsystem. However, the Linux timing system does not expose any notion of uncertainty to applications, nor does it allow applications to specify their timing requirements. The synchronization and scheduling systems always provide best effort performance. The same can be said for most operating systems. Hence, the applications are unaware of the error in their time estimate.

2.2.4 Time Stamp Determinism

Time synchronization algorithms typically estimate the relationship between two clocks by comparing the receive and transmit time stamps for the same message, and their accuracy is ultimately limited by various stochastic delays in the system:

1. **Propagation delay** - propagation distance divided by propagation velocity, which depends on medium. Note that this may not necessarily be symmetric because of different routes, multipath effects or cable properties.

2. **Transmit/Receive delay** - the delay between message time stamp and transmission.

3. **Receive delay** - the delay between message time stamp and reception.

4. **Residency delay** - only applicable to multi-hop networks, this is the duration that a packet spends in the buffer of an intermediate switch or router.

2.2.5 Time Synchronization Protocols

Time synchronization has been an important field of study, and a comprehensive software stacks with accompanying hardware are readily available. Examples of hardware include GPS and atomic clocks, switches that calculate residency delay for routed packets, and network adapters that precise time stamp a field of incoming and outgoing packets. Such infrastructure is already in use by many back-end systems. For example, cellular telephone backhaul networks use bespoke hardware for synchronizing transmissions to maximize channel utilization. Distributed database algorithms like Google Spanner [CDE13a] are already making use of this timing infrastructure

It is currently possible to synchronize to an error in the order of milliseconds with the Network Time Protocol (NTP) [Mil91] over Ethernet, or nanoseconds with the Precision Time Protocol (PTP) [LEW05] and compliant hardware. More specialized projects, such as WhiteRabbit [LWS11], attain sub-nanosecond error – enough to measure the distance light travels in a second with millimeter accuracy – by compensating for cable delay asymmetry and using Synchronous Ethernet to frequency-lock devices.

In the wireless sensor networking literature time synchronization has been approached in a different way. Rather than considering the objective to be to synchronize devices to some universal time reference, all that matters is that peer devices – which may be multiple hops away from each other – share a common sense of time, and with an emphasis on channel utilization efficiency. For the case where the root time is maintained across all nodes in the network, Flooding Time Synchronization Protocol (FTSP) [MKS04] is state of the art. It allows only one-way reference broadcasts as opposed to Timing-sync Protocol for Sensor Networks (TPSN) [S G03] and Reference Broadcast Synchronization (RBS) [J E02], which use two-way message exchange between nodes, thus effectively reducing the network traffic. All of these protocols implicitly assume that the distance between devices is sufficiently small enough (tens of meters or less) so that propagation delay can be ignored. Recently, Glossy [F F11] and PulseSync [C L14] have emerged and improved on the multi-hop accuracy of FTSP by flooding pulses at high speed throughout the network.

In recent literature, many time synchronization techniques use analytical modeling. Adaptive Clock Estimation and Synchronization (ACES) [HMZ08] model the clock directly. It applies Kalman filter to track the clock offset and skew. Based on these offset and skew estimations, it adaptively adjusts the synchronization interval to achieve desired error bounds. In [Ad07] and [Ble05] clock is synchronized over the packet switched network using Kalman filtering. But their assumption of constant clock skew over a long time is unrealistic for off the shelf unstable clocks. Xiali Li et al [LYL12] argue that sensor clock system switches between different models and modeled the clock drift using first and second order Kalman filters. Yi Zeng et al [Yi 08] considered taking offset readings from multiple parents and combining them using vector Kalman filter over a multi-hop network. What they did not consider is that multiple parents could also go out of sync and more message exchange cause an increase in communication bandwidth.

Clock synchronization techniques are also gaining momentum in distributed measurement and control systems. IEEE 1588-2008 transparent clock (TC) [Jih10] solves the problem of exponential accumulation of offset by measuring the residence time of messages in a network node. Thereof, this residence time is eliminated from the offset calculation. But it requires dedicated hardware for higher accuracy. Synchronization accuracy in nanosecond is achieved despite large queuing delays. In [Xu13] quantization effect in timestamping is taken into account. Previously, Seong et al. [C 10] compensated for this quantization error using feed forward filter preceding the PI controller. But [Xu13] uses a Kalman filter based proportional-integral (PI) clock servo to correct for this quantization error and clock offset in cascaded real time sensor networks. However, no comparison against an existing optimal PI controller has been given.

The notion of time uncertainty is not new to the field of time synchronization. NTP [Mil91] computes a bound on time for every timestamp sample and applies clock filtering algorithms to filter out the false samples. However, this bound is never exposed to an application and hence becomes invalid when a clock adjustment is made. Google Spanner [CDE13a] has shown how the knowledge of uncertainty in time, can be used to implement a TrueTime API in order to achieve external consistency of transactions in a global database. However, Spanner is a closed system and the TrueTime APIs are tailored only to database transactions. In contrast, our work seeks to provide a universal framework for making uncertainty of time observable and controllable to a broad range of time-aware applications, both at the edge and in the cloud. Outatime [LCC15] attains real-time interactivity in cloud gaming in the face of wide-area latency through speculative execution; they claim however that outatime can relax the stringent demands on speculation provided the uncertainty in their time is within 100msec.

At the programming level, time-triggered and event-triggered computation models provide timing determinism. The time-triggered architecture (TTA) [KB03] addresses issues in real-time programming by establishing a global time-base to specify interaction between nodes, whereas, an event-triggered architectures like Ptides [DFL08] maps model time to real time, only when systems react or act to the physical world, e.g, sensors and actuators. A key assumption that Ptides make is the execution times of software components are sufficiently small and can be ignored. However if execution times of software components are not sufficiently small, the design is infeasible and the deadlines specified by the Ptides model will not be met by the implementation [ZML12]. PtidyOS [ZML12] is a micro kernel for Ptides that generates target specific code for the Ptides model and runs on bare metal. Our stack operates on a different paradigm than Ptides. It focuses on assisting coordinated and distributed time-aware applications on a local as well as wide area, through a hardware independent system library as well as an entire framework built using commodity hardware/software.

2.2.5.1 Precision Time Protocol

Precision Time Protocol (PTP) [LEW05] is an IEEE 1588 compliant time synchronization protocol and it synchronizes clocks over a multicast capable network. It provides a best master clock algorithm that identifies the best clock in the network and choose it as a master. A slave clock synchronizes to the master clock using bidirectional communication. The master sends a 'Sync' packet along with the timestamp when the packet left the master node. The slave receives the 'Sync' packet and timestamps its arrival time. The slave determines its clock offset from the master by calculating the difference in 'Sync' packet's departure and arrival times, and adjusts its Network Interface Clock (*/dev/ptpX*) accordingly. The slave also compensates for the network propagation delay by exchanging 'delay request' and 'delay response' packets with the master.

2.3 Related Work

The notion of time uncertainty is not new to the field of time synchronization. NTP [Mil91] computes a bound on time for every timestamp sample and applies clock filtering algorithms to filter out the false samples. However, this bound is never exposed to applications and hence becomes invalid when a clock adjustment is made. Google Spanner [CDE13a] utilizes the True Time API to show how the knowledge of uncertainty in time can be used to achieve the external consistency of transactions in a global database. However, Spanner is a closed system and the TrueTime API are tailored only to database transactions. Additionally, both the TrueTime API and POSIX API do not treat the notion of time as an application-specified requirement. In contrast, our work seeks to provide a universal framework with associated timeline-based API that lets applications specify their QoT requirements, and also exposes the achieved QoT to applications for varied uses including coordination and adaptation. Outatime [LCC15] attains real-time interactivity in cloud gaming in the face of wide-area latency through speculative execution; they claim however that outatime can relax the stringent demands on speculation provided the uncertainty in their time is within 100msec.

At the programming level, time-triggered and event-triggered computation models provide timing determinism. The time-triggered architecture (TTA) [KB03] addresses issues in real-time programming by establishing a global time-base to specify interaction among nodes, whereas event-triggered architectures like Ptides [DFL08] map model time to real time, only when systems interact with the physical world, e.g. using sensors and actuators. PtidyOS [ZML12] is a microkernel that generates target specific code for the Ptides model. Our timeline-driven QoT architecture utilizes a different paradigm. It focuses on assisting coordinated and distributed time-aware applications on local as well as wide area networks, through a hardware-independent system library as well as an entire framework built using commodity hardware and software.

Also relevant and complementary to our work is research in time synchronization on the analytical modeling of clock uncertainties [HMZ08], and methods to compensate for them via approaches such as Kalman filtering [Xu13].

2.4 Timeline

A timeline is a virtual time base with respect to an epoch. Unlike other time bases, a timeline's notion of time is not necessarily tied to any specific reference device or time system. Timeline attributes are binding accuracy (μ), and binding resolution (r). A node[1] binds to a timeline with the desired μ and r. μ is the maximum error a node can tolerate in its time estimate from the true time. We represent μ as an asymmetric interval around true time. r – also called the timeline tick – is the minimum time increment on the timeline. r actually corresponds to the nominal or scaled hardware clock frequency. We represent both μ and r as a $\{second,\ attosecond\}$ tuple. The choice of μ and r for a timeline affects

[1] A 'node' can be a device, or an application thread on a device

the consumption of resources in a system. Higher μ and r necessitates the use of a clock with high resolution and stability, which costs energy. A proper balance between μ and r helps the system deliver the required QoT to applications, as well as optimize resource usage like energy and network bandwidth.

Each timeline maintains a virtual clock. Hence, a device can maintain multiple timelines; making use of different time bases for different applications running on the device. On the other hand, a single timeline can be shared among coordinating nodes who wish to synchronize their time. This concept is illustrated in Figure 2.2 (Left). Multiple applications running on a device may have different QoT requirements. The timeline abstraction enables the OS to provide as many disciplinable clocks as needed by the applications.

The timeline abstraction also assists developers to easily implement coordinated applications. Nodes that need to coordinate their tasks, bind to a common timeline and synchronize their time. We argue that (i) clocks should be synchronized only with the desired QoT, and (ii) only clocks of coordinating nodes should be synchronized. We represent the coordinating nodes and their respective timelines as a factor graph in Figure 2.2 (*Right*). As depicted in this figure, only those nodes which bind to a common timeline, synchronize their time, hence reducing the number of packets exchanged and the usage of timing-related hardware resources.

We design a complete architecture around the timeline abstraction, which is generalizable to any operating system. The QoT Architecture centered around timelines is described in the subsequent section.

2.5 QoT Architecture

The QoT Architecture uses the timeline abstraction along with the associated notion of QoT to make uncertainty in time observable to distributed applications. Our architecture supports multiple timelines, enabling different coordinating sub-groups with varying QoT requirements to co-exist on the same system.

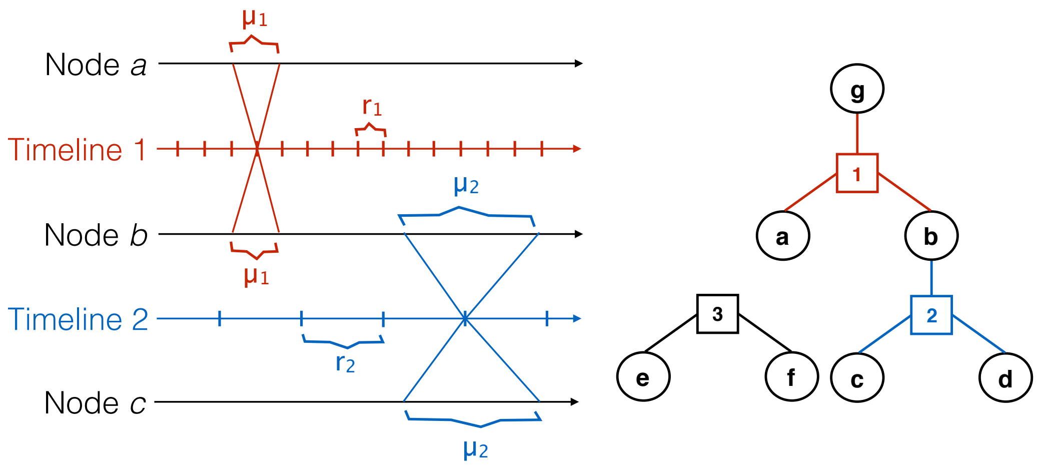


Figure 2.2: (*Left*) Node a and b bind to Timeline 1 with a desired binding accuracy μ_1, and binding resolution r_1. Node b and c bind to Timeline 2 with μ_2, and r_2. Note that the accuracy requirement of nodes on Timeline 1 is tighter than the accuracy requirement of nodes on Timeline 2 ($\mu_1 < \mu 2$). Also note that Node b binds to both Timeline 1 and Timeline 2. This shows that the timeline abstraction is flexible enough to be shared by multiple nodes, and at the same time, a node can maintain multiple timelines as well. (*Right*) Factor graph representation of Nodes and Timelines from *Left* figure. Rather than synchronizing clocks of all nodes $\{a, b, c, d, e, f, g\}$ in the network, only coordinating nodes synchronize their clocks; $\{a, b, g\}$ through Timeline 1, $\{b, c, d\}$ through Timeline 2, and $\{e, f\}$ through Timeline 3.

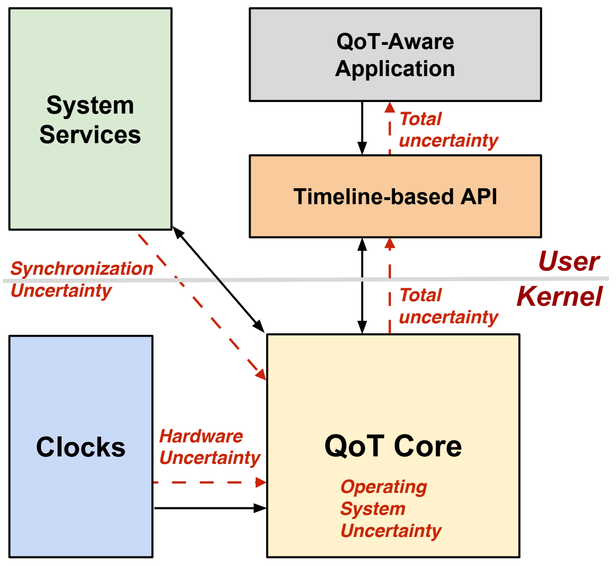


Figure 2.3: Timeline-driven Quality-of-Time Architecture

We now present the key components of the QoT Architecture as shown in Figure 2.3. It is comprised of three distinct components: (i) *Clocks*, (ii) *System Services*, and (iii) *QoT Core*. In this section, we describe the individual components of this architecture and show how they interact with each other. We believe that the core concepts of our architecture are applicable to any platform. However, the range of platforms supported by Linux make it an ideal candidate for prototyping our architecture. Our Linux implementation is called the QoT Stack for Linux, and focuses on prototyping essential functionality for embedded Beaglebone Black (BBB) [Bla] nodes connected over a Local-Area Network. The stack consists of kernel modules and system services, and does not modify the Linux kernel, ensuring portability across different kernel versions. Additionally, our implementation separates hardware-specific and hardware-independent code, providing ease of portability across platforms.

2.5.1 Clocks

The QoT architecture exposes timekeeping hardware as Clocks, which play a major role in delivering knowledge of time with associated QoT to the applications. Based on the functionality provided, we categorize them into two types:

Core Clocks drive all the functionality of the stack. All timelines derive their reference time as a projection from a core clock. For a clock to qualify as a core clock, it must provide (i) the ability to read a *strictly-monotonic* counter, which cannot be altered by any system process, (ii) the ability to schedule events along a timeline reference, and (iii) provide the hardware resolution and uncertainty associated with reading the clock. Optionally, a core clock may also expose interfaces to timestamp and generate external events.

Network Interface Clocks (NICs) assist in disciplining the local time to some global reference time. Only those network interfaces which have the ability to accurately timestamp network packet transmission and reception, at the physical layer, are exposed as NICs. This enables precise calculation of the offset between two clocks, and the propagation delay associated with a medium. A NIC is similar to a core clock in providing the ability to read time, and optionally provide I/O functionality for precisely timestamping an event, or generating a very deterministic pulse in the future. A NIC, however, differs from a core clock in that: (i) it is not necessarily monotonic, (ii) it may be disciplinable and (iii) it does not provide the ability to generate interrupts. Hence, it cannot be used to accurately schedule user-level application threads.

Our architecture supports these two clock categories, and provides mechanisms to synchronize them with each other. While every node must contain a core clock, it is not necessary for a node to contain a NIC.

2.5.2 System Services

System services are user-space processes responsible for distributing timeline metadata, quantifying timing uncertainties, and synchronizing time within and across nodes.

2.5.2.1 Data Distribution Service (DDS)

DDS [Ope] is a networking middleware service which simplifies networking programming for our architecture. It provides a publish-subscribe framework, which collects all the timeline requirements and gives participating nodes the ability to decide the reference time in a decentralized fashion.

2.5.2.2 Synchronization Service

Modern OSs expose only a single clock and synchronize it on a best-effort basis by being oblivious to the application requirements. In contrast, our timeline-driven architecture supports multiple timelines on a single node, each having its own notion of time. In Figure 2.2 (Right), node c is part of both Timelines 1 and 2. Hence, it runs two parallel instances of the synchronization service for two different timelines, each achieving only the desired accuracy.

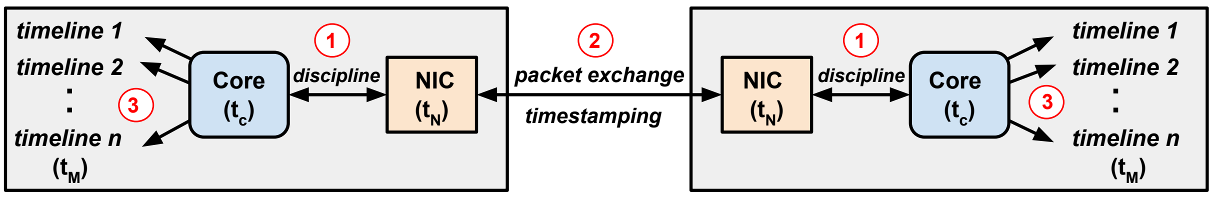


Figure 2.4: End-to-end time synchronization using timelines. (1) NIC is disciplined by the core clock on a single node ($t_N = t_C$). (2) NICs on two nodes exchange packets and timestamp them in core time. (3) The timestamps are used to work out the Core – Core mapping (t_M) and are stored in the form of a logical timeline mapping.

A timeline represents a mapping from a *local* core time to a *global* reference time. To generate this mapping we require a two-step synchronization procedure as shown in Figure 2.4. In the first step, we synchronize the NIC to the core clock on a single node. Once the NIC is aligned to core time all timestamps provided by a NIC can be mapped to core time.

In the second step, we perform *inter-node* synchronization. NICs exchange synchronization packets across nodes and timestamp them in core time, thus generating a timeline mapping using a first-order linear model, $t_M = t'_M + (1 + ppb/10^9) * (t_c - t'_c)$, where t_M is the current timeline mapping that is derived from previous mapping t'_M, frequency bias ppb (parts per billion) and current and previous core time, t_c and t'_c respectively. In Figure 2.4, multiple timelines are maintained as logical mappings and they provide the ability to synchronize with multiple nodes with totally different accuracy requirements, thus enabling the factored coordination paradigm. The Linux PTP Project [Pro] also runs two-step synchronization. It first aligns NICs across different nodes, and then it synchronizes the system clock to the NIC. However, this approach does not scale to multiple timelines. Hence, we keep the core clock strictly monotonic and maintain multiple timelines as logical mappings from the core clock.

Synchronization Uncertainty: Time synchronization performance is limited by various stochastic delays in the system: *Propagation delay*, *Transmit delay*, *Receive delay* and *Residency delay*. These delays introduce uncertainty in our time measurements. Using a statistical approach we can calculate the upper bound on synchronization uncertainty as, $E^U = \{(ppb_m - ppb + \Delta ppb)/10^9\} * (t_c - t'_c) + (e_m + \Delta e)$, and the lower bound on uncertainty,

$E^L = \{(ppb - ppb_m + \Delta ppb)/10^9\} * (t_c - t_c^{'}) + (e_m - \Delta e)$, where ppb_m is the mean bias, Δppb is the standard deviation of bias, e_m is the mean offset and Δe is the standard deviation of offset. Instead of using only local statistical information to calculate the bounds, network wide information using a Kalman filter can also be applied to calculate tighter uncertainty bounds. Whichever model is used for the uncertainty bound calculation, a global time estimate at any point in time should be $t_M - E^L < t_M < t_M + E^U$.

2.5.2.3 System Uncertainty Estimation Service

Every timestamp read by a user application contains an uncertainty value introduced by the OS, which is a function of factors like the system load and CPU operating frequency. This service continuously updates these uncertainty statistics and passes it to the the stack. These uncertainty values are appended to every timestamp as an uncertainty bound. The overhead of the OS contributes significantly to the end-to-end uncertainty associated with reading a clock.

2.5.3 QoT Core

The QoT Core (also referred to as the *core*) acts as a bridge between all the stack components, and the host OS. The core performs a range of functionality:

Timeline Management: To satisfy different QoT requirements, the core keeps track of different timelines and their associated bindings, and handles their creation and destruction. It also provides an interface for applications to bind to a timeline and specify their QoT requirements.

Clock Management: The core provides an interface for different hardware clocks to register with it, and exposes an interface for a *privileged* user to choose and switch between these different clocks. The core utilizes this chosen clock to maintain a monotonic sense of time, referred to as *core time*. The key idea is that a privileged daemon should be able to automatically select the core clock in a manner that balances clock stability/resolution with energy consumption. The core also maintains the projection parameters from the core clock to each timeline reference, and provides an interface for the synchronization service to manipulate them.

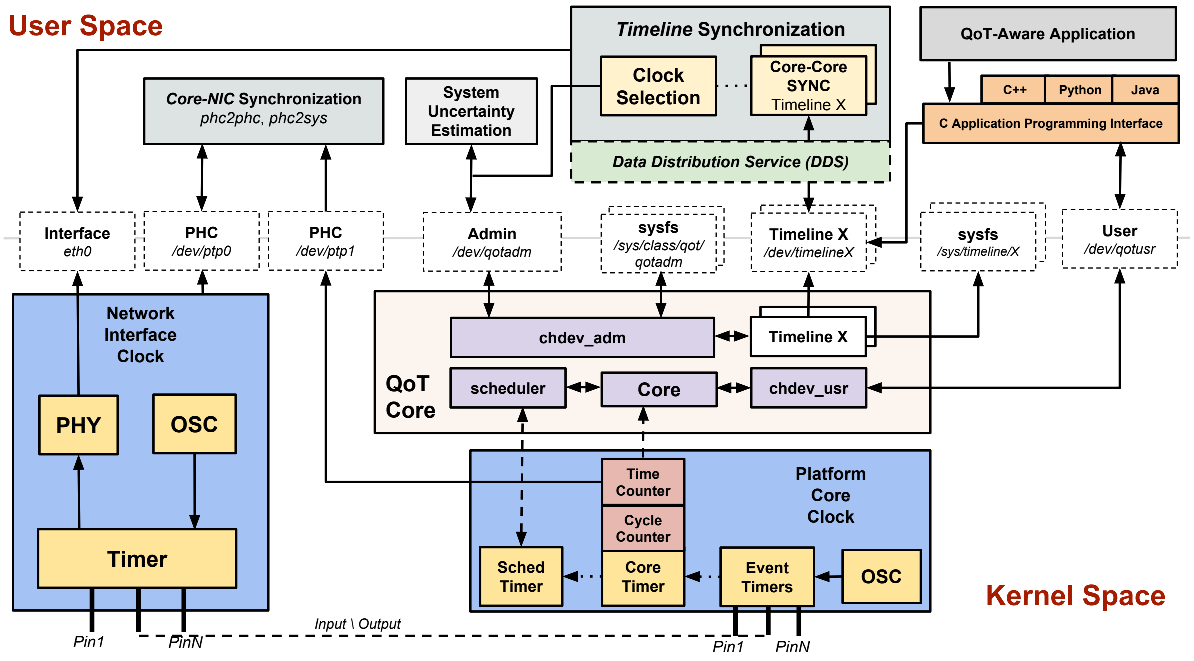


Figure 2.5: The QoT Stack for Linux

Event Scheduling: Scheduling an application on a global notion of time is important to execute distributed tasks synchronously. Hence, the core provides applications the ability to *synchronously* schedule events based on a timeline reference. The core provides this functionality in the form of *timed waits* by interfacing with the OS scheduler. Timed waits provide threads the ability to sleep for a relative duration or until an absolute time. The scheduling subsystem is also designed to dynamically compensate for any synchronization changes to a timeline reference.

QoT Propagation: One of the goals of the QoT stack is to expose the timing uncertainty to applications, so that the framework gives the current estimate of time, along with the uncertainty associated with it. As shown in Figure 2.3, the core propagates the uncertainties from different stack components, appending uncertainty to every time estimate. It also provides interfaces for the system services and the hardware clock to expose/update these uncertainty values.

2.6 QoT Stack for Linux

To demonstrate the possibility of our QoT architecture, we have developed a timeline-driven QoT Stack for Linux. Given the variety of supported software and hardware, Linux is an ideal candidate to prototype a cross-platform timeline abstraction. Since, every system has its unique timing limitations, we attempt to quantify and work with them, instead of forcing the use of a particular platform.A complete architectural diagram of our QoT Stack for Linux appears in Figure 2.5. We adopt a modular design to avoid requiring changes to the Linux kernel, instead relying on loadable kernel modules and userspace daemons to make Linux QoT-aware.

2.6.1 Timelines in Linux

Our Linux-based prototype implements a timeline as a `/dev/timelineX` character device, where X corresponds to a unique identifier. The character device exposes the *timeline reference* as a POSIX clock [IEE] to userspace, which is disciplinable by a synchronization service. The timeline character device also exposes an Input-Output Control (`ioctl`) interface, for applications to bind/unbind to a timeline, specify/update their QoT requirements, and read the timeline's reference time with an uncertainty estimate. In the Linux kernel, timelines are stored and ordered on a red-black tree which provides an $O(log(N))$ look-up time with a string identifier.

2.6.2 Clocks

Clocks (shown as *Network Interface Clock* and *Platform Core Clock* in Figure 2.5) are managed via drivers and use the Linux **`ptp_clock`** libraries to abstract away from architecture-specific sources. This abstraction provides the ability to enable or disable the clock source, configure timer pins (for timestamping inputs or pulse-width modulated outputs) and discipline the external clock (either in hardware or software). Pins are configured through the hardware timer subsystem using **`.enable`** and **`.verify`** function callbacks. The time can be observed or set through **`.gettime64`** and **`.settime64`** function callbacks. The kernel drivers implement the correct function callbacks, and register the existence of the precise clock through **`ptp_clock_register`**, with the kernel's PTP subsystem. In PTP terminology, these clocks are referred to as *Precise Hardware Clocks* (PHC), which is any clock or network interface that supports hardware timestamping and GPIO capabilities. These clocks are exposed to userspace as /dev/ptpX character devices and they register their capabilities, uncertainties, resolution and function hooks with the core module.

2.6.3 System Services

Data Distribution Service: OpenSplice [Ope] is used as the data distribution service in our stack. It disseminates timeline metadata across the entire network. Once every node has a complete picture of *timelines* on all nodes, they compete for providing the reference time and in our baseline implementation, the node with the highest accuracy requirement is chosen to provide the reference time to the timeline's subgroup. The synchronization rate is determined by the highest accuracy requirement in the network. Hence, the node which has the highest requirement in its timing subgroup can become a master and push packets with a rate corresponding to its accuracy requirement.

Synchronization Service: The synchronization service operates in userspace and comprises of *Core-NIC Synchronization* and *Timeline Synchronization* daemons, as shown in Figure 2.5. The Timeline Synchronization daemon is implemented by patching the Linux PTP Project [Pro]. It calculates clock discipline parameters, and disciplines the `/dev/timelineX` character devices through the **`.settime`** and **`.adjtime`** POSIX clock APIs. The mappings are stored in the kernel so that the timeline reference can be easily returned using the **`.gettime`** POSIX clock API.

We also create a synchronization service *phc2phc* that aligns two *Precise Hardware Clocks* (PHC): clocks which support hardware timestamping and GPIO with external hardware timestamping, and deterministic hardware interrupt capabilities. Our implementation performs *Core-NIC* synchronization using *phc2phc*. If one of the clock is not a PHC, we

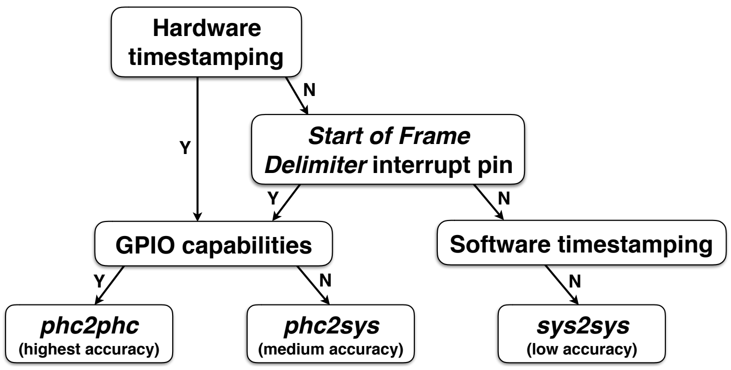

Hardware timestamping
Y
N
Start of Frame Delimiter interrupt pin
Y
N
GPIO capabilities
Software timestamping
Y
N
N
phc2phc (highest accuracy)
phc2sys (medium accuracy)
sys2sys (low accuracy)

Figure 2.6: Decision tree for choosing a time synchronization service based on hardware capabilities

use the *phc2sys* [Pro] service to synchronize clocks. The decision tree in Figure 2.6 shows how timestamping and GPIO capabilities of a clock influence our choice of synchronization service. Certain network interfaces do not support hardware timestamping, but provide a hardware interrupt upon the Start of Frame Delimiter (SFD) of a synchronization packet. In this case, if the core clock is a PHC, it can timestamp the SFD interrupt in hardware and run *phc2phc* across multiple nodes for high accuracy. However, certain network interfaces neither expose a PHC, nor support SFD. In this case, the core clocks resort to software time stamping and perform *sys2sys*. Table 2.1 lists some example network interfaces with different hardware capabilities and the corresponding synchronization service.

Table 2.1: Network Interface Capabilities

NIC	**Capabilities**	**Service**
TI CPSW	PHC, GPIO interrupt	*phc2phc*
AT86RF233	PHC, SFD interrupt	*phc2phc*
DW1000	PHC, SFD interrupt	*phc2phc*
IEEE 802.11	None	*sys2sys*

System Uncertainty Estimation Service: This service tries to get a probabilistic estimate of the OS clock read uncertainty by reading the core clock in a tight loop from userspace, via a privileged interface (`/dev/qotadm`). By taking the difference of consecutive timestamps, the service calculates the uncertainty distribution.

2.6.4 Linux QoT Core Kernel Module

The Linux QoT Core, shown as the central component in Figure 2.5, is implemented as a loadable kernel module. It consists of the following sub-modules.

Scheduler Interface: Each active timeline maintains a red-black tree of waiting threads, ordered by their wake-up times in the timeline reference. When an application thread issues a timed wait request, the thread is suspended and en-queued on a red-black tree corresponding to the timeline to which it is bound. Waking up applications from their suspended state relies on the interrupt functionality of the core clock. When the callback triggers, the interrupt handler checks each active timeline for tasks that need to be woken up, and moves such tasks from the wait queue to the ready queue. Subsequently, the task is scheduled as per its priority, and the policy being used by the scheduler. This introduces scheduling uncertainty, as other threads may also be present on the ready queue. Before the task is actually scheduled, the core returns a timestamp of the scheduling instant along with an uncertainty estimate. This enables an application to take a decision, based on the received QoT. The scheduling policy *agnostic* design, enables the stack to be portable to a range of different Linux kernels, and prevents it from being tied down to a specific kernel version. It also gives the opportunity for OS developers to use scheduling policies best suited for the target platform. Future implementations of the stack will include techniques to probabilistically compensate for the scheduling uncertainty.

Decisions on waking up a task, or programming the next interrupt callback, rely on the projections between core time and the timeline references. The scheduling interface compensates for any synchronization changes to these projections. When a synchronization event occurs, the interface checks the head of the timeline queue, to decide whether the change in the projection, necessitates a task to be scheduled earlier than previously estimated.

User Interface: The core exposes a set of thread-safe `ioctl` interfaces in the form of a character device, `/dev/qotusr`, to userspace. It gives user applications the ability to create/destroy a timeline, read timestamps with uncertainty estimates, as well as issue timed waits on a timeline reference. The user interface also provides applications the ability to access the external timestamping and event triggering functionality of the core clock (if supported by hardware).

Admin Interface: This is a special character device `/dev/qotadm`, which enables a privileged daemon to control specific parameters of the QoT stack. It provides an `ioctl` interface, which allows a privileged user to get information on clocks, switch between different core clocks, as well as get/set the OS uncertainty associated with reading timestamps.

Sysfs Introspection: The core provides a `sysfs` interface for a user to view and change the state of the system using file operations. It can be used to develop complex visualization tools or to integrate with existing monitoring systems.

On the Beaglebone Black platform, the memory footprint of the QoT Stack for Linux is 4.071 MB. The current implementation re-implements a number of existing components for easier debugging, leading to a large code size. Future implementations will focus on optimization.

2.7 Application Programming Interface

We provide an API that allows programmers to easily develop coordinated distributed applications shown in Table 2.2.

It is essential to have a core set of APIs that are independent of the platform and OS. At the same time, our APIs are extensible to support platform-specific extensions. Our APIs are based on the timeline abstraction and support coordination through timelines. The APIs enable applications to (i) bind/unbind from a timeline, (ii) specify/update their QoT requirements, (iii) schedule sensing, computation, and actuation based on a shared notion of time, (iv) timestamp events, and (v) exchange messages using the publish-subscribe framework. All our API calls return the achieved QoT, providing applications the ability to adapt to changes in their QoT.

Table 2.2: Quality of Time APIs

Category	API	Return Type	Functionality
Timeline Association	**timeline_bind** (name, accuracy, resolution)	timeline	Bind to a timeline
	timeline_unbind (timeline)	status	Unbind from a timeline
	timeline_getaccuracy (timeline)	accuracy	Get binding accuracy
	timeline_getresolution (timeline)	resolution	Get Binding resolution
	timeline_setaccuracy (timeline, accuracy)	status	Set Binding accuracy
	timeline_setresolution (timeline, resolution)	status	Set Binding resolution
Time Management	**timeline_gettime** (timeline)	uncertain_timestamp	Get timeline reference time with uncertainty
	timeline_getcoretime ()	uncertain_timestamp	Get core time with uncertainty
	timeline_core2rem (timeline, core_time)	uncertain_timestamp	Convert a core timestamp to a timeline reference
	timeline_rem2core (timeline, time)	uncertain_timestamp	Convert a timeline reference timestamp to core time
Event Scheduling	**timeline_waituntil** (timeline, absolute_time)	uncertain_timestamp	Absolute timed wait
	timeline_sleep (timeline, interval)	uncertain_timestamp	Relative timed wait
	timeline_setschedparams (timeline, period, start_offset)	status	Set period and start offset
	timeline_waituntil_nextperiod (timeline)	uncertain_timestamp	Absolute timed wait until next period
	timeline_timer_create (timeline, period, start_offset, count, callback)	timer	Register a periodic callback
	timeline_timer_cancel (timer)	status	Cancel a periodic callback
	timeline_config_events (timeline, event_type, event_config, enable, callback)	status	Configure events/external timestamping on a pin

To demonstrate the use of APIs, we first present a code snippet for a *Time Division Multiple Access (TDMA)* application in Listing 2.1, and another code example of a simplistic distributed-control application in Listing 2.3.

Listing 2.1: QoT-Aware TDMA Application

```
name = "tdma-timeline"; /* Timeline UUID */
/* Timeline accuracy equivalent to TDMA guard band */
timeinterval_t accuracy = {
  .below = TL_FROM_nSEC(0),
  .above = TL_FROM_nSEC(TDMA_GUARD_BAND),
};
/* Timeline resolution equivalent to TDMA period */
timelength_t res = TL_FROM_nSEC(TDMA_PERIOD);
timelength_t period = TL_FROM_nSEC(TDMA_PERIOD);
timepoint_t start_offset
  = TP_FROM_nSEC(get_my_slot()*TDMA_SLOT_LENGTH);
/* Bind to a timeline with requested UUID */
timeline_t timeline = timeline_bind(name, accuracy, res);
/* Set period and start offset */
timeline_setschedparams(timeline, period, start_offset);
/* Periodic TDMA Transmission */
while(tdma_running) {
  /* Sleep until start of next transmit slot */
  status = timeline_waituntil_nextperiod(timeline);
  if(status == QOT_OK) {
    /* Transmit if uncertainity within bound */
```

```
        transmit_packet(message);
    } else {
        hold_off();
    }
}
timeline_unbind(timeline); /* Unbind from a timeline */
```

The TDMA application needs to ensure that each node transmits in its own time slot to avoid packet collisions. This implies that all nodes must have a shared time base, along with the knowledge of associated uncertainty. Traditionally, TDMA application compensates for timing uncertainty using guard bands. If this uncertainty increases beyond these guard bands, packet collisions will occur. Our method of delivering the achieved QoT to the TDMA application gives it the ability to take a decision to transmit a packet or not, and avoid collisions.

For the TDMA application, the nodes first bind to a timeline, along with specifying their desired accuracy and resolution using **timeline_bind**. Transmitting in a TDMA slot is inherently periodic, therefore, the application can set its period and start offset using **timeline_setschedparams**. Subsequently, the application periodically calls **timeline_waituntil_nextperiod**, which wakes up the task using the programmed parameters. This call also returns the achieved QoT. The application can utilize this information to decide on packet transmission. Finally, before termination, the application unbinds from the timeline using **timeline_unbind**.

We also contrast our QoT-aware TDMA application with one written using the Linux POSIX API. The Linux API based TDMA application does not have a notion of QoT and cannot provide end-to-end estimates on timing uncertainty. The application computes its wake up time every period, and uses the **clock_nanosleep** system call to wake up and transmit a packet. If the timing uncertainty exceeds the guard bands, then packets will collide. On the other hand, a QoT-aware TDMA application can take a decision on packet transmission based on the returned QoT.

Alternatively, a developer may create a daemon to compute the end-to-end timing uncertainty. However, this involves significant effort, complex interactions with existing timing systems, and privileged system access. Given that this functionality is commonly required across a range of applications, our stack provides it as a system service.

Listing 2.2: TDMA Application using Linux API

```
clock_gettime(CLOCK_REALTIME, now); /* Get time */
/* Current TDMA cycle number and start time */
int tdma_cycle_no = timespec2ns(now)/TDMA_PERIOD;
uint64_t cycle_start_ns = tdma_cycle_no*TDMA_PERIOD;
/* Slot time offset from start of cycle */
uint64_t slot_offset = get_my_slot()*TDMA_SLOT_LENGTH;
uint64_t transmit_time_ns = cycle_start_ns + slot_offset;
/* Periodic TDMA Transmission */
while (tdma_running) {
  /* Time of next transmission */
  transmit_time_ns = transmit_time_ns + TDMA_PERIOD;
  /* Sleep till transmit slot. Also handle signals */
  while(clock_nanosleep(CLOCK_REALTIME, ns2timespec(transmit_time_ns) == EINTR);
  clock_gettime(CLOCK_REALTIME, now); /* Get time */
  /* Transmit if wakeup time within bound */
  /* Packets may collide due to bad sync or clock */
  if(timespec_compare(now, transmit_time_ns + TDMA_GUARD_BAND) <= 0 ) {
    transmit_packet(message);
  } else {
    hold_off();
  }
}
```

We now demonstrate the use of our API through a second example. Control theory assumes that sensing and actuation happens at the same time. This assumption however does not hold in distributed control systems due to time delays, jitter and uncertainties. The clocks at distributed sensors and actuators should be synchronized and controller takes decision based on the synchronization uncertainty in sensing timestamps and the actuation signals. Control applications are sensitive to timing uncertainty, and sensor timestamps with uncertainty exceeding tolerable limits can lead to controller instability, or incorrect actuation. Therefore, if the exposed uncertainty from our API exceeds the threshold, the controller discards those values to avoid controller instability, or incorrect actuation.

Our application consists of a sensor node and an actuator node. Both nodes first bind to a timeline, along with specifying their desired accuracy and resolution using **`timeline_bind`**. When the sensor generates a value with a timestamp (containing its associated uncertainty), the sensor node then publishes this information, using **`timeline_publish`**. The actuator node subscribes to all messages on a timeline, using **`timeline_subscribe`**, and waits for a message (sensor value), using **`timeline_waituntil_message`**. When a sensor value is received, the actuator node checks the uncertainty of the timestamp, along with the uncertainty in its own knowledge of time. If the uncertainty is within tolerable limits, the actuation node runs the controller, and performs the required actuation. If the uncertainty exceeds tolerable limits the application can stop. Alternatively, the application may decide to switch to a more robust controller. Finally, before termination, the application components unbind from the timeline using **`timeline_unbind`**.

Listing 2.3: QoT-Aware Distributed Control

```
/* Sensor Code Snippet */
name = "control-timeline"; /* Timeline UUID */
/* Application-desired Timeline Accuracy */
accuracy = REQUIRED_ACCURACY;
/* Application-desired Timeline Resolution */
resolution = REQUIRED_RESOLUTION;
/* Bind to a timeline with requested UUID */
timeline_t timeline = timeline_bind(name, accuracy, res);
while(running) {
 /* Wait for sensor to write a value */
 waituntil_sensor_ready(&sensor_val, &timestamp);
 /* Publish sensor val with timestamp and uncertainty */
 timeline_publish(timeline, sensor_val, timestamp);
}
timeline_unbind(timeline); /* Unbind from a timeline */

/* Actuator Code Snippet */
name = "control-timeline"; /* Timeline UUID */
/* Application-desired Timeline Accuracy */
accuracy = REQUIRED_ACCURACY;
/* Application-desired Timeline Resolution */
resolution = REQUIRED_RESOLUTION;
/* Bind to a timeline with requested UUID */
timeline_t timeline = timeline_bind(name, accuracy, res);
/* Subscribe to Messages on Timeline */
timeline_subscribe(timeline);
```

```
while(running) {
 /* Wait for arrival of a message */
 waituntil_timeline_message(&sensor_val, &timestamp);
 /* Run controller using sensor values and timestamps */
 if(timestamp.uncertainty < REQUIRED_UNCERTAINTY && timeline.uncertainty <
    REQUIRED_UNCERTAINTY) {
   run_controller(sensor_value, timestamp);
   perform_actuation();
 } else {
   break; // If uncertainty exceeds requirement
 }
}
timeline_unbind(timeline); /* Unbind from a timeline */
```

2.8 Experimental Evaluation

Our prototype stack provides hardware support for the popular Beaglebone Black (BBB) embedded Linux platform [Bla]. We implement drivers to support the Texas Instruments (TI) AM335x ARM Cortex-A8 System-on-Chip (SoC) found on the BBB. The SoC supports the IEEE 1588 standard [LEW05] (Precision Time Protocol) over Ethernet, and has the ability to timestamp network packets at the physical layer. The drivers serve as a reference implementation, and provide core concepts which can be ported to a variety of platforms. Corresponding to the two types of clocks that we defined in Section 2.5.1, we have implemented platform-specific drivers as follows.

Beaglebone Black QoT Clock Drivers To support the QoT stack functionality, the following clock drivers were implemented for the Beaglebone Black platform:

TI CPSW Network Interface Clock: The Linux kernel already ships with TI's Common Platform Ethernet Switch (CPSW) Drivers (which also supports the AM335x SoC), which can be found in the kernel at **Linux/drivers/net/ethernet/ti**. The Common Platform Time Stamping (CPTS) module inside the CPSW ethernet subsystem is used to facilitate host control of time synchronization related operations. CPTS supports ethernet receive events, ethernet transmit events, and hardware and software timestamp push events. By default, hardware timestamp push events are disabled in the CPTS module, so we patched it and enabled time stamp inputs (HW1/4_TS_PUSH) to load the timestamp push events into the FIFO. These time stamp inputs can be triggered from Timers 4-7 of the AM335X. This setup enables NIC to Core synchronization (described in Section 2.5.2) with high accuracy.

BBB-AM335x Core Clock: This driver makes use of the on board dual mode timers (**dmtimers**) to provide various core clock related functionality. The AM335x contains 7 timers. All of them can be driven by on board oscillators, and some provide external clocking ability. Timers 1 and 2 are already used by the Linux kernel, hence, we use timers 3-7 to demonstrate the range of functionalities of a core clock. All the timers are clocked by an onboard 24 MHz crystal. The timers and their corresponding function are as follows:

- **Timer 3:** Drives the monotonic core clock.
- **Timer 4:** Enables scheduling, by providing the ability to trigger interrupts in the future.
- **Timer 5:** Generates a precise Pulse-per-second (PPS) which is used to discipline the Network Interface Clock.
- **Timer 6:** Provides the ability to timestamp external events on a pin.
- **Timer 7:** Provides the ability to generate a periodic output on a pin.

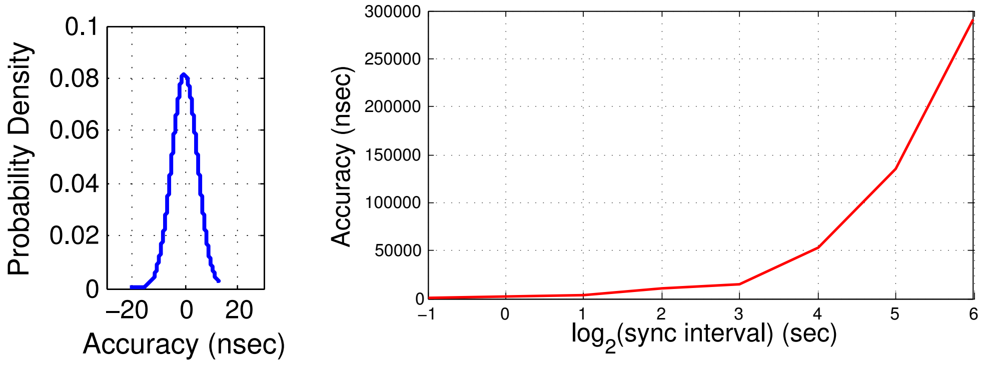
In order to deliver the functionality of timers 5-7 the system needs to configure the GPIO pins. For ARM based platforms, this is done by using a device tree. A device tree enables a user to configure the peripherals of an embedded ARM platform at run-time. Our prototype stack also provides a device tree for the Beaglebone Black platform.

Our testbed comprises multiple BBB nodes, with the Linux 4.1.12-rt kernel, connected via an IEEE 1588-compliant switch [RSG], running the *synchronization service* which is a patched version of the Linux PTP Project [Pro] and Network Time Protocol (NTP) [Mil91] implementation *ntpv4*[2] synchronization services, and term them as *ptp-qot* and

[2]http://ntp.org

ntp-qot respectively. Instead of disciplining the ethernet controller's NIC on the node (`/dev/ptp0`) or *ntpv4* that synchronizes CLOCK_REALTIME, *ptp-qot* and *ntp-qot* support simultaneous QoT-based synchronization of multiple timelines (`/dev/timelineX`). Using this testbed, we now present multiple micro-benchmarks which demonstrate the ability of our stack to perform synchronization, expose uncertainty and perform *choreographed* scheduling.

2.8.1 Synchronization Uncertainty

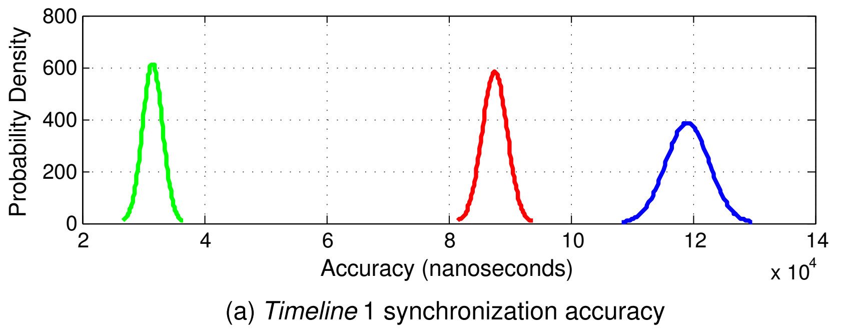


(a) Core-NIC (b) Accuracy v/s Synchronization Interval

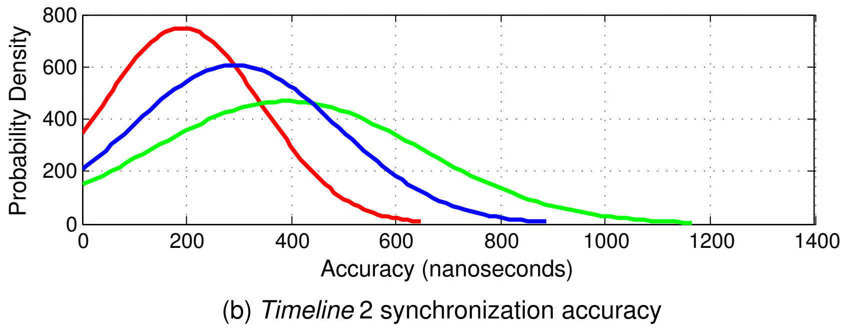
Figure 2.7: (a) Core-NIC synchronization accuracy (b) Illustrating the adjustable synchronization parameter

The prerequisite for end-to-end synchronization – mapping *local* core time to a *global* timeline reference – is to first synchronize the on-board NIC with the local core clock. We use a programmable hardware timer on the BBB AM335x to trigger very deterministic and periodic outputs on a pin in core time, which is then timestamped by the NIC. The difference between the core and NIC timestamps is used to work out the clock disciplining parameters. We plot the distribution of this difference in Figure 2.7a, which indicates the Core-NIC synchronization accuracy, which is in the order of nanoseconds. A similar approach can work on other hardware platforms as well.

The ability to control the synchronization accuracy, is a key goal of the QoT stack. We use the transmission rate of synchronization packets as a control knob to adjust the accuracy, and the resulting plot is shown in Figure 2.7b. Note that increasing the



(a) *Timeline* 1 synchronization accuracy



(b) *Timeline* 2 synchronization accuracy

Figure 2.8: (a) shows pair-wise error probability density of three nodes a, b, c bound to timeline 1 in Figure 2.2(Right) with 100 μsec accuracy requirement, (b) shows pair-wise error probability density of three nodes c, d, e bound to timeline 2 with 1 μsec accuracy (Note that x-axis units are in nanoseconds, and x-axis scale changes in (a) and (b)). Note that c maintains mappings of both timelines, and the achieved accuracy for all the nodes is almost equal to their desired accuracy

synchronization packet transmission rate reduces synchronization error and increases timing accuracy. This proves the existence of such adjustable parameters, which can be exposed to the userspace services so that they can control the system performance and meet the QoT requirements.

Now that we have synchronized the NIC to the core clock and established the relationship between synchronization rate and accuracy, we use a topology similar to the one in Figure 2.2(Right) for end-to-end synchronization. There are two timing subgroups: nodes a, b and c bound to Timeline 1 with an accuracy of 100 μsec; and nodes c, d and e bound to Timeline 2 with an accuracy of 1 μsec. The system sets a synchronization rate of 0.05 Hz for Timeline 1, and 2 Hz for Timeline 2 according to their accuracy requirements. We conducted experiments on this topology to demonstrate that the QoT stack runs multiple and parallel synchronization sessions on a single node, which disciplines multiple timelines

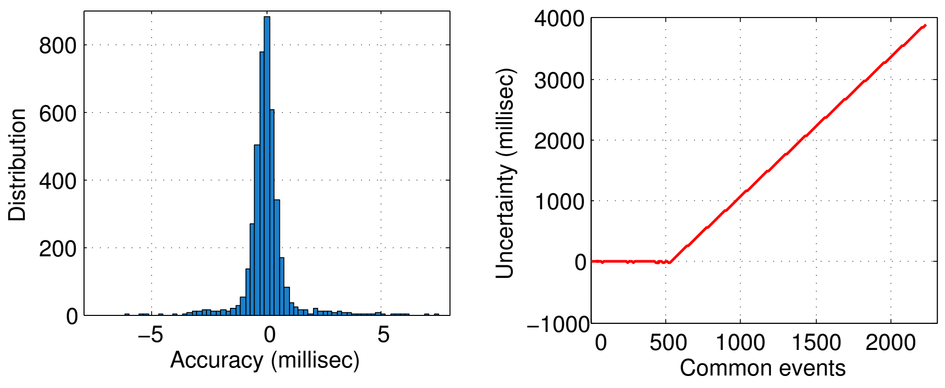
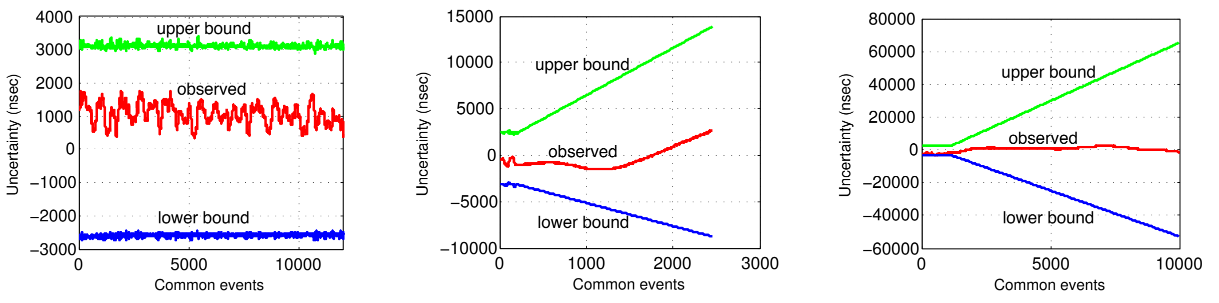


Figure 2.9: $(Left)$ The timing error distribution for two nodes is in the range of 10 millisecond. These nodes bind to a timeline with 10 millisecond desired accuracy and are synchronized using *ntp-qot*. $(Right)$ We first run *ntp-qot*, resulting in a horizontal line (negligible uncertainty). Subsequently, we turn the *ntp-qot* synchronization off and within just few minutes, the timing uncertainty increases 1000x from 4 millisecond to 4000 millisecond

simultaneously. The results are shown in Figure 2.8, where node c maintains two timelines with very different accuracy requirements of 100 μsec and 1 μsec, with respect to Timelines 1 and 2 respectively. This validates our initial claim that the timeline-driven architecture not only supports multiple virtual time references on a single node, but also synchronizes only to the desired accuracy, hence conserving resources like bandwidth and energy.

In general, NTP is used to synchronize nodes distributed in a wide-area network. NTP servers provide reference global time and are globally-distributed in different stratum. However, software timestamping limits the accuracy of NTP to the order of milliseconds. The results of our timeline-based NTP implementation *ntp-qot* are shown in Figure 2.9. We see that the nodes' clocks are no more apart than 10 milliseconds from each other, thus validating our initial claim that nodes bound to timeline, only synchronize to the desired accuracy.

For applications to adapt and be robust to failures, the system should continue to report the estimated uncertainty. When we turn off the synchronization service, the uncertainty bounds increase at a high rate as shown in Figure 2.9 (Right), because of the lack of drift compensation by NTP implementations in general. Hence, nodes synchronized using NTP have large uncertainties when synchronization stops[3].



(a) Synchronization on for 1 hour (b) Synchronization on, then off for 5 mins (c) Synchronization on, then off for 1 hour

Figure 2.10: Upper and lower bounds around the observed uncertainty with and without synchronization. Note the change in y-axis scale, increasing from (a) to (c)

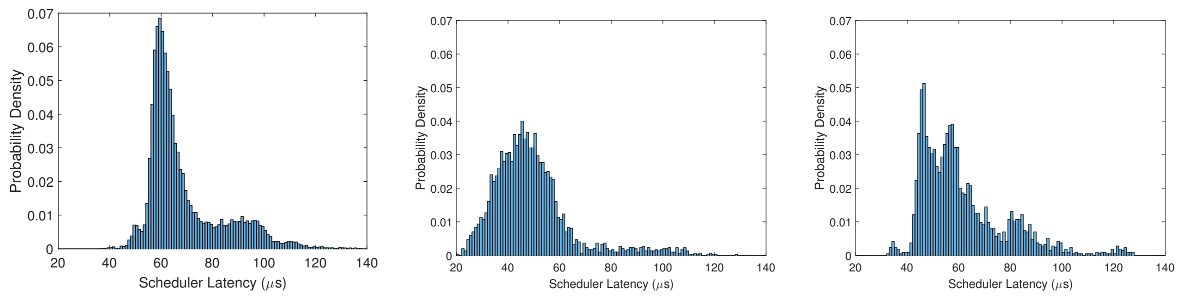
In Figure 2.10, we show the QoT stack's ability to estimate the uncertainty in synchronization and expose it. Uncertainty captures the variance in time introduced by various sources of errors, that cause the time to deviate from its true value. The red plot provides the ground truth i.e, the actual uncertainty between the *local* timeline reference and the *global* timeline reference, $e = t_{global} - t_{local}$, whereas the green plot is an upper bound on uncertainty estimated by the stack, $e_u = t_{upper} - t_{local}$ and the blue plot is the lower bound on uncertainty estimated by the stack, $e_l = t_{lower} - t_{local}$. Note that the bounds are valid that is, $t_{upper} > t_{global}, t_{local} > t_{lower}$, only when, $e_u > e > e_l$, which is what is achieved in Figure 2.10. The uncertainty bounds estimated by the stack are applicable, both when synchronization is running or not. Figure 2.10a shows the bounds when the synchronization is running. Note that these bounds tend to increase when we turn the synchronization off (Figures 2.10b and 2.10c). The bounds extend in both directions as a function of variance in frequency bias, and they will always bound the actual uncertainty. The longer the period for which the synchronization is off, the higher will be the uncertainty bounds. Thus, the QoT stack not only reports precise time to the applications but also the uncertainty in time with high confidence bounds.

[3]due to possible transient/permanent network outages

2.8.2 Scheduler Uncertainty

We benchmark the QoT Core's scheduling interface against the Linux *Real-Time* (RT) scheduler by using a periodic pin-toggling application which toggles a memory-mapped GPIO pin, at every second boundary. All the following experiments were conducted under identical load conditions for a duration of 3000 seconds, with the pin-toggling application being the highest real-time priority user application in the system. Multiple sporadic tasks with lower real-time priorities, which used the QoT stack functionality, were also running on the same system.

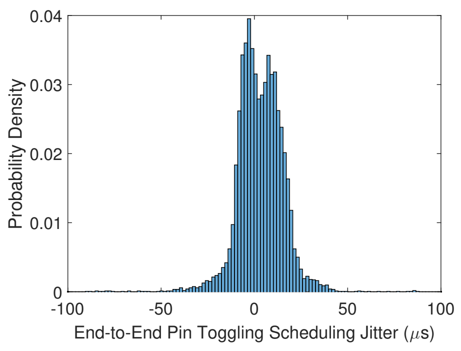
To measure scheduler uncertainty, we devise the following experiment. On a single node, an application periodically calls the **`timeline_waituntil_nextperiod`** API call, such that the pin toggle is scheduled at every second boundary on a timeline. When the task wakes up, the QoT stack provides a timestamp (with uncertainity) for when the event was actually scheduled. The scheduler latency can be estimated by taking the difference of the timestamps: when the task was supposed to wake up, and when it was actually scheduled. We also empirically measure the scheduler latency by using a Salae Logic Pro 16 logic analyzer [Sal]. The logic analyzer measures the latency for each pin toggle event by comparing against a deterministic PWM with edges at every second boundary on a timeline reference.



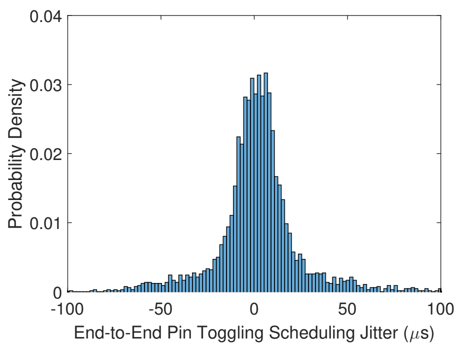
(a) Estimated QoT Scheduler Latency (b) Measured QoT Scheduler Latency (c) Measured Linux RT Scheduler Latency

Figure 2.11: Scheduler Latency Distributions, for a periodic pin toggling application on a single node

Figure 2.11a plots the distribution of the scheduler latency as estimated by the

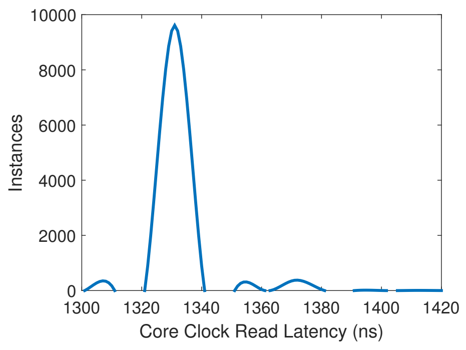


(a) QoT Stack for Linux

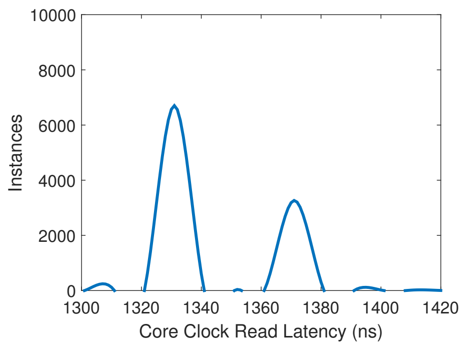


(b) Linux RT Scheduler with PTP

Figure 2.12: End-to-end scheduling jitter distributions for synchronous pin toggling on two nodes



(a)



(b)

Figure 2.13: Clock read latency histograms in different time intervals, estimated by the system uncertainty estimation service

QoT stack, while Figure 2.11b shows the empirically-measured distribution. Observe that the empirical distribution and the distribution provided by our stack share similar characteristics. Thus, the uncertainty estimate provided by the QoT stack holds up to empirical measurement.

For the Linux RT scheduler, using real-time priority scheduling (**`SCHED_FIFO`**), Figure 2.11c shows the empirically-measured latency distribution, where the **`clock_nanosleep`** system call was used to schedule a periodic pin toggle. Note that the QoT-aware Linux scheduler and the Linux RT scheduler share similar statistical properties. The QoT-aware scheduler provides adherence to our *timeline*-driven architecture with no significant overhead.

The ability to perform choreographed scheduling is key to our stack, and hence we next characterize the end-to-end synchronous scheduling jitter. In our setup, we have two identical applications running on separate nodes. Both applications bind to the same timeline and synchronize with each other. Using the **`timeline_waituntil_nextperiod`** API call, the applications synchronously toggle a memory-mapped GPIO pin at every second boundary on the timeline reference. The synchronization service is also running on both nodes. In Figure 2.12a, we plot the distribution of the end-to-end jitter between the pin toggles of the distributed application. The instants at which the pins toggled were captured by a logic analyzer, and the difference in timestamps was used to compute the obtained distribution.

We conduct a similar experiment using the Linux **`clock_nanosleep`** system call on two distributed nodes synchronized by PTP. Figure 2.12b plots the distribution of the end-to-end scheduling jitter for Linux and PTP. Our stack runs a patched PTP synchronization service, and hence the distribution obtained has a similar jitter profile to that obtained using PTP. Note that our interface is policy-agnostic and does not incur additional overhead, while at the same time providing a range of QoT-based functionality. However, the scheduling jitter can be reduced using more suitable policies in the kernel.

Figure 2.13 shows two histograms for the estimated latency in reading the core clock from userspace, over different one-second durations, as estimated by the system uncertainty estimation service. Observe that the distributions change over time and are a function of system load. Each peak in the distribution corresponds to different *locks* which cause contention in reading the core clock. This measured distribution plays a key role in continuously keeping track of the uncertainty introduced by the OS in reading the clock.

2.9 Key Findings

Summarizing all the empirical results, we have shown multiple capabilities of timeline. The ability of our system to report the uncertainty gives an application confidence in its time estimate. Applications are able to adapt to variations and be robust to failures should the system continue to report the estimated uncertainty. These uncertainties arise form hardware, software and network component of a time stack. For the synchronization uncertainty contributed by the network component, we extend it as a function of variance in frequency bias when synchronization has been turned off and the clocks are free to drift. Our system is able to capture this drift and adjust the error accordingly. Scheduling uncertainty is also a key concern for systems. We have shown that our system is able to estimate the scheduling uncertainty and expose that to the application with an error of few microseconds. Clock read latency also contributes to the end-to-end uncertainty in timing. Our results show that this latency is in the order of 100's of nanoseconds. In short, we have collected uncertainty information from the hardware component (clock read latency), software component (OS and scheduling uncertainty), and the network component (synchronization uncertainty). We sum these uncertainties and expose it to the application. We provide an upper and lower bound to the actual (ground truth) uncertainty with an error margin of 6 microseconds on average. Another capability of timeline is to control application's performance by specifying their requirements. We show that an application bound to a timeline with an accuracy of 1 microsecond achieves an average accuracy of 0.6 microsecond i.e. it is well below the error the application can tolerate. In short, our results show that an application can not only control its timing performance but also be aware of its timing performance.

2.10 Conclusion & Future Directions

The timeline abstraction with its associated notion of Quality of Time (QoT) helps virtualize time-related resources in a system and plays a role analogous to that of sockets with associated Quality of Service (QoS) bindings in network stacks. QoS-aware networking applications can read, write, open and close sockets, and specify QoS parameters. Similarly, QoT-aware time-sensitive applications can bind and unbind from timelines, read and schedule events on the timeline reference, and specify QoT requirements. We make QoT visible and controllable in our timeline-driven architecture. This enables QoT-aware applications to specify their timing requirements, while the system manages clocks and synchronization protocols to provide the appropriate levels of QoT. In the future, this architecture would be extended to address challenges introduced by multiple processing cores, hardware accelerators, and peripherals.

Our initial implementation of the QoT Stack for Linux delivers most of our early goals. However, it presently takes advantage of only the *accuracy* attribute of timelines. Future implementations of our stack will also make use of the *resolution* attribute and provide the ability to dynamically switch between hardware clocks based on application requirements. We also plan to support multiple network interfaces and different oscillators that could be adjusted in hardware. The stack could then switch between different core clocks, use different NICs across heterogeneous networks, and use different synchronization protocols, to best strike a balance between desired performance and resource consumption. Finally, a co-optimization of timelines and synchronization sessions would help conserve network and system resources.

The QoT Stack for Linux is open-source and under development. In the future, we plan to support multiple hardware platforms. The code repository can be found at, *https://bitbucket.org/rose-line/qot-stack/src*

Part II

Systems for Timing Integrity

CHAPTER 3

Securing Time in Untrusted Operating Systems with TimeSeal

3.1 Introduction

Emerging temporal use cases in the Internet of Things (IoT) applications have a critical dependence on high precision and accurate *relative* time. For instance, distance and speed calculations rely on precise round trip times [MPP07] [TSS16], schedulers build upon elapsed and remaining times [RPM13] [MW13], network telemetry depends on residency delays [LPJ15], code profiling requires execution times [CZR17], and data sampling needs timestamps [HSG18]. Attacking a system's sense of time has ramifications such as location theft, network outages, higher delays, and data inconsistencies. Furthermore, critical functionalities for securing applications in shielded execution environments such as Haven [BPH15], SCONE [ATG16], Panoply [STT17] and Graphene-SGX [TPV17] have no access to secure time. Instead, they rely on untrusted operating system (OS) time. A compromised OS may lie about the time or signal early timeouts, and although traditional cryptographic techniques, trusted execution technologies, and network security mechanisms may guarantee data security, they do not cater to *time security.*

In an attempt to provide this security of time, researchers have tried to implement secure clocks for shielded execution in Trusted Execution Environments (TEE) [SLK17]. A well known industrial approach fTPM [RSW16] tries implementing a secure clock on Arm TrustZone but it relies on untrusted OS acknowledgments for clock writes. On the other hand, Déjá Vu [CZR17] leverages hardware transactional memory to provide a high resolution clock for Intel SGX but it is susceptible to frequent aborts. Aurora [LLZ18] also leverages hardware support of System Management RAM (SMRAM) to provide an absolute clock for SGX. Although it relies on specialized hardware capabilities, it makes use of untrusted timers and kernel devices. All of these clocks are prone to attacks in the presence of a compromised OS.

To reason about the generalized security of a clock, we argue that it is essential to secure all layers in a time stack giving rise to three main challenges: first, find a trusted timer that cannot be modified by a privileged adversary [TKG18], second, provide a secure path to read the trusted timer in a timely manner [SWG17], third, protect timekeeping software from adversarial attacks [CZR17]. In this paper, we propose TimeSeal, a secure time architecture designed to tackle these challenges.

To address the first challenge, we compare the timing capabilities of different Trusted Execution Environments (TEE) [SLK17]. Based on our analysis, TimeSeal leverages hardware-based protection of Intel SGX that gives access to a trusted timer. A privileged adversary in a compromised OS cannot write to the SGX trusted timer.

With respect to the second challenge, the SGX community has confirmed that the path to SGX trusted timer is not secure [lin18b]. The SGX platform service transfers trusted time packets over a secure session via OS inter-process communication (IPC) [CZ17]. As these packets are encrypted and integrity protected, a compromised OS cannot change the packet contents. However, an attacker may violate timely arrival of these packets by *delaying* them and consequently settles on the wrong perception of elapsed time. We address this challenge of securing trusted timer access by mitigating the effects of this *delay attack* on trusted time values.

Unfortunately, coarse-grained SGX time–which increments once per sec– is insufficient to protect against delay attacks. TimeSeal first aims to increase SGX trusted time resolution by providing a subtick service that interpolates the SGX time and divides it into granular subticks, This work is based on a coarse-grained TEE trusted timer, and uses counting threads to increase its resolution while adopting compensation mechanisms to mitigate accumulative errors and maintain monotonicity. Using both the SGX time and subticks, we build a high-resolution and monotonic SGX clock that is capable of measuring intervals as small as 0.1msec and timestamp events that are apart by $\geq$ 0.1msec. TimeSeal then utilizes this improved precision to detect and mitigate delay attacks. This high resolution also serves applications with high precision requirements [TKG18] [TKA17].

The third and final challenge requires us to defend against attacks on timekeeping software, where a compromised OS may downgrade the time resolution to seconds by scheduling out the associated timekeeping threads. TimeSeal overcomes these *scheduling attacks* and adopts multithreaded counting policies to induce uncertainty in malicious scheduling and reduce the efficacy of this attack on resolution degradation. As a result, the system maintains msec-level resolution that is also substantial to detect and compensate for delay attacks.

Our contributions are summarized as follows,

- We provide a complete guideline for time security by enumerating challenges in securing a clock.
- We identify and verify that the path to reading SGX time is not secure by implementing a delay attack in OS and disrupting the timely arrival of SGX time.
- We present TimeSeal, a secure time architecture that addresses the aforementioned challenges.
- We provide a high-resolution SGX time and use the improved resolution to mitigate delay attacks.
- We devise policies that reduce the effect of malicious scheduling on time error.
- We prototype TimeSeal on Intel SGX and evaluate the complete secure time architecture.

3.2 Related Work

In this section we review related works that motivated our design of TimeSeal.

Secure clocks. Researchers have attempted to implement trusted clocks for secure computation inside Trusted Execution Environments (TEE) [KHH17] [ZCC16] or to enforce time-based policies [KRR12]. For example, Chen et al. [CRS14] provides coarse-grained secure clock on a trusted cloud with the assumptions that the network link has a bounded delay, and a trusted counter is present in the cloud. Raj et al. [RSW16] face numerous challenges in implementing a secure clock. The absence of a secure timer forces them to rely on a compromised OS to acknowledge clock writes and make the clock persistent. It is to be noted that there is no secure clock on ARM TrustZone. Though a peripheral such as a timer can be mapped to the secure world, a peripheral's controller can still be programmed by the normal world [RSW16].

Time in shielded execution. For line rate processing in middleboxes such as ShieldBox [TKG18] and Slick [TKA17] implemented over SGX, fine-grained cycle-level measurements are made inside enclaves via reasonably fast Network Interface Card (NIC) clock. However, as noted by the authors, this clock is not secure against OS attacks but they argue that there is no precise trusted time source for SGX enclaves and it remains an open problem [TKG18] [SWG17]. Other secure systems such as SCONE [ATG16], Haven [BPH15], and Panoply [STT17] also rely on untrusted time via OS system calls.

High resolution trusted clocks. TrustedClock [LL18] for SGX and Aurora [LLZ18] tries to provide a high resolution and absolute secure clock for SGX enclaves. Their absolute clock leverages System Management RAM (SMRAM) for timekeeping, and relies on a kernel daemon to trigger system management interrupt (SMI) for a clock read request. A malicious OS can make time fuzzy by arbitrarily delaying these requests. Also, SMI handler reads legacy timers on Intel Architecture that can be written to by the OS in a consistent manner to avoid detection. Thus relying on hardware timers that OS can manipulate and reading from kernel devices that can be delayed makes their system not secure.

On-demand high resolution timers. Much closer to our work is an entire body of research that uses high resolution timers either to launch side channel attacks [SWG17] or to detect side channel attacks [CZR17] in shielded execution. The absence of high resolution timers inside shielded execution environments, particularly Intel SGX, motivated researchers to find alternative solutions. Malware Guard Extension [SWG17] emulates a short-lived precise timer inside an enclave to perform a Prime+Probe cache side-channel attack against co-located enclaves. Déjá Vu [CZR17] detects page faults by measuring execution time of the victim code in SGX with a reference-clock. This clock is incremented within a Hardware Transactional Memory [HM93] to detect OS interruptions but it is affected by high frequency TSX aborts restricting the applications to execute in less time. Researchers in this domain are mostly interested in constructing on-demand, high resolution timers to measure very short durations [KS16] [SWG17]. They do not quantify performance of a clock source capable of providing high resolution monotonic sense of time.

In short, secure clocks are non-existent, and current secure systems have to rely either on untrusted clocks or coarse-grained clocks, leaving out many applications in need of precise trusted time.

3.3 Background

This section covers concepts that influence our design choices for TimeSeal. We start by explaining a traditional time stack in all systems, possible attacks on the time stack, and the timing capabilities of TEE.

Attacks on the Time Stack. Every system maintains a time stack. Discrete components that make up a time stack are hardware timers and timekeeping software. A hardware counter/timer counts the number of cycles of a periodic signal obtained from an oscillator. Timekeeping software maintains time by converting cycle counts into human understandable time.

A hardware timer in a time stack is not considered secure if a malicious software is able to write to its registers. In Intel architecture, RDTSC/RDTSCP results are not immune to influences by privileged software, e.g., the Time Stamp Counter (TSC) can be written to by the OS [sgx16] [for17]. Similarly, other timers such as the High Precision Event Timer (HPET) can be controlled by the OS. Basically, the design principle of the OS dictates that its high privilege allows it to write to all registers. Virtualizing timer via trusted hypervisor also does not protect against timer modifications. First, OS can consistently alter timer to avoid detection. Second, no hypervisor can detect malicious time delays. Furthermore, OS is responsible for timekeeping, and it can lie about time, signal timeouts early or late, delay time transfer to applications, or gradually change the notion of time for applications to deceive them. Hence, a privileged software is capable of adding discontinuities in time. To protect these timers, it is essential to restrict timer writes only to trusted entities.

Trusted Time in SGX. Intel SGX supports trusted time service by leveraging the security capabilities of Intel Converged Security and Management Engine (CSME) [CZ17]. The CSME consists of an embedded hardware engine that runs its own firmware. This firmware allows the host to load and execute Java applets in the CSME at runtime through a dynamic application loader. SGX hosts different architectural enclaves that provide key services to application enclaves. Platform Service Enclave (PSE) is an architectural enclave that provides trusted time and monotonic counter service. Architectural Enclave Service Manager (AESM) is a background service that hosts the PSE and automatically loads and starts the Platform Services DAL Applet (PSDA) inside CSME to securely expose the CSME battery backed Protected Real-Time Clock (PRTC) timer. As CSME is backed up with a battery, it's not affected by CPU power management states, i.e., it always has power to keep the PRTC running. PSDA and PSE communicate through a Management Engine Interface (MEI) driver. This MEI interface supports data exchange through a secure, memory-mapped mechanism not accessible by OS.

Intel SGX gets its notion of time from the PRTC timer in CSME. The OS can neither access nor manipulate the PRTC and, hence, SGX provides access to a trusted timer. This trusted time is managed by a PSE that reads the PRTC and transforms it into *SGX time* by appending an epoch to it. An application enclave first establishes a secure session with PSE, then invokes the *sgx_get_trusted_time()* API to get SGX time.

The encrypted and integrity protected messages between the application enclave and PSE pass through the OS layer. These messages can be captured or replayed by the OS. However, a PSE message includes a sequence number that helps reset a session if replay attacks are detected. SGX time is in seconds and too coarse-grained to be useful for measuring short durations within a single enclave.

Secure Clock in TPM [Spe17]. Secure clock in TPM provides time in msec since TPM boot. This clock can be aligned with any time reference by shifting it forward or by changing the clock frequency using owner or platform authorizations [Spe17]. Its time is written to non-volatile memory at most every 2^{22} msec for persistence. Applications query TPM's time using standard API. The API TPM2_ReadClock() returns uncertified (not signed) values, and it is used by the OS to manage the timing resources of the TPM. TPM2_GetTime() API returns a structure and an optional signature over it for time attestation. This attestation mechanism helps in detecting attacks on TPM time. A TPM clock is mostly used for internal data timestamping. It is not used as a timekeeping source for external applications because of restricted frequent accesses and large communication delays.

3.4 TimeSeal Design

To provide a secure time architecture, the challenges to address are namely the availability of a trusted timer, secure access to that timer, and secure clock software. In this section, we present design overview of TimeSeal overcoming these challenges under a generalized threat model.

3.4.1 Challenges

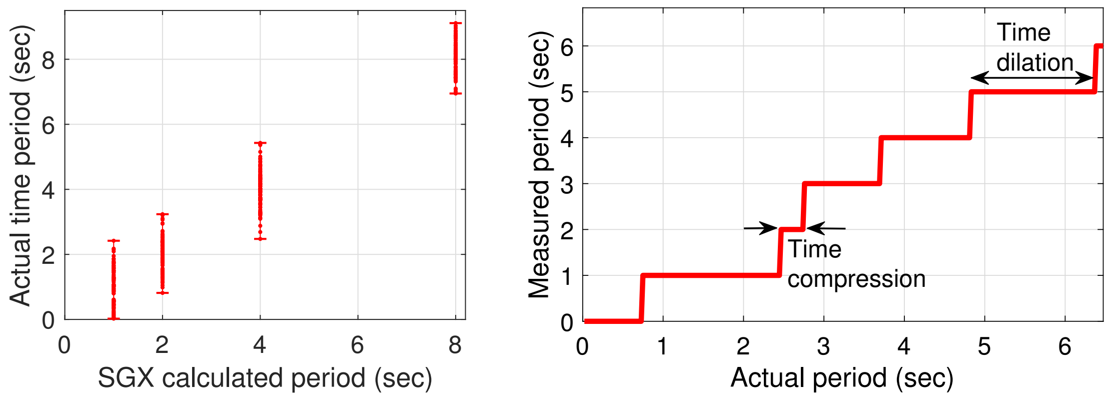
The choice of a trusted timer, one that cannot be modified by a privileged adversary, is critical for TimeSeal's design. To make this choice, we compare the timing capabilities of SGX and TPM, the only two clocks provided in TEE.

[Challenge 1] A Trusted Timer: SGX Trusted Time versus TPM Secure Clock. There are a number of limitations in providing secure time over Intel SGX. First, peripheral registers such as hardware timer's registers cannot be mapped into protected enclave memory to protect the timer from OS manipulation. Only the Process Reserve Memory (PRM), a special DRAM for secure enclaves in SGX, can be mapped into the virtual address of an enclave page – or in SGX's terminology – Enclave Page Caches (EPC). Second, SGX trusted timer has coarse-grained, one-sec resolution [CZ17] that can only be used for enforcing policies spanning large time intervals. In contrast, TPM has a high resolution clock. The effective resolution of TPM clock however is reduced when accessed from an enclave. We empirically compared the access latencies to SGX trusted time and TPM secure clock from an application enclave. The latency in retrieving SGX time from CSME has a mean of 13msec, whereas TPM clock has an access latency of almost 32msec that is three times more than SGX time latency. Note that both CSME and TPM are separate from the CPU, justifying their large access latencies from enclaves.

The effective clock resolution that an application sees is either the base timer resolution or the clock access latency, whichever is higher. This makes theoretical time resolution of 1sec for SGX and approximately 32msec for TPM. Although TPM has higher resolution, it cannot be accessed by multiple applications at the same time due to limited TPM resources, thus decreasing it's effective resolution for applications. TPM's clock is mostly used to timestamp stored keys and data inside it, and not designed for timestamping network packets or sensor data. TPM clock does not satisfy the needs of broad applications, hence we choose SGX trusted time as a trusted timer for TimeSeal's secure time architecture.

[Challenge 2] A Secure Path to Trusted Timer. The encrypted SGX time value passes through a secure channel established between PSE and an application enclave. This communication happens via IPC [CZ17]. We know that a compromised OS cannot attack the encrypted and integrity protected packet. We show, however, that it can still cause damage by *delaying* the time packet. Delaying affects timely arrival of of SGX time, hence distorting time for application enclaves. Note that TPM time is also prone to IPC delay attacks.

We implement a delay attack in the OS by delaying all SGX time packets with a random value sampled from a uniform distribution of 0 to 1sec. The result in Figure 3.1a shows that SGX notion of 1sec fluctuates within 0 to 2.5sec, while 4sec varies between 2.2



(a) Fuzzy SGX time (b) Time dilation & compression

Figure 3.1: Two ways to visualize the effect of delay attacks on SGX trusted time

to 5.5 sec. Time fuzziness due to delay attacks affects many applications. For example, Timecard [RPM13] servers need to provide consistent response times to clients within few msec of target delays. Fuzzy time in the order of seconds distorts their sense of elapsed time resulting in wastage of computation resources.

The effect of delay attacks can also be visualized in another way shown in Figure 3.1b. An application relying on SGX time measures 1sec durations on y-axis, which in reality are distorted durations on x-axis. Time advances without a fundamental fixed frequency, resulting in either time dilation or compression across different intervals. Thus, delay attacks cause SGX time to dilate and constrict, and we establish that the path to our choice of trusted timer – SGX time – is not secure.

[Challenge 3] A Secure Timekeeping Software. A timekeeping service maintained by an enclave process and threads is secure from memory manipulation. However, these threads can still be attacked by malicious OS scheduling that can make time inconsistent. This is because a thread running in enclave mode is the same as a thread running in normal mode from the OS perspective [Int18], We refer to attacks on timekeeping threads through malicious scheduling as *scheduling attacks* and these also result in time dilation or compression. Hence, we establish that timekeeping software within secure enclave is also prone to attacks.

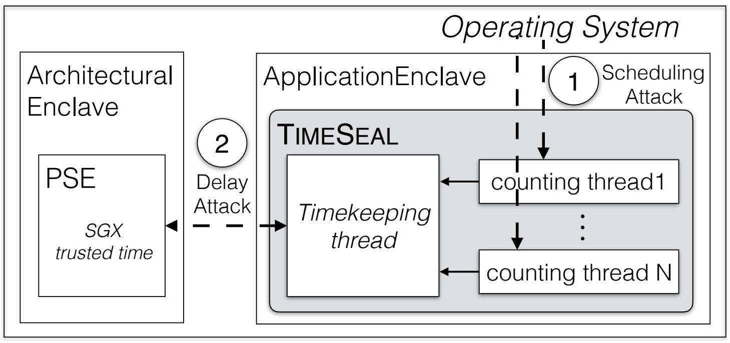


Figure 3.2: TimeSeal components inside an Application Enclave. It is subject to two main attacks: ① OS scheduling attacks on TimeSeal threads, and ② OS delay attacks on SGX trusted time

3.4.2 TimeSeal Overview

TimeSeal is a secure time architecture that overcomes the above mentioned challenges, i.e., the availability of a trusted timer, secure access to that timer, and an attack free timekeeping software. It solves `Challenge 1` by leveraging SGX time derived from a trusted timer. `Challenge 2` is addressed by timely detection and mitigation of delay attacks, while `Challenge 3` is resolved by devising policies that overcome the effect of scheduling attacks.

TimeSeal's components are shown in Figure 3.2. SGX time provided by PSE to an application enclave is coarse-grained and incapable of detecting and eliminating scheduling and delay attacks. Hence, a high resolution clock is critical to secure time. TimeSeal provides a high resolution clock comprised of a *timekeeping thread* that keeps track of SGX time, and *counting threads* that interpolate between SGX time to provide sub-sec resolution.

3.4.3 Threat Model

The goal of an attacker is to compromise TimeSeal's sense of time while maintaining stealthiness. Although more prominent attacks could cause damage, e.g., a denial-of-service attack on any component, we consider attacks that have a more enduring impact over time, e.g., an attacker that gradually makes time fuzzy successfully establishes false proximity for various applications that rely on co-location detection [HCS18] [SP05].

Compromised OS and TimeSeal. Our threat model considers TEE by different vendors such as ARM TrustZone and Intel SGX as trustworthy. TimeSeal does not trust the OS nor hypervisors as they can be corrupted. To maximize damage, an attacker may stay undetected throughout the system's operation because consistent time uncertainty is worse for the system as a single time jump can easily be detected. We also assume that the threads in TimeSeal are subject to a normal OS scheduling policy and can be aborted/scheduled out at any instant, i.e., they will compete for CPU time with the rest of the system.

Attack vectors. Because the attacker is trying to compromise the timing components of TimeSeal in a stealthy fashion, the only two attack vectors specific to TimeSeal's attack surface are the aforementioned (1) *delay attacks* or (2) *scheduling attacks*. For delay attacks, prior knowledge of the system and physical clock characteristics helps the attacker launch an attack that degrades system performance without detection. For example, the OS knows that 'aesmd daemon' encapsulates PSE and handles SGX trusted time packets. Therefore, it launches *delay attacks* on SGX time packets by intercepting all transmitted/received packets to/from aesmd daemon. Hence the attacker is capable of making SGX time fuzzy as established in `Challenge 2`.

It is to be noted that an attack strategy of delaying all SGX packets by a constant value does not harm a system. Adding a constant value to true time does not affect the rate at which time is elapsed. Rather, delaying SGX packets by a different value adds variations to the clock rate and distorts the passage of time. Therefore, our threat model incorporates incremental, random, and distribution based delay attacks on SGX packets.

Detectability of DoS Attacks. The attacker also knows that SGX time increments every sec, referred to as *SGX tick*. It may choose to delay a packet by any arbitrary value. This causes an application polling SGX time to detect missing SGX ticks. We equate this scenario to a denial of service, which becomes an availability issue rather than a security issue. To maintain stealthiness, it would choose to delay by a sec or so. For

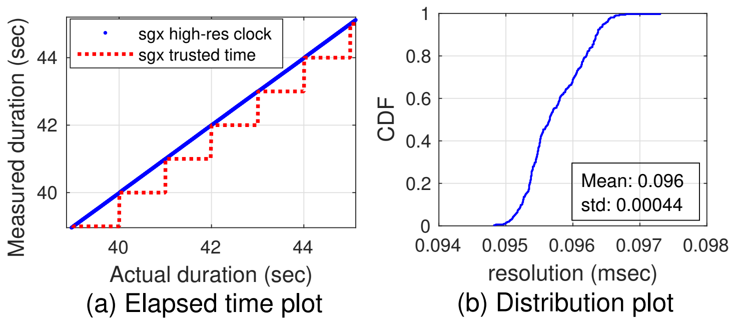


Figure 3.3: (a) High-resolution SGX clock measures sub-second durations compared to SGX tick that is only capable of measuring durations greater than a second. (b) Achieved mean resolution of high-res clock is 0.1msec with 0.4microsec standard deviation (std)

scheduling attacks (established in `Challenge 3`), we assume that an attacker does *not* want to schedule out all threads of a process in order to maintain stealthiness–if there are no threads running, this is considered a detectable denial of service because the count value is less to none.

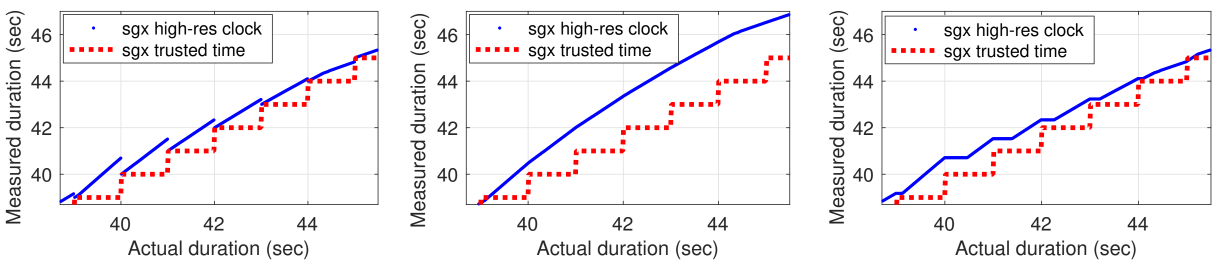
TimeSeal strives to protect against attackers capable of launching scheduling attacks on SGX enclave threads as well as delay attacks on SGX time packets.

3.5 Achieving High Resolution Secure Clock in SGX

The resolution of SGX time is only one sec. This low resolution is not enough for many IoT applications.

We provide a *subtick service* that builds a high resolution SGX clock on top of coarse SGX time. We refer to a 1sec SGX time increment as an *SGX tick*. To get fine time granularity, we develop a subtick service that interpolates SGX ticks. This interpolation mechanism uses an SGX tick, which has a large, known one-sec period, and a subtick, which has a short, unknown sub-sec period. SGX tick is used to establish the period of the subtick, and together they can provide time with high resolution.

To build a high resolution SGX clock–which we call "high-res clock"–using SGX ticks and subticks, we use a clock model to calculate the current time, i.e., $t_{local} = SGXticks + \frac{subticks}{MA(subticks\_per\_sec)}$, where t_{local} is the local time reported by our high-res



(a) Non-Monotonic Clock (b) Monotonic but not load resistant (c) Monotonic and load resistant

Figure 3.4: Comparison of different clock models to build a high resolution SGX clock. A clock should always be monotonic and compensate for time errors during high system load

clock, $SGXticks$ represents seconds, and $subticks$ divided by the moving mean (MA) of multiple

$subticks\_per\_sec$ values in a window represents the fractional part of a sec. Thus, we are able to provide sub-sec clock resolution.

The subtick service is comprised of a *timekeeping thread* inside an enclave that continuously polls SGX time, and a *counting thread* that counts for one SGX tick as shown in Figure 3.2. Note that counting thread runs software executing a loop such that subticks correspond to instruction cycles. We test the resolution of our high-res clock by measuring fine time intervals. The smallest duration that a clock is able to measure in a stable manner is its resolution. As shown in Figure 3.3a, the sloped dotted line (blue) shows that the high-res clock is capable of measuring sub-sec time durations. The dashed (red) line shows that SGX time is not capable of measuring durations that are less than a sec, i.e., a new value comes once every sec. Figure 3.3b shows that our high-res clock is able to achieve a mean resolution of 0.1 msec, i.e., the clock is capable of measuring durations as small as 0.1 msec or timestamp events that are apart by 0.1 msec. This 0.1 msec resolution is a result of software instructions in subtick service that take up CPU cycles. We can configure the code to improve or relax this resolution.

Threads scheduled under high system load cause subtick variations. We conduct experiments by running a large number of stressing threads along with the subtick service to overload the system by 80%. This causes fluctuations in $subticks\_per\_sec$ that result in discontinuities in t_{local} based on our current clock model. Figure 3.4a shows a non monotonic clock with time discontinuities shifting the clock back in time. t_{local} exceeds true SGX time as it advances at a rate higher than the nominal rate.

To make the high-res clock monotonic, we revise our clock model by advancing local time from previous local time instead of the latest SGX tick, i.e. $t_{local} = t_{prev\_local} + \frac{subticks}{MA(subticks\_per\_sec)}$. The result of this, as shown in Figure 3.4b, is a monotonic clock where t_{local} significantly deviates from true time due to accumulated errors over time. We rely on the SGX tick boundary to calculate the accumulated error, $t_{error} = t_{local} - SGXticks$, and compensate for it. *We thus propose a new clock model that not only advances time with respect to previous time, but also takes into account the accumulated error in local time at every SGX tick boundary.*

Adding huge offsets to remove error from time is not a good practice as it can lead to negative durations or high error fluctuations. Thus, we divide this error into smaller chunks equal to the high-res clock resolution. We then remove the error by subtracting small error chunks from local time at every iteration until no error remains. This process of removing error using smaller chunks is called slewing time. Thus, our new clock model is, $t_{local} = t_{prev\_local} + \frac{subticks}{MA(subticks\_per\_sec)} - slew(error)$. Figure 3.4c shows that the slewed clock is monotonic and slowly converges to true time during peak load.

3.5.1 Scheduling Attack and Mitigation

An enclave counting thread of the subtick service is continuously counting. This thread is subject to normal OS scheduling policy and can be aborted/scheduled out at any instant. As a result, the number of subticks per sec over multiple SGX ticks are inconsistent. A compromised OS may issue sophisticated attacks and schedule out the counting thread to downgrade the time resolution to seconds, making the subtick service useless for achieving high resolution.

A single counting thread is unlikely to provide a consistent count every sec under malicious scheduling as there is a higher probability that an attacker can identify the counting thread. Our goal is to provide a stable count every sec in the presence of high system load and malicious OS scheduling, both of which may cause huge variations in count values over multiple SGX ticks. We attain this goal by inducing uncertainty in the OS scheduling policy: TimeSeal employs multiple threads and a thread counting policy design that reduces the efficacy of an attack on the time resolution per sec.

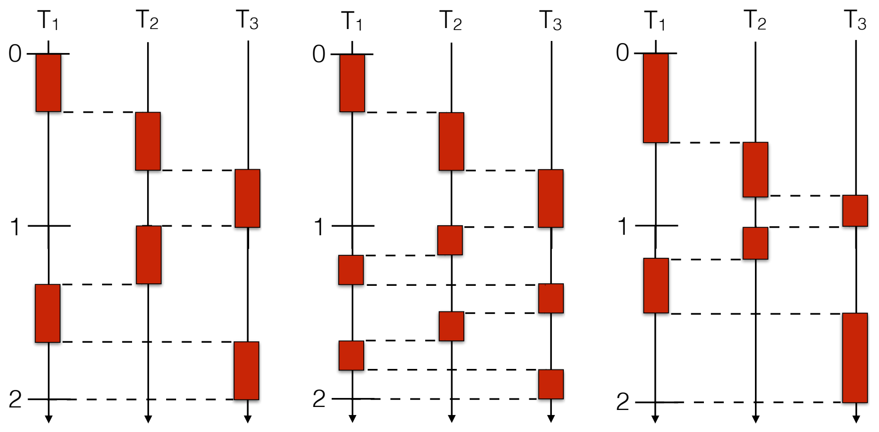
3.5.1.1 Thread Counting Policy Design

One naive approach for inducing scheduling uncertainty is to let all threads count all the time and choose one maximum count value at the end of every SGX tick. This is not an effective counting policy because a fair OS scheduling interrupts all threads for the same amount of time, and results in same reduced resolution as with one counting thread. Another approach is to allow only one thread to count at a time for a specific duration before switching to the next thread. If the order in which the threads are scheduled and their count intervals are known, a malicious OS can locate and interrupt the thread that is ready to count. As such, we need to design policies that reduce this predictability.

Policy design variables. The design variables of a thread counting policy include the number of threads, the counting interval assigned to each thread, as well as the order in which each thread counting interval occurs. We first design three policies based on the latter two variables and discuss how varying the number of threads will affect each policy. For the purpose of clarity, Figure 3.5 depicts the three general approaches for a counting policy consisting of three threads[1]: T_1, T_2, and T_3. Each policy assigns an order and counting interval to each thread. In summary,

- Policy A: assigns a different order but the same count interval to the threads every sec
- Policy B: chooses a different order and count interval every sec
- Policy C: chooses a different thread order every sec while assigning a different count interval to every thread within one sec.

[1]Note that this figure only depicts the counting threads for clarity. These threads will most likely compete with the rest of the OS.



(a) Policy A (b) Policy B (c) Policy C

Figure 3.5: Multiple threads consuming equal CPU time either in one SGX sec or across multiple seconds. Note that threads count in a different order every sec

All threads get the same amount of counting time over a small (Policies A and B) or a long period (Policy C). In order to assess the efficacy of each defense, we will first describe possible attack scenarios that aware of TIMESEAL's counting policy design variables.

Attacker strategies. Based on the aforementioned design variables of TIMESEAL, a clever attacker may craft one of the three following approaches:

- Attack 1: Choose n out of N counting threads randomly to be scheduled out for one sec, where n could be any value from 1 to $N-1$.
- Attack 2: Schedule out all counting threads for the same interval delay of c_d secs one after the other in any order, where $c_d < 1$.
- Attack 3: Schedule out all threads for the same interval delay of c_d secs at the same time.

In all cases, we assume the attacker not only knows the design variables of TIMESEAL, but can also identify the candidate set of counting threads of the associated application. In reality, there may be several other threads associated with the application that may further obfuscate the counting process.

Policy efficacy. To assess the efficacy of each policy against each defense, we define a degradation metric, D, as the portion of *subticks* that will be omitted from the overall *subticks* count across one SGX tick, e.g., if $subticks\_per\_tick = 1000$, and D= 0.5, that would mean an attack caused the subtick service to lose 50% of its resolution. We further determine what is the maximum and minimum degradation an attack may achieve, D_{max} and D_{min}, respectively, as well as the probability of achieving the maximum degradation, $\mathrm{P}(D)$.

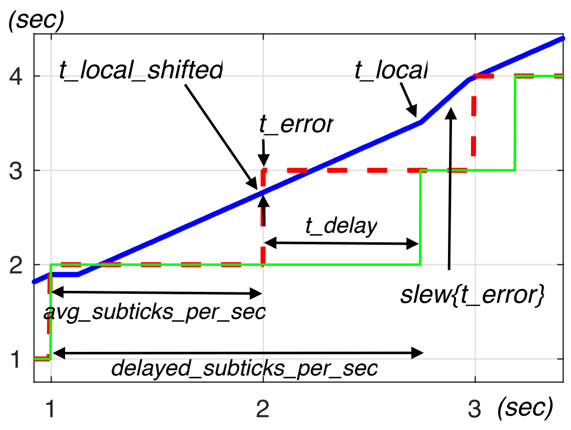
Table 3.1 summarizes the results of our formalization for each attack's efficacy across all policies. In general, Attack 3 is the most powerful and has complete control of the subtick count degradation, D, but causes a detectable denial of service. Attack 1 has the most consistent degradation under stealthy conditions assuming that an attacker can accurately identify the set of counting threads. Attack 2 has the highest possible degradation but with a much lower probability of success. In terms of choosing a policy, they all perform the same on average across all attacks. However, Policy B's performance is greater than or equal to Policy A across SGX ticks, and provides more consistent performance than Policy C–whose degradation fluctuates across SGX ticks since it is possible that an attacker schedules out the threads with the largest count intervals. As such, we hypothesize that Policy B will be the most robust approach against scheduling attacks. We will validate this hypothesis empirically in evaluation section.

3.5.2 Overcoming SGX Delay Attacks

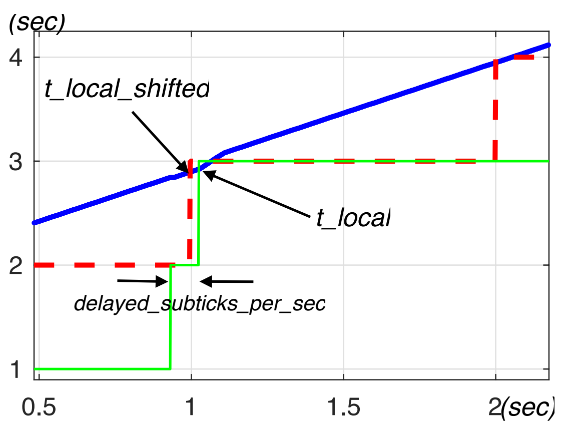
Referring back to TimeSeal's design overview in Figure 3.2, we provide a high resolution clock by overcoming SGX trusted time limitations. Assuming we choose an optimal scheduling policy–tentatively, Policy B–that provides stable subtick values across stealthy attacks, we can also use it to detect delay attacks. If an attacker delays one SGX time

	Policy A			**Policy B**			**Policy C**		
	$\mathbf{P}(D)$	D_{max}	D_{min}	$\mathbf{P}(D)$	D_{max}	D_{min}	$\mathbf{P}(D)$	D_{max}	D_{min}
A1	1	$\frac{n}{N}$	$\frac{n}{N}$	1	$\frac{n}{N}$	$\frac{n}{N}$	1	$\frac{n}{N}$	$\frac{n}{N}$
A2	$(\frac{1}{N})^N$	1	0	$(\frac{1}{N})^N$	1	0	$(\frac{1}{N})^N$	<1	0
A3	1	c_d	c_d	1	c_d	c_d	1	c_d	c_d

Table 3.1: Efficacy of each attack against all policy designs enumerated in Figure 3.5. The degradation metric D refers to the portion of subticks that will be lost due to the attack over one SGX tick



(a) Condition 1: delayed tick duration is much greater than true tick duration



(b) Condition 2: delayed tick duration is much smaller than true tick duration

Figure 3.6: Measured local time (blue dotted line) is slewed by restoring true SGX ticks (dashed red line) from delayed ticks (solid green line)

packet by n^{th} of a sec, the previous SGX tick is $1 + n$ sec away from the current tick, while the next one will be $1 - n$ sec away from the current tick. A stealthy attacker that wants to avoid detection delays SGX time packets by a little bit more than a sec and makes SGX time fuzzy by twice the delay as established in Figure 3.1a. The effect of a delay attack is time dilation and compression. Some elapsed seconds span more than a few seconds, while other seconds only span a few msecs as shown in Figure 3.1b. As a result, a time-aware application that relies on the physical notion of elapsed time calculates wrong intervals [SP05].

Our design relies on the intuition that subticks over multiple seconds show large variations under delay attacks. The attacker cannot avoid these large variations even when it launches coordinated delay and scheduling attacks. With no knowledge of the time value inside an SGX packet, the attacker does not know when a new SGX tick starts. There are two ways an attacker can avoid subtick variations: it can delay packets by a small value–which reduces the time error, or it can delay all packets by the same value–which is not an attack because relative time stays the same. Both ways do not result in an attack.

To provide an accurate high resolution clock, the local time t_{local} should join the SGX ticks by a straight line. Figure 3.6a, 3.9a shows that t_{local} significantly deviates from true SGX ticks due to delay attacks. The first step to overcome delay attacks and align

Algorithm 1 Delay Free Clock Model

```
1: procedure RestoreTrueSGXTicks(avg_subticks_per_sec, subticks_persec_upperlimit, subticks_persec_lowerlimit)
2:     $a \leftarrow$ avg_subticks_per_sec
3:     $u \leftarrow$ subticks_persec_upperlimit
4:     $l \leftarrow$ subticks_persec_lowerlimit
5:     $true\_tick\_dist \leftarrow 1$
6: new delayed tick:
7:     $d \leftarrow delayed\_subticks\_per\_sec$
8:     if $d > u$ then                                  ▷ true tick found: Condition 1
9:         $true\_tick\_dist \leftarrow 1$
10:    else if $d < u$ & $d > l$ then                  ▷ true tick passed
11:        $true\_tick\_dist \leftarrow true\_tick\_dist + 1$
12:    else if $d < l$ then                            ▷ true tick found: Condition 2
13:        $true\_tick\_dist \leftarrow 1$
14:        $d \leftarrow a$
15:    $n \leftarrow true\_tick\_dist$
16:    $t_{delay} \leftarrow \frac{\sum_1^n d}{a} - n$
17:    $t_{local\_shifted} \leftarrow t_{local} - t_{delay}$
18:    $t_{error} \leftarrow t_{local\_shifted} - SGXticks$
19:    goto new delayed tick
1: procedure Advance Time($t_{prev\_local}$, avg_count)
2:     $t_{error} \leftarrow procedure\{$Restore True SGX TICKS$\}$
3: new subtick:
4:     $t_{local} \leftarrow t_{prev\_local} + \frac{subticks}{avg\_count} - slew\{t_{error}\}$
5:     goto new subtick
```

t_{local} with true SGX ticks is to approximately locate these ticks. We make an observation that large delay variation gives rise to two conditions that are an indication of a delayed tick being close enough to true SGX tick. `Condition 1`, as shown in Figure 3.6, arises when *delayed_subticks_per_sec* is large enough for a delayed tick to be aligned with at least one of the true SGX ticks. Once a true tick is identified, we shift the delayed tick back to true tick by t_{delay}. This delay approximately equals *delayed_subticks_per_sec* minus the average of delayed subticks *avg_subtick_per_sec*. We can find the error in local time t_{error} by subtracting t_{delay} from t_{local} and shifting it to true SGX Tick at $t_{local\_shifted}$. This error helps slew the clock to true time. Figure 3.6b shows a scenario that gives rise to `Condition 2` of detecting a true SGX tick., This condition states if *delayed_subticks_per_sec* is small enough, the delayed tick overlaps the true tick within an error tolerance. In that case t_{delay} is zero and t_{error} would be the difference in t_{local} and true SGX tick. Algorithm 1 provides the details of how we calculate t_{delay} and t_{error} to overcome delay attacks and slew the clock towards true time.

Calculating the right t_{delay} value is critical to construct a delay-free clock. Algorithm 1 shows in detail how t_{delay} is calculated from different set of parameters. The range of delay variation helps select values for these parameters: *subticks_persec_upperlimit* determines how large *delayed_subticks_per_sec* should be to satisfy `Condition 1`, *subticks_persec_lowerlimit* satisfies `Condition 2`, whereas *avg_subticks_per_sec* is the mean of multiple *delayed_subticks_per_sec*. This mean provides a good approximation of true *subticks_per_sec*. The upper and lower limits parameters are adjusted based on the maximum, minimum, and average *delayed_subticks_per_sec*.

To align t_{local} with true SGX ticks, it is necessary to find a true tick and extrapolate time from there. As the occurrence of Condition 1 and 2 is not high, we maintain a parameter *true_tick_dist* that determines how many delayed ticks have elapsed since the last found true tick. In essence, it makes sure that t_{delay} is always calculated with respect to true tick. In reality, error in time gets accumulated due to inaccuracies in t_{delay} calculation with every passing delayed tick since the true tick. Therefore, the *true_tick_dist* value should not exceed an accumulated error threshold that deems the time unreliable.

3.6 Implementation and Evaluation

We provide a scalable TimeSeal implementation on a SGX enabled computer with an i7-6700K processor and 16 GiB memory running Ubuntu Linux 16.04.3, kernel version 4.10.32. The *sgx_get_trusted_time* API provides time from the hardware management engine. An application that wishes to acquire secure time instantiates TimeSeal within its own process to limit OS interactions and avoid delay attacks. Counting policies and their associated parameters, e.g., the number of threads, counting order, and count values, are maintained within the SGX process memory. To randomize the selection of threads, and count values every SGX sec, we use *sgx_read_rand* API to generate a true random

	Attack 1				**Attack 2**				**Attack 3**	
	Single Thread		N-1 Threads		Single Thread		N-1 Threads		50% c_d	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Policy A	$0.5\|\ 3$	$7\|\ 7$	$0.5\|\ 3$	$9\|\ 8.4$	$0.5\|\ 1.8$	$7\|\ 7$	$0.5\|\ 3$	$9\|\ 8.5$	$0.6\|\ 8$	$25\|\ 24$
Policy B	$0.002\|\ 3$	$6.6\|\ 5.2$	$0.068\|\ 3$	$9.6\|\ 8.5$	$0.076\|\ 1.5$	$6\|\ 5$	$0.05\|\ 0.6$	$10\|\ 7.4$	$0.08\|\ 9.4$	$22\|\ 24$
Policy C	$8\|\ 2.7$	$300\|\ 200$	$0.3\|\ 0.6$	$1500\|\ 1600$	$1\|\ 11$	$81\|\ 60$	$1\|\ 25$	$100\|\ 99$	$0.02\|\ 5$	$31\|\ 26$

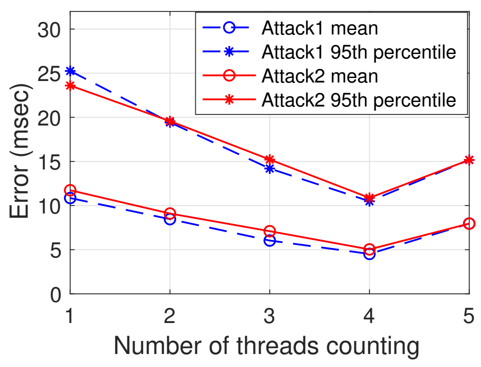
Table 3.2: Counting policies results for single and multiple-thread contexts, where μ and σ are the mean and standard deviation of induced error in msec. A "Single" Thread attack implies only one thread is scheduled out while an "(N-1) Threads" attack implies only one thread is counting at a time. Format of error is $\{error_{tick} \parallel error_{OS}\}$

number. Every thread increments its own counter based on the counting policy instead of incrementing a common global counter. This is to avoid reliance on OS mutual exclusion locks for race conditions. To manipulate thread count, OS may give two threads the lock at once or may not give a lock to any thread at all [PG08].

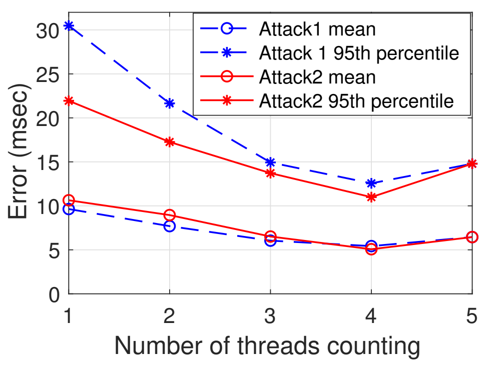
Evaluation metrics. We choose two evaluation metrics. One metric is the time difference of TimeSeal's high resolution secure clock with SGX trusted time at the boundary of a true SGX tick. We term this accumulated error per SGX tick as $error_{tick}$. It represents frequency error in TimeSeal's clock. Note that TimeSeal's clock is derived from SGX time by constantly polling it. The SGX time access latency is around 13 msec on average. Therefore, $error_{tick}$ would always have a standard deviation comparable to access latency.

The other evaluation metric compares TimeSeal's jitter with respect to OS time. We term it as $error_{OS}$. Considering an OS's monotonic clock as the ground truth for evaluation purposes, we generate small durations with respect to OS time. TimeSeal timestamps these durations and the resultant jitter indicates its stability w.r.t. OS's monotonic clock. Because the oscillators for OS's clock and TimeSeal's clock are different, there will be sub millisec level relative drift between both clocks. However, this drift is masked by millisec level access latencies.

Countering scheduling attacks. Table 3.2 provides a summary of TimeSeal's errors in the presence of scheduling attacks. We run different experiments with three threads (N = 3) counting under different policies (A, B, C) experiencing the three categories of scheduling attacks (1, 2, 3) under 50% system load. The performance of Policy B is comparable to Policy A in terms of empirical errors except that Policy B also decreases the

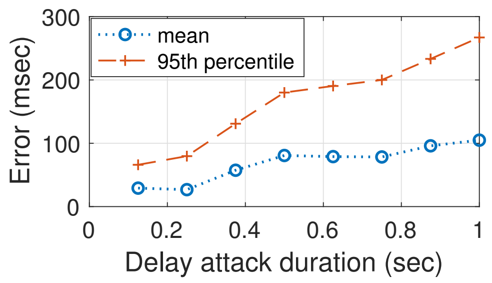


(a) TimeSeal error w.r.t OS time

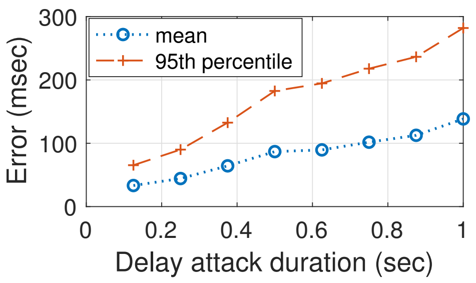


(b) TimeSeal error w.r.t SGX ticks

Figure 3.7: Scheduling attacks 1 and 2 aborts either 1, 2, 3, or 4 threads out of 5 counting threads. Policy B bounds the error to within tens of msec



(a) TimeSeal error w.r.t OS time

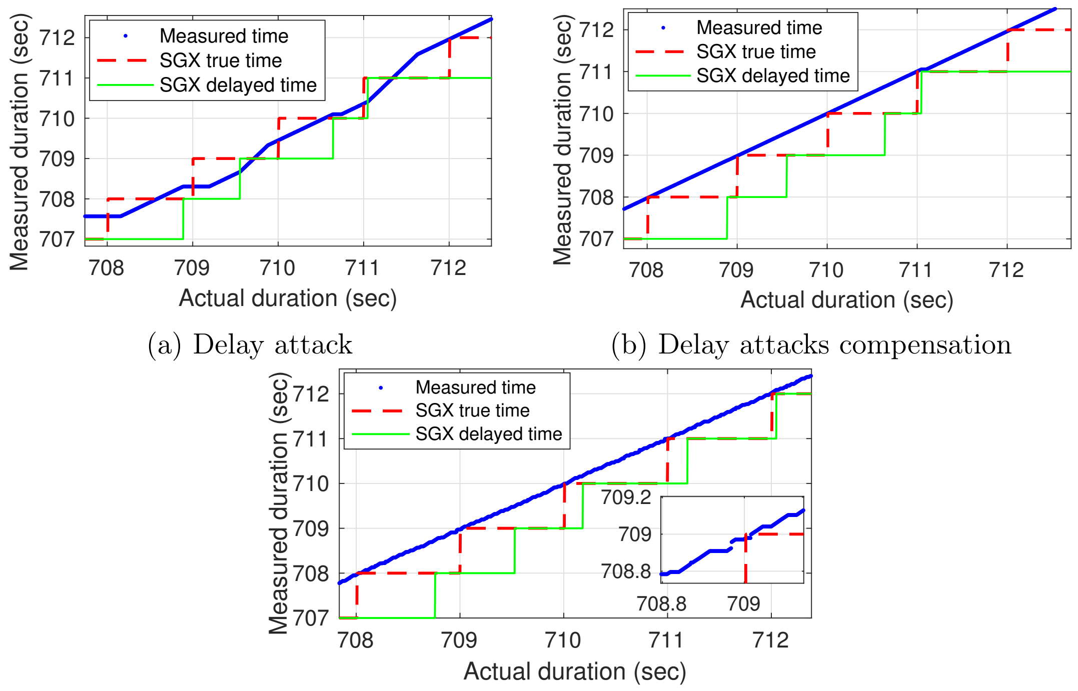


(b) TimeSeal error w.r.t SGX ticks

Figure 3.8: Delay attacks of different durations

attack's success probability. Policy C performs worse when few threads are allowed to run because of uneven count distribution among threads per sec. Lastly, Attack 3 degrades all policies equally because it causes a detectable denial of service for a duration it is launched in. Figure 3.7 shows the relationship between the number of threads and TimeSeal's errors. Using Policy B, we see a decrease in errors with an increase in number of threads because of small attack success probability. Also note that 95th percentile $error_{tick}$ for Attack2 is smaller as compared to Attack 1 because of its lower attack success probability as discussed in Section 3.5.1.

Countering delay attacks. Time error is directly proportional to delay attack duration. The more the SGX time packet is delayed, the more error an attacker can accumulate. We test different delay attack intervals ranging from 0 to 1sec under 50% system load. In Figure 3.8, although TimeSeal's error increases with an increase in delay duration, our delay mitigation technique bounds it to be within 100s of msec. For example, for a delay



(a) Delay attack (b) Delay attacks compensation

(c) Both attacks compensation

Figure 3.9: Time plot of true SGX, delayed SGX, and TimeSeal measured time. (a) Error in *subticks_per_sec* due to delayed SGX ticks affects TimeSeal's accuracy. (b) Delay attack mitigation technique, connects measured time to true SGX ticks. (c)Scheduling attacks on top of delay attacks decrease resolution as shown in zoomed-in plot

duration of 1sec, $error_{tick}$ has a 140msec mean and 342msec 95th percentile. $error_{OS}$ has 137msec mean with 356msec 95th percentile. We can also detect and bound delay attacks above 1sec by adjusting the subtick related parameters discussed in Section 3.5.2.

Overcoming scheduling and delay attacks. Figure 3.9 shows the effects of delay attacks and scheduling attacks on time plots. SGX true time advances every sec (red dashed line), delayed SGX time advances with variations around 1 sec (green solid line), and TimeSeal's clock advances with a msec resolution (blue dotted line). Delay attacks distort frequency of TimeSeal's clock such that it advances at a different rate every sec as shown in Figure 3.9a.

Our delay mitigation technique restores true SGX ticks, adjusts TimeSeal's frequency, and slews accumulated errors of delay attacks. By doing so, Figure 3.9b shows that the high resolution TimeSeal clock traces SGX ticks precisely and advances with a stable frequency. If an attacker also launches scheduling attacks on top of delaying SGX

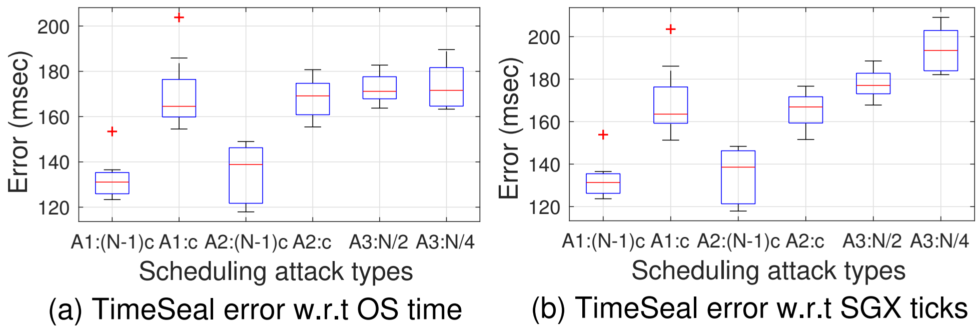


Figure 3.10: An attacker delays SGX packets by 1 sec and launches scheduling attacks 1, 2, 3. The mean error distribution for all attacks remain within 100s of msec

packets, our policies make sure that the error remains bounded to within 100s of msecs. Depending upon the scheduling attack type and c_d value per thread, Figure 3.9c's zoomed-in plot shows a decrease in TimeSeal's effective resolution. The wavy plot with small time discontinuities is a result of different threads counting at different times due to scheduling attacks. TimeSeal's resolution degradation is much less than the clock errors. Figure 3.10 presents the mean $error_{tick}$ and $error_{OS}$ distributions for different scheduling attack types and 1sec delay attacks. The mean of errors are a result of delay attacks while the interquartiles (iqr) are a result of scheduling attack. Note that fewer number of threads yield slightly large errors. For Attack 1 with N-1 threads counting ($A1 : (N-1)c$), the mean $error_{os}$ is 135msec with 11msec iqr, and mean $error_{tick}$ is 138msec with 9msec iqr. For Attack 1 with only one thread counting ($A1 : c$) the mean errors increase to 165msec with 15msec iqr for both errors.

System resources overhead. TimeSeal threads are not given a high priority and they are scheduled as normal threads. Systems under high load may give less CPU time to TimeSeal threads resulting in errors due to varying *subticks_per_sec* as shown in Figure 3.4. Scheduling attack achieves same degradation maliciously. Hence, we argue that our clock model and counting policies are equally resilient to high load scenarios and an attacker can't obfuscate an attack during high load.

For every application that needs secure time, TimeSeal's model of polling PSE for SGX time is similar to current SGX model. To enforce certain time based policies, SGX enclaves also poll PSE continuously to make sure that a certain duration has passed [sgx18].

Therefore, we argue that there is no bandwidth increase over current SGX use cases. However, although more counting threads under any scheduling policy decreases the probability of an attack's success, CPU usage increases. This implies a performance degradation after a certain number of threads. Figure 3.7 supports this claim where four threads provide the best performance, and degradation is minimal with decreasing threads. Therefore, an application can strike a balance among the number of threads, CPU usage, and required time error.

3.7 Discussion

The following are key points for TimeSeal's design.

TimeSeal placement: TimeSeal's components are enclosed inside application enclavesto avoid delay attacks on additional communication channels via the OS. This design requires applications to have their own instances of TimeSeal threads–increasing the CPU usage. One possibility to avoid delay attacks between PSE and TimeSeal is to provide a high resolution clock within PSE. A modified PSE, though, will not be able to use a secure channel with CSME because the root of trust isn't established by Intel servers [lin18a].

Attacks Validation. A diverse set of timers such as TSC, APIC, HPET, PIT are present in Intel architectures. These timers are controlled by the OS. The diversity of these timers give us the opportunity to query multiple timers to detect misbehaving OS. The OS however can remain undetected by adopting an attack strategy of consistently lying for all the timers. Moreover, the diverse OS timers cannot be leveraged against delay attacks on SGX trusted time as OS and SGX time sources cannot be correlated. In future, we can formulate a game theory based approach by giving time based challenges to OS that can restrict its attack strategies.

Agnostic to Synchronization Protocol. In our TimeSeal implementation, we do not necessitate the use of one synchronization protocol or the other. In fact we argue that any synchronization protocol or a filter in the time synchronization literature can be used in place of our linear regression filter.

TimeSeal for other TEE: TimeSeal's architecture and design principles are not dependent on one TEE. Though we used SGX, any TEE that provides access to a trusted timer can be considered. Exploring the possibility of enabling TimeSeal for ARM TrustZone is in consideration as it dominates the embedded market.

TimeSeal performance: The number of applications running TimeSeal is limited by the number of logical cores on a CPU. If a CPU has 8 cores, only 8 applications can run TimeSeal without being a victim of Attack 3 where all TimeSeal threads are scheduled out at the same time. However, we do quantify and provide bounds on time degradation for threads running only 50% or 75% of a sec in Figure 3.10.

Recommendations for vendors: To avoid complex and incomplete designs of secure clocks–which is still an open problem–we list a set of requirements for hardware vendors that fulfill all conditions of a secure clock to the best of our knowledge. First, there should exist a hardware timer with associated registers that no privileged hardware or software can write to. Second, this timer should be derived from a high frequency oscillator that should not be overclocked or under-clocked by a malicious software. Third, the timer should have a sufficient number of bits–preferably 64 bits–so that it never overflows. Finally, access to the timer value should be securely memory mapped for fast access and independent of OS manipulation and delays. These requirements can only be fulfilled by a hardware vendor.

3.8 Key Findings

In this work, we have shown that time can be attacked at all layers of a time stack; via hardware timers, timekeeping software, and time transfer network packets. We also demonstrate the time in trusted execution technologies such as Intel SGX and ARM TrustZone can also be attacked. Exploiting vulnerabilities in the OS, we are able to launch a delay attack on SGX trusted time and accumulate a timing error of multiple seconds. Another demonstrated attack is scheduling attack also with an error accumulation of multiple seconds. We thwart both these attacks via TimeSeal design and provide a high-resolution secure clock on commodity TEE with 10's of milliseconds accuracy in the presence of delay and scheduling attacks.

3.9 Conclusion

Securing time in an untrusted OS is a challenge. We present TimeSeal, a new secure time architecture that leverages TEE for hardware timer protection and eliminates timing limitations and vulnerabilities in TEE to secure time. TimeSeal provides a local secure clock that is good for measuring time durations. There are a plethora of applications that require secure global time [ZCC16] [CDE13b] [DR17]. Researchers have addressed global time security by protecting time transfer packets in the network. This is an orthogonal area of research, where we synchronize TimeSeal to global time. TimeSeal protects time or in other words “seal" time so that no privileged adversary can arbitrarily change the notion of time, and compromise safety and performance of applications.

CHAPTER 4

A Case for Feedforward Control with Feedback Trim to Mitigate Time Transfer Attacks

4.1 Introduction

Time Transfer is a way of sharing reference time among physically or geographically separated entities. A shared sense of time is critical for many applications such as to correlate observations in astronomical [AAA19] and financial [PH16] systems, coordinate tasks between cell towers [HSA11], and choreograph acts among autonomous agents [KSB14]. Widely used time transfer systems either rely on *one-way* packets or a *two-way* packets exchange. In a one-way time transfer system, a server/master sends its current time over a network to multiple clients. This technique is simple yet its accuracy suffers from uncompensated propagation delays. Global Positioning System (GPS) [AW80] is an example of one-way time transfer system with tightly calibrated delays. In a two-way time transfer systems, both master and a client exchange their current time with each other. The four timestamp measurements from two packets help calculate round trip delays. A major drawback of two-way packet exchange is the assumption that delays are symmetric in both directions. In reality, these delays are asymmetric and translate to clock errors. Clock synchronization protocols make use of two-way time transfer to calculate delays and clock offsets. These offsets are fed to a feedback controller that adjusts a clock to compensate for the offset.

4.1.1 Time Transfer Attacks

Network elements that assist time transfer can be malicious. They can develop various Man in the middle (Mitm) capabilities such as dropping, replaying, pre-playing, and delaying packets. Prior works [MDA16] [YAY13] [DSD18] talk about attacks on time transfer packets and mitigating these attacks using cryptographic [AFZ17a] [IW17] and network security [sgx16] [AFZ17b] mechanisms. Unfortunately, delay attacks are considered too strong and immune to proposed mitigations [UV09] [MVV17]. Researchers have attempted to secure systems against delay attacks under many assumptions. Widely used assumptions relate to the availability of excess or redundant information in terms of (1) multiple masters providing reference time, (2) multiple communication paths to those masters, and (3) at least two-thirds of those paths to not be compromised. These assumptions are not realistic in the presence of a malicious gateway router that can potentially delay 'all' packets [DSD18]. There exist numerous synchronization architectures where redundant resources come at a cost [Sym09]. The ultimate goal of a Mitm attacker is to move the client's clock away from true time towards its malicious time for illegal activities in high-frequency trading [PH16], digital rights violation [CRS14], and spoofing location [SP05], etc.

Prior work has identified two key problems in current synchronization protocols that make them vulnerable to delay attacks. First, an attacker can induce large asymmetric delays by delaying one packet in a two-way time transfer [MVV17] [DSD18]. Second, a feedback control based clock adjustment mechanism amplifies the effects of delay asymmetry by feeding the error back to the controller and steering the clock towards inaccurate time.

4.1.2 Contributions

It is already well known that one-way packets with a feedforward control can syntonize clocks, however in this paper, we make a case that *one-way* time transfer and a *feedforward* controller can achieve clock *syntonization*[1] even in the presence of delay attacks. Syntonized

[1]The process of aligning frequencies of different clocks is called *syntonization*

clocks help systems perform various operations such as measuring precise inter-event times in localization [BHE00], residency delays in high-speed networks [GLY18], and execution times for code profiling [CZR17]. We highlighted a key advantage of a feedforward controller to compensate for delay attacks i.e, its ability to calculate relative frequency even in the presence of *constant delays*. Feedforward control leverages the property that *equally delayed* packets cannot give correct instantaneous offset, but they do preserve the rate of change in offset i.e., the relative frequency of the two clocks.

In contrast to feedback control, *feedforward* does not rely on the instantaneous time offset. Instead, it monitors the “rate of change" in this offset over multiple one-way time transfer packets to calculate relative frequency error. A feedforward controller uses timestamps from a free-running clock that is never adjusted. This is because the relative frequency of two clocks is a property of their hardware oscillators. Comparing timestamps of free running clocks derived from these oscillators gives the best estimate of relative clock frequency.

We make an observation that if two packets are received at the same interval at which they are transmitted, they either experience no delay or equal delays in the network. An adversary may choose not to delay packets equally and degrade the feedforward controller's performance by injecting random delays. In this case, the received packets do not arrive at the same interval at which they are transmitted, hence they experience unequal delays. Leveraging this observation and the fact that feedforward control calculates accurate relative frequency only for equally delayed/periodic packets, we propose a technique that *restores periodicity* among unequally delayed packets. Given that a master transmits packets periodically – which is the generalized case – we are able to filter received packets and transform their timestamps to emulate received periodic packets under the right conditions. Once we *restore periodicity* of the delayed one-way packets, we feed the transformed timestamps of the emulated packets to a feedforward controller to calculate the relative frequency error.

After aligning frequencies of different clocks in a secure manner, we present a secure clock *synchronization* approach in the presence of delay attacks. To synchronize two clocks, we cannot rely on a feedforward controller as it only calculates relative frequency using raw timestamps. We need to calculate the accurate clock offset i.e, the error between a reference time and the adjusted time. Unfortunately, delay attacks make it harder to calculate the exact offset. Therefore, we propose a packet filtering technique that looks for delay patterns in consecutive packets to search for minimally delayed or, as we call them, *delay-free* packets. These delay-free packets provide a good estimate of the clock offset. Rather than directly adjusting a clock with this offset, we design a *frequency shaper* that uses this offset to trim the relative frequency calculated by the feedforward controller. The resultant trimmed frequency is used to synchronize the clock. In short, we calculate the offset in a feedback loop and utilize it to shape the relative frequency for clock adjustment. We call this clock adjustment mechanism *feedforward control with feedback trim.*

4.1.3 Advantages

Our proposed approach of feedforward control with feedback trim can syntonize clocks for constant delays, and synchronize clocks in the presence of random delays and other delay patterns. It comes with an additional benefit of reduced bandwidth as we mostly rely on one-way time transfer with occasional reliance on two-way time transfer. This is in contrast to traditional synchronization protocols that rely on two-way time transfer. Our standalone software implementation can easily be integrated into various synchronization protocols with minimal code changes. For proof of concept and to support one synchronization protocol, we implement our packet filtering techniques along with feedforward controller with feedback trim for Precision Time Protocol (PTP) [LEW05] and evaluate our system on a real embedded platform [Bla] used in many clock synchronization applications. Our evaluation shows promise and comparable performance in reference to systems that are not under attack.

To summarize our contributions,

- We present a detailed experimental analysis of how different delay patterns affect feedback and feedforward control based clock synchronization.

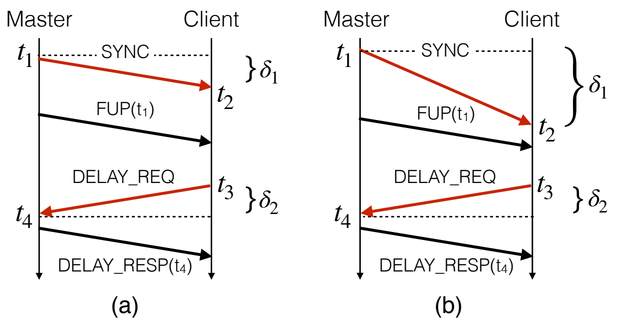


Figure 4.1: (a) Symmetric delays in forward and reverse direction ($\delta_1 = \delta_2$), (b) Asymmetric delays in both directions ($\delta_1 > \delta_2$)

- We identify a key property of equally delayed packets. These packets preserve the relative frequency between two clocks. Hence we leverage equally delayed packets to syntonize clocks with feedforward control.
- We propose two packets filtering techniques: one to *restore periodicity* of variably delayed packets to calculate relative frequency, and the other to find *delay-free packets* to calculate offset.
- We design a secure clock synchronization architecture under delay attacks. It consists of a *feedforward control with feedback trim* based clock adjustment mechanism and relies on a *frequency shaper* that trims the relative frequency based on the offset.
- Finally, we evaluate our proposed architecture on a hardware-supported testbed and support new feedforward controllers for widely used precision time protocol.

4.2 Background

4.2.1 Time Synchronization Basics

In any time synchronization protocol, there is always a disciplinable clock and a `timestamping` mechanism that captures event times from that clock. Packets exchanged in the network act as events to be timestamped. These packets can either be timestamped in software or hardware depending upon the desired synchronization accuracy and/or the availability of hardware timestamping in a given platform. A time synchronization protocol relies on one-way or two-way time transfer packets to align a clock with a reference time. To explain a time synchronization protocol, lets take an example of Precision Time Protocol (PTP) [LEW05] that is widely used in high precision measurement and control applications.

The packet exchange for PTP is shown in Figure 4.1. Master node sends the `SYNC` message at t_1 and clients receive it at t_2. The actual value of t_1 is sent to client in a follow up (`FUP`) message. The time offset of two clocks are, $offset = t_2 - t_1 - delay$. To determine this $delay$, the client sends a delay request (`DELAY_REQ`) message at t_3 that reaches the master at t_4. The master sends the timestamp t_4 to client in a response message (`DELAY_RESP`). In PTP, the timeliness of SYNC and DELAY_REQ message is important as their timestamps are used to determine clock synchronization parameters. The calculated $delay = (t_2 - t_3 + t_4 - t_1)/2$ is round trip delay divided into two symmetric delays. The delay in SYNC message to reach the client is termed as forward path delay (δ_1), whereas the delay that DELAY_REQ message incurs is the reverse path delay (δ_2). PTP like other protocols also assumes that both forward and reverse path delays are equal i.e. symmetric with a very small error margin as shown in Figure 4.1(a). If an attacker launches a delay attack either on the forward or the reverse path, the delays are no longer symmetric as shown in Figure 4.1(b), and the quality of time synchronization is degraded.

4.2.2 Time Transfer Attacks and Mitigations

There are many ways for an adversary in a network to attack time transfer packets and degrade the accuracy of a time synchronization protocol. It can launch a `removal attack` that selectively drops the synchronization packets before they reach the receiver. Note that not all packets are dropped so as not to cause a DOS attack. `Replay attack` records previous synchronization packets and replay them at a later time providing inaccurate time information to the receiver. `Pre-play attack` injects new messages in the network and tricks the receiver into believing that the newly forged messages are from a legitimate sender. `Substitution attack` replaces time value inside a valid packet with a new value. Finally, `delay attack` simply holds the packet for some time and send it later. Various cryptographic techniques protect against pre-play and substitution attacks by verifying that the messages have indeed been sent by a legitimate sender [AFZ17a] [IW17]. A replay attack is thwarted by checking the freshness of a packet by observing a monotonically increasing sequence number [sgx16] [AFZ17b].

Delay attack however is considered too powerful to be completely mitigated [UV09] [MVV17]. Cryptographic and network security mechanisms are insufficient to detect and estimate delay attacks. Some attempts have been made to bound these attacks [MDA16] [YAY13]. Most of these techniques rely on the diversity in servers that provide reference time [DSD18], and the paths towards them [Miz12] [KB18] to bound delay attacks, while assuming that not all paths to these servers are attacked. The assumption of a communication architecture with multiple servers per client is valid for a Network Time Protocol (NTP) [Mil91]. However, all paths to those servers are compromised if a single gateway router is malicious and delays all NTP traffic. In PTP however the availability of multiple masters is not realistic. PTP networks rely on few grandmasters due to cost issues and utilize switches that contain boundary clocks and transparent clocks to scale the network [Sym09]. In this scenario, a single compromised switch can delay all the packets to PTP clients.

Prior works tried to calibrate round trip delays of different paths in the network. During time synchronization, the two-way time transfer is not considered secure if the packets exceed the expected round trip delay [SNW06]. Annessi et al. have tried to give upper bounds on delay by modeling the clock drift [AFZ17b]. The offline modeling based approaches have conservative bounds and give a smart attacker sufficient slack to launch attacks and remain unnoticed. Narula et al. [NH18] presents a theory to assess security of protocols in a generic setting. Their necessary and sufficient conditions give an idea of how to prove a protocol's security. By doing so, they prove that PTP [LEW05] is not secure. In securing PTP, their assumptions of communicating over line of sight channel or shortest possible path are not realistic. They also give a necessary condition of bounding delays by estimating round trip delays a priori. Distance bounding protocols [SP05] leverage the same idea for bounding distances but there are numerous attacks [HK08] [ABG17] possible on these protocols.

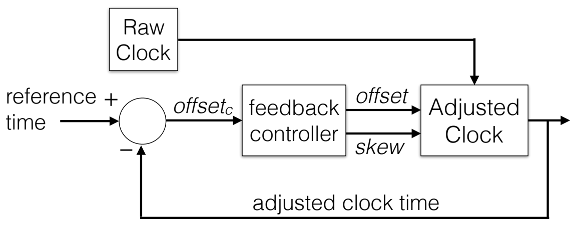
Researchers have also correlated different sensing modalities to timing signals. Dima et al. [RTY17] secured NTP communication against delay attacks by detecting and estimating path asymmetries through the use of power grid voltages. Their solution requires specialized hardware and only works for grid-connected distributed systems. Hence making it unsuitable for embedded and wireless sensor networks applications. Seismic deployments are also used to recover temporal integrity [LDC09]. The availability of seismic modality is not widespread and its accuracy is sub-seconds that is insufficient for most applications.

4.2.3 Why clock synchronization protocols are vulnerable to delay attacks

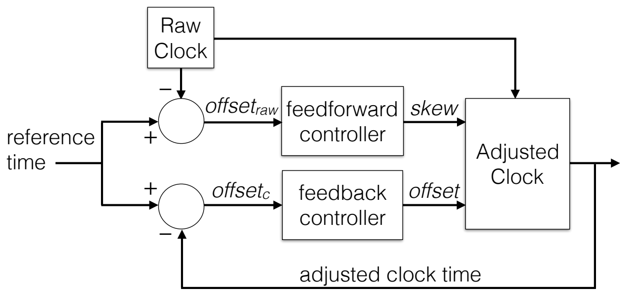
There are two key problems in a clock synchronization architecture that an attacker exploits to manipulate the time,

1. Clock synchronization protocols calculate offset between two clocks by utilizing near-symmetric delays in a two-way time transfer. Both network variations and adversarial components in the network can violate the symmetric delay assumption in synchronization protocols. An attacker delays one of the two packets in a two-way time transfer to achieve desired delay asymmetry. This asymmetry directly translates to a clock error.

2. In a synchronization protocol, a controller adjusts a clock by aligning it to a reference clock both in time and frequency. Most protocols use a feedback controller as shown in Figure 4.2a. This controller either jumps time by a large $offset$ or it uses a small offset to tune the frequency of a clock called $skew$. In a closed loop feedback control, next measurements are affected by the previous adjustments. In case of an attack, adjustments are based on offsets calculated from asymmetric delays. These adjustments affect the next measurements, thus accumulating clock errors.



(a) Feedback control based clock adjustment



(b) Feedforward and feedback control based adjustment

Figure 4.2: Global clock synchronization based on different clock adjustment mechanisms

4.3 Feedforward Control

In contrast to feedback control in current clock synchronization protocols, we propose a feedforward control for frequency correction in the presence of delay attacks. Unlike feedback, it decouples the time and frequency calculation so that the error in one does not affect the other. A comparison of feedback and feedforward based clock adjustment is shown in Figure 4.2b. In a feedforward system, the control is not driven by an error in the output. Instead, it is based on the knowledge of the process. In context of clocks, the process knowledge we use is that all clocks are derived from hardware oscillators that tick at a nominal frequency. Due to manufacturing variations, the oscillators drift from their nominal frequency. These frequency drifts are unique to a hardware oscillator. A free running counter or a clock is directly impacted by this frequency drift. Comparing timestamps of two free running clocks gives an estimate of their relative frequency drift. This is termed as frequency error in context of a clock synchronization protocol. Feedforward control directly compares the timestamps of two free running / raw clocks to get an accurate estimate of relative frequency error. It uses this relative frequency to adjust the clock. Thus a feedforward control is able to *syntonize* clocks i.e. aligns the frequency of two clocks. An accurate estimate of this frequency reduces the need to synchronize often thus saving network bandwidth. But the accuracy of feedforward control depends on how accurate our process model is.

A feedback control continuously exchange messages to calculate and adjust the offsets.

Frequently adjusting offsets indirectly compensates for the relative frequency error, but it comes at a cost of increased bandwidth. Any error in offset affects the relative frequency as well. Note that the feedback control sample timestamps from an adjusted clock to calculate the offset, whereas a feedforward control gathers timestamps from a free running clock to calculate relative frequency. Feedforward control also bypasses various sources of error in an adjusted clock due to noisy measurements and asymmetric delays.

RADclock [RVB12] was the first to introduce a feedforward clock model. The difference between RADclock and our architecture is that RADclock provides a feedforward clock model [BRV09], whereas we propose a feedforward controller to adjust a general clock model. RADclock derives frequency and time adjustments from two-way time transfer and feedback controllers to adjust their feedforward clock. In our case, we use feedforward controllers to calculate clock adjustments for the regular clock models in the kernel. Unlike RADclock, our feedforward controller does not rely on a specific hardware functionality. It can also be integrated with any clock synchronization protocol and work with any clock model with minimal code changes.

We implement a feedforward controller for PTP, and switch from the regular PTP feedback controller to this feedforward controller. Using multiple timestamps of a local free running clock with respect to the reference clock over the passage of time, we calculate the rate of change in clock errors. This rate of change determines how fast or how slow the local clock should tick to be aligned with the reference clock. Once the relative frequency is known, feedforward control can switch to infrequent packets exchange. One disadvantage of feedforward control is that it cannot handle disturbances or transient events.

4.4 Inadequacy of the Status Quo

Before designing a feedforward based delay-tolerant architecture, an important step is to analyze the performance and compare the behaviors of feedforward and feedback controllers under attacks.

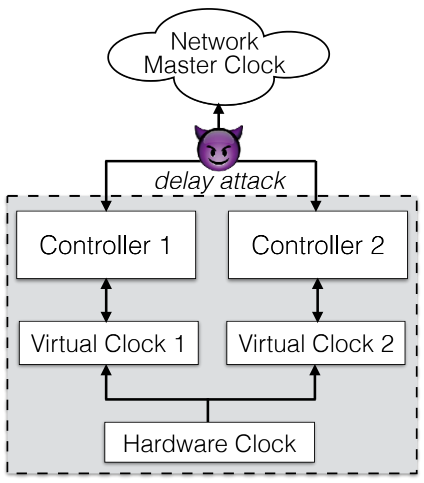
4.4.1 Experimental Methodology

We introduce our experimental methodology early on to use it in our experimental analysis and later on in evaluation.
The basic setup is shown in Figure 4.3a. It incorporates our own implementation of virtual clocks derived from the same timing hardware on a single platform to alleviate performance bias due to hardware variabilities. Each virtual clock is disciplined by a separate controller. Time transfer packets flow between the master and the client for synchronization purpose, typically with a poll period of half a second. These packets are delayed in the network before reaching the client. Then the same network traffic is fed to all the controllers to remove the effects of network variations.

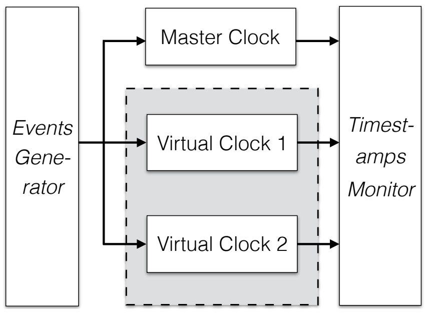
In these experiments, we intend to compare the performance of a feedback control with a feedforward control under delay attacks. Though our feedforward controller is generalized and applicable to a wide variety of synchronization protocols, we choose PTP for our experiments as it is widely used in many applications. PTP provides a set of feedback controllers to choose from. We compare the commonly used Proportional Integral (PI) controller with our implementation of a linear regression based feedforward controller. It is essential to note that we use hardware timestamping capability to timestamp incoming and outgoing packets.

For evaluation purposes, a separate box shown in Figure 4.3b generates events at the rate of 8 events per second. These events act as common timestamping opportunities for multiple virtual clocks under test. A monitor analyses the timestamps in the following two ways,
1. Frequency stability: Finds timestamping jitter in all clocks to compare relative frequency error of these clocks with respect to the master clock.
2. Time accuracy: Finds the absolute time accuracy of virtual clocks with respect to the master.

All the experiments for synchronization and validation are run concurrently so that clocks to be compared experience identical conditions, and show a real time performance.

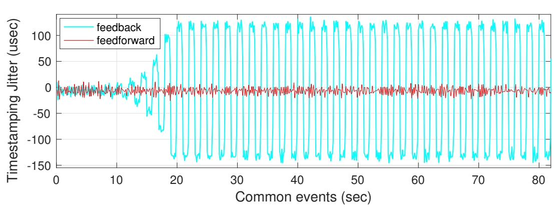


(a) Testbed to synchronize multiple clocks on a single platform

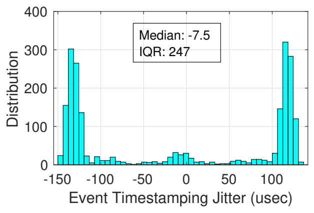


(b) Testbed to verify synchronization accuracy of multiple clocks on a single platform

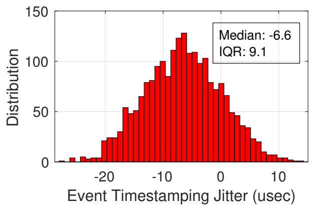
Figure 4.3: (a) A single platform testbed to synchronize virtual clocks to a master in the network. Each virtual clock is disciplined by a separate controller. (b) Once the virtual clocks are disciplined, we test their accuracy with respect to master by supplying common events to all clocks with an event generator. All clocks timestamp the same events, and a monitor analyses these timestamps.



(a) Feedback versus Feedforward clock jitter



(b) Feedback controller



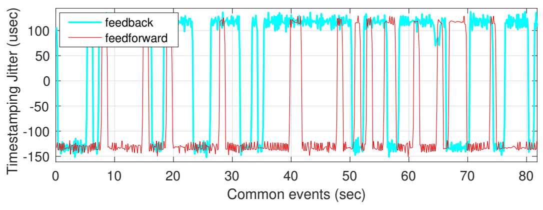
(c) Feedforward controller

Figure 4.4: Periodic events occurring every 125 milliseconds are timestamped by two clocks, where one clock is adjusted by feedback control and the other clock with feedforward control. Both of these clocks experience forward path delays of 1 second. (a) feedback clock jitter gradually oscillates to a steady state whereas feedforward clock jitter is smooth from the start. (b) Timestamping jitter distribution for feedback based clock sync varies in the order of 247 μseconds, whereas (c) jitter distribution for feedforward based clock is only 9μseconds

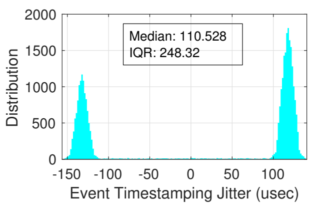
Timestamping Jitter under constant delay attack: We first run an experiment where we constantly add one second delays to all one-way SYNC packets in PTP. This induces asymmetry of 1 second, and an error of half a second in offset estimation. We synchronize two virtual clocks on the same platform, one with feedback and the other with feedforward control. Then generate periodic events separated by 125 milliseconds and use both clocks to timestamp these periodic events. Using these timestamps, we calculate the interval between consecutive events. Note that this interval should ideally be 125 milliseconds but due to errors in relative frequency calculation, there is a jitter in interval measurements. We plot the timestamping jitter for both clocks. Nodes only need to align their frequencies to measure fine inter-event times. Our results show that the frequency error calculation for our feedforward clock has less variations under constant delay attack as shown in Figure 4.4c because feedforward control is indifferent to asymmetric delay. Whereas, feedback clock utilizes asymmetric delays to calculate the offset and then use this offset to calculate frequency error. This results in high jitter as shown in Figure 4.4b. Figure 4.4a shows how the jitter in both clocks varies over time.

Timestamping Jitter under random delay attacks: In reality, an adversary can delay synchronization packets both in forward and reverse paths by any value. To illustrate the effect of a random attacker, we imitate an adversary that delays packets by a random value from a uniform distribution of 0 to 1 seconds. We synchronize one clock with feedback control and the other with feedforward control using SYNC packets delayed randomly in the network. In a separate process, we repeat the same validation mechanism; the two clocks timestamp 125 milliseconds periodic events from an event generator. We plot the clock jitter in Figure 4.5a. There is a jitter in the order of sub-milliseconds both for feedback and feedforward clock. The jitter distributions in Figure 4.5b and 4.5c are comparable. The Inter Quartile Range (IQR) for both clock's jitter is around 250 μseconds. We conclude that the feedback has similar clock jitter behavior for constant and random delays, whereas the feedforward clock performs worse in the presence of random attacks.

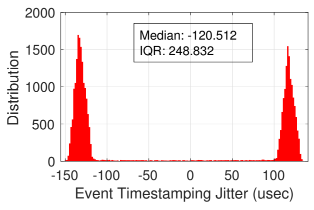
Synchronization error under constant/random delay attacks: We also compare the synchronization error of the feedback and feedforward clocks. Again the master



(a) Feedback versus Feedforward clock jitter



(b) Feedback controller

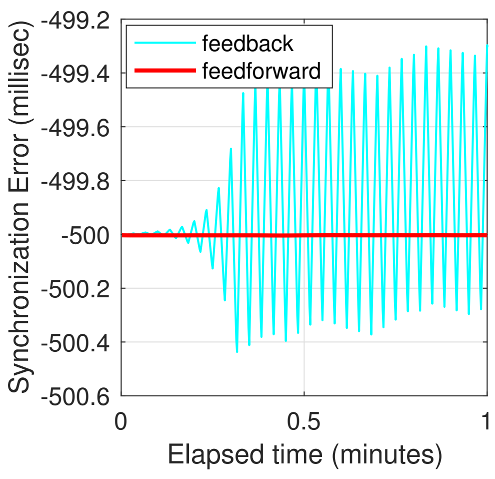


(c) Feedforward controller

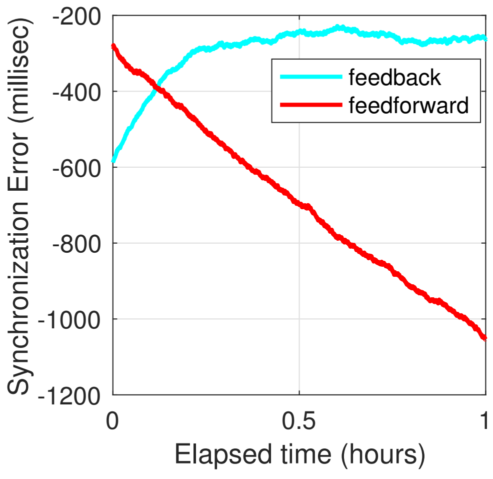
Figure 4.5: Clock under random delay attack from 0 to 1 second. (a) Timestamping jitter of both feedback and feedforward clocks have large variations in the range of -150 μsec to 150 μsec. The jitter distributions are shown in (b) and (c) with approximately 250 μsec IQR.

and the client clocks timestamp periodic events. The errors are shown in Figure 4.6. The synchronization errors in Figure 4.6a are influenced by constant delay attacks. Note that the feedforward clock error is almost constant at -0.5 second with little fluctuations that are hard to visualize in comparison to large error variations for feedback clock. The variations in feedback clock are in the range of 800 μseconds. We conclude that a constant delay attack only adds a large offset to a feedforward clock with no effect on the clock's frequency. Whereas, feedback clock is affected both in terms of an offset and large peak oscillations due to frequency error. In Figure 4.6b however, we see that random delay attack largely influences the open loop relative frequency calculation in a feedforward controller. The random delay gradually steers the frequency in one direction and the error of feedforward clock never converges. On the other hand, a feedback clock gradually accumulates error and reaches a steady state value of one fourth of the maximum random attacks. It oscillates in the order of milliseconds after reaching the steady state. This gives us an idea that feedforward works well for constant attacks, but it performs far worse under random attacks.

Delaying both forward and reverse path packets with equal delays in both directions is not an attack because in this case the delay is symmetric. Feedback clock error however takes longer time to converge in the presence of high symmetric delays, while feedforward clock error quickly converges. Another delay attack model is to gradually increase delay to a maximum attack value and then gradually decrease the delay to a minimum value. The



(a) Constant delay attacks



(b) Random delay attacks

Figure 4.6: Clock errors of feedback and feedforward based clock synchronization under different attacks

behavior of both feedback and feedforward control is quite predictable in this scenario. For both controllers, the clock error gradually increases to a maximum offset and then decreases to zero. They both exhibit oscillating behavior with a period that depends on the rate of change of delay.

Through our experimental analysis, we came to a conclusion that an attacker can cause maximum damage by first launching a constant delay attack and then switching to random delays. With this attack, both feedback and feedforward controllers jump to a clock error that equals half the initial delay value and then oscillate at a maximum frequency due to random delays. In case of feedforward clock, the error never converges.

Observations: We make three key observations after analyzing the behavior of both controllers in the presence of different attacks,

1. The frequency error calculated by the feedforward controller is unaffected by constant delay attacks. This gives us an intuition that we can *syntonize* clocks with a feedforward controller if consecutive packets are equally delayed.

2. The problem of delay asymmetry does not affect one-way clock syntonization. But delay asymmetry greatly influences two-way clock synchronization.

3. Both feedback and feedforward controllers accumulate clock errors under different attacks. The only way to reliably calculate offset between two clocks is through one-way packets that do not experience delays, or through two-way packets that are delayed by the same value in both directions.

4.5 Delay Attack-Tolerant Architecture for Clock Synchronization

After observing the behavior of controllers under delay attacks, we first state our threat model. Then leverage some of the observations we made in our experimental analysis to put forward a delay attack-tolerant clock synchronization architecture for systems under time transfer attacks.

4.5.1 Threat Model

Delay attacks are considered too strong to protect against [MVV17]. Prioir work [DSD18] offered delay attacks mitigations for NTP with a weak assumption that only one-third of the time servers and network links are compromised. We argue that an adversary sitting on a gateway router can easily delay or attack all the packets for a client. In our work, we consider a much stronger threat model, where all the links between clients and master can be compromised. In other words, an attacker sitting on a network element is capable of delaying all the packets by any value. A practical assumption however is the attacker delays packets by a finite value to avoid detectable denial of service.

Clock synchronization protocols employ sanity checks through prior knowledge of the environment, network and physical clock characteristics to reject outliers that exceed the delay bounds. These bounds are conservative and gives attacker enough slack to launch delay attacks. A smart attacker can significantly alter the time value within the specified bounds. In our threat model, an attacker can delay packets by a value smaller or greater than the synchronization period. Small, large, incremental, or distribution based delay variations with prior knowledge of the network delay are in scope.

In this work, we trust the master or any entity that provides the reference time to the system. In many industrial systems, the nodes that provide reference time are mostly considered secure. Most of the servers are equipped with GPS modules that provide reference time in many clock synchronization protocols such as NTP, and PTP. Any GPS spoofing attack is not in our scope. We do not consider any hardware clock manipulation both at the master and the clients. Prior works have addressed hardware based protections [sgx16] [RSW16]. Our threat model also excludes those network attacks that are mitigated by cryptographic techniques. We are interested in addressing the most powerful attack considered in clock synchronization literature; a delay attack that is immune to any cryptographic or network security mechanisms.

4.5.2 Overview

The delay attack-tolerant clock synchronization architecture along with its major components is shown in Figure 4.7. Our architecture is designed for a *feedforward* controller that aligns clock frequencies for equally delayed packets. As not all packets are equally delayed, we put forward a packet processing technique that *restores periodicity* of variably delayed packets such that the feedforward controller can better estimate the relative frequency error. As our focus is also to synchronize clocks in the presence of attacks, we search for a property in *consecutive one-way* packets that can safely declare them *delay-free.* These delay-free packets provide good offset estimates that are used by a *frequency shaper* to trim the relative frequency. We refer to it as a *feedback trim* to the feedforward calculated frequency. Finally the trimmed frequency is used to steer the clock towards true time.

We make the following key contributions to bound the effect of delay attacks on clock synchronization protocols,

- **Restore periodicity of delayed one-way packets** under certain conditions with a frequency tolerance. These transformed periodic packets at the client are assumed to be equally delayed in the network.

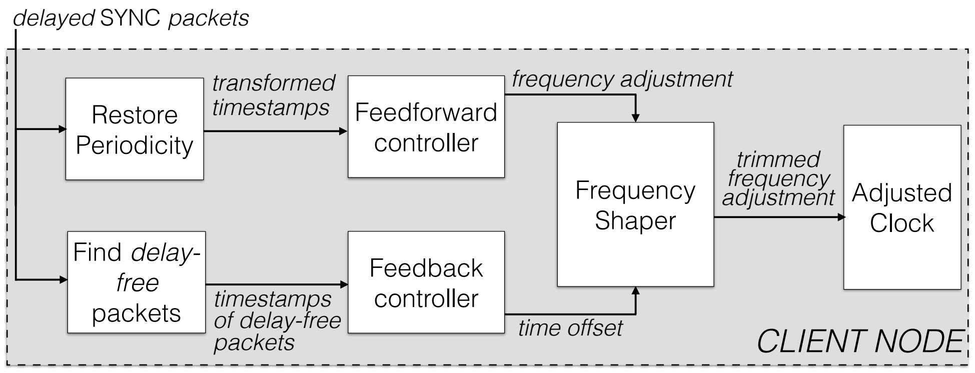


Figure 4.7: Delay attack-tolerant clock synchronization architecture

- **Implement a feedforward controller** that leverages the transformed one-way packets to calculate the relative frequency error with respect to the master. This frequency error is used to adjust the client clock's frequency.
- **Find delay-free packets** or packets that experience less delays in the network to calculate the time offset between the clocks. The occurrence of these packets depend upon the maximum delay an attacker is capable of launching.
- **Design a feedback trim** to synchronize clocks. The feedforward calculated frequency is trimmed based on clock offset to compensate for the accumulated error due to inaccuracies of feedforward controller and the error in finding delay-free packets.

4.5.3 Design

We now explain the detailed system design.

4.5.3.1 Restore one-way packets periodicity

Periodic one-way packets from the master are delayed in the network by an adversary. Our approach transforms the receive timestamps of these packets in a way that restores their original periodicity. Maintaining original period of one-way packets does not reflect zero delays in the network, rather it emulates equal delays for all the packets.

A master sends one-way packets with a predefined period. If two packets are sent with a time interval of s seconds, upon receiving the packets we mark those packets valid for our clock synchronization if they satisfy one of the three conditions shown in Figure 4.8. Packet a is sent before packet b, and both are s seconds apart.

Algorithm 2 Restore Periodicity

```
 1: procedure TRANSFORM TIMESTAMPS(sync_period, tolerance)
 2:     $s \leftarrow$ sync_period
 3:     $\epsilon \leftarrow$ tolerance
 4: sample new packet:
 5:     $t_a \leftarrow t_{prev\_pkt}$                  ▷ Timestamp of previous packet
 6:     $t_b \leftarrow t_{new\_pkt}$                   ▷ Timestamp of latest packet
 7:     $inter\_pkt\_dist \leftarrow t_b - t_a$
 8:     if $inter\_pkt\_dist > 0$ then                  ▷ In order packets
 9:         if $inter\_pkt\_dist = s \pm \epsilon$ then   ▷ Condition I
10:             $t_a \leftarrow t_a$
11:             $t_b \leftarrow t_b$
12:         if $inter\_pkt\_dist \leq \epsilon$ then      ▷ Condition II
13:             $t_a \leftarrow t_a$
14:             $t_b \leftarrow t_b + s$
15:         if $inter\_pkt\_dist = ns \pm \epsilon$ then  ▷ Condition III
16:             $t_a \leftarrow t_a + (n-1)s$
17:             $t_b \leftarrow t_b$
18:     goto sample new packet.
```

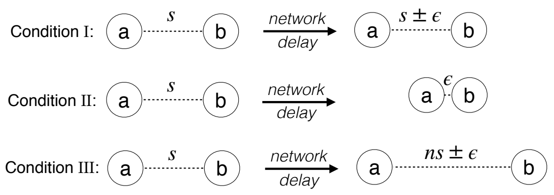


Figure 4.8: Two packets sent at a predefined interval of s seconds experience different delays in the network. Packet a and packet b may incur same delay with a small tolerance ϵ as shown in Condition I. Packet a could be delayed s seconds more than packet b such that both packets are only ϵ seconds apart from each other as shown in Condition II. Packet b could be delayed s times more than packet a such that they have large delays between them as shown in Condition III

1. *Condition* I: If the received packets arrive at the client in order and they are s seconds apart within some tolerance ϵ, then they hold the periodicity property and do not need to be transformed.

2. *Condition* II: If the received packets arrive in order and the time interval between them is negligibly small then packet a is delayed s seconds more than packet b. In this case, we transform the receive timestamp of packet b such that it reflects the same delay as packet a and both packets remain periodic.

3. *Condition* III: If the received packets are in order and their inter-packet time is much higher than the actual period, then it can be safely said that packet b is delayed multiples of s times the packet a. We add the same amount of delay to packet a timestamp to align it with the desired period.

Refer to Algorithm 2 for the steps to restore periodicity among received packets by transforming their timestamps. The *sync_period* and *tolerance* are the input parameters

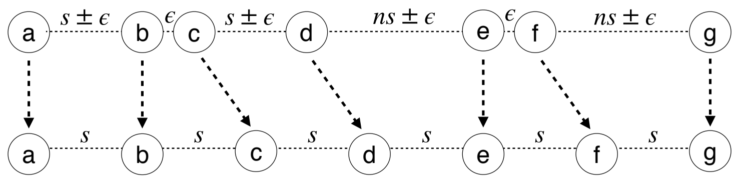


Figure 4.9: An illustrated example of receive timestamps transformation to keep the packets periodic

that can be tuned to achieve desired synchronization performance. When a new packet is received, its receive timestamp is compared with the previous packet's timestamp. If their difference is greater than zero, we received in order packets. If the inter-packet time difference is equal to the *sync_period* with a small tolerance, the timestamps are not changed (Condition I). If the packets arrive at a very small time difference, we move the second packet's timestamp ahead by the period (Condition II). If the packets have big enough of a difference, we transform the first packet's timestamp so as to maintain the *sync_period* (Condition III). We demonstrate an example in Figure 4.9 to show how the transformation works. Packets a and b maintains the same period s hence condition I is applied and their timestamps are used as it is. Packets b and c are very close hence condition II dictates that packet c be delayed by s seconds. Finally, packet f is farther away from g, and condition III transforms timestamp of f to be only s seconds away from packet g.

The key to restore periodicity is the knowledge of the one-way packets period. We assume that the master is trustworthy and sends one-way packets at a known rate. Current feedback based clock adjustment in PTP also makes use of the knowledge of packets' period. If the received packets at the client adhere to the same known period, they are assumed to be equally delayed and do not alter frequency error calculation in a feedforward controller.

4.5.3.2 Implement a feedforward controller

We filter out the packets satisfying the three conditions mentioned above, and transform their timestamps such that the filtered packets are periodic with an error tolerance. Note that these timestamps come from a free running client clock that is not adjusted. This free running clock is labeled as 'Raw Clock' in Figure 4.2b. After transformation, these timestamps emulate packets that experience same delay in the network. The rate of change of these timestamps w.r.t master timestamps over the course of one period determines the frequency error of client clock w.r.t master clock. For example, given two sets of transformed timestamps $(t_{1previous}, t_{2previous})$ and $(t_{1current}, t_{2current})$ for two consecutive one-way packets, the frequency error is calculated as,

$$skew = \frac{(t_{2current} - t_{1current}) - (t_{2previous} - t_{1previous})}{t_{2current} - t_{2previous}}$$

Note that this equation to calculate frequency error ($skew$) based on previous and current packets with equal delays in the network only holds true for a feedforward controller. The benefit of feedforward over feedback under delay attacks is its ability to calculate frequency error in the presence of large and varying delays. The transformed timestamps are not advantageous to the feedback controller as it cannot make use of timestamps of equally delayed packets.

We know that feedforward controller is only capable of aligning frequencies i.e. syntonization between a master and a client because we gather timestamps from a free running undisciplined clock. However, the feedforward controller gradually builds up error in its *skew* calculation due to environmental disturbances, error tolerance, path noise, and inaccuracies in timestamping. The accumulation of error due to inaccuracies in *skew* estimation results in increased time error between the two clocks. To overcome this error and also to synchronize two clocks, we propose a feedback trim to our feedforward controller.

4.5.3.3 Search for delay-free packets

There are two components in a clock adjustment mechanism: one is the frequency and the other is a time component. Two clocks are *syntonized* if we only adjust the frequency component of a clock w.r.t a reference clock. On the other hand, two clocks are *synchronized* when we adjust the time component of a clock. Syntonized clock is enough for applications interested in precisely measuring small inter-event times. Whereas, synchronized clocks are necessary for applications that coordinate tasks or choreograph acts.

We already provide syntonization to clocks under delay attacks through a feedforward controller. Now we attempt to synchronize two clocks in the presence of delay attacks. To synchronize two clocks, we have to find an offset between them. As the offsets are affected by adversarial network delays, we first filter the packets that are least affected by network delays. In other words, we search for *delay-free* packets. To find packets with less delays, we again rely on periodic one-way packets. Except now our conditions to filter packets are different. In Figure 4.10, a master sends two packets at a known interval s. We assume that an adversary is able to delay packets by at most x seconds. This assumption is realistic as it can easily be checked by a sanity check. We assert that one of the consecutive packets experience small delay if both packets satisfy the following two conditions,

Algorithm 3 Find Delay Free Packets

```
1:  procedure PACKET FILTERING(sync_period, offset_tolerance, max_delayattack)
2:      $s \leftarrow$ sync_period
3:      $\gamma \leftarrow$ offset_tolerance
4:      $x \leftarrow$ max_delayattack
5:  sample new packet:
6:      $t_a \leftarrow t_{prev\_pkt}$ ▷ Timestamp of previous packet
7:      $t_b \leftarrow t_{new\_pkt}$ ▷ Timestamp of latest packet
8:      $inter\_pkt\_dist \leftarrow t_b - t_a$
9:      if $inter\_pkt\_dist > 0$ then ▷ In order packets
10:         if $inter\_pkt\_dist > x + s - \gamma$ then ▷ Condition IV
11:             $(t_1, t_2) \leftarrow t_a$
12:     if $inter\_pkt\_dist < 0$ then ▷ Out of order packets
13:         if $inter\_pkt\_dist > x - s - \gamma$ then ▷ Condition V
14:             $(t_1, t_2) \leftarrow t_b$
15:     goto sample new packet.
```

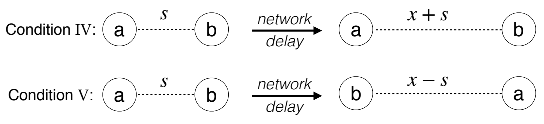


Figure 4.10: To calculate clock offset, only those packets are chosen that experience very small delay. The adversary delays packets by no more than x seconds. If two packets sent at a predefined interval s seconds experience delays such that packet a and packet b are $x + s$ apart then packet a incurs negligible delay (Condition IV), or if packet a gets ahead of packet b by $x - s$ then packet b incurs negligible delay (Condition v)

1. *Condition* IV: Given two packets sent at an interval of s seconds, capable of being delayed in the network by at most x seconds, the *first packet* is said to be minimally delayed if the two packets are *in order* and the received interval between these packets is at least $x + s - \gamma$, where γ is the error tolerance in offset calculation.

2. *Condition* V: Given two packets sent at an interval of s seconds, capable of being delayed by at most x seconds, we consider the *second packet* delay-free if the two packets are *out of order* and the resulting interval between the packets is at least $x - s - \gamma$.

Refer to Algorithm 3 for details.

There must be enough delay variation among two consecutive packets to satisfy the above two strict conditions. We do not put the expectation of providing delay variation on the attacker. Instead, we argue that less variations in delay attack would take longer for the Condition IV, and V to be met and the clocks to be synchronized. But these less variations in delay comes at a benefit. It supplements our approach to restore periodicity and the ability of our feedforward controller to precisely calculate frequency adjustment. On the other hand, huge variations in delay speed up our efforts to synchronize clocks. Nonetheless, our clock syntonization and synchronization architecture works with a wide variety of delay attacks.

4.5.3.4 Design a feedback trim for the feedforward controller

Feedback control based clock adjustment is only capable of providing synchronized time whereas, our implementation of *feedforward control with feedback trim* (feedforward w/ feedback trim) shown in Figure 4.11 is capable of providing both syntonized frequency and synchronized time.

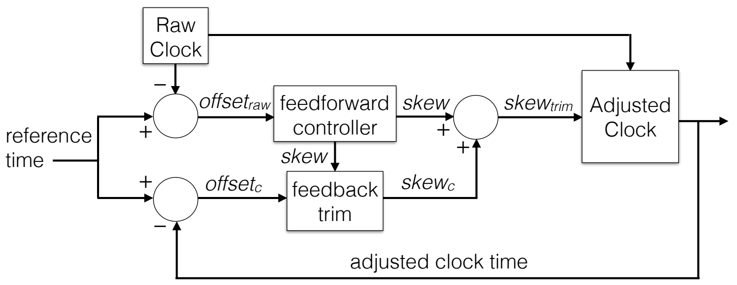


Figure 4.11: Feedforward control with feedback trim based clock adjustment

Algorithm 4 Feedback Trim

```
1: procedure TRIM FREQUENCY ADJUSTMENT ($skew$, $offset_c$, $sync\_period$, $max\_adj$)
2: $delay\text{-}free\ packet$:
3:     $f \leftarrow \frac{offset_c}{sync\_period}$          ▷ time error converted to frequency error
4: $loop$:
5:     $skew\_adj \leftarrow skew + f$
6:     if $skew\_adj < -max\_adj$ then          ▷ lower frequency adjustment limit
7:         $skew_c = -max\_adj - skew$
8:         $f = skew\_adj + max\_adj$          ▷ residual error
9:     else if $skew\_adj > max\_adj$ then          ▷ upper frequency adjustment limit
10:        $skew_c = max\_adj - skew$
11:        $f = skew\_adj - max\_adj$          ▷ residual error
12:    else
13:        $skew_c = f$
14:        $f = 0$
15:    if $f != 0$ then
16:        goto $loop$          ▷ loop again for residual error
17:    else
18:        goto $delay\text{-}free\ packet$          ▷ no residual error
```

After identifying the delay-free packets, we record their timestamps (t_1, t_2). In Figure 4.11, reference time represents t_1, and the timestamp from the client's adjusted clock is t_2. The $offset_c = t_2 - t_1$ is calculated from the timestamps of delay-free packets. A clock's time can be adjusted in two ways. Either add the time error, $offset_c$, to the clock or increase/decrease the clock frequency to gradually compensate for the clock error. We refrain from adding $offset_c$ to adjust the clock because it would cause discontinuities in time, and a clock should not have time discontinuities greater than it can tolerate. Instead we feed $offset_c$ and the frequency error ($skew$) from the feedforward controller to the *feedback trim* block. This block transforms the time error to a frequency error $f = offset_c/sync\_period$, and trims the frequency $skew$ of feedforward controller. Details of feedback trim algorithm is found in Algorithm 4.

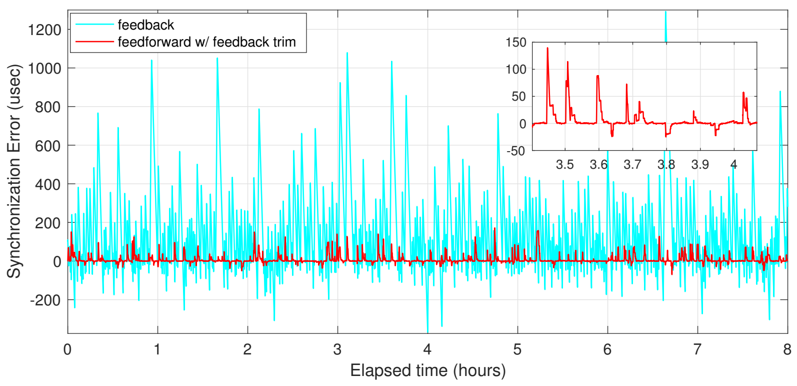
4.5.3.5 Putting it all together

An adversary in the network delaying packets can cause a lot of damage to the clock synchronization accuracy. We propose a delay attack-tolerant clock synchronization architecture that bounds the effect of delay attacks on clock errors. Our clock synchronization architecture consists of four major blocks as shown in Figure 4.7. The timestamps of delayed one-way packets are transformed in an effort to restore their periodicity such that a feedforward controller utilizes them to calculate frequency adjustment of a clock. Our architecture is also capable of finding delay-free packets to assist in offset calculation. The feedback control utilizes timestamps of delay-free packets to calculate the clock offset. Using this offset, we trim the adjusted frequency and provide the final trimmed frequency to discipline the clock. Thus presenting our architecture that uses feedforward control w/ feedback trim based clock adjustment.

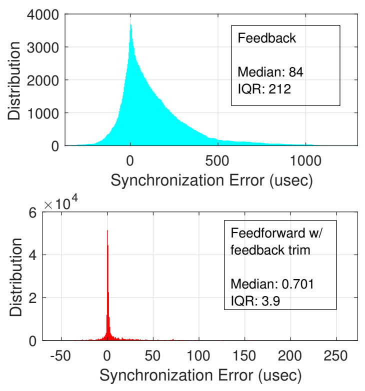
4.6 Implementation & Evaluation

We implement a testbed to evaluate the clock synchronization architecture. Our testbed consists of Beaglebone Black (BBB) [Bla] embedded Linux platform used in many applications in industry [ET03], fog computing [VDK17], smart grids [SBH12], financial market [PH16], and autonomous agents [KSB14]. The ethernet interface on BBB is IEEE 1588 standard compliant, and provides hardware timestamping capability essential for PTP protocol. Our testbed consists of two BBB nodes, one serving as the master and the other as a client. These nodes are connected via an IEEE 1588 compliant switch [Mox].

We do not necessitate the use of one synchronization protocol for our approach. However, we choose PTP [LEW05] for prototyping because of its demand in various high precision applications. We run a modified version of linuxPTP [Pro] on BBB nodes. One modification in PTP is we do not discipline the PTP clock. Instead we modify virtual clocks derived from that PTP clock. This concept of virtual clocks is not new; the need of multiple disciplinable clocks on a single platform has motivated this architecture [ADS16] [AAS18]. Posix standard [IEE] virtual clocks derived from the same hardware timer are also part of the Linux kernel. Our approach does not depend on any single architecture. It can work with POSIX clocks in the kernel as well as *timeline* in QoT stack [ADS16]. The only assumption we make is the access to a free running raw clock from which we derive a disciplinable clock. In the context of Linux kernel, CLOCK_MONOTONIC_RAW serves as the free running clock whereas CLOCK_REALTIME is a disciplinable clock derived from this raw clock. Our software implementation can be plugged into any



(a) Time series plot of feedback versus feedforward control w/ feedack trim for a total of 8 hours duration. Feedback clock error has large and continuous fluctuations in the range of -300 to 1300μseconds, while feedforward w/ feedback trim clock error ranges from -50 to 150μseconds with large durations of stable clock error.



(b) Distribution of clock errors shown in part (a). Feedback gives hundred times high error median and IQR as compared to feedforward w/feedback trim.

Figure 4.12: Synchronization error of two client clocks with respect to a master clock. One clock is disciplined by a feedback controller, and the other is adjusted by a feedforward controller with feedback trim. For a fair comparison, both controllers process transformed timestamps of periodic one-way packets, and timestamps of delay-free packets. Note that these packets are delayed in the network randomly by 0 to 2 seconds.

synchronization protocol as a standalone combination of a filter and a controller. We already provide controllers for PTP that work alongside PTP's traditional PI and Linear regression controllers. A user can switch between any controller of its choice.

We simulate delay attacks by adding delay to the received PTP SYNC packets at the client as shown in Figure 4.1(b). Note that our approach utilizes only one-way SYNC packets and does not require DELAY_REQ packets as feedforward controller does not rely on calculated delay. We established in Section 4.4 that feedforward calculated frequency error from one-way packets that are delayed equally is valid. Therefore, a constant delay attack is not considered an attack for a feedforward controller. Instead it complements frequency error estimation for the controller. But our results for random delay injection show that feedforward based frequency estimation never converges. Therefore, we evaluate our architecture for random attacks throughout this section, and discuss countermeasures for a combination of attacks.

Feedback-only versus Feedforward w/ feedback trim: We compare a *Feedback-only* controller used in all synchronization protocols with our proposed *Feedforward with feedback trim* controller in the presence of delay attacks. In this test case, a master periodically sends one-way packets at 2Hz (0.5seconds). An adversary is delaying these packets by randomly choosing a delay value from a uniform distribution of 0 to 2 seconds. Both controllers are fed the same delayed packets. The adjustable clocks for the controllers are derived from the same timing hardware ticking at 24MHz. This test scenario is depicted in Figure 4.3a, where we alleviate the bias caused by network and hardware variations on our results. In this test case, both the controllers process filtered one-way packets using our *Restore Periodicity* and *Delay-Free packets* techniques explained in Section 4.5. We choose the frequency error tolerance $\epsilon = 10$ milliseconds for restoring the periodicity of delayed packets, whereas, offset error tolerance $\gamma = 100$ milliseconds to find delay-free packets. While both controllers discipline their respective clocks, we continuously measure the time error of the adjustable clocks w.r.t the master clock using the setup shown in Figure 4.3b. We run this experiment for 8 hours and the results are shown in Figure 4.12.

The time series plot of the synchronization errors in Figure 4.12a shows large fluctuations in feedback clock error. While the clock error for feedforward w/ feedback trim has comparatively small variations. We zoomed into this result to show the range of clock error fluctuations for feedforward w/ feedback trim control. The feedforward controller calculates the frequency adjustment / skew from raw timestamps of a free running clock. The property of skew is that it does not vary much over time. Hence you see regions in Figure 4.12a for feedforward control that do not have much variations. As a result, the median of clock error distribution for feedforward w/ feedback trim control in Figure 4.12b(2) is almost 0.7μsecond with an IQR of only 4μsecond. On the other hand, the clock error distribution for feedback control has a median of 84μsecond with and IQR of 212μsecond. These results show 100 times synchronization improvement of feedforward w/ feedback trim control over feedback-only control in the presence of delay attacks.

Figure 4.13 shows a zoomed in six seconds duration in Figure 4.12a. This result gives the reason behind large clock errors in feedback than feedforward control. A you can see,

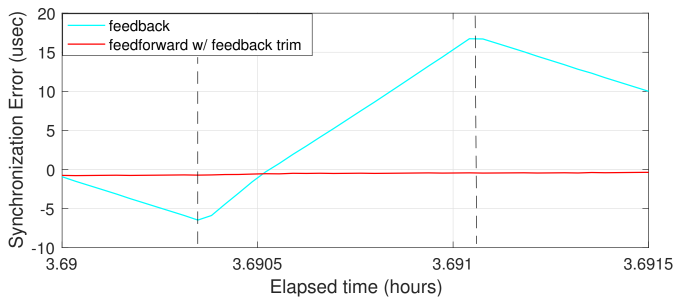


Figure 4.13: Zoomed in plot of 6 seconds duration from Figure 4.12a. Feedback control steers clock towards high errors with high frequencies whereas, feedforward control maintains clock error with consistent frequency

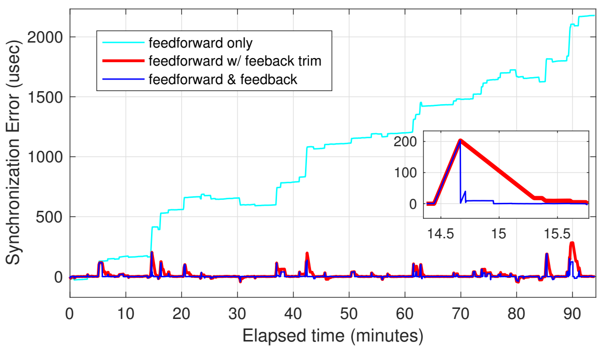
the clock error for feedback control keeps on increasing while the error for feedforward control is quite consistent. We know that feedback control only relies on the clock offset. This offset is calculated from delay-free packets, and the duration between two delay-free packets is not predictable. The feedback controller adjusts the frequency of the clock calculated from the clock offset, and waits for the next delay-free offset calculation. In the meantime, it updates the clock using the previous frequency adjustment and the error keeps on accumulating as shown in Figure 4.13 between two dashed lines. In this particular case, the delay-free packet come after few seconds but it can take longer as well. When the new delay-free packet comes, feedback control identifies the large clock error and steers the error in the other direction. On the other hand, feedforward w/ feedback trim control does not solely rely on the offset value from delay-free packets. For our controller, the time and frequency components are independent of each other. While waiting for a delay-free offset, it continuously adjusts the clock frequency calculated from transformed timestamps of periodic one-way packets. Hence feedforward w/ feedback trim maintains clock synchronization through syntonization.

As we establish that our approach of feedforward w/ feedback trim performs better than feedback-only approach, we now present other controller combinations and find the optimal combination among all. Other controller combinations are, *feedforward-only*, and independent *feedforward & feedback* under delay attacks. It is to be noted that none of these combinations exist in the current synchronization architectures. The only purpose of this comparison is to validate our choice of feedforward w/ feedback trim over other

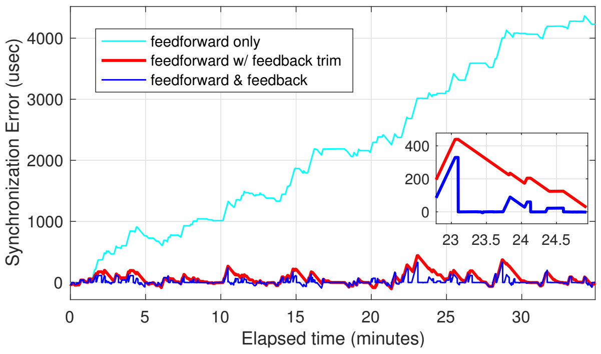
combinations using our filtering and frequency shaping techniques.
(1) A *feedforward-only* approach only calculates the frequency adjustment using timestamps sampled from a free running raw clock. It does not account for clock offset and accumulated clock error due to inaccuracies on feedforward frequency calculation. There is no loop to feed the error back for compensation. This control can syntonize but cannot synchronize the clocks.
(2) A *feedforward & feedback* approach uses two independent controllers. One feedforward controller to adjust the clock frequency, and one feedback controller to fix the clock offset. feedforward component syntonizes the clock, and feedback component synchronizes the clock.
(3) A *feedforward w/ feedback trim* approach is similar to feedforward & feedback, except here the two controllers are not independent. The feedback calculated offset is used to trim the feedforward calculated frequency.

Comparison of feedforward-only, feedforward & feedback, and feedforward w/ feedback trim: Now we compare the three possible controller combinations to find the optimum one in the presence of delay attacks. In this test case, we try to synchronize three virtual clocks on one client to a master. The reason of putting all clocks on a single client is to remove hardware and network variations related biases in results. Each of these clocks is disciplined form one of the above 3 controllers. For a fair comparison, all the controllers process the same filtered packets after restoring packets periodicity and finding delay-free packets using approaches in Section 4.5. We plot the clock errors in Figure 4.14.

In the presence of random 0 to 2 seconds delay attacks, Figure 4.14a shows that the performance of feedforward & feedback and feedforward w/ feedback trim clock errors is comparable. Their clock errors fluctuate in the range of -50 to 200μseconds with large durations of stable error. The feedforward & feedback median error is 0.27μsecond and 0.93μsecond IQR. While feedforward w/ feedback trim median error is slightly larger as 1μsecond with 11μsecond IQR. The slightly better results of feedforward & feedback control comes at a benefit of increased clock discontinuities as shown in the zoomed plot



(a) Clock synchronization performance of three controllers under random 0 to 2 seconds delay attack. This experiment runs for 95 minutes (x-axis), and the maximum accumulated error is approximately 2500μseconds (y-axis)



(b) Clock synchronization performance of three controllers under random 0 to 4 seconds delay attack. This experiment runs for 35 minutes (x-axis), and the maximum accumulated error is approximately 5000μseconds (y-axis)

Figure 4.14: Clock errors for three different controllers. Less error fluctuations for feedforward & feedback control as well as feedforward control w/ feedback trim. An added advantage of smooth error adjustment by the latter controller shown in zoomed in subplots in (a) and (b). feedforward-only control never compensates for clock offset and accumulates error at a small rate. Note the change in x-axix and y-axis for both (a) and (b)

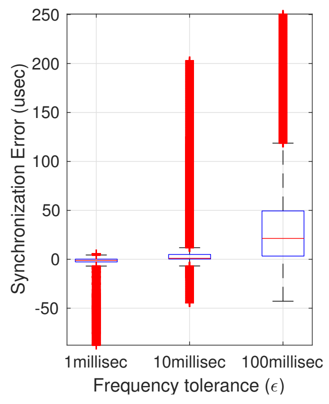
inside Figure 4.14a. This one minute zoomed plot shows how both controllers adjust their clock offsets. As feedforward & feedback adjusts offset independently, it has no choice but to jump the time to adjust the clock. These jumps cause discontinuities in time that are not desirable. On the other hand, feedforward w/ feedback trim transforms the calculated offset to a frequency adjustment. Then gradually adjust the frequency with maximum and minimum bounds to compensate for the offset. Thus avoiding time discontinuities at the cost of increased clock error.

We have yet to explain the performance of feedforward-only approach. We plot the normalized clock error in Figure 4.14a by subtracting all the errors from the first error. Thus the plot represents the rate of increase in error because feedforward control does not adjust clock offsets. The increased clock error is due to inaccuracies in calculated frequency adjustment. Note the sudden jumps in error. These jumps occur when a wrong feedforward based frequency adjustment makes the clock progress at the maximum frequency. We have set the limits of frequency adjustment to not exceed 10μseconds/second. In our test duration of almost 90 minutes, the clock error increased to approximately 2 milliseconds at the rate of 22μseconds/minute median and 15μseconds/minute IQR. Even though this error accumulation rate is low, an adversary may end up accumulating large errors over large durations. Therefore it is required to compensate for these errors. Hence feedforward-only approach could only work for measuring small durations inter-event times where the 20μseconds/minute error accumulation can be tolerated. In cases where better syntonization or where synchronized clocks are required, this approach would not work.

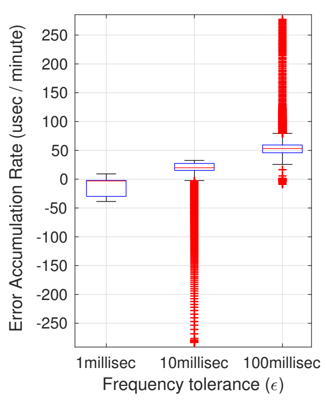
Performance under large delay attacks: So far our experiments had a random delay injection in the range of 0 to 2 seconds. We run the same set of experiments of comparing different controller under large delay attacks of 0 to 4 seconds. The clock errors of feedforward & feedback control and feedforward w/ feedback trim control in Figure 4.14b are not as stable as errors for these controllers in Figure 4.14a. The median error for feedforward & feedback control is -0.4μseconds with 38μseconds IQR, while feedforward w/ feedback trim control provides 47μseconds median error with 124μseconds IQR. Even though the high IQR of clock errors shows less error stability, we are still able to provide decent synchronization accuracy that many applications require. With different delay distributions, choosing the right frequency and offset tolerance to filter delayed packets greatly affects the synchronization performance.

Optimizing Frequency tolerance (ϵ) and offset tolerance (γ):

Frequency tolerance ϵ is the upper error limit our system can ignore in calculating frequency adjustment when checking for different conditions to restore delayed packets' periodicity as explained in Section 4.5. Offset tolerance γ is the system's upper error limit in its offset calculation when finding delay-free packets. Filtering delayed packets that satisfy conditions in Figure 4.8 and 4.10 is affected by ϵ and γ value. Too high tolerance values result in high false positives. Less packets are filtered out increasing the probability of accepting those delayed packets that do not necessarily satisfy the desired conditions. Too low tolerance produces false negatives and valid syntonization and synchronization opportunities are missed. Therefore, it is necessary to find the optimal tolerance values



(a) Affect of frequency tolerance on clock errors



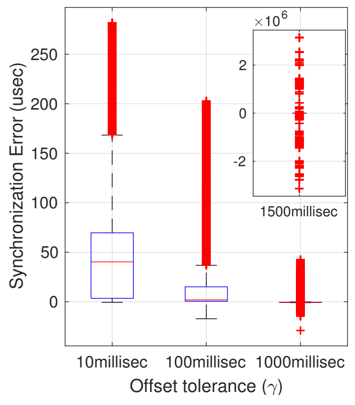
(b) Affect of frequency tolerance on clock errors accumulation rate

Figure 4.15: Showcasing the relationship between frequency tolerance and clock error rate. Higher the tolerance, higher the rate of error accumulation raising the instantaneous clock error.

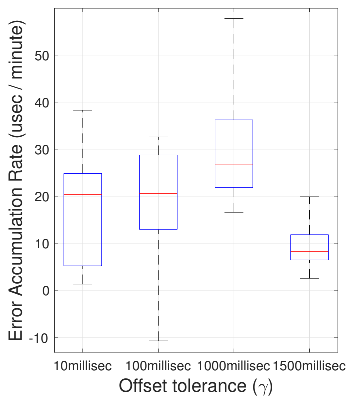
when filtering packets.

We evaluate our delay-tolerant synchronization approach with three different ϵ values. The result in Figure 4.15a shows the clock error for different ϵ values. For $\epsilon = 1$ millisecond, the clock error is small with relatively less number of high outliers. For same ϵ value, the conditions to restore periodicity do not occur often, hence there are less chances for the feedforward controller to calculate frequency adjustment. That is why a wrong frequency adjustment keeps on doing the damage before the next frequency adjustment can be calculated. The result are a few outliers in Figure 4.15a for 1 millisecond. On the flip side, the small tolerance range reduces false positives resulting in valid frequency calculation most of the time, hence the small clock error. The error accumulated rate due to wrong frequency calculation is also less for $\epsilon = 1$ millisecond in Figure 4.15b. The high IQR with respect to median error accumulation rate signifies occasional wrong frequency calculations. The clock error and the rate at which this error gets accumulated keep on increasing with the rise in ϵ value. As ϵ increases, the high chances of calculating wrong frequency on false positives increases clock error as well as error accumulation rate.

The clock errors are also dependent on the offset tolerance γ. Note in Figure 4.16a that the clock error keeps on decreasing with an increase in γ. A controller with small γ



(a) Affect of offset tolerance on clock error



(b) Affect of offset tolerance on error accumulation rate

Figure 4.16: Showcasing the relationship between offset tolerance and clock error. Higher the tolerance, lesser the clock error but only till a certain limit beyond which the errors are marked as attacked. Offset tolerance does not affect error accumulation rate

value takes more time to synchronize because high tolerance puts strict bounds on offset error and reduces the chance of getting a delay-free packet. This results in high clock error accumulated due to the inaccuracies in frequency adjustment. A large γ value means getting delay-free packets more frequently, hence increased opportunities to fix the clock offset and overcome accumulated error due to wrong frequency. But there is a limit to this increase as too high a value increases false positives and the clock error has large outliers and huge discontinuities as shown for $\gamma = 1500$ milliseconds in the subplot in Figure 4.16a. There are huge jumps in the order of seconds that are heavily influenced by delay attacks. Thus a large γ value is useful but only up to a certain limit. The error accumulation rate is only a function of ϵ, hence it remains unaffected by γ in Figure 4.16b.

Our architecture can dynamically select tolerance values by keeping track of error accumulation rate, time taken to synchronize, and calculated offset outliers. High error accumulation rate gives an indication to decrease ϵ while frequent and large offset outliers is an indication to decrease γ. Reducing γ also speeds up synchronization. Ultimately, an optimal combination of frequency and offset tolerance gives small and stable error.

Incremental delay attacks and countermeasures: There are three major kinds of delay attacks: constant delay, random delay, and incremental delay. Most of the other attacks are a combination of these attacks. We already analyzed and evaluated constant and random delay attacks. Indeed our approach works best under large delay variations but it can also detect attacks with small delay variations over a longer time. Over small duration, an incremental attacker injects small delays. If these increments are smaller than the frequency tolerance (*epsilon*), our approach can potentially makes use of them to calculate the adjustment frequency. If left unchecked, a patient adversary can accumulate large error over a long period by inducing small errors in adjusted frequency. We can thwart this attack by not only applying our restoring periodicity conditions to consecutive packets but also every tenth or hundredth packet, depending on the value of incremental attack. This value can be dynamically adjusted in many ways. Keeping the same ϵ, our approach can detect the attack and filter packets accordingly. Thus it is able to bound the errors induced by incremental attacks. Similarly, small variations between consecutive packets can never satisfy conditions for delay-free packets. We repeat the same method above by applying the conditions on every tenth or more of a packet. When an adversary is able to accumulate an error in the order of our offset tolerance γ over multiple periods, our conditions for delay-free packets will be satisfied. Depending upon the value of γ, the conditions may occur sooner or later.

4.7 Key Findings

This work is the first to overcome network delay attack under a very strict threat model i.e. delaying all time transfer packets to a client node. We evaluated our design by preserving the accuracy of precision time protocol in the order of microseconds by overcoming constant, random, and incremental delay attacks. Our design features a feedforward controller for synchronizing clock frequency with a reference, and a feedback trim to synchronize the clocks. Our architecture also works in non malicious networks. The idea of restoring packets periodicity in the presence of attacks is equally valid for packets that are not delayed. In a benign network, the packets reach at almost the same interval hence it can be used to calculate frequency adjustment. As we mostly rely on one-way packets and rarely upon two-way packets over long durations in clock synchronization, our system consumes less bandwidth as compared to traditional protocols. Summarizing all the results, feedforward control w/ feedback trim outperforms other controllers with high clock accuracy, no time discontinuities, and achieving tunable syntonization as well as clock synchronization in the presence of different kinds of delay attacks.

4.8 Conclusion & Future Directions

Security in the context of time is important as many applications are emerging that have moved away from traditional "clockless" assumption [GLY18]. Systems are increasingly making use of synchronized clocks to enhance the accuracy of network measurements and reduce the complexity of distributed system protocols. On the other hand, adversaries are targeting timing primitives for copyright theft, illegal trade, and location theft, etc. Cryptography and network security mechanisms have thwarted various attacks on time transfer packets but delay attack is too strong to be mitigated completely. We pointed out the key issues in current clock synchronization architectures that make them vulnerable to delay attacks and propose a new delay attack-tolerant synchronization architecture. Built on top of a feedforward control with feedback trim clock adjustment mechanism coupled with packet filtering techniques, the architecture is capable of bounding delay attack errors.

In the future, we intend to provide a formal security analysis of our delay attack-tolerant clock synchronization architecture and provide provable error bounds under all possible attacks. Showing an implementation for NTP is also an interesting direction where we can utilize the excess information from multiple servers to tighten the error bounds for strong delay attacks.

We are also considering some interesting future research directions. We know that securing time transfer packets is meaningless if the timing hardware and software stack are stealthy. Researchers show that an adversary can cause hardware faults by overclocking digital circuits and not satisfying their timing constraints [TSS17]. A malicious OS can manipulate timer registers [sgx16], and a host can lie about time [BPH15]. It would be beneficial to protect all layers of the time stack – the hardware timers, system software, and the network packets.

Part III

Systems for Timing Precision

CHAPTER 5

gPHC: generalized Precise Hardware Clock

5.1 Introduction

Modern distributed systems comprise of many heterogeneous devices. In order to choreograph their actions, these devices need to share a common notion of time, which may or may not be pinned to some global time coordinate like UTC. Various hardware and software solutions exist for providing standalone and networked devices with precise knowledge of time. For example, GPS and atomic clocks, network adapters that perform hardware timestamping [LEW05], and synchronization algorithms that achieve low-nanosecond time synchronization between devices [LWS11]. These precise time solutions are mostly available for wired interfaces, preferably ethernet. We have yet to see precise time solutions for applications in Low Range Wireless Personal Area Networks (LR-WPAN) even though numerous applications in LR-WPAN require precise time-awareness such as real-time positioning systems, formation flying, distributed sound systems, and foraging applications.

IPv6 over Low power Personal Area Network (6LoWPAN) is a powerful technology that defines the upper layers for the LR-WPAN MAC and PHY layers [CC09]. The key idea is to provide IP connectivity for resource-constrained devices so that they may participate in the IoT with their sensing, communication, and data delivery capability. In addition, 6LoWPAN is interoperable with IPv6 based networks, and software written for 6LoWPAN is compatible with IPv6 based solutions. These characteristics along with access to cloud-based services increase the chances of penetration of 6LoWPAN for applications in LR-WPAN.

This motivates us to enable PTP over 6LoWPAN interface to bring precision timing to applications in LR-WPAN. In fact, we provide a generalized Precise Hardware Clock (gPHC) abstraction that enables high precision for all processors, co-processors, radios, and other interfaces equipped with the necessary hardware capabilities. As a proof of concept, we prototype it for LR-WPAN and 6LoWPAN, though the purpose of this paper is not only to implement timestamping capability and a PHC for a specific radio but also to provide a generalized guide for developers to implement their own gPHC over desired hardware.

We make use of the existing PTP Clock infrastructure in the Linux kernel [CM10], and the networking stack to implement hardware timestamping for LR-WPAN and 6LoWPAN interface. We also expose these radio interfaces as a PTP clock. In PTP terminology, the PTP clock is referred to as a Precise Hardware Clock (PHC). We provide the wireless version of a PHC along with the capability to precisely timestamp external events, generate precise hardware interrupts, and provide pulse-per-second (pps) signal for onboard time alignment of peripherals and processor.

There are indeed many drivers written to expose the network interfaces as PHC such as National Semiconductor PHYTER (dp83640), AMD 10GbE Ethernet Soc (amd-xgbe), Freescale eTSEC gianfar (gianfar), and Intel 82574 (e1000e). All of these PHCs are pinned to an ethernet network interface. Contrary to these, we put forward a wireless PHC[1]. We implement the wireless PHC driver on top of the PTP class driver to expose the LR-WPAN and 6LoWPAN interface as a PHC (*/dev/ptpX* character device) to the userspace services and applications. Using this hardware clock and the hardware timestamping functionality implemented in Linux, we run PTP synchronization service [Pro] as an IPv6 based protocol to discipline Precise Hardware Clocks (PHCs) in a distributed network.

Our experimental testbed consists of beaglebone black devices interfaced with the Decawave DW1000 radios, which is a short-range wireless Ultra-wideband (UWB) radio [SPN05]. UWB radios are LR-WPAN and 6LoWPAN compliant. We provide PTP support over UWB radios and achieve synchronization accuracy in the order of nanoseconds. One of

[1]Intel is working on PHC support for WiFi but they have not released yet

the results shows that the propagation delay in the UWB wireless medium is deterministic. This is due to the fact that UWB signaling has low power spectral density and have reduced interference with other radio frequencies.

5.2 Background

Before diving into details, we go through the fundamentals of few networking standards and timestamping capabilities of the networking stack.

5.2.1 IEEE 802.15.4 (LR-WPAN) and 6LoWPAN

The IEEE 802.15.4 standard defines the protocols for low-rate, low energy, and low-cost wireless data communication devices transmitting RF signals over short range in personal, local, or metropolitan area networks. This is in contrast to WiFi, which offers more bandwidth at the cost of high power [LSS07]. IEEE 802.15.4 standard is implemented as a Low Rate Wireless Personal Area Network (LR-WPAN). Key features of LR-WPAN are optional allocation of guaranteed time slots, channel access protocols, energy detection, and link quality indication. LR-WPAN standard define these features only at the PHY and MAC layers in the network stack. Other standards like Zigbee, Thread, and 6LoWPAN define upper layers for LR-WPAN to provide the entire networking stack. 6LowPAN provides LR-WPAN nodes with IP communication capabilities. The advantages of IPv6 over LR-WPAN frames is that IP based technologies already exist; the pervasive nature of IP networks allows use of existing infrastructure, IPv6 is more suitable for higher density, and IP based applications can make use of socket interface for programming. Low-power, IP-driven nodes and large mesh network support make 6LoWPAN technology a great option for Internet of Things. It open doors for many applications such as precision Real Time Location System (RTLS), home automation, building security, distributed sound system, and gaming.

5.2.2 Ultra-wideband (UWB) radios with Hardware Timestamping support

UWB radios are LR-WPAN and 6LoWPAN compliant. They use very low energy levels for short range communication. The key characteristic of UWB radios is that they transmit over an absolute bandwidth of more than 500MHz. Spreading information over a large bandwidth decreases the power spectral density, reducing interference with other systems and enabling coexistence with conventional radio frequencies with minimal interception. The large bandwidth also alleviates small scale fading [GTG05]. UWB pulses are timed very precisely across a wide spectrum, and the receiver signal detector should match the transmitted signal in bandwidth, signal shape and time. A mismatch results in loss of margin for the UWB radio link. UWB signaling combines low rate precision communication with positioning capabilities, and allows centimeter level ranging [GTG05].

During frame transmission or reception, the Start of Frame Delimiter (SFD) detection event marks the end of the preamble and the start of the PHR (PHY header). The IEEE 802.15.4 UWB standard nominates the time when the start of the PHR arrives at the antenna as the significant event to capture the transmit or receive timestamp in a hardware register [Man]. Antenna delay effects the transmit and receive timestamps accuracy. Antenna delay occurs between internal processor digital timestamp at the start of the PHR and when the start of PHR is actually transmitted or received at the antenna. Antenna delays can be calibrated to get more accurate timestamps.

Table 5.1: Flag options for network sockets based hardware timestamping in Linux

Flag	**Description**
SOF_TIMESTAMPING_TX_HARDWARE	Generate transmit timestamp in the hardware by Network Interface Clock
SOF_TIMESTAMPING_RX_HARDWARE	Generate receive timestamp in the hardware by Network Interface Clock
SOF_TIMESTAMPING_TX_SOFTWARE	Generate transmit timestamp in kernel driver by Network Interface Clock
SOF_TIMESTAMPING_RX_SOFTWARE	Generate receive timestamp in kernel driver by Network Interface Clock
SOF_TIMESTAMPING_RAW_HARDWARE	Report any generated hardware timestamp when available

5.2.3 Network Sockets based Timestamping in Linux

Networking stack in Linux provides socket attributes to query software or hardware timestamps. The socket attribute used to get the software timestamps is *SO_TIMESTAMPNS*.

Socket attribute to request the generation of hardware timestamps, and report the hardware timestamps is $SO\_TIMESTAMPING$. Bitmap of flags provided by $SO\_TIMESTAMPING$ socket attribute determines the generation or reporting of hardware timestamps. Different timestamping options corresponding to different flags are summarized in Table 5.1. Socket types that support timestamping are RAW, UDP, and Stream sockets. The hardware timestamping capability however is contingent upon a radio's capability of generating timestamps in PHY or MAC.

A process reads the timestamp by calling $recvmsg(intsockfd,\ structmsghdr\ *msg, intflags)$ linux api [pagb] on a network socket. The $flags$ field determines if the reported timestamp came for a received packet or a transmitted packet. In the case of packet transmission, the outgoing packet is looped back to the socket's error queue with the transmit timestamp appended to the message ancillary data (CMSG). The flag used to retrieve this transmit timestamp is $MSG\_ERRQUEUE$. On packet reception, the receive timestamp is also stored in the message ancillary data CMSG ($cmsg\_data$). $cmsg\_data$ consists of $struct\ timespec\ ts[3]$. This structure returns three timestamps. Software timestamps are passed in $ts[0]$ while hardware timestamps are passed in $ts[2]$. The attributes of CMSG determines the type of socket and the type of timestamping. If the CMSG attributes $cmsg\_level$ equals SOL_SOCKET, and $cmsg\_type$ is $SO\_TIMESTAMPING$, then $ts[2]$ has a value of transmit or receive hardware timestamp.

5.3 Related Work

It is currently possible to synchronize system clock in Linux to the order of milliseconds with the Network Time Protocol (NTP) [Mil91], or nanoseconds with the Precision Time Protocol (PTP) [LEW05] and compliant hardware. More specialized projects, such as WhiteRabbit [LWS11], attain sub-nanosecond error – enough to measure the distance light travels in a second with millimeter accuracy – by compensating for cable delay asymmetry and using Synchronous Ethernet to frequency-lock devices. Authors of [CM10] were the first to provide a Precise Hardware Clock (PHC) infrastructure in the Linux kernel, also refereed to as a PTP clock. Authors in [CMR11] makes use of the PHC support in Linux to synchronize the PTP clock with the Linux system clock. In short, the PTP protocol is implemented around powerful abstractions and a complete system support for precise timing. PTP however is exclusive to ethernet interface; select companies provide PTP-compliant ethernet interface and drivers. We argue that other network interfaces particularly wireless NICs can leverage the PTP capabilities.

There have been attempts to enable PTP for WiFi radios. Authors in [MGT11] tried wireless clock synchronization based on PTP and software timestamping using specialized drivers. The accuracy however is compromised by software timestamping in WiFi drivers. Authors in [Exe12] proposed hardware timestamping for WiFi radios. They estimate delay between the received signal and the local clock through timing recovery measurements with an FPGA based implementation. The use of additional and specialized hardware limits the use of their approach, and multipath propagation was also a liming factor. Other wireless technologies like Ultra-wideBand (UWB) radios have emerged recently, where [DFR11] achieved high precision timestamping based on UWB signaling.

One interesting direction in time synchronization literature is for the applications to know what sort of synchronization accuracy or Quality of Time the underlying system provides. Authors of [ADS16] coined the term Quality of Time and reported the timing uncertainty to the applications. In a similar context, [AF07] provides a detailed comparison of timestamping accuracy different hardware and software provides.

5.4 PTP over LR-WPAN and 6LoWPAN

To be able to run PTP over any network interface, it is essential to follow three steps: (a) implement socket based hardware timestamping in the radio driver and MAC software, (b) expose the radio as a PHC clock, and (c) discipline the PHC clocks in a distributed network through packet exchange. These three steps are covered in detail in this section.

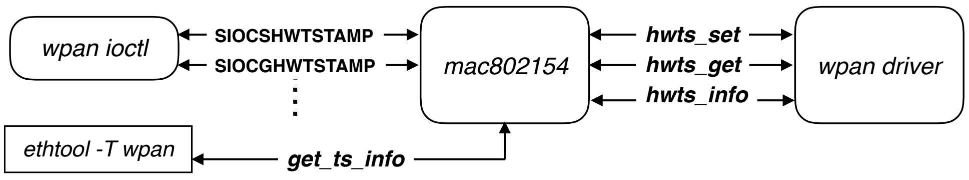


Figure 5.1: Flow chart of hardware timestamping configuration for LR-WPAN (8021.5.4 standard)

5.4.1 Enable socket based hardware timestamping in LR-WPAN and 6LoWPAN stack

Various Network Interface Cards (NIC) have the capability to shadow capture timer values in hardware registers upon packet transmission and reception. The drivers written for these NICs however may not make use of these timestamps stored in hardware registers. Even though the networking stack in Linux provides sockets and socket buffers to store and transfer timestamps up the stack. Currently, various Ethernet NICs support network sockets based hardware timestamping. The drivers of these NICs read the timestamps in hardware registers and push these timestamps to the socket buffer. In order to see the timestamping capabilities of a particular NIC, run $ethtool\ -T\ ethX$, where $ethX$ corresponds to the network interface.

In this section, we will walk through the necessary steps to add socket based hardware timestamping support for a particular network interface. Our steps are explained in reference to WPAN[2] (802.15.4) and 6LoWPAN interface but the procedure is generalized enough to be applicable to other network interfaces as well.

5.4.1.1 Hardware timestamping configuration

The first step is to configure which outgoing and incoming packets should be timestamped. The NICs should not be timestamping all packets since the code responsible for timestamping may slow down the network stack on a critical path. After all, not every user of the network stack requires accurate hardware timestamping. Applications use the ioctl

[2]LR-WPAN will be referred to as WPAN from here on for simplicity

SIOCGHWTSTAMP to choose which outgoing packets to timestamp in hardware (*tx_type*) and which incoming packets to timestamp in hardware (*rx_filter*). This ioctl is also used to retrieve the timestamping configuration (*tx_type* and *rx_filter*) that may have already been set. Refer to the file, */include/linux/net_tstamp.h* for complete values of timestamping configuration.

Figure 5.1 illustrates the entire flow of hardware timestamping configuration for the WPAN NIC. The *SIOCSHWTSTAMP* and *SIOCGHWTSTAMP* ioctls are initiated in the userspace (shown as *wpan ioctl* block in Fig 5.1). These ioctls are passed down to WPAN MAC software in the kernel (*mac*802154 block in Fig 5.1). *mac*802154 handles these ioctls by invoking the driver's registered callback functions. We used three registered callbacks for timestamping configuration: *hwts_get*, *hwts_set*, and *hwts_info*, and implemented these callback functions in the WPAN driver (*wpan driver* block in Fig 5.1). The following code snippet shows how *mac*802154 handles the hardware configuration ioctls,

```
mac802154_wpan_ioctl(...) {
  struct ieee802154_local *local = sdata->local; ...
    case SIOCSHWTSTAMP:
      if (local->ops->hwts_set)
        err = local->ops->hwts_set(&local->hw, ifr);

    case SIOCGHWTSTAMP:
      if (local->ops->hwts_get)
        err = local->ops->hwts_get(&local->hw, ifr);
... }
```

*mac*802154 defines the hardware timestamping configuration callback functions along with other network operations such as, asynchronous frame transmission, channel access, and address assignment. The callback function (*hwts_info*) which returns what kind of packets the NIC is capable of timestamping, and the functions (*hwts_get* and *hwts_set*) which return what kind of packets the application wants the NIC to timestamp are shown in the following listing,

```
struct ieee802154_ops {
int (xmit_async)(struct ieee802154_hw *hw,
                        struct sk_buff *skb); ...
int (hwts_get)(struct ieee802154_hw *hw,struct ifreq *ifr);
int (hwts_set)(struct ieee802154_hw *hw,struct ifreq *ifr);
int (hwts_info)(struct ieee802154_hw *hw,
            struct ethtool_ts_info *info);
};
```

Ethtool [paga] is a useful tool that provides visibility into the timestamping capabilities of a NIC. To provide ethtool support in the WPAN driver, *mac*802154 makes use of *ethtool_ops* and implements its callback function *.get_ts_info*, which further invokes a nested callback function *hwts_info* to get the desired timestamping capabilities from the driver. The complete ethtool flow is shown in Fig 5.1 and the following code would make it more clear,

```
static const struct ethtool_ops mac802154_ethtool_ops = {
  .get_ts_info = mac802154_get_ts_info,
};
static int mac802154_get_ts_info(struct net_device *ndev,
                  struct ethtool_ts_info *info) { ...
  if (local->ops->hwts_info)
    return local->ops->hwts_info(&local->hw, info);
}
```

5.4.1.2 Hardware timestamping implementation

Socket Buffer (*sk_buff*) is a core data structure in Linux networking stack, designed to store protocol PDUs for packet transmission and reception. As the packets enter or leave NIC, they are processed up or down the stack through socket buffers. These buffers optionally store hardware timestamp for packet transmission and/or reception inside a structure *skb_shared_hwtstamps* which is a part of *skb_shared_info* as shown in Fig. 5.2. Network driver (*wpan driver* in Fig. 5.2) is responsible for copying the timestamps captured in hardware registers to the *skb_shared_hwtstamps* structure in socket buffer. The following code snippet shows how a timestamp is copied from a register to socket buffer upon frame reception.

```
static void wpan_async_rx() { ...
  struct sk_buff *skb;
  struct skb_shared_hwtstamps *sshptr;
```

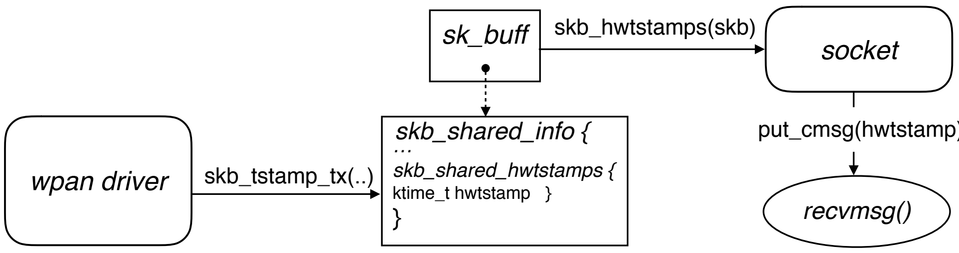


Figure 5.2: Timestamps are copied to socket buffer by the wpan driver. The socket buffer store the timestamp in the message ancillary data (CMSG) to be read by an application

```
    sshptr = skb_hwtstamps(skb);
    sshptr->hwtstamp = RXhardwareTimestampValue;
}
```

In the above code, $skb\_hwtstamps(skb)$ function returns the pointer to $skb\_shared\_hwtstamps$ structure in the socket buffer. Using this pointer, a network driver can set the $hwtstamp$ to the timestamp value captured in hardware register ($RXhardwareTimestampValue$ in the code above). The procedure of copying transmit timestamp is a bit different though. Since transmit timestamps are reported by looping the outgoing packet to the socket error queue (as mentioned in previous section), the function, $skb\_tstamp\_tx()$ (shown in the following code snippet) is used to copy the captured transmit timestamps in hardware registers to the socket buffer.

```
static void wpan_async_tx() { ...
  struct skb_shared_hwtstamps sshptr;

    ssh.hwtstamp = TXhardwareTimestampValue
    skb_tstamp_tx(skb, &ssh);
}
```

The entire flow of hardware timestamp implementation is shown in Fig 5.2. The *wpan driver* is responsible for writing the hardware timestamps in registers to socket buffer. *socket* reads the timestamp from the buffer and put it in the ancillary data (CMSG) of the message. A userspace application retrieves timestamp in CMSG by calling the *recvmsg* function on the network socket.

5.4.1.3 Hardware timestamp configuration & implementation for 6LoWPAN

After setting up hardware timestamp configuration and implementation for WPAN, the next step is to enable these functionalities for 6LoWPAN interface. Since 6LoWPAN represents the upper layers of WPAN in a network stack, we just need to add function hooks in 6LoWPAN software to pass the ioctl calls to the underlying WPAN driver.
The $SIOCSHWTSTAMP$ and $SIOCGHWTSTAMP$ ioctls for harwdare timestamping configuration, called on a 6LoWPAN interface, are passed down to the ioctl of WPAN driver.

```
static int lowpan_ioctl(...){
  struct net_device *real_dev = lowpan_dev_info(dev)->real_dev;
  real_dev->netdev_ops->ndo_do_ioctl(real_dev, ifr, cmd);
...}
```

In this listing, the *real_dev* represents the WPAN device of the 6LoWPAN interface. The *ndo_do_ioctl* is forwarding the arguments of 6LoWPAN ioctl to WPAN ioctl. We implement the ethtool functionality for 6LoWPAN in a similar manner by forwarding ethtool call to the WPAN driver.

5.4.2 Expose Radio as a PHC Clock

PTP clock is also referred to as a *Precise Hardware Clock* (PHC). A clock that supports hardware timestamping for packets transmission and reception, and has associated pins with external event hardware timestamping and deterministic hardware interrupt capabilities is a PHC. Richard et al. [CM10] was the first to provide PTP clock infrastructure in Linux kernel along with a standard API for user space programs and kernel clock drivers. The structure used to represent a PHC is *ptp_clock*. This is a powerful abstraction with drivers written around it to abstract away from hardware-specific code.

After enabling hardware timestamping for WPAN in Linux, the next step is to expose the WPAN interface as a *ptp_clock*. This is essential because it provides a clock source that is central to maintain time and discipline time. A *ptp_clock* is an abstraction on top of an oscillator that is used to drive a hardware timer, from which an overflow-safe logical clock is derived using the *cyclecounter* and *timecounter* structures in Linux. The key functionalities to implement this clock are listed here,

```
static struct ptp_clock_info wpan_ptp_info = {
  .owner     = THIS_MODULE,
  .name      = "WPAN Clock",
  .max_adj   = 1000000,
  .n_ext_ts  = 0,
  .n_per_out= 0,
  .pps       = 0,
  .n_pins    = 0,
  .adjfreq   = wpan_ptp_adjfreq,
  .adjtime   = wpan_ptp_adjtime,
  .gettime   = wpan_ptp_gettime,
  .settime   = wpan_ptp_settime,
  .enable    = wpan_ptp_enable,
};
```

A *ptp_clock* is a wrapper around the *posix_clock* interface and provides additional/optional pin capabilities. *ptp_clock* provides an interface to enable or disable the clock source, configure timer pins (for timestamping inputs or pulse-width modulated outputs), and discipline the clock (either in hardware or software). Pins are configured through the hardware timer subsystem using *.enable* and *.verify* function callbacks. The time can be observed or set through *.gettime* and *.settime* function callbacks, which reduce to read from or write to instructions on the *cyclecounter* and *timecounter*. The WPAN driver implements the correct function callbacks, and register the existence of the precise clock through *ptp_clock_register*, with the kernel's PTP subsystem. This clock is exposed to userspace as */dev/ptpX* character device.

5.4.3 Synchronize Distributed gPHCs

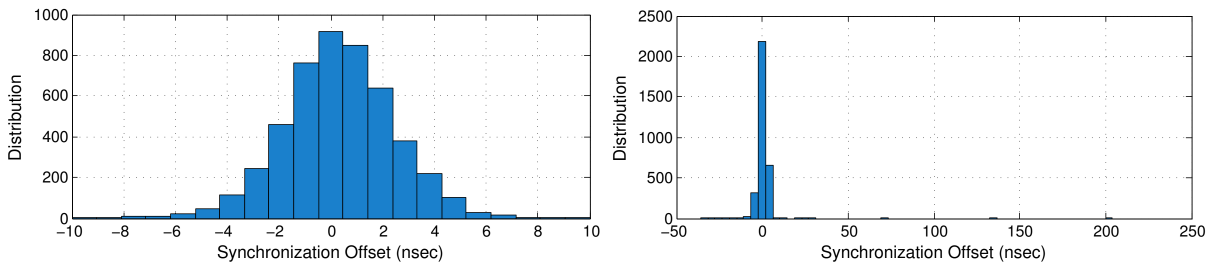
We have configured and implemented hardware timestamping for WPAN and 6LoWPAN in Linux and exposed these interfaces as a PHC. Note that a single PHC (*/dev/ptpX*) represents WPAN as well as 6LoWPAN since both are a part of the same networking stack. The next step is to synchronize the PHCs in a distributed network. Linuxptp [Pro] is a synchronization daemon for PTP and it supports synchronization for three kind of network packets; Ethernet (IEEE 802.3), IPv4, and IPv6 packets. 6LoWPAN supports IPv6 packets and it provides an adaptation layer that is used to fragment packets if IPv6 packets are large enough to exceed the maximum MTU (127 bytes) of 802.1.5.4 packets. Since PTP packets are multicast, the packet sizes will not exceed the 802.15.4 maximum MTU. We used the IPv6 packet option for running Linuxptp over the 6LoWPAN interface. We choose the end-to-end (E2E) mode over peer-to-peer (P2P) for PTP because E2E is more flexible and does not require the entire infrastructure (switches and routers) to be PTP-compliant as required in P2P mode.

5.5 Evaluation

We set up a test bed comprised of Beaglebone Black devices (BBB) [Bla]; it is a low cost development platform running Linux operating system. We interface a UWB decawave radio DW1000 [Man] to the BBB through SPI using the device tree overlay. DW1000 is a low power, low cost transceiver IC compliant to the IEEE 802.15.4-2011 standard. It has the capability of hardware timestamping. We package the DW1000 driver as a loadable kernel module that supports the following functionalities,

- Configure hardware timestamping in network stack by implementing MAC callback functions; $hwts\_set$, $hwts\_get$, and $hwts\_info$. These functions are invoked through ioctl calls from userspace to set, get and query hardware timestamping capability.
- Upon frame transmission and reception, copy timestamps from hardware registers to socket buffers.
- Develop all the functions necessary to implement a *ptp_clock* and expose it to userspace as a */dev/ptpX* character device.

We also modify the Linux kernel source code such as mac802154, 6lowpan, and UDP socket interface files to implement hardware timestamping for wpan and 6lowpan.



(a) Periodic *Sync* messages (b) Periodic *Sync* and *Delay_ Req* messages

Figure 5.3: Synchronization accuracy over WPAN and 6LoWPAN based radio (DW1000), (a) without *Delay_ Req* message (b) with *Delay_ Req* message, Note the change in x-axis scale

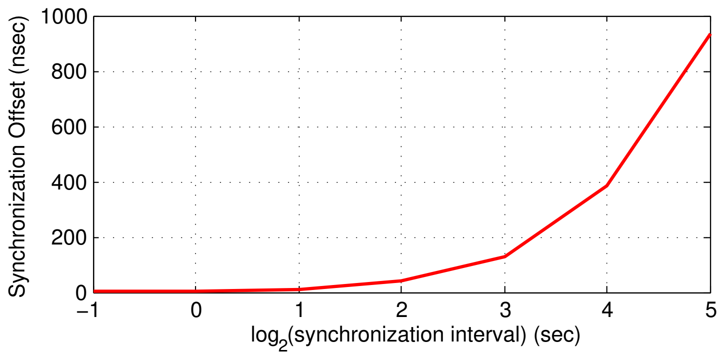


Figure 5.4: Synchronization Accuracy is a function of the synchronization period over UWB DW1000 radios

The instructions to set up a node for wpan & 6lowpan time synchronization, and the complete source code repository for this work can be found at,

https://bitbucket.org/rose-line/ptp-wpan-6lowpan

The master in PTP synchronization sends periodic *Sync* messages to slaves while the slaves send periodic *Delay_ Req* messages to the master in E2E mode. *Delay_ Req* messages are used to compensate for the packet propagation delay in the medium. Since the variation in wireless propagation delay is minimal for UWB transmissions, we statically computed a fixed propagation delay to compensate for the synchronization bias. We then test the synchronization accuracy (offset) of two nodes (one acting as a master and the other as slave) with and without the *Delay_ Req* messages. As seen in Fig 5.3(a), the mean offset is almost 0.3 nsec with a standard deviation of 2.27 nsec for the case when no

Delay_ Req messages are exchanged. The offset standard deviation increases to 5.3nsec as shown in Fig 5.3(b) when we enable periodic *Delay_ Req* messages, which is mostly because of the transmission delay jitter, and clock quantization noise.

We conclude from these results that in this wireless setting, where path delay does not change much, and we can estimate this delay through statistical means, it is better to reduce the number of synchronization messages by keeping the periodic *Sync* messages, and exchanging *Delay_ Req* messages only when needed and not in a periodic fashion. This takes us to our next result. We plotted the effect of synchronization period to the synchronization accuracy in Fig 5.4. We argue that accuracy is a logarithmic function of the synchronization interval, and a synchronization protocol should exchange messages only as often as needed to achieve the required synchronization accuracy, and not more. There is no need to send *Sync* and *Delay_ Req* messages beyond the required accuracy.

5.6 Key Findings

gPHC abstraction assist developers to access high precision over commodity devices and peripherals such that the same synchronization protocols can be run on heterogeneous devices greatly reduces developer burden, and eases the reuse of existing and well tested protocols. Also, since most of the coordinated IoT applications are composed of heterogeneous devices and posses high precision requirements, gPHC provides ease of integration and hardware-agnostic solutions. Our results show that we provide the same level of synchronization performance using a software based solution with no dependence on a specific hardware.

5.7 Conclusion

For NICs capable of timestamping events in hardware, we have provided a generalized guide to enable network sockets based hardware timestamping in Linux. This is the key to run PTP synchronization over a network interface. We have also implemented a precise hardware clock for a wireless network interface using the PTP clock infrastructure. We also explored the PTP clock abstractions beyond the radio interfaces, and extend the precision time support to other peripherals like processors and co-processors over gPHC.

Part IV

Systems for Testing Timing Robustness

CHAPTER 6

OpenClock: A Testbed for Clock Synchronization Research

6.1 Introduction

Hardware capabilities required for clock synchronization have developed significantly in the past decade; hardware timestamping feature is introduced for many processors, co-processors [AAZ17], and network interface cards [AS17], and new timing abstractions have been added in operating systems for precise timing [ADS16]. Systems have extensively made use of time-based technology developments to push for higher timing accuracy. These systems, however, lack a comprehensive testing environment to test and compare synchronization algorithms.

Industrial and automotive applications heavily rely on clock synchronization. These applications operate under uncertain environments and prone to hardware faults, network failures, or man in the middle attacks. To develop a clock synchronization algorithm that is robust to faults, failures, and attacks, comprehensive testing is necessary before practical deployments in safety-critical applications. Unfortunately, many algorithms are not tested for faults and attacks as it is hard to reproduce them on distributed devices.

Due to hardware characteristics of a clock, no two clocks are the same; Cho et al. [CS16] have used unique clock characteristics for fingerprinting electronic control units in cars. To compare multiple synchronization algorithms in a fair manner, their disciplinable clock models should be derived from the same hardware clock. A clock model is affected by short and long term variations in jitter, wander, and skew due to physical characteristics

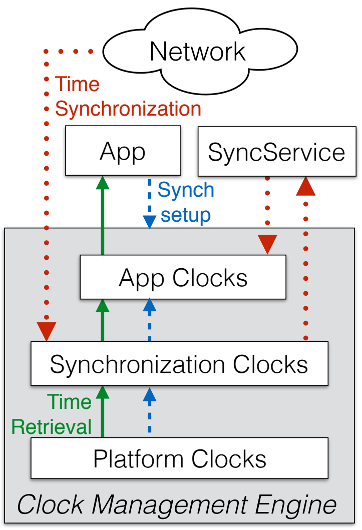


Figure 6.1: Testbed architecture

of oscillators and environmental variations due to temperature and aging.

For fair algorithmic comparison, and the ability to test algorithms under faults and attacks, we assert that it is essential to provide a clock synchronization testbed on a single host, as a single platform is subject to the same hardware and network conditions. The big challenge however in testing clock synchronization algorithms on a single platform is the absence of multiple similar clocks. We propose OpenClock, a testbed that supports multiple virtual clocks derived from the same physical clock, alleviating the bias in results from the physical and network characteristics. OpenClock consists of multiple components as shown in Figure 6.1. A clock management engine initializes and manages three layers of clocks: platform clocks that define the hardware timing capabilities, synchronization clocks that assist time synchronization protocols, and application clocks that provide the notion of time to applications.

As shown in Figure 6.1, OpenClock supports multiple clock synchronization protocols in three key steps. First, the dashed (blue) line represents the initial setup that bootstraps the required clocks for synchronization. Second, the dotted (red) line shows the interactions among clocks, synchronization services, and the network to perform clock synchronization and discipline the virtual clocks. Finally, the solid (green) line retrieves time from disciplined virtual clocks and transfer it to applications. These three steps are necessary for testing a clock synchronization algorithm offered by OpenClock testbed.

OpenClock also has the capability of testing algorithms in the presence of attacks. It consists of a network attack simulator that can be used to inject different kinds of attacks. The attack simulator imitates an adversary sitting on a network element that can arbitrarily delay victim packets in the network. Users leverage the attack simulator to test the conditions under which their algorithms fail. Users can also design new algorithms and test their resilience to attacks. To demonstrate the usage of OpenClock, we test different use cases in this paper. The design of OpenClock is modular, extensible, and it is open source. Developers can extend the testbed functionalities will minimal effort.

In this work, we lay the foundation of a testbed for clock synchronization research. This testbed can be extended in many ways, from supporting tunable synchronization parameters to providing attacks and faults primitives.

6.2 Hierarchy of Clocks

Clocks are the essential part of time synchronization protocols because they do timekeeping, timestamping, and scheduling. A clock is represented by a timing stack. This stack consists of an oscillator that oscillates at a particular frequency. The frequency of oscillation corresponds to the clock resolution. A counter counts the oscillations, and a software converts those counts to a human readable time in seconds. The ability of a clock to measure small time intervals is limited by its resolution, and a clock can be no more accurate to some reference time than its resolution.

A clock is either a software abstraction in the operating system or a logical mapping in the application. Our proposed clock synchronization testbed OpenClock consists of a hierarchy of clocks as shown in Figure 6.1. The clocks at the bottom of the hierarchy are *(A) platform clocks*; they represent the timing characteristics of a particular hardware. The middle layer in the clock hierarchy is comprised of *(B) synchronization clocks*; they are derived from the platform clocks and assist time synchronization protocols. The clocks at the top of the hierarchy are *(C) application clocks*; they are derived from synchronization clocks, and exposed to applications for timekeeping, and timestamping.

6.2.1 Platform Clocks

Platform clocks define the timing capabilities of a given platform. There are three types of platform clocks; a 1) system clock, a 2) precise hardware clock, and a 3) peripheral clock. Every device has at least one of these clocks. Below, we explain the types of platform clocks in reference to the Linux operating system.

1. System Clock provides a local sense of time to the operating system and user processes. System time is the number of time units passed since an epoch e.g, POSIX-compliant systems such as Linux count seconds since 1st January 1970. It's timing stack is shown in Figure 6.2 (middle vertical path). Linux uses hardware counters as the basis for higher-level clock abstractions: the `clocksource` encapsulates a non-wrapping hardware counter. The clocksource contains a member-function for reading the hardware counter, and `mult` and `shift` parameters that convert the counter value into nanoseconds for timekeeping. `CLOCK_REALTIME` is the Linux system clock that is exposed to userspace via the standardized POSIX clock interface. This interface allows the system clock to be disciplined using synchronization algorithms such as NTP [Mil91] and PTP [LEW05]. We refer to the system clock as `SYS` from now onward.

2. Precise Hardware Clock (PHC) is capable of timestamping events and scheduling tasks in hardware [CM10]. Unlike system clock, a PHC provides hardware pins for precise timestamping and accurate interrupt generation. In Figure 6.2 (left vertical path), an oscillator source is used to drive a hardware counter, from which an overflow-safe logical PTP clock `ptp_clock` is derived using the `cyclecounter` and `timecounter` abstractions. This clock is exposed to userspace as a PTP character device. It extends a POSIX clock interface and implements hardware pin functionalities. A PTP clock is also called a PHC. A userspace daemon can synchronize this clock to other PHCs in the network. PHC provides high synchronization accuracy and low synchronization jitter because it timestamps network packets in hardware. We refer to a PHC clock as `PHC` from now

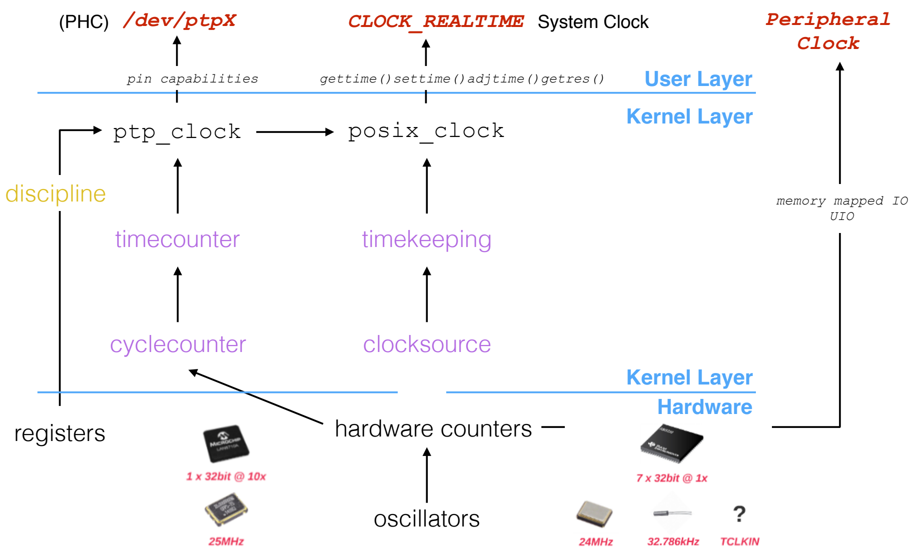


Figure 6.2: The timing stack of three different Platform Clocks in Linux OS

onward.

3. Peripheral Clock is a bare-metal peripheral clock on a platform such as a co-processor clock. The applications running on an operating system can access this clock via the Userspace I/O (UIO) Linux kernel subsystem as shown in Figure 6.2 (right vertical path). UIO maps regions of the peripheral clock memory and registers directly into userspace, with a small amount of kernel-space code to handle interrupts. This allows most of the driver logic to run in userspace instead of kernel-space, reducing the need for debugging kernel modules. UIO is used because it has low latency and is supported by both old and new Linux kernels. Thus a peripheral's clock is accessed at the userspace via UIO. We refer to this clock as `PPHL` from now onward.

6.2.2 Synchronization Clocks

For the working of a time synchronization protocol, two clocks are needed. A local clock that is used for timekeeping, and a network interface clock that timestamps the incoming and outgoing packets at the interface. For most of the platforms and synchronization protocols, a single clock is used both for timekeeping and packets timestamping (NTP [Mil91], PTP [LEW05], FTSP [MKS04]). In OpenClock, we assert that the usage of different clocks for timekeeping and timestamping in a synchronization protocol provides a modular, and extensible design. We refer to these clocks as synchronization clocks that are derived from platform clocks.

1. Core Clock is a synchronization clock in OpenClock testbed. It does timekeeping by providing a `core` sense of time to the entire platform. For a clock to qualify as a core clock, it must provide the ability to read a strictly monotonic counter that cannot be altered. Core clocks may also provide interfaces to expose platform-specific functionality such as, timestamping and generating interrupts at precise time.

2. Network Interface Card Clock (NICC) is the second synchronization clock in OpenClock testbed. It timestamps the synchronization packets at the network interface. Only those network interfaces that can accurately timestamp packet transmission and reception – at the physical or MAC layers – are exposed as NICC. This enables clock synchronization protocols to precisely estimate the offset between two clocks, and the propagation delay associated with a medium. Like core clocks, NICC provide the ability to read time, and optionally provide I/O functionality for precisely timestamping an event, or generating a very deterministic pulse in the future. However, a NICC is not necessarily monotonic and it can be disciplined. In addition, NICC do not provide the ability to generate interrupts, and cannot be used to accurately schedule user-level application threads.

NTP only has a core clock that also timestamps network packets, while PTP does not have a core clock and disciplines a network clock. A user process cannot use PTP clock to schedule events. In OpenClock, we provide synchronization clocks and mandates the use of a core clock while making NICC usage optional. However, we believe that both types of clocks are required for precise clock synchronization and scheduling events. Depending upon the hardware capabilities of a platform, and synchronization requirements of an application, a core clock can be derived from any platform clock, whereas, NICC can only be derived from a `PHC` or a `PPHL`.

6.2.3 Application Clocks

For fair algorithmic comparison, we establish that multiple disciplinable clocks are needed on a single platform. Currently, the platforms and operating systems only provide a single disciplinable clock. In OpenClock, we leverage a timing abstraction called a `timeline` [ADS16] that derives its time from a core clock projection. That way, multiple timelines or virtual clocks are derived from a single core clock on one platform. The timeline abstraction was first proposed by Anwar et al. [ADS16]; they used timelines to synchronize distributed clocks with desired accuracy. Note that in OpenClock, timelines are the application clocks, and the synchronization protocols discipline these application clocks. The synchronization clocks (core and NICC) are not disciplined, they are only used to assist the synchronization protocol.

Timeline maintains a virtual time base with respect to an epoch. The timeline abstraction enables the OS to provide as many disciplinable clocks as needed by the applications. Timelines enforce isolation among different synchronization algorithms and their respective clock adjustment routines by providing a unique disciplinable virtual clock to each synchronization protocol. An application creates a timeline with a unique uuid and specifies its synchronization requirements and protocols. These protocols synchronize their respective timelines under the given requirements. Hence the timeline abstraction provides a natural isolation among multiple protocols running on the same platform.

6.3 OpenClock Architecture

OpenClock is a clock synchronization testbed that is used to compare different synchronization algorithms on a single platform. Figure 6.3 shows the OpenClock architecture. It consists of a *clock management engine* that interacts with *synchronization services* to manage and discipline timelines. The detailed procedure is: (1) an application registers its synchronization requirements with the clock management engine. (2) This engine assigns a platform clock to core clock and NICC according to the requirements. (3) If a

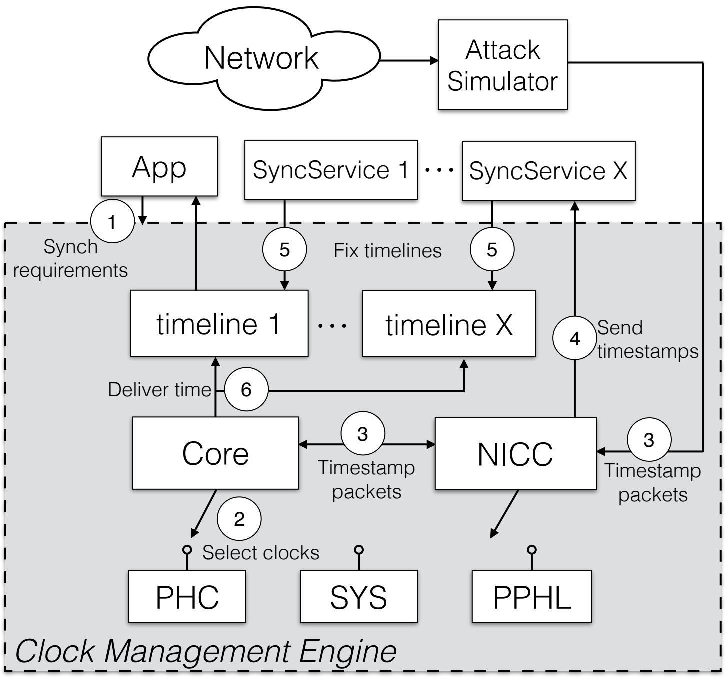


Figure 6.3: Clock synchronization steps in OpenClock architecture

different platform clock is assigned to core and NICC, they both should synchronize to each other and present the same time. NICC timestamps network packets to synchronize to a reference time, and (4) send those timestamps to all the synchronization services. (5) These services utilize the timestamps in their algorithms to calculate clock disciplining parameters, and fix their respective timelines. Note that each synchronization service disciplines its corresponding timeline. (6) The applications can then retrieve synchronized time from their timelines, and compare the performance of different synchronization algorithms. We now cover the key components of OpenClock in detail.

6.3.1 Clock Management Engine

The clock management engine initializes and manages the hierarchy of clocks (explained in Section II). This engine assess the hardware timing capabilities of a platform and expose them as platform clocks. `SYS` is present on every platform and operating system. However, this clock does not have the hardware capabilities to measure precise intervals or schedule tasks at precise time instants. `PHC` have these hardware capabilities but it's

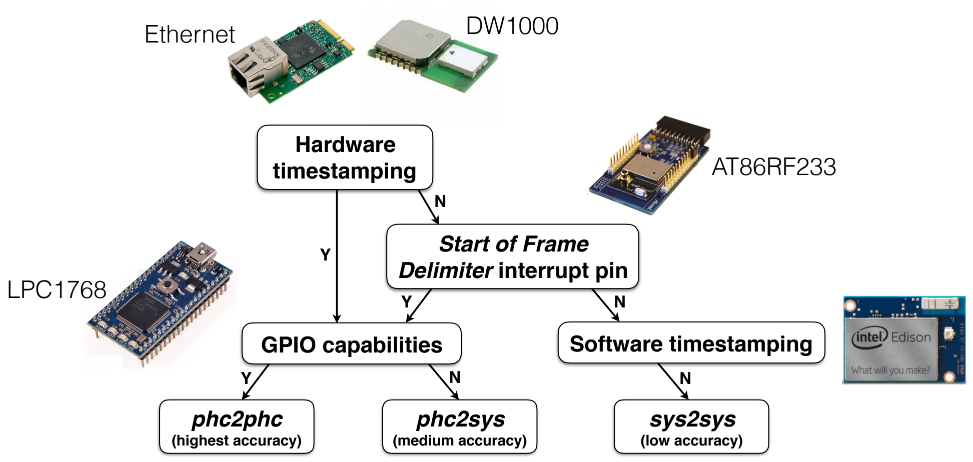


Figure 6.4: Hardware capabilities of clocks influence synchronization performance (adapted from [ADS16])

been tied to ethernet interfaces on certain platforms. In our previous work [AS17], we show if a network interface support certain hardware features, we can expose it as a PHC by writing drivers for it. We propose that besides network interfaces, a processor clock or a co-processor clock can also be converted to a PHC, given that they have necessary hardware functionalities. In this work, we expose a processor (AM335X) as a PHC because it is capable of timestamping and scheduling hardware events. We use the processor timer and wrote a kernel module to transform it into a PHC. The clock management engine also initializes any PPHL on a platform. For example, on the beaglebone black platform, there is a Programmable Realtime Unit (PRU) that has good timing capabilities. Our previous work cyclops [AAZ17] exposes PRU as a timing device and synchronizes it to the processor clock. We utilize the PRU clock as PPHL in OpenClock, and we believe similar peripherals on other platforms can provide a PPHL platform clock. Hence a SYS, PHC, and PPHL are initialized and managed by the clock management engine.

After initializing the platform clocks, the clock management engine assigns the platform clocks to synchronization clocks based on the synchronization requirements of an application. The engine also dynamically switches core clocks and NICC to different platform clocks. For example, if a user wants to run NTP, the engine selects core clock as SYS and leave the NICC empty. If the user wants to run PTP, core clock is selected as SYS and NICC as PHC. To run the QoT Stack [ADS16] with highest accuracy, both core clock and NICC are

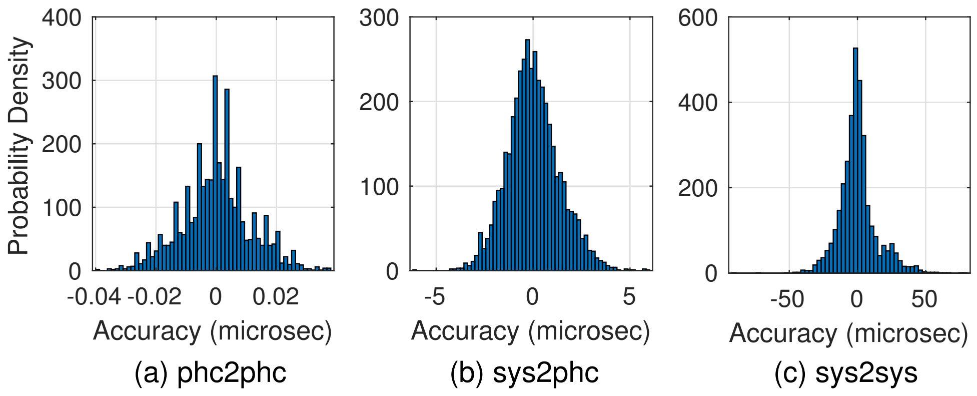


Figure 6.5: Comparison of time accuracies for different synchronization clocks

chosen as PHC.

We know that the hardware capabilities influence the accuracy of a synchronization protocol. In Figure 6.4, the flow chart and the accompanying hardware platforms provide the following guideline: given certain capabilities, how would you traverse down the flowchart to determine which accuracy the synchronization protocol will achieve. For example, given an Intel edison platform, the clock management engine cannot select the core clock or NICC as a PHC because edison is incapable of hardware timestamping. On the other hand, for a DW1000 platform interfaced with Beaglebone Black, the engine selects both the core and NICC to be a PHC [ADS16]. For LPC1768, only core clock is a PHC, while NICC is empty.

To test the effect of platform clock on the synchronization accuracy, we run an experiment to synchronize the core clock with the NICC on one device. We use three different combinations of SYS and PHC. In the first experiment, both the core and NICC are PHC, and the synchronization accuracy achieved is in the order of nanoseconds as shown in Figure 6.5a. In the second experiment, we choose core as SYS and NICC as PHC, and the accuracy reduced to microseconds in Figure 6.5b. In the last experiment, we choose both the core and NICC as SYS. The accuracy reduced even further to 10s of microseconds in Figure 6.5c. Thus we establish that the choice of PHC for a core clock and NICC provides the highest accuracy.

In OpenClock, besides using the traditional protocols, users can write their own synchronization protocols and choose any platform clock to act as core or NICC. The user also has the advantage of testing established time synchronization algorithms such as NTP and PTP with different clock settings. A user can also specify its own clocks of choice, if however a user doesn't specify the synchronization clocks, the engine choose `SYS` to be the default core clock as it is available on every platform. The engine also maintains default clock settings for known synchronization protocols but they can be overridden if desired.

After assigning desired platform clocks to synchronization clocks, the engine creates timelines as application clocks. Recalling from Figure 6.3, a timeline is a projection of core clock's time. The engine maintains the projection parameters for all timelines, and provides an interface to all synchronization services to change the projection parameters of their respective timelines.

6.3.2 Synchronization Service

As shown in Figure 6.3, multiple synchronization services work with the clock management engine in OpenClock. A synchronization service utilizes the engine's interface to discipline its timeline. OpenClock also provides multiple parameters that tune the performance of a synchronization algorithm. The two tunable parameters are, the 1) synchronization interval, and the 2) clock discipline mechanism. These tunable parameters can be changed during initialization or at runtime. The performance is enhanced by reducing the first tunable parameter i.e. the synchronization interval. To tune the other parameter, one has to choose between two different mechanisms to discipline a clock. A `feedback` mechanism timestamps packets and calculates synchronization parameters from the disciplined clock. On the other hand, a `feedforward` mechanism is based on a clock that is never disciplined. The calculated synchronization parameters reflect the local clock's relative drift with respect to global time.
Using OpenClock, users can write their own synchronization algorithms and specify their own tunable parameters.

6.3.3 Network Attacks Simulator

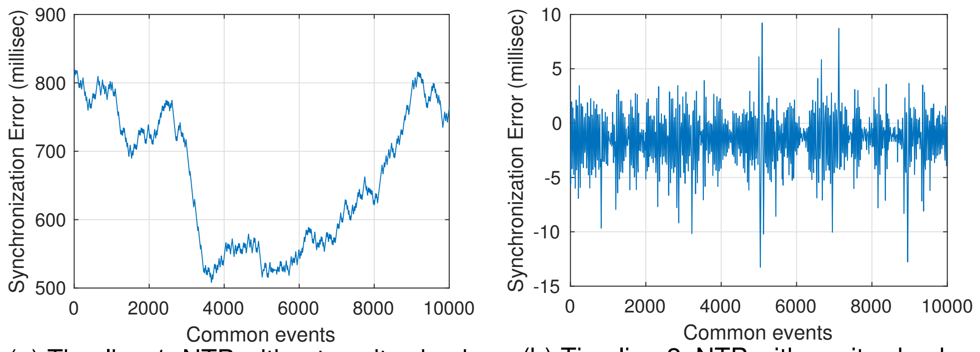
OpenClock supports comparison of multiple algorithms under fair conditions. It also provides an opportunity to compare algorithmic resilience to attacks on network packets. With the system and network attacks on the rise, there is a need to design algorithms that are both resilient to faults and attacks. OpenClock lets the user test its algorithms for various kinds of attacks by providing a network attack simulator. This simulator injects delays in packets transmission and reception as shown in Figure 6.3. These attacks compromise the accuracy of time by delaying the synchronization packets. The attacks can be injected in both forward and reverse paths in the network. The attacked packets are fed to the synchronization algorithms that need to be compared.

To use the OpenClock testbed, a user provides a configuration file to the system. This file specifies the name of timelines, types of clocks, protocols for timelines along with their tunable parameters, and an attack indicator. One example configuration file is shown below,

```
timeline1{
            Core: SYS
            NICC: PHC
            SYNC:{
                    protocol: PTP
                    servo: feedforward
                    interval: 1
            }
            attack: true
}

timeline2{ ... }
```

Here, the user has defined two timelines. On one timeline, she configures core clock as SYS and NICC as PHC running PTP algorithm that synchronizes to a master every second through an attacked network medium using a feedforward clock disciplining mechanism.



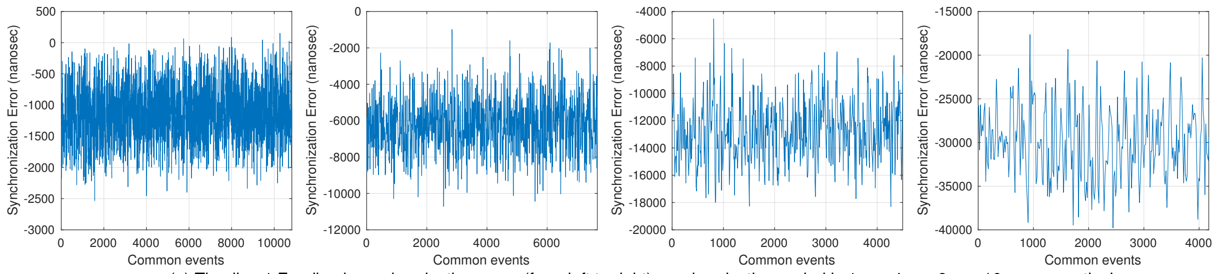
(a) Timeline 1: NTP without sanity check (b) Timeline 2: NTP with sanity check

Figure 6.6: Comparison of synchronization accuracies for unmodified NTP running in Timeline 1, and modified NTP running on Timeline 2 under network attacks. (a) Timeline 1 fluctuates and accumulates large error over time, while (b) Timeline 2 accumulates less error

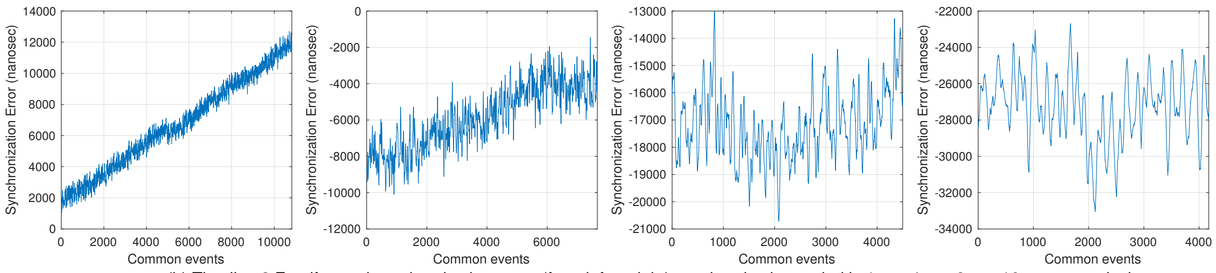
6.4 Evaluation

The purpose of providing a testbed for clock synchronization protocols is to compare multiple algorithms under same hardware and network conditions. OpenClock leverages timeline to provide multiple disciplinable clocks on a single platform and run multiple protocols. Below, we provide three use cases that show case different ways in which OpenClock could be used. Nonetheless, the usage of this testbed is not limited to these test cases.

Use Case 1: Effect of network attacks on synchronization error: When a network packet moves from client to server, the adversary in a network router can delay the packet. We refer to it as the *forward path attack*. The adversary can also delay the packet in the opposite direction i.e. when the packet moves from server to client in the network. We term it as *reverse path attack*. Multiple protocols are subjected to these attacks. In NTP, if an attacker is able to attack sufficient number of packets, it can manipulate the Marzullo's algorithm [MO85] to converge to a time desired by the attacker. We simulate both forward and reverse path attacks in OpenClock. We use these attacks to understand how NTP algorithms can be fooled. We run unmodified NTP on one timeline, and assume that the attacker is compromising more than half the packets from the NTP servers. By adding 2 second forward path delay to 4 out of 6 packets coming from 6 different NTP servers in



(a) Timeline 1 Feedback synchronization error: (from left to right) synchronization period is 1sec, 4sec, 8sec, 16sec respectively



(b) Timeline 2 Feedforward synchronization error: (from left to right) synchronization period is 1sec, 4sec, 8sec, 16sec respectively

Figure 6.7: Visualizing effect of two tunable parameters in this figure, the first parameter is clock discipline mechanism that changes from top to bottom row. The second parameter is synchronization period that changes from left to right. Synchronization error increases with an increase in synchronization period. This is evident in both (a) and (b) going from left to right. Comparison of feedback (top) and feedforward (bottom) and for 1 second synchronization period shows that feedback performs well for small periods. As we move to higher periods, the feedforward error tends to converge

the network, we slowly shift the timeline's notion of time away from the global time. We then modify NTP algorithm by adding a sanity check. This check sorts the timestamps from multiple servers based on their offsets, then carefully discards first half or the last half of NTP timestamps. We run the unmodified NTP on timeline1 and modified NTP on timeline2 at the same time processing the same attacked network packets. Our results in Figure 6.6 show that the attacks led to an increase in unmodified NTP's synchronization error, while the modified NTP experience less degradation in synchronization accuracy.

So far in this work, we simulate network attacks and test for resilience against those attacks. We can also test for hardware attacks (temperature variations etc.) by simulating their effect in the testbed.

Use Case 2: Effect of clock discipline mechanism on synchronization error: To understand which clock discipline mechanism enhances system performance, we run PTP on two timelines. Timeline 1 employs feedback discipline using PTP PI servo while

Timeline 2 runs feedforward discipline using linear regression. Both timelines choose `SYS` for core clock and `PHC` for NICC. The results for both the timelines is shown in Figure 6.7. We explain the results with the next use case.

Use Case 3: Effect of synchronization parameters on error: There exists multiple tunable parameters in a synchronization protocol that can affect the performance. For example, to study the affect of synchronization period on the error, we run PTP with a feedback PI servo on two timelines. Both timelines use `SYS` for core clock and `PHC` for NICC. The only difference between the two timelines is that Timeline 1 has a synchronization period of 1 second while Timeline 2 has a period of 4 seconds. The results in Figure 6.7 show that both feedback and feedforward have different impact on synchronization error with increase in period. Feedforward performs better at higher periods because it disciplines clock by measuring relative drift that can only be measured accurately over long durations.

6.5 Key Findings

OpenClock is the first testbed for clock synchronization research that alleviates hardware and network biases in testing mechanisms and provide new means to test multiple algorithms for failures and attacks at the same time. It provides a rich suite of clocks with a modular and extensible design. We evaluate the performance of a number of protocols and show hardware and network biases in the results when these protocols are tested on different devices, whereas biases are alleviated from results when these protocols are tested on OpenClock. Our results also show that algorithms on OpenClock can be tested simultaneously by simulating various conditions and attacks in real time.

6.6 Conclusion

When designing new time synchronization algorithms, they are compared with the established algorithms on distributed platforms. We argue that such comparisons are not fair as distributed platforms are subject to hardware and network variabilities. Moreover, many algorithms are not tested for attacks as it is hard to simulate the same attacks on multiple platforms. We design and implement OpenClock, a real testbed on a single platform that overcomes the above limitations: (1) it provides multiple disciplinable clocks on a single platform to circumvent hardware variability, (2) it provides adjustable parameters for timelines to tune the synchronization performance. (3) OpenClock also presents an attack simulator that injects the same attack to all algorithms under test. The benefit of an attack simulator is two folds: it can help find vulnerabilities in an algorithm, and help test resilience of algorithms under those attacks. OpenClock can be extended in multiple ways. Developers can write their own synchronization algorithms, present new dynamically tunable parameters, and define new hardware and network attacks. Our modular design based on clocks hierarchy and timelines provides programming and porting flexibility.

CHAPTER 7

Conclusion and Future Research

7.1 Conclusion

This dissertation presented Quality of Time (QoT) that brings a new paradigm for distributed system design particularly for Cyber-Physical Systems and Internet of Things. We conclude this thesis by summarizing the systems designed to enable QoT for applications running on commodity platforms and operating systems under timing variations and vulnerabilities. Our thesis contribution to enable QoT is categorized into four main parts:

7.1.1 Systems for Characterizing Timing Uncertainty

In this part, we presented a new operating system abstraction – timeline – that helps design coordinated applications through a thorough set of API. Built around the timeline abstraction is an architecture that orchestrates system resources to provide the required QoT to applications. The timeline abstraction with its associated notion of Quality of Time (QoT) helps virtualize time-related resources in a system and plays a role analogous to that of sockets with associated Quality of Service (QoS) bindings in network stacks. QoS-aware networking applications can read, write, open and close sockets, and specify QoS parameters. Similarly, QoT-aware time-sensitive applications can bind and unbind from timelines, read and schedule events on the timeline reference, and specify QoT requirements. We make QoT visible and controllable in our timeline-driven architecture that enables time-aware applications to specify their timing requirements, while the system manages clocks and synchronization protocols to provide the appropriate levels of QoT.

7.1.2 Systems for Timing Integrity

In this part, we present challenges in providing a secure clock shared across distributed entities. Security in the context of time is vital as many applications are emerging that have moved away from traditional "clockless" assumption [GLY18]. Systems are increasingly making use of synchronized clocks to enhance the accuracy of network measurements and reduce the complexity of distributed system protocols. On the other hand, adversaries are targeting timing primitives for copyright theft, illegal trade, and location theft, etc. To provide timing integrity, we present TimeSeal, a new secure time architecture that leverages trusted execution for hardware timer protection and eliminates timing limitations and vulnerabilities in trusted execution to secure time. TimeSeal provides a secure local clock that is good for measuring time durations. In other words, it "seals" time so that no privileged adversary can arbitrarily manipulate a local clock, and compromise the safety and performance of applications. For a secure global clock, we establish that cryptography and network security mechanisms thwart various attacks on time transfer packets but delay attack is too strong to be mitigated completely. We pointed out the key issues in current clock synchronization architectures that make them vulnerable to delay attacks and propose a new delay attack-tolerant synchronization architecture. Built on top of a feedforward control with feedback trim clock adjustment mechanism coupled with packet filtering techniques, the architecture is capable of bounding delay attack errors.

7.1.3 Systems for Timing Precision

High timing precision is required by many applications across a broad spectrum of applications. We have provided a generalized Precise Hardware Clock (gPHC) abstraction that enables high precision for applications. For NICs capable of timestamping events in hardware, we have provided a generalized guide to enable network sockets based hardware timestamping in Linux that is the key to run PTP synchronization over a network interface. We have also implemented a precise hardware clock for a wireless network interface using the PTP clock infrastructure as a proof of concept. We also support other peripherals and processors beyond the radio interfaces and extend the precision time support to other peripherals like processors and co-processors over gPHC.

7.1.4 Systems for Testing Timing Robustness

When designing new time synchronization algorithms, they are compared with the established algorithms on distributed platforms. We argue that such comparisons are not fair as distributed platforms are subject to hardware and network variabilities and security vulnerabilities. We design and implement OpenClock, a real testbed on a single platform that overcomes the above limitations: (1) it provides multiple disciplinable clocks on a single platform to circumvent hardware variability, (2) it provides adjustable parameters for virtual clocks to tune the synchronization performance. (3) OpenClock also presents an attack simulator that injects the same attack to all algorithms under test with two-fold benefits: it can help find vulnerabilities in an algorithm, and help test resilience of algorithms under those attacks.

The implementations of the systems designed in this thesis are open-source and available at the following code repositories,

- QoT Stack: `https://bitbucket.org/rose-line/qot-stack`
- TimeSeal: `https://bitbucket.org/rose-line/linux-sgx-secure-clock`
- gPHC: `https://bitbucket.org/rose-line/ptp-wpan-6lowpan`
- OpenClock: `https://bitbucket.org/fatimanwar/openclock-testbed`

7.2 Future Directions

In the next few years, modern applications in IoT such as health care, connected vehicles, digital assistants, and augmented/virtual reality are going to bridge the gap between the edge devices and the cloud. Similar to the current theme, the research goal would be to design and build high-performance, secure systems that tackle various issues across all levels in a cooperative fashion with an emphasis on developing practical and functional systems.

7.2.1 Spatiotemporal awareness in distributed applications

Time and space are intertwined and their relationship assists distributed applications spanning the edge and the cloud. Enabling spatiotemporal awareness for distributed applications – such as swarms – in the presence of intermittent network connection and adversarial scenarios remains an exciting research challenge. A principled approach to design is suitable, where we first design the essential components such as precise time and frequency synchronization, accurate localization, high-resolution inter-event time measurement, distance bounding technique, and co-location detection mechanisms. Afterward, integrate these components to build reliable and secure spatiotemporal services. Along this direction, developing high-level abstractions and API to simplify access to spatiotemporal services across heterogeneous platforms is also a research challenge.

7.2.2 Increased responsiveness in interactive applications

IoT applications such as cloud gaming and AR/VR require real-time interactivity in the face of wide-area latency. Round trip delays as little as 60 milliseconds significantly depreciate users experience. The challenges are manifolds in this domain: to bring distributed servers closer to clients increase cost, average network latencies are high, and buffering does not work for interactive applications. There exist several research challenges in this domain, including designing systems that mask the high latency for end users in a secure fashion using speculative execution with machine learning models.

7.2.3 Secure peripherals in trustworthy platforms

Emerging Trusted Execution Environments (TEE) such as ARM TrustZone and Intel SGX limit security features to the boundary of a CPU with no support of secure resources/peripherals, i.e., secure storage, clock, counter, and entropy. Dedicated trusted hardware such as Trusted Platform Module (TPM) provides secure resources. However, TPM does not have a powerful processor, no flexibility for application development, and only accessed by a fixed set of API guided by a standard. Even worse, applications running on ARM TrustZone secure world and Intel SGX enclaves only access TPM resources via an untrusted and unreliable communication channel. There is a need to conduct extensive research in hardware and system security to provide software systems running on trusted CPU with similar security guarantees as dedicated, trusted hardware. One hurdle in the way of trusted hardware support is the lack of full stack open source implementations of current TEEs. There are some initiatives such as Keystone project that provides a new open source TEE based on RISC-V architecture, but they lack secure peripherals. A preliminary investigation by thoroughly examining the shortcomings of current TEE implementations and putting forward recommendations for vendors to support secure resources outside the CPU perimeter is needed. Providing trusted resources beyond the CPU remains a fundamental challenge, and requires research to design trustworthy platforms.

7.2.4 Side-channels on secure computation

Hyperthreading is a side-channel by design. Two notable systems that are vulnerable to side-channel are microarchitectures and browsers. The key attack enabler is the attacker's ability to precisely measure the time a victim process takes to access its contents. Caches and Translation Lookaside Buffer (TLB) have been identified as microarchitectural resources assisting side-channel attacks. Shared event loops are also susceptible to timing attacks in browsers. Recent work has also shown that it is possible to emulate a short-lived precise timer inside an enclave to perform a Prime+Probe cache side-channel attack against co-located enclaves. Previous prevention techniques remain ineffective: minimizing the effect of a process's state on microarchitectural resources requires a clean slate design; making all computations in a process take constant time is a challenging task, and restricting precise time access only to trusted entities is an open research area. A future direction is to rethink designs from the ground up, and overcome attacks without compromising system performance.

REFERENCES

[AAA19] Kazunori Akiyama, Antxon Alberdi, Walter Alef, Keiichi Asada, Rebecca Azulay, Anne-Kathrin Baczko, David Ball, Mislav Baloković, John Barrett, Dan Bintley, et al. "First M87 Event Horizon Telescope Results. II. Array and Instrumentation." *The Astrophysical Journal Letters*, **875**(1):L2, 2019.

[AAS18] Fatima M Anwar, Amr Alanwar, and Mani B Srivastava. "OpenClock: A Testbed for Clock Synchronization Research." In *2018 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS)*, pp. 1–6. IEEE, 2018.

[AAZ17] Amr Alanwar, Fatima M Anwar, Yi-Fan Zhang, Justin Pearson, Joao Hespanha, and Mani B Srivastava. "Cyclops: PRU programming framework for precise timing applications." In *Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS), 2017 IEEE International Symposium on*, pp. 1–6. IEEE, 2017.

[ABG17] Gildas Avoine, Xavier Bultel, Sébastien Gambs, David Gerault, Pascal Lafourcade, Cristina Onete, and Jean-Marc Robert. "A terrorist-fraud resistant and extractor-free anonymous distance-bounding protocol." In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 800–814. ACM, 2017.

[Ad07] L. Auler and R. d'Amore. "Adaptive kalman filter for time synchronization over packet-switched networks (an heuristic approach)." In *IEEE COMSWARE*, 2007.

[ADS16] Fatima Anwar, Sandeep D'souza, Andrew Symington, Adwait Dongare, Ragunathan Rajkumar, Anthony Rowe, and Mani Srivastava. "Timeline: An Operating System Abstraction for Time-Aware Applications." In *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pp. 191–202. IEEE, 2016.

[AF07] Patrik Arlos and Markus Fiedler. "A method to estimate the timestamp accuracy of measurement hardware and software tools." In *International Conference on Passive and Active Network Measurement*, pp. 197–206. Springer, 2007.

[AFZ17a] Robert Annessi, Joachim Fabini, and Tanja Zseby. "It's about Time: Securing Broadcast Time Synchronization with Data Origin Authentication." In *Computer Communication and Networks (ICCCN), 2017 26th International Conference on*, pp. 1–11. IEEE, 2017.

[AFZ17b] Robert Annessi, Joachim Fabini, and Tanja Zseby. "SecureTime: Secure multicast time synchronization." *arXiv preprint arXiv:1705.10669*, 2017.

[AS17] Fatima M Anwar and Mani B Srivastava. "Precision time protocol over LR-WPAN and 6LoWPAN." In *Precision Clock Synchronization for Measurement,* *Control, and Communication (ISPCS), 2017 IEEE International Symposium on*, pp. 1–6. IEEE, 2017.

[ATG16] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark Stillwell, et al. "SCONE: Secure Linux Containers with Intel SGX." In *OSDI*, volume 16, pp. 689–703, 2016.

[AW80] David W Allan and Marc Abbott Weiss. "Accurate time and frequency transfer during common-view of a GPS satellite." In *34th Annual Symposium on Frequency Control. 1980*, pp. 334–346. IEEE, 1980.

[BCR10] Timothy Broomhead, Laurence Cremean, Julien Ridoux, and Darryl Veitch. "Virtualize Everything but Time." In *OSDI*, volume 10, pp. 1–6, 2010.

[BHE00] Nirupama Bulusu, John Heidemann, and Deborah Estrin. "GPS-less low-cost outdoor localization for very small devices." *IEEE personal communications*, **7**(5):28–34, 2000.

[Bla] Beaglebone Black. "https://beagleboard.org/black.".

[Ble05] A. Bletsas. "Evaluation of kalman filtering for network time keeping." In *IEEE Trans. on Ultrasonics, Ferroelectrics, and Frequency Control*, 2005.

[BPH15] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding applications from an untrusted cloud with haven." *ACM Transactions on Computer Systems (TOCS)*, **33**(3):8, 2015.

[BRV09] Timothy Broomhead, Julien Ridoux, and Darryl Veitch. "Counter availability and characteristics for feed-forward based synchronization." In *Precision Clock Synchronization for Measurement, Control and Communication, 2009. ISPCS 2009. International Symposium on*, pp. 1–6. IEEE, 2009.

[C 10] W. Choi C. Seong, S. Lee. "A new network synchronizer using phase adjustment and feedforward filtering based on low-cost crystal oscillators." In *IEEE TRANSACTIONS ON INSTRUMENTATION AND MEASUREMENT, VOL. 59, NO. 7*, 2010.

[C L14] R Wattenhofer C Lenzen, P Sommer. "PulseSync: An Efficient and Scalable Clock Synchronization Protocol." In *IEEE/ACM TRANSACTIONS ON NETWORKING*, 2014.

[CC09] David Culler and Samita Chakrabarti. "6LoWPAN: Incorporating IEEE 802.15. 4 into the IP architecture." *IPSO Alliance, White paper*, 2009.

[CDE13a] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. "Spanner: Google's globally distributed database." *ACM Transactions on Computer Systems (TOCS)*, **31**(3):8, 2013.

[CDE13b] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. "Spanner: Google's globally distributed database." *ACM Transactions on Computer Systems (TOCS)*, **31**(3):8, 2013.

[CM10] Richard Cochran and Cristian Marinescu. "Design and implementation of a PTP clock infrastructure for the Linux kernel." In *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2010 International IEEE Symposium on*, pp. 116–121. IEEE, 2010.

[CMR11] Richard Cochran, Cristian Marinescu, and Christian Riesch. "Synchronizing the Linux system time to a PTP hardware clock." In *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2011 International IEEE Symposium on*, pp. 87–92. IEEE, 2011.

[CRS14] Chen Chen, Himanshu Raj, Stefan Saroiu, and Alec Wolman. "cTPM: A Cloud TPM for Cross-Device Trusted Applications." In *NSDI*, pp. 187–201, 2014.

[CS16] Kyong-Tak Cho and Kang G Shin. "Fingerprinting Electronic Control Units for Vehicle Intrusion Detection." In *USENIX Security Symposium*, pp. 911–927, 2016.

[CZ17] Shanwei Cen and Bo Zhang. "Trusted Time and Monotonic Counters with Intel SGX Platform Services." `https://software.intel.com/sites/default/files/managed/1b/a2/Intel-SGX-Platform-Services.pdf`, 2017.

[CZR17] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. "Detecting privileged side-channel attacks in shielded execution with Déjá Vu." In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 7–18. ACM, 2017.

[DFL08] Patricia Derler, Thomas H Feng, Edward A Lee, Slobodan Matic, Hiren D Patel, Yang Zheo, and Jia Zou. "PTIDES: A programming model for distributed real-time embedded systems." Technical report, DTIC Document, 2008.

[DFR11] CM De Dominicis, A Flammini, S Rinaldi, E Sisinni, A Cazzorla, A Moschitta, and P Carbone. "High-precision UWB-based timestamping." In *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2011 International IEEE Symposium on*, pp. 50–55. IEEE, 2011.

[DR17] Sandeep D'souza and Ragunathan Rajkumar. "Time-based coordination in geo-distributed cyber-physical systems." In *Proceedings of the 9th USENIX Conference on Hot Topics in Cloud Computing*, pp. 9–9. USENIX Association, 2017.

[DSD18] Omer Deutsch, Neta Rozen Schiff, Danny Dolev, and Michael Schapira. "Preventing (Network) Time Travel with Chronos." In *Network and Distributed Systems Security Symposium (Proceedings of NDSS 2018). San Diego, CA, USA. http://dx. doi. org/10.14722/ndss*, 2018.

[EB12] P. V. Estrela and L. Bonebakker. “Challenges Deploying PTPv2 in a Global Financial Company.” In *Intl. IEEE Symposium on Precision Clock Synchronization for Measurement & Communication (ISPCS)*, 2012.

[ET03] John C Eidson and John Tengdin. “IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems and applications to the power industry.” In *Proc. Distributech*, pp. 4–6, 2003.

[Exe12] Reinhard Exel. “Clock synchronization in IEEE 802.11 wireless LANs using physical layer timestamps.” In *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2012 International IEEE Symposium on*, pp. 1–6. IEEE, 2012.

[F F11] L Thiele F Ferrari, M Zimmerling. “Efficient network flooding and time synchronization with Glossy.” In *Information Processing in Sensor Networks (IPSN)*, 2011.

[for17] intel sgx forums. “Intel(R) Software Guard Extensions forum, TSC discussion topic 743186.” `https://software.intel.com/`, 2017.

[G 13] B. Lantos G. Regula. “Formation control of a large group of UAVs with safe path planning.” In *IEEE 21st Mediterranean Conference on Control & Automation (MED)*, 2013.

[GLY18] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. “Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization.” In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, 2018. USENIX Association.

[GN06] Thomas Gleixner and Douglas Niehaus. “Hrtimers and beyond: Transforming the linux time subsystems.” In *Proceedings of the Linux symposium*, volume 1, pp. 333–346. Citeseer, 2006.

[GTG05] Sinan Gezici, Zhi Tian, Georgios B Giannakis, Hisashi Kobayashi, Andreas F Molisch, H Vincent Poor, and Zafer Sahinoglu. “Localization via ultra-wideband radios: a look at positioning aspects for future sensor networks.” *IEEE signal processing magazine*, **22**(4):70–84, 2005.

[HCS18] Jun Han, Albert Jin Chung, Manal Kumar Sinha, Madhumitha Harishankar, Shijia Pan, Hae Young Noh, Pei Zhang, and Patrick Tague. “Do you feel what I hear? Enabling autonomous IoT device pairing using different sensor types.” In *Do You Feel What I Hear? Enabling Autonomous IoT Device Pairing using Different Sensor Types*, p. 0. IEEE, 2018.

[HK08] Gerhard P Hancke and Markus G Kuhn. “Attacks on time-of-flight distance bounding channels.” In *Proceedings of the first ACM conference on Wireless network security*, pp. 194–202. ACM, 2008.

[HM93] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[HMZ08] Benjamin R Hamilton, Xiaoli Ma, Qi Zhao, and Jun Xu. "ACES: adaptive clock estimation and synchronization using Kalman filtering." In *Proceedings of the 14th ACM international conference on Mobile computing and networking*, pp. 152–162. ACM, 2008.

[HSA11] Mohammad Kamrul Hasan, Rashid Abdelhaleem Saeed, Aisha-Hassan Abdalla, Shayla Islam, Omer Mahmoud, Othman Khalifah, Shihab A Hameed, and Ahmad Fadzil Ismail. "An investigation of femtocell network synchronization." In *Open Systems (ICOS), 2011 IEEE Conference on*, pp. 196–201. IEEE, 2011.

[HSG18] Sean Hamilton, Dhiman Sengupta, and Rajesh Gupta. "Introducing Automatic Time Stamping (ATS) with a Reference Implementation in Swift." In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 138–141. IEEE, 2018.

[IEE] 2004 IEEE Standard 1003.1. "http://pubs.opengroup.org/onlinepubs/009695399/functions/clo

[Int18] Intel. "Intel(R) Developer Zone, Enclave Thread Scheduling Discussion, Topic 635939." `https://software.intel.com/en-us/forums/`, 2018.

[IW17] Eyal Itkin and Avishai Wool. "A security analysis and revised security extension for the precision time protocol." *IEEE Transactions on Dependable and Secure Computing*, 2017.

[J E02] D Estrin J Elson, L Girod. "Fine-grained network time synchronization using reference broadcasts." In *ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, 2002.

[Jih10] Deog-Kyoon Jeong Jiho Han. "A Practical Implementation of IEEE 1588-2008 Transparent Clock for Distributed Measurement and Control Systems ." In *IEEE TRANSACTIONS ON INSTRUMENTATION AND MEASUREMENT, VOL. 59, NO. 2*, 2010.

[KB03] Hermann Kopetz and Günther Bauer. "The time-triggered architecture." *Proceedings of the IEEE*, **91**(1):112–126, 2003.

[KB18] Anantha K Karthik and Rick S Blum. "Estimation Theory-Based Robust Phase Offset Determination in Presence of Possible Path Asymmetries." *IEEE Transactions on Communications*, **66**(4):1624–1635, 2018.

[KHH17] Seong Min Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. "Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments." In *NSDI*, pp. 145–161, 2017.

[KR11] Michael Kuperberg and Ralf Reussner. "Analysing the fidelity of measurements performed with hardware performance counters." In *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*, pp. 413–414. ACM, 2011.

[KRR12] Ramakrishna Kotla, Tom Rodeheffer, Indrajit Roy, Patrick Stuedi, and Benjamin Wester. "Pasture: Secure Offline Data Access Using Commodity Trusted Hardware." In *OSDI*, pp. 321–334, 2012.

[KS16] David Kohlbrenner and Hovav Shacham. "Trusted Browsers for Uncertain Times." In *USENIX Security Symposium*, pp. 463–480, 2016.

[KSB14] Andrew J Kerns, Daniel P Shepard, Jahshan A Bhatti, and Todd E Humphreys. "Unmanned aircraft capture and control via GPS spoofing." *Journal of Field Robotics*, **31**(4):617–636, 2014.

[LCC15] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Alec Wolman, Yury Degtyarev, Sergey Grizan, and Jason Flinn. "Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming." *GetMobile: Mobile Computing and Communications*, **19**(3):14–17, 2015.

[LDC09] Martin Lukac, Paul Davis, Robert Clayton, and Deborah Estrin. "Recovering temporal integrity with data driven time synchronization." In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pp. 61–72. IEEE Computer Society, 2009.

[LEW05] K Lee, John C Eidson, Hans Weibel, and Dirk Mohl. "Ieee 1588-standard for a precision clock synchronization protocol for networked measurement and control systems." In *Conference on IEEE*, volume 1588, p. 2, 2005.

[lin18a] linuxsgx. "Intel(R) Software Guard Extensions for linux OS, PSE modification Discussion." `https://github.com/intel/linux-sgx/issues/194`, 2018.

[lin18b] linuxsgx. "Intel(R) Software Guard Extensions for linux OS, Trusted Time Discussion." `https://github.com/intel/linux-sgx/issues/161`, 2018.

[LL84] Jennifer Lundelius and Nancy Lynch. "An upper and lower bound for clock synchronization." *Information and control*, **62**(2), 1984.

[LL18] Hongliang Liang and Mingyu Li. "Bring the Missing Jigsaw Back: TrustedClock for SGX Enclaves." In *Proceedings of the 11th European Workshop on Systems Security*, p. 8. ACM, 2018.

[LLZ18] Hongliang Liang, Mingyu Li, Qiong Zhang, Yue Yu, Lin Jiang, and Yixiu Chen. "Aurora: Providing Trusted System Services for Enclaves On an Untrusted System." *arXiv preprint arXiv:1802.03530*, 2018.

[LPJ15] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. "Accurate Latency-based Congestion Feedback for Datacenters." In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 403–415, Santa Clara, CA, 2015. USENIX Association.

[LSS07] Jin-Shyan Lee, Yu-Wei Su, and Chung-Chou Shen. "A comparative study of wireless protocols: Bluetooth, UWB, ZigBee, and Wi-Fi." In *Industrial Electronics Society, 2007. IECON 2007. 33rd Annual Conference of the IEEE*, pp. 46–51. Ieee, 2007.

[LWS11] Maciej Lipiński, Tomasz Włostowski, Javier Serrano, and Pablo Alvarez. "White rabbit: a PTP application for robust sub-nanosecond synchronization." In *Intl. IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2011*. IEEE, 2011.

[LYL12] Xiali Li, Shaona Yu, cYuan Lin, and Min Xi. "A Multi-model Kalman Filter Clock Synchronization Algorithm based on Hypothesis Testing in Wireless Sensor Networks." In *2nd International Conference on Electronic & Mechanical Engineering and Information Technology*, 2012.

[Man] Decawave Radio User Manual. "http://www.decawave.com/support.".

[MDA16] Bassam Moussa, Mourad Debbabi, and Chadi Assi. "A detection and mitigation model for PTP delay attack in an IEC 61850 substation." *IEEE Transactions on Smart Grid*, 2016.

[MGT11] Aneeq Mahmood, Georg Gaderer, Henning Trsek, Stefan Schwalowsky, and Nikolaus Kerö. "Towards high accuracy in IEEE 802.11 based clock synchronization using PTP." In *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2011 International IEEE Symposium on*, pp. 13–18. IEEE, 2011.

[Mil91] David L Mills. "Internet time synchronization: the network time protocol." *Communications, IEEE Transactions on*, **39**(10), 1991.

[Miz12] Tal Mizrahi. "Slave diversity: Using multiple paths to improve the accuracy of clock synchronization protocols." In *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2012 International IEEE Symposium on*, pp. 1–6. IEEE, 2012.

[MKS04] M Maroti, B Kusy, G Simon, and A Ledeczi. "The flooding time synchronization protocol." In *SenSys, Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004.

[MO85] Keith Marzullo and Susan Owicki. "Maintaining the time in a distributed system." *ACM SIGOPS Operating Systems Review*, **19**(3):44–54, 1985.

[Mox] IEEE 1588 compliant Moxa switch. "https://store.moxa.com/a/cat/industrial-ethernet/ethernet-switches/managed.".

[MPP07] Catherine Meadows, Radha Poovendran, Dusko Pavlovic, LiWu Chang, and Paul Syverson. “Distance bounding protocols: Authentication logic analysis and collusion attacks.” In *Secure localization and time synchronization for wireless sensor and ad hoc networks*, pp. 279–298. Springer, 2007.

[MVV17] Aanchal Malhotra, Matthew Van Gundy, Mayank Varia, Haydn Kennedy, Jonathan Gardner, and Sharon Goldberg. “The security of NTP’s datagram protocol.” In *International Conference on Financial Cryptography and Data Security*, pp. 405–423. Springer, 2017.

[MW13] Ratul Mahajan and Roger Wattenhofer. “On consistent updates in software defined networks.” In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, p. 20. ACM, 2013.

[NG09] S. Natarajan and A. Ganz. “SURGNET: An Integrated Surgical Data Transmission System for Telesurgery.” In *International Journal of Telemedicine and Applications Volume, Article ID 435849.*, 2009.

[NH18] Lakshay Narula and Todd Humphreys. “Requirements for secure clock synchronization.” *IEEE Journal of Selected Topics in Signal Processing*, 2018.

[Ope] OpenSplice. “dds for Real-Time System, PrismTech.”.

[paga] Ethtool: Linux man page. “https://linux.die.net/man/8/ethtool.”.

[pagb] recvmsg: Linux man page. “https://linux.die.net/man/2/recvmsg.”.

[PG08] Dan RK Ports and Tal Garfinkel. “Towards Application Security on Untrusted Operating Systems.” In *HotSec*, 2008.

[PH16] Mark L Psiaki and Todd E Humphreys. “GNSS spoofing and detection.” *Proceedings of the IEEE*, **104**(6):1258–1270, 2016.

[Pro] The Linux PTP Project. “http://linuxptp.sourceforge.net.”.

[RPM13] Lenin Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. “Timecard: Controlling user-perceived delays in server-based mobile applications.” In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 85–100. ACM, 2013.

[RSG] Siemens RUGGEDCOM RSG2488. “http://w3.siemens.com.”.

[RSW16] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, et al. “fTPM: A Software-Only Implementation of a TPM Chip.” In *USENIX Security Symposium*, pp. 841–856, 2016.

[RTY17] Dima Rabadi, Rui Tan, David KY Yau, and Sreejaya Viswanathan. "Taming Asymmetric Network Delays for Clock Synchronization Using Power Grid Voltage." In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 874–886. ACM, 2017.

[RVB12] Julien Ridoux, Darryl Veitch, and Timothy Broomhead. "The case for feed-forward clock synchronization." *IEEE/ACM Transactions on Networking (TON)*, **20**(1):231–242, 2012.

[S G03] MB Srivastava S Ganeriwal, R Kumar. "Timing-sync protocol for sensor networks." In *SenSys, Proceedings of the 1st international conference on Embedded networked sensor systems*, 2003.

[Sal] 2015 Salae Logic Pro 16. "http://downloads.saleae.com.".

[SBH12] Daniel P Shepard, Jahshan A Bhatti, Todd E Humphreys, and Aaron A Fansler. "Evaluation of smart grid and civilian UAV vulnerability to GPS spoofing attacks." In *Proceedings of the ION GNSS Meeting*, volume 3, pp. 3591–3605, 2012.

[sgx16] intel sgx. "Intel(R) Software Guard Extensions Software Developer manual.", 2016.

[sgx18] linux sgx. "Intel(R) Software Guard Extensions for linux OS, DRM Sample Code." `https://github.com/intel/linux-sgx/tree/master/SampleCode/SealedData`, 2018.

[SLK17] James Seibel, Kevin LaFlamme, Fred Koschara, Reinhard Schumak, and Jeremy Debate. "Trusted execution environment.", July 27 2017. US Patent App. 15/007,547.

[SNW06] Kun Sun, Peng Ning, and Cliff Wang. "TinySeRSync: secure and resilient time synchronization in wireless sensor networks." In *Proceedings of the 13th ACM conference on Computer and communications security*, pp. 264–277. ACM, 2006.

[SP05] Dave Singelee and Bart Preneel. "Location verification using secure distance bounding protocols." In *Mobile Adhoc and Sensor Systems Conference, 2005. IEEE International Conference on*, pp. 7–pp. IEEE, 2005.

[Spe17] Library Specification. "Trusted Computing Group." `https://trustedcomputinggroup.org/resources/tpm_library_specification`, 2017.

[SPN05] Robert A Scholtz, Davida M Pozar, and Won Namgoong. "Ultra-wideband radio." *EURASIP Journal on Advances in Signal Processing*, **2005**(3):758540, 2005.

[STT17] Shweta Shinde, DL Tien, Shruti Tople, and Prateek Saxena. “Panoply: Low-tcb linux applications with sgx enclaves.” In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, p. 12, 2017.

[SWG17] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Malware guard extension: Using SGX to conceal cache attacks.” In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 3–24. Springer, 2017.

[Sym09] Symmetricom. “Best Practices for IEEE 1588/ PTP Network Deployment.”, 2009.

[TKA17] Bohdan Trach, Alfred Krohmer, Sergei Arnautov, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. “Slick: Secure middleboxes using shielded execution.” *arXiv preprint arXiv:1709.04226*, 2017.

[TKG18] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. “ShieldBox: Secure Middleboxes using Shielded Execution.” In *Proceedings of the Symposium on SDN Research*, p. 2. ACM, 2018.

[TPV17] Chia-Che Tsai, Donald E Porter, and Mona Vij. “Graphene-SGX: A practical library OS for unmodified applications on SGX.” In *Proceedings of the USENIX Annual Technical Conference (ATC)*, p. 8, 2017.

[TSS16] Remi Tachet, Paolo Santi, Stanislav Sobolevsky, Luis Ignacio Reyes-Castro, Emilio Frazzoli, Dirk Helbing, and Carlo Ratti. “Revisiting street intersections using slot-based systems.” *PloS one*, **11**(3):e0149607, 2016.

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. “CLKSCREW: exposing the perils of security-oblivious energy management.” In *26th USENIX Security Symposium*, 2017.

[UV09] Markus Ullmann and Matthias Vögeler. “Delay attacks—Implication on NTP and PTP time synchronization.” In *Precision Clock Synchronization for Measurement, Control and Communication, 2009. ISPCS 2009. International Symposium on*, pp. 1–6. IEEE, 2009.

[VDK17] Peter Volgyesi, Abhishek Dubey, Timothy Krentz, Istvan Madari, Mary Metelko, and Gabor Karsai. “Time synchronization services for low-cost fog computing applications.” In *Proceedings of the 28th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype*, pp. 57–63. ACM, 2017.

[WYM02] H Wang, L Yip, D Maniezzo, JC Chen, RE Hudson, J Elson, and K Yao. “A wireless time-synchronized COTS sensor platform, part II: Applications to beamforming.” *Center for Embedded Network Sensing*, 2002.

[Xu13] Xiong Xu. “A New Time Synchronization Method for Reducing Quantization Error Accumulation Over Real-Time Networks: Theory and Experiments Evaluation of kalman filtering for network time keeping.” In *IEEE Trans. on Industrial Informatics*, 2013.

[YAY13] Qingyu Yang, Dou An, and Wei Yu. “On time desynchronization attack against IEEE 1588 protocol in power grid systems.” In *Energytech, 2013 IEEE*, pp. 1–5. IEEE, 2013.

[Yi 08] Shunjia Liu Yi Zeng, Bo Hu. “Vector Kalman Filter Using Multiple Parents for Time Synchronization in Multi-Hop Sensor Networks.” In *Sensor, Mesh and Ad Hoc Communications and Networks, SECON*, 2008.

[ZCC16] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. “Town crier: An authenticated data feed for smart contracts.” In *Proceedings of the 2016 aCM sIGSAC conference on computer and communications security*, pp. 270–282. ACM, 2016.

[ZML12] Jia Zou, Slobodan Matic, and Edward A Lee. “PtidyOS: A lightweight microkernel for Ptides real-time systems.” In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*. IEEE, 2012.

[ZNK08] Hui Zhou, Charles Nicholls, Thomas Kunz, and Howard Schwartz. “Frequency accuracy & stability dependencies of crystal oscillators.” *Carleton University, Systems and Computer Engineering, Technical Report SCE-08-12*, 2008.