Hora: High Assurance Periodic Availability Guarantee for Life-Critical Applications on Smartphones

Dylan Zueck
dzueck@uci.edu
University of California, Irvine
Irvine, California, USA

Nathaniel Atallah
atallahn@uci.edu
University of California, Irvine
Irvine, California, USA

Ian Do
iydo@uci.edu
University of California, Irvine
Irvine, California, USA

Zhihao Yao
zhihao.yao@njit.edu
New Jersey Institute of Technology
Newark, New Jersey, USA

Ardalan Amiri Sani
ardalan@uci.edu
University of California, Irvine
Irvine, California, USA

ABSTRACT

Body-worn medical devices benefit from having a companion mobile application to monitor them and even program them. For example, the companion application for an insulin pump can be used to automatically monitor blood sugar level throughout the day and administer insulin without user input [1]. Unfortunately, the operating systems of modern smartphones cannot provide adequate security guarantees for these applications. Existing Trusted Execution Environment (TEE) solutions aim to alleviate these problems by removing the system software (and even most of the hardware [19]) from the TCB. However, no existing TEE solution provides a critical guarantee needed for these applications: *periodic availability*. This is needed to ensure that the application is executed according to a requested schedule, e.g., multiple times a day to read the patient's blood sugar and administer insulin. We present our ongoing work on Hora[1], a high assurance TEE solution for smartphones that guarantees periodic availability of CPU and I/O with a minimal and formally-verified scheduler. We present the design of Hora as well as its scheduler, which is implemented fully in Rust (in 1583 lines of code) and (partially) formally verified using the Kani model checker [7].

[1]Spanish word meaning "hour", pronounced as 'ora.

CCS CONCEPTS

• **Security and privacy** → **Trusted computing**; **Mobile platform security**; **Denial-of-service attacks**; • **Software and its engineering** → **Software verification**.

KEYWORDS

Availability, TEE, Mobile computing, Verification

ACM Reference Format:
Dylan Zueck, Nathaniel Atallah, Ian Do, Zhihao Yao, and Ardalan Amiri Sani. 2024. Hora: High Assurance Periodic Availability Guarantee for Life-Critical Applications on Smartphones. In *15th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '24), September 4–5, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3678015.3680486

1 INTRODUCTION

The ubiquity of modern smartphones makes them an attractive option for hosting a variety of applications. More specifically, body-worn medical devices benefit from having a companion mobile application to monitor them and even program them. For example, the companion application for an insulin pump can be used to automatically monitor blood sugar levels throughout the day and administer insulin without user input [1]. We refer to such an application as *life-critical* since it can result in grave consequences to the user if its execution is tampered with in any form.

Today, modern smartphone operating systems cannot provide an adequate level of security for executing life-critical applications [18]. Mobile operating systems are ridden with vulnerabilities as evidenced by Android having 1422 CVEs reported in 2023 and 1223 in 2022 [3]. The high stakes nature of life-critical application deployment makes it difficult and dangerous to rely on operating systems to withstand the threat of malicious applications.

Mobile Trusted Execution Environment (TEE) solutions such as Arm TrustZone, Arm Confidential Compute Architecture [11], and the split-trust hardware [19] provide enhanced security guarantees by reducing the size of the Trusted Computing Base (TCB). However, no existing TEE provides a critical guarantee needed by life-critical applications: *periodic availability*. More specifically, these applications may require system resources on a strict schedule to perform life-critical operations. For example, the companion application of an insulin pump may need periodic access (e.g., every hour) to a secure execution environment as well as the Bluetooth interface to interact with and/or program a glucose monitor and an insulin pump. Without availability guarantees, companion applications on personal smartphones cannot be trusted to perform their critical actions.

Due to the inadequate security of smartphones, today, we have two options. The first option is to use a locked-down, dedicated smartphone for these programs [4, 18]. This option is inconvenient and expensive for users as it requires buying and carrying a second smartphone.

The second option is to execute these life-critical tasks on medical devices themselves [5]. This option however has several disadvantages. First, it introduces unnecessary hardware and software complexity in medical devices, which can enlarge the attack surface of the device, increase its cost, and decrease its battery life. Increased cost is an important problem as some devices are intended for one-time use and are disposable [4]. Second, medical devices do not have as much computing power as smartphones. The extra computing power enables advanced algorithms. For example, algorithms for administering insulin have proven to be more complicated than a one-size-fits-all solution [16]. The variation in individual responses to insulin has led to incorporating a user's historical blood sugar levels with predictive neural network models. Third, medical devices have no or inferior user interfaces (UI) compared to smartphones. A better UI helps keep the user more informed and involved in treatment decisions.

The objective of this paper is to present a high assurance TEE system that enables running life-critical applications alongside untrusted applications on the user's personal smartphone. To achieve this goal, we propose Hora, which is designed to guarantee periodic availability. That is, Hora periodically provides these applications with a TEE to run while simultaneously giving them guaranteed and exclusive access to all necessary I/O system resources. We present our ongoing efforts to formally verify Hora's scheduler in order to guarantee the periodic availability of system resources after it accepts a request from an application. Towards that goal, we have written the scheduler fully in Rust and (partially) verified it using the Kani model checker [7] along with utilizing the Rust ownership system for static verification.

Kani allows us to build proofs that can analyze the scheduler's behavior in all possible states. In these proofs, we utilize stubbing to directly examine the actions of the scheduler's executor to ensure Hora interacts with the hardware such that it upholds its promised schedules. To ensure Hora makes no mutually-exclusive schedule promises, we also leverage the Rust ownership system to statically prove exclusive ownership of system resources needed for each schedule.

Overall, we implemented Hora's scheduler in 1583 lines of Rust code. Moreover, as a proof of concept of Hora's effectiveness, we implemented a life-critical application that periodically takes control of a TEE and the Bluetooth Low Energy domain. This application can then be used to command an Omnipod DASH [4] insulin pump, which we have also ported successfully.

2 BACKGROUND

We build Hora on top of the split-trust hardware [19], which provides physically-isolated TEEs for security-critical applications on smartphones and supports secure I/O for these applications, all with a minimal and formally-verified TCB. Each TEE and I/O device leverages its own physically-isolated hardware domain consisting of statically-partitioned CPU and memory. Domains are connected via mailboxes, which facilitates communication between them.

A security-critical application running in a TEE domain can request exclusive access to an I/O domain (to access I/O resources) by sending a request to the scheduler (i.e., the Resource Manager domain). The scheduler can approve or deny the request. Upon approval, the scheduler resets the requested domain to a clean state, and then delegates corresponding mailboxes to the requesting domain for a specified session duration. The split-trust hardware mailbox's design ensures that once a mailbox is delegated, the requesting domain has exclusive access (at the hardware level) to that mailbox for the requested duration.

The split-trust hardware ensures *session availability*, meaning that once a resource request is approved, the resource will remain available until the session ends. A hardware reset guard ensures that the scheduler cannot reset a domain (either a TEE or I/O domain) that has an active session.

Applications can verify the correctness and freshness of a domain through the Trusted Platform Module (TPM) attestation protocol. As a result, the scheduler does not need to be trusted by security-critical applications to provide secure and exclusive access to domains, or to ensure *session availability* guarantees.

The split-trust hardware solution, however, cannot ensure *periodic availability* since the untrusted scheduler can simply refuse to execute an application or give it the I/O resources that it needs. This is the problem that we try to address in this work.

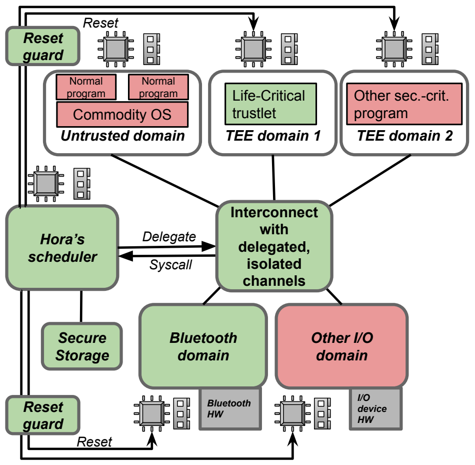

Figure 1: Hora's architecture. We show a life-critical trustlet running with exclusive access to a TEE domain and the Bluetooth domain. Red and green represent untrusted and trusted components, respectively, for the trustlet.

3 ARCHITECTURE

Hora is a high assurance TEE system for smartphones that ensures the availability of system resources for life-critical applications. Hora's scheduler receives syscalls from the various TEE domains, which can request one-time or periodic access to system resources. Hora reserves the right to accept or reject any such request in order to better utilize resources. But once it approves a periodic access request, it guarantees to fulfill it. Figure 1 shows the architecture of Hora.

Trustlets are minimal-sized companion applications that implement the crucial functionalities of their primary life-critical applications. Trustlets should be minimal in size because, in case of periodic execution, their images along with any other vital images must be kept in a non-volatile secure storage, which is small to reduce hardware costs. Vital images include any drivers for I/O domains that the scheduled trustlets rely on as well as the trustlet images themselves. The secure storage is necessary to ensure the availability of vital images without trust in the main system storage. Additionally, the secure storage allows Hora to keep track of any approved schedules after losing power.

A trustlet can issue a *schedule request*, which is for periodic and indefinite execution of the trustlet including a list of I/O domains that the trustlet will need, how long the trustlet will need to run, and at what frequency. If the scheduler approves the request, it assigns the trustlet periodic execution windows. At the start of every execution window, the scheduler guarantees to reset a TEE domain and all requested I/O domains, run the trustlet in the TEE domain, and delegate all requested I/O domains to the TEE domain for the duration of time it promised. During this period of time, the trustlet has exclusive access to the TEE domain as well as all I/O resources it requested with the guarantee that they are not contaminated by other malicious applications.

In order to follow the schedule, restrictions on when Hora can accept temporary (i.e., one-time) and schedule (i.e., periodic) requests must exist. Due to the inability to revoke access to a domain once delegated, Hora must never accept a temporary request that overlaps with existing schedules. Additionally, Hora must never accept a schedule request that overlaps with another schedule. To achieve high assurance, we formally verify all these properties to ensure the system will accurately follow all promised schedules.

Currently, once scheduled, only the trustlet itself is able to revoke its schedule. We are however designing a secure solution to enable the user to revoke a schedule as well (in order to prevent poorly-implemented trustlets from getting scheduled executions forever).

4 THREAT MODEL

The goal of the adversary is to prevent Hora from fulfilling its schedule promises and/or compromise the trustlet. We assume the adversary to have control of any application running on any of the TEE or untrusted domains, other than the trustlet that we aim to protect. The adversary can compromise any I/O domain they use, although their control over an I/O domain is lost once the domain is reset. We also assume that the adversary is not able to compromise Hora's scheduler. Since Hora is built on top of the split-trust hardware, most hardware side channels are mitigated [19]. We assume remote network attacks on the trustlet are mitigated by the trustlet using a secure channel to communicate to authenticated servers only. Hardware modifications and physical attacks are out of scope.

5 PERIODIC AVAILABILITY

Hora provides and guarantees periodic availability, which ensures that a trustlet is executed repeatedly with a consistent interval between executions. Scheduled trustlets may continue to run periodically indefinitely. Therefore, care must taken to ensure that conflicting schedules are not approved. To be able to verify that two given requested schedules will never overlap at any point in the future, we assign trustlets to time slots within a fixed repeating period, `SCHEDULE_PERIOD`. This exact value is not an inherent restriction and can be expanded, as discussed in (§9).

More specifically, the schedules are represented by time slot tables on a per-domain basis, which together represent one `SCHEDULE_PERIOD` of the system schedule. Figure 2 illustrates an example. For Hora to make any delegations, it must first acquire exclusive ownership of slots in the schedule. To that end, we utilize Rust's ownership system to statically

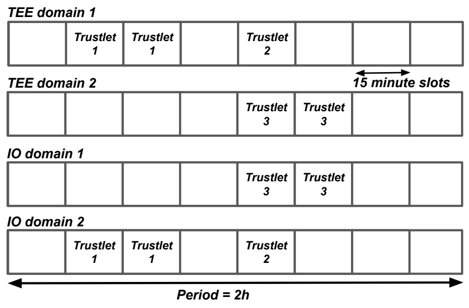

Figure 2: An example of Hora's schedule configured with `SCHEDULE_PERIOD` $= 2h$ and `NUM_SLOTS` $= 8$. All scheduled trustlets have contiguous slots.

ensure exclusive ownership of time slots. This is done by owning an `OwnedSlots struct` for both the requesting TEE domain and the requested I/O domain. Since `OwnedSlots` does not implement the `Copy` or `Clone` trait, ownership of the `struct` can only be moved, but not copied or cloned. Using this exclusive ownership combined with some additional proofs (§6), we can ensure that owning this `struct` represents mutually exclusive ownership of the set of contiguous slots indicated by the `struct`.

While we ensure that no future temporary delegation will overlap with a schedule once promised, we make no such guarantee for previous temporary delegations. This may sound like it violates our guarantees, but it does not and is actually desirable. Since contentious resources may be temporarily delegated for a majority of time, it may be difficult to establish a schedule on such resources. If we allow promised schedules to overlap with previous temporary delegations and limit temporary delegations to have a maximum time of `SCHEDULE_PERIOD` (as we do), then we can still guarantee that a promised schedule will start to execute (as defined in table 1) within two `SCHEDULE_PERIOD`s.

6 FORMAL VERIFICATION

6.1 Theorems

To formally verify that Hora will accurately follow all accepted schedules, there are a few theorems we must prove, which can be seen in Table 1. These theorems are for our current design where all schedules have a period of `SCHEDULE_PERIOD` and there is no way to remove an approved schedule.

We start by combining our first 3 theorems to show that each schedule will be executed periodically with a period of `SCHEDULE_PERIOD`. First, we demonstrate that approved schedules will be executed periodically if the executor function is called `NUM_SLOTS` times periodically. Theorem 1 and 2 demonstrate this through induction. Theorem 1 establishes the base case by showing an approved schedule request is first executed within `NUM_SLOTS` calls of the executor function. Theorem 2 then demonstrates that given the base case, Hora will continue to execute the approved schedule every `NUM_SLOTS` calls to the executor function.

Since we know that calling the executor function `NUM_SLOTS` times will repeat approved schedules, we now must create a link between a number of calls and a real world amount of time passing. If we can show that the executor function is called `NUM_SLOTS` times in `SCHEDULE_PERIOD`, then we can say that the executor executes approved schedules periodically with a period of `SCHEDULE_PERIOD`. This is what Theorem 3 demonstrates. Theorem 3 shows that the executor function is called periodically with a period of `SCHEDULE_PERIOD/ NUM_SLOTS` which means if called `NUM_SLOTS` times, `SCHEDULE_PERIOD` time has passed. We do note that Theorem 3 allows for a small bounded delay, which allows time for Hora to make scheduling decisions while not being long enough to impact the majority of applications.

While we know that the executor will take the correct actions, we must also show that these actions will not fail. There are two ways a schedule could fail to execute successfully. First, a necessary domain may already be delegated which would result in the split-trust hardware blocking the domain reset or delegation. To demonstrate that the split-trust hardware will not block necessary actions, Theorem 4 demonstrates the lack of overlapping promises, which would necessarily be blocked. Second, the necessary images to run the trustlet or reset a domain could be lost or corrupted. Theorem 5 ensures the availability of all necessary images such as the trustlet itself and the domain images. Combining all these theorems, we can prove that Hora will execute all approved schedules indefinitely.

6.2 Proof Techniques

We perform our proofs using a combination of the Kani model checker [7] and static checking from the Rust ownership system. Model checking works well for Hora as its design is stateful and it typically has to accommodate a small and limited number of approved schedules at any given time (due to the small number of existing life-critical applications that a user needs to use) limiting the state space and hence avoiding state explosion. So far, we have been able to prove Theorems 1, 2, and 4. For these, we will discuss the techniques used to prove them. For Theorems 3 and 5, we discuss the unique challenges involved in proving them.

Kani allows us to build our proofs directly in Rust. It provides the ability to assign variables to `kani::any()`, which allows Kani to reason about these variables in all possible valid states. We use `assert!` statements in our proofs to check that a property we are trying to prove holds in all valid states. Additionally, Kani will automatically fail if there is any path that leads to a panic, automatically proving the lack of panics.

Table 1: The theorems we must prove to formally verify Hora. "Executing a schedule" refers to taking all actions necessary for an approved schedule request such as resetting all requested domains and running the trustlet.

#	Theorem	Proven
1	If Hora approves a schedule request, the approved schedule will be executed within `NUM_SLOTS` calls of the executor function	Y
2	An action taken by the executor function is repeated on the `NUM_SLOTS`-th preceding call	Y
3	The executor function is called periodically (with an allowed bounded delay) with a period of `SCHEDULE_PERIOD/NUM_SLOTS`	N
4	No two approved schedules overlap and no temporary delegation overlaps with a currently approved schedule	Y
5	Scheduled trustlet and domain images for approved schedules are available	N

For Theorems 1 and 2, we also utilized the technique of *stubbing*. Stubbing is the ability to swap out certain code for other code that mimics the original code's behavior. Since our implementation is currently for the split-trust hardware emulator (§7), we use *stubbing* to avoid verifying the emulator specific code including any interactions with the split-trust hardware such as mailbox delegations. Importantly, when we transition to the hardware prototype, these actions can then be formally verified for properties such as correctness and lack of panicking. In order to keep proofs relying on these stubs sound, any response from split-trust hardware is assumed to be possible. Unfortunately, this does mean our proofs for these theorems do not currently cover our implementation of our interactions with the split-trust hardware and therefore must be assumed to be correct.

We were also able to prove Theorem 4. This theorem relies on both Kani and static checking from Rust as described in (§5). Kani is utilized to prove that `OwnedSlots` represents contiguous time slots. Rust's ownership system is used to statically verify that ownership of this *struct* represents exclusive ownership of those time slots.

Theorem 3 may present the most difficult challenge. It relies on proving the real-time characteristics of Hora, which is historically difficult for software [15]. To tackle this theorem, we plan on transitioning to a hardware prototype on the Xilinx Zynq UltraScale+ MPSoC ZCU102 FPGA board. This board includes a dual-core Cortex-R5F real-time processor, which we plan to use to run the scheduler and calculate a bounded running time for it. This is further complicated by the need to reset domains and run applications, which may also be challenging to find a bounded time for. Even with these challenges, we should be able to determine a maximum runtime due to the bounded size of images combined with the exclusive access from split-trust hardware ensuring no contention while sending images.

Proving Theorem 5 will require us to prove the correctness of a simple file system for our secure storage. Primarily, we need to be able to prove the availability and integrity of certain vital images that are actively in use by any promised schedules.

7 IMPLEMENTATION

Hora is implemented on top of the existing split-trust hardware emulator [2]. Hora mainly replaces the scheduler (i.e., Resource Manager), which previously was untrusted.

As a proof of concept, we are implementing an Omnipod Dash controller application, which is a minimal port of AndroidAPS [1]. AndroidAPS is an open source project, which has reverse engineered many diabetes pumps and continuous glucose monitors (CGM) to allow these devices to be used in an artificial pancreas system (APS). APS is a system in which little to no user input is required to manage insulin levels. Typically, these are implemented by introducing a Continuous Glucose Monitor (CGM), which can frequently report blood sugar levels throughout the day. The frequent blood sugar data can then be used to make predictions about future blood sugar levels applying insulin as necessary.

We have developed a trustlet that successfully uses Hora to get periodic access to a TEE domain and the Bluetooth Low Energy (BLE) domain needed to interact with the omnipod. We have also managed to port the Omnipod DASH driver from the AndroidAPS application to show the feasibility of implementing such an application using the Hora system.

To better understand how Hora can be used to implement a fully-featured application, we present the potential workflow for Omnipod devices. First, the user downloads the manufacture's application and runs it. Upon running, the application proves to the user that it is the correct application through its preferred method. Once the application is trusted by the user, it attempts to install its trustlet, which is responsible for communicating with the Omnipod. If rejected, the application will not be able to control the pod safely. If accepted, the application will be confident in its ability to consistently control the Omnipod and communicate with the CGM. The application then informs the user that it is now safe to pair with the Omnipod. Once paired and applied to the body, the trustlet can periodically (e.g. every 5 minutes) read the blood sugar level and update the Omnipod insulin delivery appropriately. Even if the main application is never able to be accessed again (e.g., due to some malware blocking it), the trustlet will be able to run in the background safely controlling insulin levels.

8 TCB ANALYSIS

The key component of the TCB is the scheduler. The scheduler consists of 1583 lines of Rust code, which has no unsafe sections (except emulator specific code) and uses `#no_std` disallowing the standard library.

In addition, the TCB includes the secure storage required by the scheduler. Proofs add 976 lines of Rust code and currently cover the scheduling behavior of Hora. Of course, the trustlet itself and the TCB of the split-trust hardware are also included in the TCB including the split-trust hardware's formally verified hardware and root of trust. But we note that the split-trust hardware is shown to have a minimal and formally-verified TCB [19]. Additionally, the Kani model checker and Rust compiler are included in the TCB.

9 ONGOING AND FUTURE WORK

In addition to completing the proofs (discussed in §6), there are several improvements to be made to Hora. First, the current design only allows schedule requests for periods that are equal to `SCHEDULE_PERIOD`. Removing this restriction is important as some applications may only need to run once a day, while others such as closed loop systems may need to run as often as every 5 minutes.

Additionally, it would be more efficient to allow each I/O domain to be acquired at different times throughout execution as well as for different amounts of time. For example, an application may request to run for 5 minutes, where it has network access for the first 3 and storage for the last 2.

Furthermore, secure storage is important for many life-critical applications, For example, our proof-of-concept application needs to be able to store communication details such as encryption keys. We plan to achieve this by allowing trustlets to utilize a small partition of our secure storage.

Finally, we need to consider how Hora should handle low power and power-off scenarios given that smartphones are battery powered. This includes deciding how Hora should resume schedule once the device is powered back on.

10 RELATED WORK

To our knowledge, the only prior works that provide similar guarantees are Aion and the work of Masti et al. [8, 13]. They provide CPU availability through trusted scheduling on embedded systems. While both can provide guaranteed access to I/O resources, neither can provide protection against other applications compromising needed drivers. Additionally, they place a high burden on these drivers to correctly facilitate mutually distrusting applications that share resources with additional limitations on atomic execution. Neither are formally verified, and the latter does not provide isolation needed for mutually distrustful applications.

Another notable attempt that may be able to mix availability with formal verification is the seL4 microkernel [10]. seL4 was formally verified by utilizing Isabelle/HOL [14] to create a series of refinement proofs to demonstrate that its C code accurately follows its high level specification. While seL4 at base does not provide availability guarantees, it has been shown to be usable as a base to provide similar guaranties as Aion in mixed criticality systems [12]. However, since it would provide similar guarantees, it also shares the same issues of reliance on drivers, and can only provide guarantees for the highest criticality threads.

Other system verification work exists that can formally verify security properties of systems. Sigurbjarnarson et al. [17] describes a framework for automatically verifying file system implementations with push button verification. Realms [11] introduced novel techniques to verify concurrent and interlanguage code to verify their firmware for their secure execution environment. Jitk was able to formally verify the correctness of a kernel interpreter using the Coq proof assistant [6]. However, to the best of our knowledge, no other work has formally verified the same periodic availability and security guaranties that we can utilizing a minimal TCB with formal verification.

Real time operating systems [9] can generally provide CPU availability or latency guarantees. However, guarantees are often given based on task criticality at the cost of other less critical tasks. Additionally, guarantees are often focused on latency and little consideration is given towards security/isolation especially strong TEE-grade isolation.

11 CONCLUSION

We presented Hora, a TEE solution for smartphones that can provide high assurance periodic and simultaneous access to CPU and I/O system resources. These assurances can be used to enable the creation and use of life-critical applications to run alongside mutually distrustful applications on smartphones. We provided these guarantees using a minimal TCB, in which the scheduler consists of only 1583 lines of Rust code and a small amount of secure storage. We also presented our on-going efforts to formally verify Hora's scheduler. Moreover, we presented a proof-of-concept application on top of Hora that can get exclusive control of a TEE and BLE domain according to a schedule.

ACKNOWLEDGMENTS

The work was supported in part by NSF Awards #1846230, and #1953932.

REFERENCES

[1] 2022. AndroidAPS app documentation. http://wiki.aaps.app/en/latest/.

[2] 2022. Source code for the split-trust hardware and its OS, OctopOS. https://github.com/trusslab/octopos_hardware, https://github.com/trusslab/octopos.

[3] 2024. Google » Android (Operating system): Product Details, Threats and Statistics. https://www.cvedetails.com/product/19997/Google-Android.html.

[4] 2024. Simplicity Starts With Omnipod DASH®. https://www.omnipod.com/what-is-omnipod/omnipod-dash.

[5] 2024. Simplify Life® With Omnipod® 5 Tubeless, Automated Insulin Delivery. https://www.omnipod.com/what-is-omnipod/omnipod-5.

[6] 2024. The Coq Proof Assistant. https://coq.inria.fr/.

[7] 2024. The Kani Rust Verifier. https://model-checking.github.io/kani/.

[8] F. Alder, J. Van Bulck, F. Piessens, and Jan T. Mühlberg. 2021. Aion: Enabling open systems through strong availability guarantees for enclaves. In *Proc. ACM CCS*. 1357–1372.

[9] P. Hambarde, R. Varma, and S. Jha. 2014. The Survey of Real Time Operating System: RTOS. In *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*. 34–39. https://doi.org/10.1109/ICESC.2014.15

[10] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. 2009. seL4: Formal Verification of an OS Kernel. In *SOSP*. 207–220.

[11] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell. 2022. Design and verification of the Arm Confidential Compute Architecture. In *Proc. USENIX OSDI*. 465–484.

[12] A. Lyons and G. Heiser. 2014. Mixed-Criticality Support in a High-Assurance, General-Purpose Microkernel. In *Workshop on Mixed Criticality Systems*. 9–14.

[13] Ramya J. Masti, C. Marforio, A. Ranganathan, A. Francillon, and S. Capkun. 2012. Enabling Trusted Scheduling in Embedded Systems. In *Proc. ACM ACSAC*. 61–70.

[14] T. Nipkow, M. Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer.

[15] P. Puschner and C. Koza. 1989. Calculating the Maximum Execution Time of Real-Time Programs. *Real-time systems* 1, 2 (1989), 159–176.

[16] G. Quiroz. 2019. The Evolution of Control Algorithms in Artificial Pancreas: A Historical Perspective. *Annual Reviews in Control* 48 (2019), 222–232.

[17] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *Proc. USENIX OSDI*. 1–16.

[18] DiabetesMine Team. 2019. NEWS: OmniPod Tubeless Insulin Pump to Offer Smartphone Control Soon. https://www.healthline.com/diabetesmine/omnipod-smartphone-control-diabetes.

[19] Z. Yao, S. M. Seyed Talebi, M. Chen, A. Amiri Sani, and T. Anderson. 2023. Minimizing a Smartphone's TCB for Security-Critical Programs with Exclusively-Used, Physically-Isolated, Statically-Partitioned Hardware. In *Proc. ACM MobiSys*.