2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS) | 979-8-3503-5841-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/RTAS61025.2024.00009

Trusted Timing Services with TIMEGUARD

Adeel Nasrullah
UMass Amherst
anasrullah@umass.edu

Fatima M Anwar
UMass Amherst
fanwar@umass.edu

***Abstract*—The importance of timing services in edge systems makes them a lucrative target for privileged adversaries. Malicious agents with Operating System (OS) privileges can stealthily manipulate timing services and provide altered timestamps to user applications. In this paper, we first demonstrate the adverse impact of time attacks on the accuracy of sensor fusion algorithms at the edge. Then, we introduce TIMEGUARD, our proposed architecture that protects against time attacks and provides trusted time to user applications. TIMEGUARD's design leverages the secure interrupt and memory primitives of trusted execution environments (TEEs) to bypass untrusted privileged software and acquire time securely. Yet, these secure primitives come at a high computational cost. TIMEGUARD also introduces a probabilistic security framework – bounded by a time error – to limit the cost of our timing service. We prototype our design on ARM TrustZone — the dominant secure architecture in edge systems, and evaluate the trade-off in security, accuracy, and system overhead. TIMEGUARD's secure performance ranges from a microsecond to tens of milliseconds at 3.9% and 1.2% CPU overhead respectively, catering to a variety of application requirements.**

***Index Terms*—Trusted Execution Environments, Secure, Timestamping, Operating Systems**

I. INTRODUCTION

Timekeeping is crucial for the proper functioning of any cyber-physical system (CPS). It facilitates the scheduling of competing tasks to meet performance standards and establishes causality between internal and external events [1]. Additionally, timing services in edge platforms enable applications such as pedestrian safety [2], and online legal contracts [3], to function properly. Accurate time is also essential for the optimal performance of deep learning-based sensor fusion systems [4], and the ability to manipulate the system clock enables even non-expert adversaries to attack systems, such as the perception systems of autonomous cars [5]. Furthermore, security mechanisms, including credential expiration [6] and public key infrastructure (PKI) [7], depend on the integrity of the system's clock. A malicious OS can cause its security subsystem to accept expired credentials by simply changing its clock to a date in the past, potentially granting unauthorized access to restricted resources. In short, by merely tampering with the OS's timing services, adversaries can degrade system performance, incur financial losses, breach privacy, and cause physical harm. Despite their critical role, modern systems lack end-to-end secure timing services [8] and rely on conventional cryptographic and TEE-based security measures for time security.

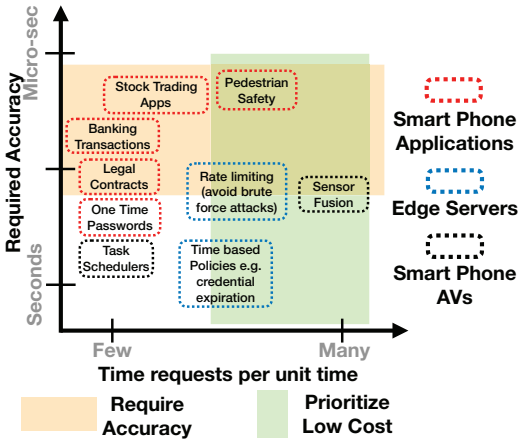


Fig. 1: Timing requirements of the edge applications

While secure timing sources exist in Trusted Execution Environments (TEEs), user applications have limited access to them. Researchers have proposed a few trusted clock mechanisms, with the most notable ones being timeseal [8] and ftpm [9], which present secure clock designs for Intel SGX and Arm TrustZone, respectively. However, these are ble within the TEEs, requiring applications to execute inside the TEE for secure access. This poses a security concern, as it significantly increases the size of the trusted computing base (TCB) [10]. Additionally, such a mode of execution may lead to a degraded user experience on interactive devices [9]. For instance, in ARM TrustZone, CPU resources are shared between untrusted and trusted software. The untrusted software, which implements most of the device functionality (including user interface), uses secure services provided by short-lived executions of the trusted software. Long-running executions inside TrustZone for securely accessing trusted time can degrade user experience and may even jeopardize the stability of the commodity OS [9]. These issues indicate that existing designs for trusted timing services are, at best, sub-optimal and, at worst, pose a security risk.

Designing a trusted timing service with system-wide availability poses challenges for several reasons. First, the untrusted OS controls access to the hardware clocks [11] and mechanisms for communication between trusted and untrusted software [8], [12], rendering the issuance of timing requests and transfer of timing information insecure. The second challenge arises from the semantic gap between the untrusted software and the TEE, as the TEE lacks a comprehensive

2642-7346/24/$31.00 ©2024 IEEE
DOI 10.1109/RTAS61025.2024.00009

view of the untrusted software's state and cannot verify the identity of the client application. Ensuring time delivery to the correct client application without incurring significant system penalties proves difficult. Finally, accuracy requirements differ among applications, as depicted in Figure 1. For example, banking transactions and legal contracts demand high accuracy, where cost is a secondary concern, whereas applications like sensor fusion and thwarting brute force attacks prioritize a trusted timing service that offers frequent timing updates with minimal overhead. Balancing accuracy and overhead in a unified design is non-trivial.

Our first challenge arises from the traditional design principle of assigning the OS the highest privileges over system hardware. This design primarily aims to enable the OS to mitigate security threats from untrusted user applications by limiting their access to critical system resources. For the same reason, TEEs like ARM TrustZone do not permit applications direct communication with secure software [1]. However, this also implies that an application's timing requests to the TEE may be tampered with or delayed by the untrusted OS. We address this challenge by leveraging the secure memory and interrupt primitives provided by the TEEs [2]. We utilize the fact that memory access violations by untrusted software trigger a secure interrupt and use this to issue secure timestamping requests. Once this interrupt is triggered, the execution control flow transfers to the secure timing stack, which then returns timing information from a trusted source. We use TEE's secure memory and interrupt primitives to ensure the OS cannot intercept the secure timing request.

When an application invokes trusted software (by issuing a secure timing request) via a memory access violation, the TEE receives notification of this event. However, the semantic gap between the TEE and the untrusted OS means the former lacks a complete view of the latter's state and cannot readily identify the software entity initiating this timing request. Our second challenge involves identifying the client that issued the timing request despite this semantic gap. In a single-core system, the application currently scheduled would be considered the client application; however, in multi-core systems, two or more applications running on different cores may request timestamps alternately, complicating the identification of the application that initiated the time request. We propose a strategy based on inter-processor interrupt (IPI) mechanisms to interrupt all cores and identify the client application, albeit this approach incurs significant system overhead. To mitigate this, we design scheduling policies aimed at predicting the next processor core likely to issue a timestamping request, thereby significantly reducing overhead.

TIMEGUARD's employment of secure primitives and IPIs renders it robust against adversaries, yet these mechanisms are costly and incur high overhead, particularly when timing requests are frequent. This design is at odds with applications that do not require high time accuracy but instead prefer a low-cost timing service (Figure 1). To introduce flexibility into our design, we add another mode to our trusted timing service that offers probabilistic security at a lower cost. In this mode, the system is configured to allow applications direct access to the timing source, while hardware state monitoring ensures untrusted software does not interfere. This is achieved by periodically inspecting the hardware state and implementing strategies that reduce the likelihood of a stealthy adversary accumulating significant time error.

In summary, provisioning secure timing services is a non-trivial task that requires adopting innovative approaches and motivates us to design a flexible and secure time service with the following contributions:

- We analyze various timing attacks by a privileged adversary and demonstrate the effects of these attacks on the accuracy of a sensor fusion algorithm.
- We present the design of TIMEGUARD, a trusted and flexible timing service that operates in two modes, i.e., *passive* and *active*. The passive mode provides highly accurate and secure timestamps, however, each timing request incurs significant overhead, which increases as timestamps are requested frequently. In such cases, the active mode provides timing information at a lower cost, determined by a secure configuration monitoring strategy. It provides probabilistic security, i.e., the adversary can reduce time accuracy within a fixed bound.
- We prototype our design on the iMX6 development board and evaluate various aspects of our trusted timing service: we quantify the cost and accuracy trade-off in the presence of an adversary.

II. BACKGROUND & MOTIVATION

This section briefly discusses the components of a time service, how they are subject to attacks, and the impact of these attacks on prediction algorithms. It then concludes with an examination of time services in trusted execution environments (TEE).

A. Trusted Time Stacks

A timing service comprises i) a time source and ii) a mechanism for transferring this time from the source to the client. Both components must be secure to establish a trusted timing service. The time source should be impervious to influence by untrusted privileged software and should be available in most TEE-based architectures. For instance, ARM implements a physical counter [13], as a part of its generic timer, which serves as a trusted timing source when configured to be monotonic and at a fixed rate by the TrustZone [14]. In RISC-V, *mtime* acts as a fixed-rate counter accessible to the secure monitor (machine mode), which controls the interaction of untrusted components with it [15]. Intel SGX, while lacking access to hardware-based, high-resolution trusted time sources, supports software-based trusted timing designs [8]. Although

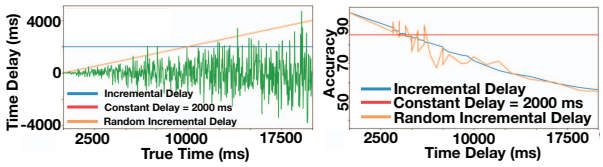
[1] ARM's *smc* instruction, which switches between non-secure and secure modes, can only be executed in privileged modes.

[2] E.g., TrustZone Address Space Controller for ARM and Physical Memory Protection (PMP) mechanism in Keystone.

a trusted timing source is found on most architectures, developing a secure time transfer mechanism to the unprivileged applications remains a challenge.

B. Case Study: Sensor Fusion under Timing Attack

Timing services rely on non-secure communication for time transfer. A privileged adversary may exploit these services using one of the following strategies: adding a **constant delay** of c time units to each timestamp requested by the victim. This constant delay, difficult to detect, appears as latency from an unknown system component from the victim's perspective. Alternatively, an adversary can add **incremental delays** to each of the victim's subsequent timestamp requests, masquerading as clock drift. Or, it can add **random incremental delays** to subsequent timestamps by varying the rate of increment every few timestamps to simulate clock drift in response to temperature changes.



(a) Perceived Time (b) Accuracy Performance

Fig. 2: (a)Effects of malicious OS attacks on time perceived by user applications. (b) The drop in accuracy of a multi-modal deep learning algorithm that uses attacked sensor data.

We evaluate a multi-modal deep learning-based sensor fusion algorithm under these timing attacks to demonstrate their effects. It is trained using a fraction of the CMActivities dataset [16], comprising inertial, and audio data from various participants engaging in physical activities. We compute the prediction accuracy of the algorithm on the remaining dataset and simulate timing attacks by delaying IMU data relative to the audio. Figure 2a illustrates the time deviations during these attacks, while Figure 2b displays the sensor fusion algorithm's accuracy deterioration due to manipulated timestamps. We observe that the performance degradation caused by constant delay is limited; however, the prediction accuracy degrades almost linearly with the accumulation of time delay in incremental attacks. Similar, sensor fusion algorithms are employed in safety-critical applications, such as LiDAR-Camera based perception systems in autonomous vehicles (AVs), where such attacks can have serious consequences [5]. *Typical attacks on these perception systems necessitate creating carefully crafted inputs and thus require significant domain knowledge. However, a non-expert adversary can induce catastrophic failures by remotely manipulating system time.*

C. Trusted Execution Environment

igure 3a shows a simplified architecture of a generic TEE. There are two modes of execution: non-secure (normal world) and secure (secure world), which *time-share* the CPU resources between them; that is, a given processor core either executes in secure or non-secure mode. The secure mode also has secure copies of some CPU registers and a protected memory area that are inaccessible to the normal world. The normal world runs a typical OS, e.g., Linux, Android, and user applications, which are treated as untrusted. The secure world runs a secure OS with a small Trusted Computing Base (TCB), providing secure services like encryption, random number generation, and secure storage. The switch between the two worlds is expensive because it involves storing and restoring processor states, and invalidating CPU and TLB caches to prevent side-channel attacks. Both the normal world and secure OS implement their own virtual memory for memory protection, creating a semantic gap between them. The secure OS is the most privileged component in the system and has access to all system resources, including those of the untrusted OS. The secure OS extends its services to the normal world via trusted applications that have limited access to secure resources, thus limiting the interface exposed to untrusted software. For example, the TEE time service provides clock source integrity but neither timeliness nor correctness for time transfer.

III. Threat Model and Design Goals

Here, we describe our threat model, assumptions regarding TEE primitives and design goals for our secure service.

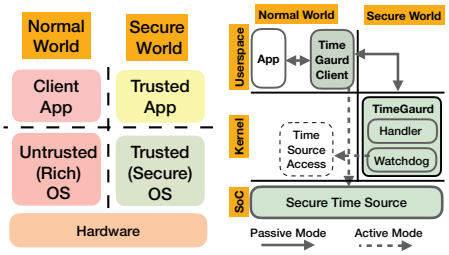
A. Threat Model and Assumptions

Our adversary aims to *stealthily* manipulate the victims' view of time. It intercepts a timing request issued by the victim or the time response from the secure time service. If successful, it either delays or alters the intercepted data to manipulate the victim's view of time.

Attacker's Capabilities. We assume a privileged adversary capable of executing code, accessing system resources, hijacking exception handlers, and programming a timer interrupt. These capabilities enable it to preempt the target program arbitrarily or exploit preemptions resulting from other sources, such as interrupts from a network device, to launch attacks. It may take over rarely used global variables of the untrusted operating system and can also profile the target program to maintain statistics, such as the number and types of system calls used by the victim. However, the adversary cannot maintain shadow data structures, such as page tables, that would require real-time synchronization with the actual page table. Similarly, it cannot decompile a program binary to perform code analysis or build its execution graph. These tasks result in significant performance degradation of the target system and create detectable signatures of the adversarial activity, identifiable by existing security solutions [17], [18]. Finally, we assume that the system boots using secure boot technologies and that the adversary only gains control after the boot sequence is complete.

Target Platform Features. We make a case for TEEs such as ARM TrustZone [19] or RISC-V Keystone [20], which are widely available and utilized in edge applications. These TEEs provide the secure primitives described earlier. If protected memory is illegally accessed by the normal world OS, a

signal is sent to the secure world OS. Secure interrupts are exclusively enabled, programmed, and received by the secure OS.



(a) TEE Architecture (b) TimeGaurd Overview

Fig. 3

B. Design Goals

Our design, TIMEGUARD, builds on secure TEE primitives and seeks to achieve the following goals:

1. Minimizing Attack Surface: To reduce the attack surface against a stealthy privileged adversary, such as a compromised OS, we enforce two security properties. First, **(P1)** the timestamp request should bypass the untrusted OS. Second, **(P2)** the timing information should be returned without alerting the untrusted software. The former ensures that the malicious agent cannot delay the timestamp request, while the latter deprives the adversary of the opportunity to forge a fake response or delay the timing information returned by TIMEGUARD.

2. Performance/Security Trade-off: The design should balance the security requirements of diverse applications while avoiding disproportionate overhead. A high overhead is acceptable for most safety-critical applications, whereas applications such as multiplayer games can tolerate some time error but require frequent timestamps. To achieve flexibility, we implement a third security mechanism, **(P3)**, that seeks to discover stealthy adversaries probabilistically. Using this component, we add another operational mode to our design with an adjustable trade-off between overhead and the accuracy of trusted time.

Moving forward, we present our design with reference to ARM-based platforms that implement TrustZone. It is to be noted that our design can be implemented on any TEE that provides secure memory and interrupt primitives as described in our threat model.

IV. DESIGN

TIMEGUARD operates in two modes: **passive** and **active**, each designed to fulfill one or more of our stated goals. The **Passive mode** addresses our first two security properties (P1 & P2). In this mode, security mechanisms activate solely upon timestamp requests, hence operating passively. This mode ensures the provision of accurate timestamps, even amidst presence of an adversary, with an overhead proportional to the frequency of timestamp requests. Conversely, the **Active mode** is more lightweight but yields less precise timestamps. Motivated by our third security property (P3), TIMEGUARD continuously scans for the presence of the privileged adversary in the active mode and imposes a fixed overhead.

A. Overview

Figure 3b presents an overview of our trusted timing service design TIMEGUARD. In passive mode, it should transfer control flow from the user space (TIMEGUARD *client*) to the trusted software (TIMEGUARD *handler*) inside the TEE, which is challenging, given the privileged position of the untrusted OS that controls all communication mechanisms. Secure interrupts can bypass the OS, but they are asynchronous and are not suitable for generating synchronous timestamping requests. We propose, *timelock*, a novel mechanism for TIMEGUARD's client to *psuedo*-synchronously trigger secure interrupt and compensate for the delays involved. Once, TIMEGUARD handler receives a request, it acquires timestamp from the secure time source and delivers it back to the application. However, the semantic gap between the untrusted and the trusted software makes it challenging to identify the correct client application. We propose data-driven policies to identify and deliver the timestamp to the correct application.

In the active mode, TIMEGUARD enables fast access to the trusted timing source. To achieve this, applications are allowed to access the timing source directly without relying on untrusted or the trusted OS software. It is challenging because untrusted OS can intercept the applications' direct accesses to the time source and launch man-in-the-middle attacks [21]. Our proposed TIMEGUARD watchdog (figure 3b) actively searches for ongoing attacks against the victim application despite having a limited view of the untrusted hardware/software states. Active and passive modes, when combined, enable TIMEGUARD to provide applications with an adjustable trade-off between security and performance.

B. Passive Mode

The passive mode's components are present in both normal and secure worlds: TIMEGUARD *client* in the unprivileged normal world and TIMEGUARD *handler* in the privileged secure world. The client issues the timestamping request while the handler returns trusted time securely to the user application.

1. Secure Psuedo-Synchronous Timing Requests. An application's request to the secure TEE software is routed through the untrusted OS. While this route ensures data integrity through encryption mechanisms, it does not guarantee *timely availability* of TEE services to the user applications. It poses a challenge for the design of trusted timing services where freshness of timing information must also be ensured in addition to the timestamp's integrity. We overcome this challenge using the secure memory and interrupts primitives implemented by the privileged TEE. A typical TEE memory protection mechanism (e.g. Trustzone Address Space Controller [22], RISC-V PMP [15]) can set permissions on different memory regions. The reserved memory regions are inaccessible to the untrusted OS and the user applications. If

the untrusted software attempts to access these protected memory regions, a secure interrupt will be generated notifying the TEE about potential memory violation attempt. TIMEGUARD client leverages this memory protection mechanism to issue a timestamp without alerting the untrusted OS.

Initiating timestamping request using a memory based mechanism poses two challenges: i) how to enable successful memory access without notifying the untrusted OS, and ii) how to intercept the asynchronous secure access in a synchronous manner. To address the first challenge, we configure memory protection controller to return an okay response upon a write attempt to the protected memory, and return zero when the same memory is read by the untrusted software. It makes our request generation transparent to the untrusted OS enabling our first security property **P1**, and depriving the adversary of the opportunity to delay this request. Refer to appendix for technical details of memory access mechanisms.

To address the second challenge, we introduce *timelock*: i) it ensures that the user application waits for the timestamp return before moving forward, and ii) it measures the time while it awaits the secure memory interrupt to trigger the TIMEGUARD *handler*. With *timelock* in place, an application continues its execution only after the timestamp request is completed. *Timelock* enables the TIMEGUARD's timing API to appear synchronous (*pseudo*-synchronous) from the application's perspective.

Algorithm 1 Time Lock

```
1: volatile branch_var ← 0
2: function TIMELOCK
3:     if branch_var == 0 then
4:         LDR R0, 0
5:         label : loop
6:         ADD R0, 1
7:         B loop
8:     else
9:         return 0
10:    end if
11: end function
```

Timelock compensates for the timestamp acquisition delay incurred while waiting for the asynchronous interrupt. It employs a busy-wait mechanism that increments a variable with each clock cycle as described in Algorithm 1. It creates variable $branch\_var$ initialized to zero. A branch condition, based on whether $branch\_var$ is zero, determines whether to engage the infinite loop (lines $5-7$) or bypass it. This branch is deliberately constructed to always trigger the infinite loop, which in turn increments a CPU register each cycle, effectively acting as a counter incrementing every two CPU cycles. This mechanism is central to our *timelock* strategy as it awaits TIMEGUARD handler's execution control and quantifies the duration spent in this state.

The *timelock* design, incorporating an infinite loop, enables precise measurement of secure memory interrupt delays but introduces two challenges: (i) compiler optimizations may eliminate code following *timelock*, considering it unreachable, and (ii) the absence of a loop exit mechanism. To counter the first issue, we declare the variable $branch\_var$ as *volatile* (line 2), which signals to the compiler that its value might change in ways unknown to it, ensuring that it does not optimize away subsequent code as unreachable. In practice, $branch\_var$ remains zero, and the branch always resolves to execute the *timelock*'s infinite loop. The TIMEGUARD handler addresses the second challenge by resuming execution from the instruction following the infinite loop (line 11) after delivering the trusted timestamp to the application. Although a branched loop could serve as an alternative, it introduces measurement uncertainty due to CPU branch prediction inaccuracies, resulting in unaccounted wasted cycles.

After the *timelock* is released and the memory interrupt delay is measured, the control flow is passed to the TIMEGUARD handler. It acquires secure timestamp t_A (reads the Physical Counter [13] for Trustzone based design), and calculate the final timestamp $t = t_A - t_L - t_S$ by compensating t_A for acquisition delay which comprises of i) secure memory interrupt delay t_L (measured by *timelock*) and ii) time elapsed during world switch t_S (platform-specific and measured empirically during bootstrapping). To transfer this secure timestamp to the application, it is critical to identify which application initiated the timing request.

2. Data-Driven Client Identification. On a single core system, client identification is not required and the handler returns the timestamp to the program that was interrupted by the secure memory interrupt. The same does not hold true for multi-core systems, where secure memory interrupt can be received by any processor core as it is not a processor-specific interrupt. Further, the semantic gap between the normal and secure world means that TIMEGUARD is unaware of the cores scheduled by the normal OS. To identify the core that initiated time request, TIMEGUARD handler checks if the current core has an active *timelock*, and completes time transfer if it detects one. Otherwise, the handler issues secure inter-processor interrupt (IPIs) to all other processor cores to check if they have an active *timelock*. This naive strategy is effective but inefficient, as each timing request generates one interrupt on each core. On a n-core system, it leads to n times the resources it would take on a single-core system.

We propose a data-driven approach to reduce the cost of dispatching trusted time on multi-core systems. Taking inspiration from branch prediction mechanism in the processor cores, we adopt policies for speculatively predicting the core that may issue next timestamp request based on historical data. TIMEGUARD handler sets the affinity of secure memory interrupt to the core predicted to receive the next timestamp. For each true prediction, we avoid $n-1$ interrupts and a good policy design would drastically reduce the overhead of *passive mode* operation.

Our proposed speculative core predication policies include **random selection** that randomly chooses a core to schedule next timing request, **frequency-driven** strategy that schedules the core with the most timestamp requests. It works well in scenarios when the OS anchors the client to a single core or schedules client programs on a small set of cores while putting others to sleep for energy conservation. It may not perform

well on a high workload system where the timestamp requests are more uniformly distributed across processor cores. **Most Recently Used (MRU)** policy schedules the same core that issued the current request for next timing request. This strategy assumes that a program often requests multiple timestamps within a short interval. It would work well when one program issues more frequent timestamps than others.

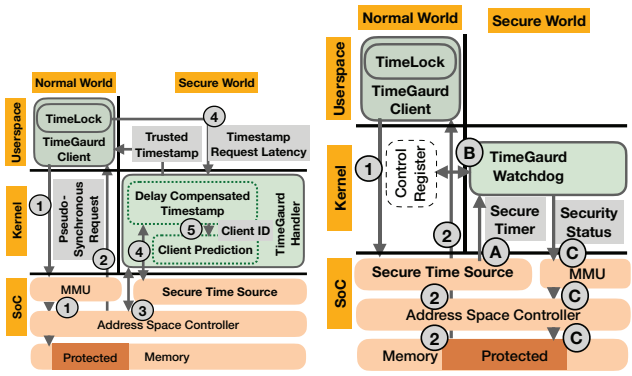
Finally, the **SchedTrace** policy enhances core prediction performance across a broader range of scenarios by monitoring scheduler and program behavior. However, it encounters a limitation as it requires access to client process IDs (PID), which are unavailable. To address this issue, each client is assigned a unique number, $RAND\_CONSTANT$, during registration, generated by the TEE random number generator, to act as a proxy for PID (*proxyPID*) (see Section IV-D). By utilizing the *proxyPID*, we track each client's most recent core affinity and the intervals between its successive timing requests from the time of its launch. Upon receiving a timestamp request, we calculate the average interval between timestamps for each client. Using this average and the timing of the last request by each client, we estimate the remaining time until its next request, t_R. The policy then assigns the core with the smallest t_R to handle the forthcoming timestamp request.

3. Secure Timestamp Transfer. Having scheduled a processor core to receive next timestamp, TIMEGUARD handler completes the timing request by transferring the secure timestamp to the client. Standard data transfer between the TEE and the untrusted realm occurs through the shared memory. If used for time transfer, a malicious OS can detect timestamp transfer as soon as timestamp is written to the shared memory. It is a violation of our second security property (**P2**) and exposes TIMEGUARD to delay attacks.

We preserve P2 security property by using CPU register $R0$ for timestamp transfer. It is the same register used by *timelock* for measuring secure memory interrupt latency (algorithm 1). TIMEGUARD handler writes the timestamp to $R0$ and switches to normal world to resume application's execution. Remember that *timelock* runs an infinite loop, and normal resumption would put the client program in forever loop. To prevent this, TIMEGUARD handler adds an $offset$ to the program counter (PC) before switching to the client. Originally, PC points to one of the two instructions shown on lines 6-7 in algorithm 1, which is next in line for execution. If the current instruction is the one on line 6, we add +8 to the program counter register[3] (+4 for the instruction on line 7) before returning to the normal world. The user application would now resume from the first instruction following the *timelock* and continues further execution.

Figure 4a shows the step-by-step operation of TIMEGUARD's passive mode: ① The user application generates a *pseudo-synchronous* timestamp request via illegal access to the protected memory and activates *timelock*, ② the address space controller returns a valid response, and ③ triggers a secure memory interrupt. ④ The TIMEGUARD handler, if

[3]Assuming each instruction is 4-bytes (or 32 bits) wide. In the case of variable length instructions, $offset$ would take on different values.



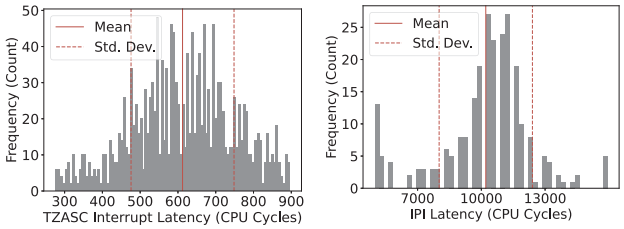
(a) TIMEGUARD's passive mode operation. (b) TIMEGUARD's active mode operation.

Fig. 4

necessary, switches to the client program's *timelocked* core, reads the secure time source and *timelock* to calculate a delay-compensated timestamp. Finally, ⑤ we set the secure memory interrupt's affinity to the core predicted to receive the next request and return the timestamp to the application via CPU registers. This sequence of steps successfully returns a trusted timestamp to a user program, bypassing the untrusted OS. However, on rare occasions, a *timelock* may be preempted by the untrusted OS scheduler, in which case the TIMEGUARD handler will not find any *timelocked* core. Such a situation could also arise when the secure memory interrupt is triggered by untrusted software accessing the secure memory (and not a TIMEGUARD client). In both cases, the TIMEGUARD handler resumes normal world execution and, to distinguish between the two scenarios, preempts the normal world execution after a short interval to check if the program has been rescheduled. If the preemption was non-malicious, the TIMEGUARD handler finds the *timelocked* core in one or more such follow-ups. Otherwise, it alerts the TEE software of a secure memory violation.

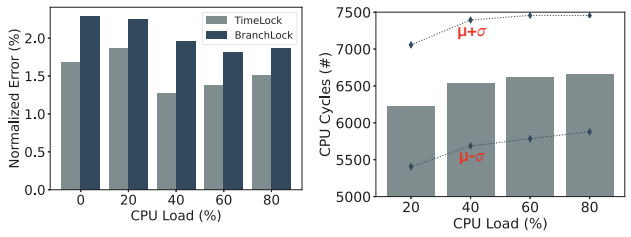
Evaluating *TimeLock*. *Timelock* is at the heart of passive mode's *psuedo-synchronous* timing requests. We perform measurement study to validate the design choices involved in it's implementation. We also look at *passive mode's* timestamp acquisition latencies and quantify TIMEGUARD's ability to measure them. Finally, we investigate how the accuracy of trusted time provided by passive mode changes with the introduction of *timelock*.

Measurement Setup: Our measurement setup consists of



(a) (b)

Fig. 5: a) Secure Memory Interrupt and b) IPI latency distributions observed over thousand measurements.



(a) (b)

Fig. 6: a) Normalized *Timelock* and Branchlock Measurement Errors b) CPU cycles consumed during switch from normal world to the secure world.

i.MX6Q Sabresd board equipped with 4 Cortex-A9 cores [23]. It's a TEE enabled SoC with Trustzone Address Space Controller (TZASC) that enforces memory protection and implements secure memory interrupt ($tzasc\_int$). We use these primitives to realize TIMEGUARD's passive mode prototype, which uses standard embedded Linux, and OPTEE as OS for the normal and secure world, respectively. For our evaluations we use CPU cycle counter as our reference clock because of its proximity to the CPU and minimum latency. It increments with each instruction cycle executed by the CPU, and stops when the CPU is idle. However, that does not affect our evaluations, as the CPU runs continuously during the completion of a timing request. Finally, we fix the CPU frequency to 996 MHz to avoid conversion errors from variable CPU frequency.

Timestamp Acquisition Latency: *Timelock* measures latencies from two sources: i) secure memory ($tzasc\_int$) and inter-processor interrupt (IPI), both of which vary widely as evident from their distributions shown in Figure 5. The standard deviations of $tzasc\_int$ and IPI latency distributions are 126 and 2160 clock cycles, respectively. Without time lock, this spread would not be captured and result in an uncertainty of $2.3\mu s$ in the trusted timestamps, which *timelock* has reduced by 12 times to $0.18\mu s$.

World Switch Latency: The total uncertainty in passive mode's timestamps also includes variations in world switch time. Figure 6b shows world switch times for our prototype, which have a standard deviation of 828 clock cycles. With no mechanism to measure the variation of world switch time, its contribution to the timestamp uncertainty would be upto $0.83\mu s$.

Timelock Accuracy: Figure 6a shows measurement errors for two implementations of *timelock*. *TimeLock* is the standard implementation from algorithm 1 while BranchLock is implemented using a finite loop, which is executed using a conditional branch instruction. To measure the accuracy of the two, we modify algorithm 1 to insert an additional instruction between lines $6-7$ to copy CPU cycle count to register $R1$. We compare the values in the two registers to obtain the reported error $\epsilon = \frac{R1-4*R0}{R1}$[4]. We see that our *timelock* implementation outperforms (by upto 48% at 40% CPU load) the branched version which is affected by CPU's mis-predictions of the branch prediction hardware.

Passive mode Overhead: The $tzasc\_int$, IPI and world switch operations, all part of passive mode timing call, takes almost 27325 clock cycles. While, IPI costs will be partially mitigated by core prediction strategies, core prediction computations and timestamp transfer would add yet more cycle time to the passive mode timestamp acquisition, thus making it expensive and not designed for frequent time calls.

[4]Each modified loop iteration takes 4 clock cycles.

C. Active Mode

This mode offers a fast timestamping interface with a relatively low overhead by trading off timing accuracy. TIMEGUARD's active mode enables a customizable balance between security and cost, catering to applications like task schedulers that require moderate accuracy but generate frequent timestamp requests [6]. Below, we describe the design components of active mode:

1. Simulated Secure Path. Modern edge platforms provide secure timing sources in the hardware, (section IIA) which are configured by the secure world but are accessible in both worlds. Further, the SoC can be configured to enable direct [5] userspace access to secure timing source. It is often done by writing to a control register on the SoC (e.g., setting the $PL0PCTEN$ and TSD bits in the $CNTKCTL$ and $CR4$ registers to 1 on ARM and Intel platforms, respectively.). We refer to this register as *control register* since it controls the userspace application's access to the trusted time source. TIMEGUARD leverages this functionality to provide user applications a direct path to the secure time source. A drawback to this approach is that untrusted OS can modify the control register to intercept timing requests and return forged timestamps. To mitigate this threat, we introduce TIMEGUARD watchdog that implements our third security property (**P3**). It monitors the state of the control register, and alerts the user application for an unauthorized update. Using TIMEGUARD watchdog, active mode simulates a secure timing path.

2. TIMEGUARD Watchdog. The watchdog monitors control register's state for a short duration to discover a stealthy attacker while preserving application's performance. Watchdog adopts a probabilistic approach with recurring inspections of the control register at random intervals. Not knowing when an inspection will take place, a stealthy adversary has an incentive to limit its attacks to short intervals. This scheme does not eliminate the attacks completely but ensures the time error introduced by the adversary stays within a bound i.e. we can trust that the timing information is accurate (secure) with a certain resolution. Additionally, the inspections should circumvent the untrusted OS, which motivates our design of watchdog inspections via another secure interrupt i.e. the timer interrupt. Unlike other secure interrupts, it can preempt the normal world execution at a pre-determined time with a high accuracy.

Watchdog Inspection Policies. A privileged attacker can manipulate control register at arbitrary duration and at arbitrary times for stealth operation using either scheduler events or untrusted timer interrupts. We design three policies for

[5]By executing a CPU instruction which is an atomic operation and transparent from OS perspective.

	Policy A		Policy B		Policy C	
	$\mu\|\sigma\|max$	P_{avg}	$\mu\|\sigma\|max$	P_{avg}	$\mu\|\sigma\|max$	P_{avg}
Attack 1	$0.5\|0.29\|0.99$	0	$0.5\|0.29\|1.99$	0.5	$0.5\|0.29\|\frac{2m}{n}$	$\frac{2n-1}{2nm}$
Attack 2	$0.25\|0.22\|0.99$	0	$0.25\|0.22\|1.99$	0.025	$0.25\|0.22\|\frac{2m}{n}$	$\frac{0.025n}{m}$
Attack 3	$0.08\|0.32\|0.1$	0	$0.08\|0.29\|0.1$	0.008	$0.08\|0.29\|0.1$	$\frac{0.008n}{m}$

TABLE I: Time errors and detection rate of privileged attacks against watchdog policies. The reported figures were calculated assuming random variables c, d, i were chosen randomly from a uniform distribution. Time error is normalized with respect to inspection period p and reported in the format $\{mean|std.\ dev.|max\}$

inspecting the control register state in the presence of these stealthy attacks. These policies stipulate that the watchdog performs one control register inspection in a given time $period$. In each $period$, the watchdog chooses the instant $i \in period$ to perform this inspection. The two variables are tuned by the watchdog to implement different policies.

Policy A: Watchdog period is $p = t_S$, and the inspection time $i = 0$ i.e. occurs at the start of the watchdog period. Here t_S is the untrusted OS's scheduling period in μ-seconds.

Policy B: Watchdog period is $p = t_S$, and the inspection time $i \in \mathbb{N}$ is selected randomly from the set $\{0, \frac{p}{10}, \frac{2*p}{10}, ..., p\}$.

Policy C: Watchdog period is $p = \frac{m*t_S}{n}$ where $m, n \in \mathbb{N}$, and the inspection time $i \in \mathbb{N}$ is selected randomly from the set $\{0, \frac{p}{10}, \frac{2*p}{10}, ..., p\}$. This policy allows us to tune the inspection period, with higher period resulting in lower costs but potentially higher time error and vice versa.

An adversary trying to evade the watchdog makes the following observations: attacks longer than the watchdog period p will almost certainly be detected, thus, an ideal attack is always smaller than p. Further, if $p > t_S$, it should make OS attacks more effective, an incentive for the watchdog to keep $p \leq t_S$. However, a $p < t_S$ increases the likelihood of the adversary being discovered, but watchdog ends up incurring higher overhead a disincentive for it to choose $p < t_S$. Given this, stealthy attacker, being unaware of p, can choose $p = t_S$ to design its attacks for maximize the damage and opt for one of the following three attack strategies:

Attack 1: The attack is launched for a fraction c of the interval $(0, t_S)$ starting at 0 where c is chosen randomly. Its a relatively simple attack that attempts to evade watchdog by introducing uncertainty in ending the attack.

Attack 2: This attack lasts from $(d, d + c)$ where both d and c are chosen randomly from intervals $(0, p)$ and (d, p) respectively. This attack further adds to the uncertainty making it difficult for the watchdog to detect the stealthy attacker.

Attack 3: This attack lasts from $(d, d + c)$ where both d and $c < \frac{p}{10}$ are chosen randomly from intervals $(0, p)$ and (d, p) respectively. It builds on previous attack by reducing the maximum attack duration.

Policy Analysis. We compute time errors (μ, σ, max) introduced by privileged attacks during a scheduling interval t_S and their detection rates (P_{avg}) against the proposed watchdog policies (Table I). Time error is normalized over the watchdog inspection period p, and we assume random variables c, d, i are uniformly distributed.

First two policies *A and B*, are identical except that the inspection instant for *Policy B* is chosen randomly while it is fixed for *Policy A*. Time error distribution is the same for the two across all three attacks. However, they have different bounds on the maximum error and their attack detection rates also differ. At a first glance, *Policy B*'s error bound is twice the *Policy A*'s bound. The non-zero detection rate P_{avg} of *Policy B* means it does not allow unchecked error accumulation over consecutive time periods, which is not the case with *Policy A*. Note that *Policy B*'s detection rate is modest for a given detection period yet it puts a bound on the maximum accumulated time error. For example, it would detect the *attack 2* and *attack 3* in 40 and 125 inspection periods respectively.

Policy C generalizes the success of *Policy B* in imposing a maximum accumulated error bound on the stealthy attacks. It provides us with knobs (m & n) to adjust *accuracy-cost* trade-off. We can increase the inspection period using m which would extend it to $m*t_S$, raising the bound on the accumulated time error while reducing the cost of inspections. For $m = 2$, the time error would accumulate up to $20p$ before the attack is detected, i.e. twice the time error at half the cost. On the other hand, n increases the number of inspections in one scheduling period t_S. It bounds the accumulated error to a lower value at n times the cost. With $n = 2$, both *attack 2* and *attack 3* only accumulate time error up to $5 * p$ at double the cost. To summarize, time error bound achieved by TIMEGUARD watchdog is *inversely* proportional to the overhead it imposes on the system.

3. TIMEGUARD Trust Signal TIMEGUARD watchdog implements probabilistic strategies to discover a stealthy attacker. This discovery is only useful, if it is communicated to the user application. Watchdog uses domain-shared read-only[6] memory to signal trust to the user application. When watchdog is launched it writes 1 to this memory location which signals absence of an adversary, and updates it to 0 if and when it detects an adversary. The user application checks this memory location with every timing request to decide if it should trust the timing information it read from the trusted timing source.

Figure 4b shows TIMEGUARD's active mode operation. The watchdog inspection operations i.e. Ⓐ secure timer interrupt, Ⓑ control register inspection, and Ⓒ trust status update are recurring events, and together they implement watchdog inspection policy. To access time, ① an application reads the trusted time source in the hardware and subsequently ② it reads trust signal to determine if the timing information is trustworthy.

D. Bootstrapping Trusted Time

For an application, secure registration with TIMEGUARD is a prerequisite for using the API. Applications follow a four-step process to securely register with TIMEGUARD. ① The user application invokes TIMEGUARD through a TEE driver that uses an smc call to switch to the secure world and passes a secret already known to TIMEGUARD. ② Upon receiving this secret, TIMEGUARD saves the page table base register pointing to the application's page table, and then ③ adds three protected memory mappings to the application's page table. One

[6]for the normal world

of these mappings is used by TIMEGUARD's passive mode to generate a secure timing request. The second location is used by the active mode to signal trust, while TIMEGUARD writes a random number $RAND\_CONSTANT$ to the memory location indicated by the third entry. This number is used for identifying client applications in passive mode. It is important to highlight that TIMEGUARD only marks these three entries as protected to avoid the overheads that would arise from normal to secure world context switches for each page table modification request if the entire page table were marked as protected. ④ Finally, TIMEGUARD returns an encrypted secret that will be sent to the trusted remote server, informing it of the availability of the trusted time service.

V. SECURITY ANALYSIS

In this section, we present a security analysis of how our design preserves security properties $P1-3$ from section IV. We describe attacks that may be employed to subvert one or more of these security properties, and how TIMEGUARD protects against such attacks:

Hijacking Timestamp Requests. If an adversary can predict *when* a timestamp request will be generated via memory access by the passive mode, it can issue an interrupt right before the timing request. If successful, the adversary can put delay between the event that victim wants to timestamp and when the actual timestamp will be obtained. Similarly, in active mode, the adversary will momentarily change the control register ($CNTKCTL$ [21]), launch the attack, evading detection by the watchdog. However, to launch this attack successfully, an adversary needs to precisely monitor each instruction being executed and interrupt just before the timing request instruction. This task will require detailed profiling of the program behavior including branch prediction, as well as determining when the target instruction will be executed. It means that attacking the TIMEGUARD takes significant time and computational resources on the attacker's behalf. With the attacker not having access to the physical device, such profiling becomes extremely difficult.

Interrupting the *Timelock*. An attacker may want to exploit the fact that victim application would put the core in a timelock. On a mult-core system, cores can monitor each other to detect timelock. Once detected, it can be pre-empted using a high priority IPI. Again to launch such an attack, the adversary needs to launch continuous interrupts, whose period will have to be smaller than the *timelock*'s duration. As shown in the figure 5a, this period is small (in the order of few hundred cycles) and the adversary would need to launch frequent interrupts degrading system performance and revealing itself. Even when interrupted successfully, it will be preempted by the execution of the secure memory interrupt, as secure interrupts typically suspend all normal world activities including interrupt. This attack is not only difficult to launch but also ineffective given the use of secure memory interrupts in our design.

Page Table Manipulation Attack. An adversary can easily identify the table entries added by TIMEGUARD, and it might modify these entries to trigger a fault on memory access. If successful, the adversary would imitate TIMEGUARD behavior but return incorrect timing information without ever invoking TIMEGUARD handler. In case of active mode, it would return a false trust signal. However, the page table entries used by TIMEGUARD are marked read only during bootstrapping, and any attempts to modify them will trigger a secure memory interrupt informing TIMEGUARD of a potential threat.

Hijacking Timestamp Transfer. In passive mode, an attacker may want to interrupt the target program right after it returns the timestamp and just before the timestamp is consumed. To achieve this, processor cores would monitor each other to check if one or more of them are executing in the secure state. If yes, an IPI would be sent to this core. Now, when TIMEGUARD handler returns the timestamp, the CPU will immediately receive the IPI and switch to kernel mode which gives the adversary a chance to modify the returned timestamp. However, TIMEGUARD handler masks the IPI for the core receiving the timestamp. This design decision ensures that there is no immediate interruption to the client programs execution. Other cores may unmask the IPI, but that gives the program enough time to consume the received timestamp.

Timing Attacks on Real-time Tasks For certain real-time applications (such as an event-triggered system), adversaries could reasonably assume that a timestamp is requested shortly after the victim is scheduled. This would allow them to intercept and manipulate the timing requests. However, TIMEGUARD can reliably detect such instances. In case of the passive mode, once the timestamp request is initiated it would always trigger TIMEGUARD handler. And as we discussed in section IV-B, the absence of a *timelocked* core would indicate an on-going attack. In case of the active mode the attack window i.e. the active-mode timestamp return duration (< 100 cycles), is very small and significantly reduces the likelihood of precise interception by the stealthy adversary. Increasing the likelihood of successful spoofing requires the adversary to expand the window size, increasing the chances of its detection.

Exploiting Client Identification Strategies for Denial of Service Attack. An adversary may launch several user applications to flood the passive mode with fake timing requests through secure memory violations. These spurious requests significantly affect the accuracy of client identification strategies and cause the TIMEGUARD handler to interrupt all system cores in response to each timing request. Although this attack would severely degrade system performance, it would also be detected immediately as the system is rendered dysfunctional, prompting system administrators to take mitigation steps.

VI. IMPLEMENTATION & EVALUATION

For the evaluation of our trusted timing service, we extend our prototype from our measurement study in section IV-B to include active mode components. Notably, we use secure physical timer [24] available in ARM Trustzone to implement TIMEGUARD watchdog. We start by evaluating the perfor-

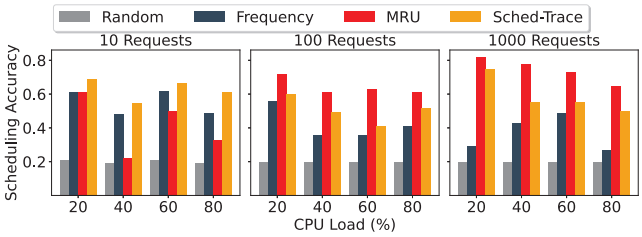


Fig. 7: Client Identification strategy performance with number of requests received per second.

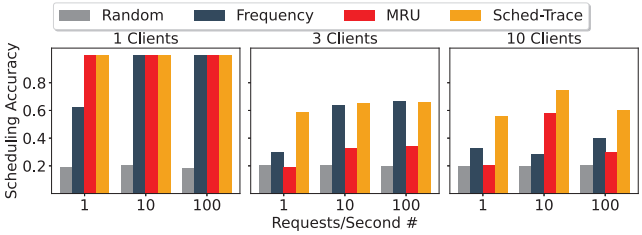
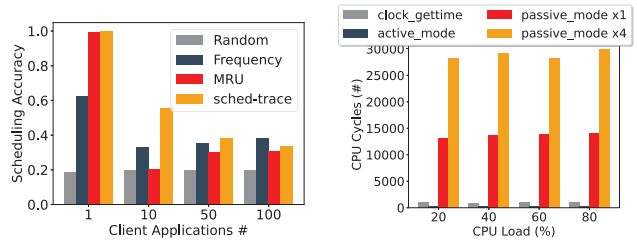


Fig. 8: Client identification strategy performance with with varying number of requests per second received from varying number of clients.

mance of core prediction policies that are integral to passive mode's responsiveness and cost on multi-core systems.



(a) (b)

Fig. 9: a) Client identification strategy performance with number of clients using the API simultaneously, b) TimeGaurd's active mode is 20 times more responsive than the Linux system-call for time *clock_gettime* which is 12 times less expensive than the TIMEGUARD passive mode API.

A. Data Driven Client Identification Strategies

Figure 7 shows the performance of our client identification strategies with three different rates of requests issued by 4 clients, each pinned to one of the four processor cores, under different system loads. We see that *MRU* and *SchedTrace* perform well across different request rate and under different system loads. MRU's good performance is expected since its design suits our experimental setup of few clients which are distributed uniformly across available processor cores. Similarly, with few programs to keep track of, $SchedTrace$ demonstrates better predictive power.

We also test our policies using varying number of time stamping clients. Figure 8 shows performance of our strategies with 1, 3 and 10 clients (not pinned to any specific core) requesting timestamps at different rates. We see that MRU performance degrades with higher number of clients because they are more likely to issue timestamp requests one after the other. This is not conducive to MRU design, which thrives upon receiving repeated requests from a single processor core. This is further validated by our results from Figure 9a which shows policy performance with even higher number of clients. $SchedTrace$ performance also degrades significantly with more than 10 client programs because of complex scheduling. However, $SchedTrace$ consistently performs well across different levels of CPU utilization, timestamp frequencies and number of clients. Its performance does degrade significantly with very high number of time-stamping clients but it still remains our best prediction strategy. Hence, we use $SchedTrace$ as our core prediction policy for further evaluations in this section.

B. Responsiveness

We measure responsiveness in terms of number of clock cycles taken to complete a timing request. It determines the smallest duration that can be measured with a timing API and also decides the timing service's overhead on the system. We evaluate the responsiveness of both the active and passive mode APIs and compare them against Linux timing API $get\_clocktime$. Figure 9b shows the responsiveness of TIMEGUARD APIs with varying system load. Notably, our active mode API is extremely responsive, it takes 47 clock cycles on average and is $20x$ more responsive the Linux's time API. It is because, in active mode, client directly reads the time from the trusted source without any context switch to the kernel or the TEE. It also means active mode's overhead comes from watchdog operation, and should stay fixed irrespective of timestamp frequency.

The passive mode's API has an average response time of approximately 13000 clock cycles on a single core system, and its 12x less responsive than Linux API. On a multi-core system, the response time increases significantly to an average of more than 27000 clock cycles. Even though our cross-prediction schemes (*SchedTrace* in this case) mostly predict the correct destination, the failures result in more than double the cost of correct prediction. This low responsiveness hints at high overhead that is proportional to the frequency of the API use.

C. Accuracy

Our evaluations suggest that TIMEGUARD's passive mode has low responsiveness and higher overhead compared to active mode. On the other hand, passive mode API provides trusted time accurate to 1 microsecond (section IV-B). To analyze active mode's performance, we empirically evaluate watchdog's policies against privileged adversaries attacks laid out in section IV-C. For this experiment, we launch four different user applications, one for each processor core, where the OS scheduling period is $t_S = 10ms$. Similarly, TIMEGUARD watchdog is active on all four cores as does our adversary. We run the experiment for an hour and take repeated measurements and report the results in Table II. Similar to our theoretical analysis, we report the mean (μ), standard deviation (σ) and maximum (max) time error induced by a stealthy adversary during one scheduling period. Further, we

	Policy A		Policy B		Policy C ($m = 2$)		Policy C ($n = 10$)	
	$\mu\|\sigma\|max$	P_{avg}	$\mu\|\sigma\|max$	P_{avg}	$\mu\|\sigma\|max$	P_{avg}	$\mu\|\sigma\|max$	P_{avg}
Attack 1	1.67|2.36|9.38	0	1.68|2.42|9.67	0.446	1.83|2.40|9.35	0.1995	1.69|2.35|9.66	0.9426
Attack 2	0.651|1.48|7.997	0	0.685|1.55|9.06	0.248	0.56|1.35|7.68	0.1165	0.589|1.38|8.50	0.7922
Attack 3	0.026|0.12|0.996	0	0.044|0.16|0.973	0.0795	0.037|0.154|0.995	0.0358	0.041|0.159|0.977	0.6918

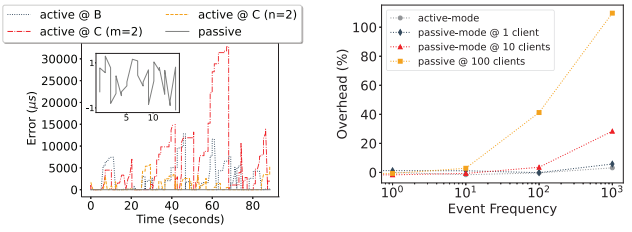
TABLE II: Efficacy of TIMEGUARD watchdog efficacy against attacks launched by the privileged adversary. Mean (μ), standard deviation (σ) and maximum error (max) are given in milliseconds, while attack detection rate $P_{avg} \in (0, 1)$.

report detectability of an attack P_{avg}, which determines the maximum accumulated error.

We see that time error statistics are all inline with those computed theoretically (table I). However, the detection probabilities of *Policy B* and *C* are smaller than the theoretical estimations, which increases the bound on maximum accumulated error by a small margin. Also, *Policy C* is evaluated with two settings: i) with twice the inspection period of *Policy B* (m=2) and ii) with $\frac{1}{10}th$ the *Policy B's* period. In the first case, the detection probabilities are halved which doubles the accumulated error bound, while in the second case, the detection probabilities have increased significantly reducing the bound on maximum error.

Figure 10a shows time errors accumulated under *Attack 2* for policies *B*, $C_{m=2}$ and $C_{n=2}$. We see that the accumulated time error for these policies are $11.89ms$, $34.18ms$ and $5.24ms$ respectively. We see that with watchdog inspection periods p, $2p$ and $p/2$ the accumulated error is well under theoretical bounds of $10p$, $20p$ and $5p$. This is because the theoretical analysis implicitly assumes that the victim program requests timestamps uniformly over the OS scheduling period t_S. However, our client programs for this test requested a fixed number of timestamps each scheduling period but at irregular intervals, which significantly decreases the error an adversary manages to accumulate since it intercepts fewer number of timestamps than our theoretical analysis assumed.

Our evaluations validate our theoretical bounds on the time error accumulation under stealthy attacks. Further, it also demonstrates the adjustable nature of cost versus security (measured in terms of error in trusted time) trade-off provided by TIMEGUARD's active mode.



(a) (b)

Fig. 10: a) Error introduced by an adversary in the trusted time provided by TIMEGUARD b) CPU utilization by active and passive modes with varying request frequency

D. System Benchmarks

Having established the responsiveness and accuracy of TIMEGUARD's two modes, we evaluate its impact on the whole system. Figure 11 shows the overhead of TIMEGUARD's both modes using Sysbench test suite as our benchmarking tool. We see that in active mode TIMEGUARD incurs very low overhead, the maximum CPU utilization is 1.2% with 1000 watchdog inspections (secure interrupts) per second. Interestingly, memory and thread fairness benchmarks show a negative overhead i.e. an improvement in performance when secure interrupt frequency increases to 100 and above. On each secure interrupt, the data and code caches are cleared along with the TLB caches. At 100 (1000) secure interrupts per second, the caches are cleared once (10 times) every scheduling period, and the memory and thread fairness improves. The *fileio* results do not show an established pattern because we use network based filesy-stem and network latencies skew the results based on network conditions. On the other hand, passive mode incurs high overhead and equally impact CPU, memory and thread fairness performance with frequent timing requests.

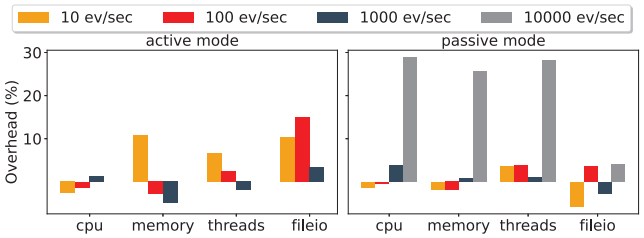


Fig. 11: Sysbench benchmark overheads for active (left) and passive mode (right) APIs with different timestamp request rates from the clients

Figure 10b contrasts the CPU overhead between active and passive modes across varying client numbers and timestamping frequencies. In passive mode, the overhead escalates with an increase in clients utilizing the API, a consequence of the heightened frequency of timestamp requests, which, as previously observed, increases the overhead. Thus, passive mode is best suited for applications requiring infrequent, precise timestamps, such as online legal contracts and banking transactions. Conversely, the overhead in active mode remains constant, predominantly due to TIMEGUARD's watchdog, whose inspections occur at a steady rate, unaffected by the volume of timestamp requests. This consistency, given the negligible cost per timestamp (refer to Figure 9b), makes active mode ideal for applications, such as credential verification and task scheduling, needing frequent but less precise timestamps.

E. Sensor Fusion with TIMEGUARD

Having validated our design for trusted timing services, we revisit our deep learning based sensor fusion case study presented in section II. Figure 12 shows the sensor fusion

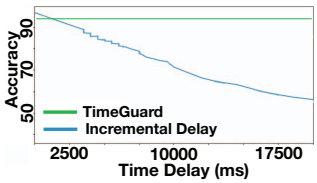


Fig. 12: Deep Learning performance using trusted time provided by TIMEGUARD

Attack	TimeSeal	TIMEGUARD (Active)	TIMEGUARD (Passive)
Scheduling Attack	Yes | ≥ 42	No | 0	No | 0
Delay Attack	Yes | ≥ 163	Yes | $\leq$11.56	No | 0
Packet Forging Attack	No | 0	Yes | $\leq$11.56	No | 0

TABLE III: Susceptibility of different trusted timing services against timing attacks. Each cell is formatted as {susceptibility | time error (ms) }, where susceptibility indicates if the service will degrade under this attack while time error is the average time error observed during the attack.

algorithm's performance in presence of a stealthy adversary but using trusted time provided by TIMEGUARD. It's using the active mode (with $m = 2$) because sensor fusion does not demand accuracy as high as provided by the passive mode. TIMEGUARD's flexible design means, it can opt for active mode timestamping which incurs lower overhead while providing time accuracy sufficient for optimal sensor fusion performance.

F. Comparison with TimeSeal

We compare TIMEGUARD with another trusted timing service TimeSeal [8] that provides high-resolution trusted timing service to applications inside Intel SGX. For our comparison, we implement TimeSeal with ARM Trustzone by emulating platform service enclave's (PSE) trusted clock using Trustzone's secure timer. And the time is transferred from the trusted clock to the user application via domain shared memory instead of inter-process communication (IPC) which was used by original Timeseal implementation with Intel SGX.

Table III shows three categories of time attacks. Timeseal's performance is affected by scheduling and delay attacks with an error greater than $42ms$ and $163ms$ respectively. On the other hand, TIMEGUARD's passive mode is resistant to all three attacks while in active mode it may experience delay or packet forging attacks by a stealthy adversary. However, TIMEGUARD watchdog ensures that this error is bounded to a maximum value of 11.56ms, with watchdog implementing *policy C* with $n = 10$. To conclude, TIMEGUARD demonstrates high degree of robustness against a privileged attacker and provides high accuracy trusted time as compared to TimeSeal.

An important measure of a system's security is its TCB size. Table IV shows that TimeSeal has a small TCB like TIMEGUARD, but its size increases linearly with increase in number of applications opting to use them. On the contrary, TIMEGUARD's TCB remains fixed irrespective of the number of events being timestamped or applications using it.

VII. RELATED WORK

Trusted Timing Solutions. TrustedClock [25] and Aurora [26] present secure clock designs that use system management interrupt (SMI) but rely on a kernel daemon to trigger SMI which can be arbitrarily delayed by a compromised OS. In addition to being vulnerable, it is specific to Intel SGX and intended for servers and cloud environments. Timeseal [8] and T3E [27] present an improved design of a secure timestamping service using Intel SGX, however, they only provide trusted time inside the TEE which limits their benefits to trusted world and increase the TCB. Further, T3E [27] does not protect against delay attacks by the privileged adversaries. Further, platforms such as Android do not give direct TEE access to developers who may need trusted time. Our design provides a flexible trusted timing service that provides secure time to applications outside TEE.

System	LoC (1 App)	LoC (n Apps)
Existing TEE (e.g. ARM TZ)	x	$n * x$
TimeSeal	250	$n * 250$
TimeGaurd	150	150

TABLE IV: Trusted computing base comparison (approximate number of *Lines of Code* (LoC).)

Realtime Kernel Protection. Samsung Knox [17] – a real-time kernel protection mechanism – de-privileges the untrusted OS and prevent it from executing privileged instructions. However, timing attacks are not complex; they can be as simple as delaying time requests [8]. SATIN [18] is another work, that aims to detect the presence of adversary through changes made to the kernel. However, such an inspection may not be sufficient to prevent timing attacks. A privileged adversary can hide inside a device driver and launch timing attacks without modifying the kernel or its data. Similar attack vector could also be used against SPROBES [28] and SKEE [29], integrity protection solutions that aim at protect kernel integrity by monitoring each page table operation. Our design defends against all timing attacks, especially delay attacks, and it can be implemented alongside these holistic solutions to harden them against timing attacks.

Secure Communication Channels between TEE and User Programs. TrustICT [30] establishes a secure communication channel between the applications in normal world and the TEE, by interposing context switches between the user programs and the untrusted kernel. SeCRet [31] establishes such a channel using expensive cryptographic operations and inspecting page table operations. These solutions may be used to transfer trusted time from the secure world to user applications. While these solutions do protect the integrity of timing information they do not prevent delay attacks.

VIII. CONCLUSION

TIMEGUARD provides trusted time to applications using secure primitives of the modern TEEs. Our design enables an adjustable trade-off between accuracy of the trusted time and the system overhead. Some future directions to consider are TIMEGUARD integration with kernel integrity protection solutions such as Knox [17] and SATIN [18], and do security analysis under double page mapping attacks, and enabling multitenancy with a low overhead and high accuracy.

ACKNOWLEDGEMENTS

We thank the anonymous RTAS reviewers for their insightful comments and feedback. This research is supported by NSF grants 2237485 and 2230143.

REFERENCES

[1] S. Hamilton, D. Sengupta, and R. Gupta, "Introducing automatic time stamping (ats) with a reference implementation in swift," in *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2018, pp. 138–141.

[2] S. Li, X. Fan, Y. Zhang, W. Trappe, J. Lindqvist, and R. E. Howard, "Auto++ detecting cars using embedded microphones in real-time," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 3, pp. 1–20, 2017.

[3] S. Mirzamohammadi, Y. M. Liu, T. A. Huang, A. A. Sani, S. Agarwal, and S. E. S. Kim, "Tabellion: Secure legal contracts on mobile devices," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 220–233. [Online]. Available: https://doi.org/10.1145/3386901.3389027

[4] S. S. Sandha, J. Noor, F. M. Anwar, and M. Srivastava, "Time awareness in deep learning-based multimodal fusion across smartphone platforms," in *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, 2020, pp. 149–156.

[5] R. S. Hallyburton, Y. Liu, Y. Cao, Z. M. Mao, and M. Pajic, "Security analysis of Camera-LiDAR fusion against Black-Box attacks on autonomous vehicles," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1903–1920. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/hallyburton

[6] F. Alder, G. Scopelliti, J. Van Bulck, and J. T. Mühlberg, "About time: On the challenges of temporal guarantees in untrusted environments," in *Proceedings of the 6th Workshop on System Software for Trusted Execution*, ser. SysTEX '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 27–33. [Online]. Available: https://doi.org/10.1145/3578359.3593038

[7] J. Selvi, "Breaking ssl using time synchronisation attacks," in *DEF CON Hacking Conference*, 2015.

[8] F. M. Anwar, L. Garcia, X. Han, and M. Srivastava, "Securing time in untrusted operating systems with timeseal," in *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2019, pp. 80–92.

[9] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon *et al.*, "ftpm: A software-only implementation of a tpm chip." in *USENIX Security Symposium*, vol. 16, 2016, pp. 841–856.

[10] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1416–1432.

[11] ARM. (2022, June) What is the generic timer? [Online]. Available: https://developer.arm.com/documentation/102379/0000/What-is-the-Generic-Timer-

[12] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "Boomerang: Exploiting the semantic gap in trusted execution environments." in *NDSS*, 2017.

[13] ARM. (2022, June) Armv8 architecture registers, cntpct_el0. [Online]. Available: https://developer.arm.com/documentation/ddi0595/2021-06/AArch64-Registers/CNTPCT-EL0--Counter-timer-Physical-Count-register

[14] ——. (2022, June) System counter. [Online]. Available: https://developer.arm.com/documentation/102379/0100/System-Counter

[15] U. B. EECS. (2022, June) Risc-v - instruction set manual. [Online]. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf

[16] S. Sandha and T. Xing. (2019, June) Github: Cmactivities dataset. [Online]. Available: https://github.com/nesl/CMActivities-DataSet

[17] U. Kanonov and A. Wool, "Secure containers in android: the samsung knox case study," in *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2016, pp. 3–12.

[18] S. Wan, J. Sun, K. Sun, N. Zhang, and Q. Li, "Satin: A secure and trustworthy asynchronous introspection on multi-core arm processors," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 289–301.

[19] ARM. (2023, October) Arm security technology building a secure system using trustzone technology. [Online]. Available: https://developer.arm.com/documentation/PRD29-GENC-009492/latest/

[20] K. Enclave. (2023, October) Keystone: An open framework for architecting trusted execution environments. [Online]. Available: https://keystone-enclave.org/

[21] ARM. (2022, June) Armv8 architecture registers, cntkctl_el1. [Online]. Available: https://developer.arm.com/documentation/ddi0595/2020-12/AArch64-Registers/CNTKCTL-EL1--Counter-timer-Kernel-Control-register

[22] ——. (2022, June) Region security permissions using trustzone address space controller. [Online]. Available: https://developer.arm.com/documentation/ddi0431/b/functional-description/functional-operation/region-security-permissions

[23] NXP. (2023, October) Sabre board for smart devices based on the i.mx 6quad applications processors. [Online]. Available: https://www.nxp.com/design/development-boards/i-mx-evaluation-and-development-boards/sabre-board-for-smart-devices-based-on-the-i-mx-6quad-applications-processors:RD-IMX6Q-SABRE

[24] ARM. (2023, October) Timers, security extensions implemented, virtualization extensions not implemented. [Online]. Available: https://developer.arm.com/documentation/ddi0406/c/System-Level-Architecture/The-Generic-Timer/About-the-Generic-Timer/Timers?lang=en

[25] H. Liang and M. Li, "Bring the missing jigsaw back: Trustedclock for sgx enclaves," in *Proceedings of the 11th European Workshop on Systems Security*, 2018, pp. 1–6.

[26] H. Liang, M. Li, Q. Zhang, Y. Yu, L. Jiang, and Y. Chen, "Aurora: Providing trusted system services for enclaves on an untrusted system," *arXiv preprint arXiv:1802.03530*, 2018.

[27] G. M. Hamidy, P. Philippaerts, and W. Joosen, "T3e: A practical solution to trusted time in secure enclaves," in *International Conference on Network and System Security*. Springer, 2023, pp. 305–326.

[28] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the trustzone architecture," *arXiv preprint arXiv:1410.7747*, 2014.

[29] A. M. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning, "Skee: A lightweight secure kernel-level execution environment for arm." in *NDSS*, vol. 16, 2016, pp. 21–24.

[30] J. Wang, Y. Wang, L. Lei, K. Sun, J. Jing, and Q. Zhou, *TrustICT: An Efficient Trusted Interaction Interface between Isolated Execution Domains on ARM Multi-Core Processors*. New York, NY, USA: Association for Computing Machinery, 2020, p. 271–284. [Online]. Available: https://doi.org/10.1145/3384419.3430718

[31] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "Secret: Secure channel between rich execution environment and trusted execution environment." in *NDSS*, 2015, pp. 1–15.

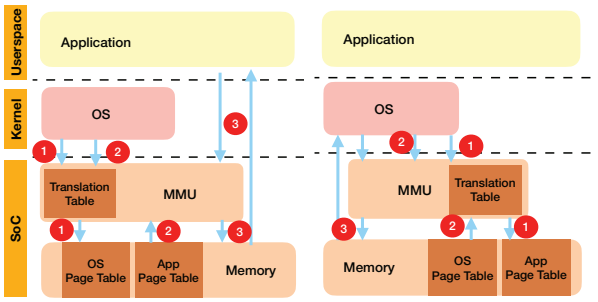


Fig. 13: Address Space Translation on a typical system: a) to run an application, ① the OS saves its own page table, ② loads the pointer to the application's page table, ③ application now runs in less privileged mode, and MMU performs address translation without any interaction with the OS; b) the reverse happens when application performs a system call or OS performs a context switch.

APPENDIX

Figure 13 shows the memory access mechanism in a typical system. When an application is launched, OS programs the Memory Management Unit (MMU) with the corresponding application's page table pointer, and hands over CPU execution to the application. The application's accesses to the memory are performed by the MMU without any interaction with the OS, which continues until the application requires a new memory allocation or when the OS loads its own page table to the MMU and performs switch to the kernel mode. Other situations when the OS will be alerted of the access to a particular memory location is when that memory location does not belong to the user application.