Hey Kimya, Is My Smart Speaker Spying on Me? Taking Control of Sensor Privacy Through Isolation and Amnesia

Piet De Vaere
ETH Zürich

Adrian Perrig
ETH Zürich

Abstract

Although smart speakers and other voice assistants are becoming increasingly ubiquitous, their always-standby nature continues to prompt significant privacy concerns. To address these, we propose KIMYA, a hardening framework that allows device vendors to provide strong data-privacy guarantees. Concretely, KIMYA guarantees that microphone data can only be used for local processing, and is immediately discarded unless a user-auditable notification is generated. KIMYA thus makes devices accountable for their data-retention behavior. Moreover, KIMYA is not limited to voice assistants, but is applicable to all devices with always-standby, event-triggered sensors. We implement KIMYA for ARM Cortex-M, and apply it to a wake-word detection engine. Our evaluation shows that KIMYA introduces low overhead, can be used in constrained environments, and does not require hardware modifications.

1 Introduction

Over the past five years, we have witnessed a massive rise in the popularity of voice assistants such as Apple's Siri, Amazon's Alexa, or Google Assistant. In fact, in January 2021, more than one in three adults living in the U.S. owned a smart speaker [59]. Moreover, voice assistants are increasingly being integrated into everything from headphones to glasses and cars, and recent technologies such as ARM Helium promise to facilitate voice assistants even on the smallest devices [65].

Despite their success, the "always on" nature of voice assistants has given rise to significant concerns about the privacy and societal implications of these devices [35]. In particular, a significant fraction of people, including voice assistant users, are worried about voice assistants being hacked [7, 24], or that they are covertly recording and being used to spy on them [27, 37, 38]. This fear has been further amplified by reports that voice assistant interactions—including false positives—are recorded and reviewed by humans [4]. Moreover, current hardware capabilities and attacks go well beyond speech recognition [52]. For example, smart-speaker hardware is sufficiently precise to perform sonar ranging and thereby detect movements as small as a heartbeat [60].

Today, voice assistant vendors are addressing these privacy issues by allowing users to mute device microphones, and by adding status indicators which show when devices are actively recording audio. Although these methods can provide some level of protection, they also have significant shortcomings. Concretely, (i) muting a voice assistant removes almost all of its (useful) functionality, and (ii) both microphone muting and status indicator mechanisms are typically implemented as opaque features, making it unclear how strong the privacy guarantees they aim to provide really are. To illustrate the latter, while Amazon requires devices to "implement *(...)* a hardware-based microphone on/off control" [2], it only requires "a dedicated microphone status indicator" [2], without any further security requirements. If such a status indicator is controlled by standard software, it can potentially be disabled by a remote adversary.

On a more general level, the proliferation of sensors in our daily lives is making it increasingly more difficult for individuals to know when they are (not) being monitored. We consider this trend to be undesirable, and believe that people deserve the assurance that they are not being unknowingly observed when they are in a private space [42].

Focusing on voice assistants, this leads us to the following research question: How can we ensure that voice assistants only record when spoken to, even when they have been compromised? However, voice assistants are just one instantiation of a more general class of devices: those with always-standby, event-triggered sensors. That is, devices that contain sensors that are continuously on *standby* (i.e., processing the sensor data for event detection), but only rarely *triggered*. Although voice assistants are currently the dominant device in this class, others, such as "always-on" cameras for smartphones, are already on the horizon [47]. When considering this more general device class, our research question generalizes to: How can we ensure that always-standby devices only record when triggered, even when they have been compromised?

Answering this question requires us to unify the apparently

conflicting requirements of a device that is always sampling its sensor, but should only record when triggered. Past work on sensor privacy under adversarial settings has either focused on restricting *all* access to sensors [6, 28], or on generating a notification for *any* sensor sampling activities [40, 41]. However, such approaches cannot differentiate between an always-standby device in standby mode (i.e., waiting for its trigger event), or in triggered mode (i.e., actively processing sensor data), and therefore do not address our research question.

In contrast, this paper makes the following observation: the key challenge is to guarantee that always-standby devices only locally process sensor data, and, if no event has occurred, immediately discard the sampled data. If this guarantee is met, the device cannot eavesdrop and its privacy implications are minimal. Applied to a voice assistant, they must immediately discard sampled audio if no wake word is detected.

To enable vendors to guarantee this property, we propose KIMYA[1], a hardening framework that restricts direct access to sensor data, but provides an isolated, *amnestic* execution container, inside of which applications can execute event-detection routines. When using KIMYA, application code maintains access to all sensor data, but can neither store nor transmit it without generating a user-auditable *notification*.

By using user-auditable notifications, KIMYA can allow application code to self declare when an event has occurred. This is necessary, as generally no ground truth data about event occurrences is available. After all, if such data were available, no event detection would have to be performed in the first place. Yet, the device user will often intuitively know when an event has occurred. Hence, KIMYA's notifications make a device accountable for its detection behavior.

When a user notices that notifications are generated when no trigger event has occurred, this strongly indicates that a device either (i) has been compromised and is being used to eavesdrop, or (ii) generates excessive false positives and is therefore not privacy-preserving. Based on this information, users can then decide to mend or, when this is not possible, to stop using the device.

KIMYA runs on commodity microcontroller units (MCUs). It achieves isolation by partitioning memory-mapped resources into multiple regions, and restricts access to these regions based on execution *phases*. Amnesia is achieved by routinely erasing memory regions that store sensor data or derivatives thereof. We design KIMYA's erasure schedule to not affect the continuity of event-detection algorithms. KIMYA can run together with existing application code on the same MCU and does not require additional hardware.

KIMYA allows for arbitrary code to be executed inside the event-detection container and is not dependent on cryptography. This is possible because KIMYA's security properties are based on isolation and amnesia rather than software attestation. Moreover, it ensures that KIMYA does not inhibit device vendors from updating their event-detection algorithms once devices are in the field. It also ensures that KIMYA is lightweight and applicable to constrained hardware. This is important because (i) it has been shown that applications such as wake-word detection are already possible on such hardware [67], and (ii) industry is actively working to further facilitate digital signal processing (DSP) and machine learning applications on constrained devices [65].

Further, KIMYA's lightweight design translates itself into a small trusted computing base (TCB) size when implemented. Combined with the strong properties KIMYA provides, this allows security audits to focus on a small, reusable, module with clearly defined functionality, which in turn facilitates the work of independent certification centers, such as the new Swiss National Test Institute for Cybersecurity (NTC).

We demonstrate KIMYA's applicability by implementing it on an ultra-low-power Cortex-M33 MCU with ARM TrustZone. We design our implementation to be minimally intrusive, allowing it to coexist with existing application code on the same device. Further, we demonstrate KIMYA's practical applicability by applying it to a wake-word detection engine running on a Cortex-M33 processor.

Concretely, this work presents the following contributions:

1. We design KIMYA, a hardening framework that provides strong privacy guarantees for event-triggered, always-standby sensors.
2. We implement KIMYA on ARM Cortex-M33, demonstrating its applicability, even on low-end hardware. We release our implementation as an open-source project.[2]
3. We implement a wake-word detection engine for ARM Cortex-M33 and apply our KIMYA implementation to it. We use this prototype to evaluate KIMYA's performance.

2 Background: TrustZone on Cortex-M

We implement KIMYA using TrustZone on the ARM Cortex-M architecture, the basics of which we describe below. However, KIMYA can also be implemented on other architectures, as discussed in Section 9.3.

TrustZone on Cortex-M introduces two new processor security states: secure and non-secure [31]. These states are orthogonal to traditional processor states such as thread vs. handler mode and privileged vs. non-privileged mode. The active security state is determined by the instruction pointer and a security map which partitions executable addresses into secure and non-secure regions.

Beyond executable memory, other MCU resources are assigned a security attribute. Resources marked as secure are only accessible to code running in the secure state. Resources marked as non-secure are accessible to all code. Because of this separation, the security states are also referred to as the secure and non-secure *worlds*.

[1] Swahilli for "silence".

[2] https://github.com/KimyaGateway

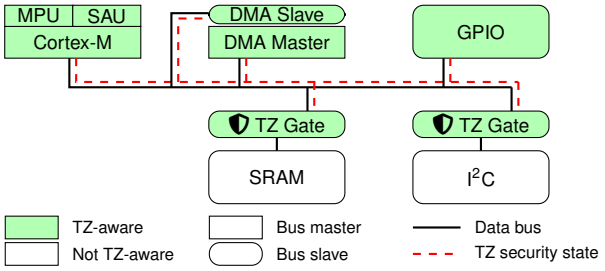


Figure 1: A high level overview showing how TrustZone (TZ) on Cortex-M extends beyond the processor core.

There are two principal ways to configure which resources are placed in which world. First, the Cortex-M core is extended with a secure attribution unit (SAU). The SAU is functionally similar to a memory protection unit (MPU) and can be used to configure specific memory regions as secure or non-secure. Second, in order to extend the concept of security states beyond the core, the data bus is extended to carry the security state of each transaction. Individual peripherals can either be made TrustZone-aware, or must be protected by a security gate, as shown in Fig. 1 [31]. Other bus masters (i.e., direct memory access (DMA) controllers) must also indicate the security state of their bus requests. Further relevant for our work is that the MPU is duplicated, with one instance being active when the core is in the secure state, and the other one when the core is in the non-secure state [30]. Both instances can be independently configured.

Contrary to TrustZone for Cortex-A, there is no secure monitor in the Cortex-M architecture. Transitions from the non-secure world to the secure world are facilitated through jumps to developer-defined *secure gateway* (SG) instructions, which provide a limited set of entry points into the secure world. Secure functions can also make calls to the non-secure world. After the non-secure function returns, control is then automatically returned to the secure world. To speed up transitions between the secure and non-secure worlds, some processor registers are banked.

3 Adversary Model & Security Setting

We consider a setting in which a user has equipped a private space, such as a home or workspace, with an Internet-connected device that has an always-standby sensor. The goal of the attacker is to access data from the always-standby sensor when no trigger event has occurred, and to do so with stealth, i.e., without generating a notification and without leaving an auditable trace.

The KIMYA trust model distinguishes between the platform vendor and application vendor of a device. The platform vendor constitutes the entity that produces the hardware platform and provides the KIMYA firmware. The application vendor implements the device functionality. It is possible for the platform and application vendor to be the same entity, but they can also be different entities within the same company, or different entities in different companies.

KIMYA requires the platform vendor to be trusted, as it provides the TCB upon which KIMYA's features are based. However, the application vendor can be untrusted. As the application vendor does not contribute to the TCB, companies providing both platform and application can focus their security resources on a smaller entity and a minimal code base. Platform vendors can obtain a trusted status through reputation or auditing.

We consider an adversary with full control over the device's network connectivity who can view, inject, and drop packets. Moreover, the adversary can exploit device vulnerabilities and can execute code on the device. The adversary might have already infected the device at the time of production. However, we explicitly do not consider attacks against trusted execution environments [11] or the platform code therein [34]. We also do not consider physical attacks, as adversaries with physical access could install their own covert sensors in the private space, significantly reducing the relevance of defenses against such adversaries.

4 Design

4.1 Design Goals

KIMYA's primary goal is to facilitate the privacy-preserving use of always-standby sensors. More concretely, this goal can be broken down into the following two subgoals.

G1, Availability: Sensor data must be made available to an event-detection algorithm.

G2, Isolation: It must be ensured that only sensor data related to an event can be used for purposes other than event detection. This is a high-level goal and will be refined in Section 4.3.

Additionally, we have the following secondary goals.

G3, Lightweight: KIMYA should be lightweight and deployable on microcontrollers. Its TCB should be small.

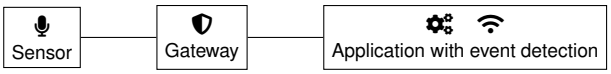
G4, Low-cost: KIMYA should not require designs to include additional hardware.

G5, Non-restrictive: KIMYA should not restrict which event-detection algorithms can be used.

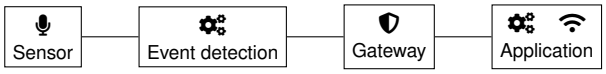
G6, Agile: KIMYA should not prevent application vendors from pushing updates to their devices, in particular, updates to the event-detection logic.

4.2 Straw-Man Proposals

Because our solution should be lightweight **(G3)**, we do not consider mechanisms that rely on dynamic code analysis. Additionally, both because of the lightweightness goal **(G3)**, and



(a) Using a sensor gateway.



(b) Using a sensor gateway and an isolated container.

Figure 2: Straw-man defenses for always-standby sensors.

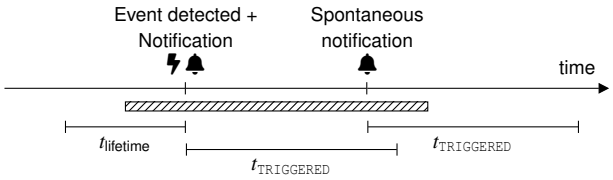


Figure 3: The interaction timeline. ▨ indicates an interaction.

to avoid the complexity of traditional attestation mechanisms, we refrain from using software attestation. Combining this restriction with the requirement for devices to be easily updatable **(G6)**, this means that we must consider all application code, including the event-detection code, to be untrusted.

Therefore, we focus on approaches whose properties are independent of the executed code, and instead are based solely on the properties of the environment in which code is executed. Concretely, we focus on approaches that directly restrict access to the always-standby sensor. Figure 2 displays two such approaches which serve as straw men for our final design.

The first design (Fig. 2a) places a gateway between the always-standby sensor and the application which also contains the event-detection logic. This approach is most similar to previous designs for peripheral access control, e.g., SeCloak [28]. However, once the gateway provides sensor access to the application (as required by **(G1)**), it can no longer control what purpose the sensor data is being used for, thus violating the isolation goal **(G2)**.

The second design (Fig. 2b) attempts to addresses this issue by separating event-detection functionality from the rest of the application. Event detection is then performed in an isolated environment that has direct access to the sensor. Doing so provides the event-detection code with continuous access to the sensor, while still allowing the gateway to restrict access for the other application code. Once the containerized code declares that a sensor event was detected, a notification is generated by the gateway and the application code is granted access to the event-detection container and sensor data.

Although this design represents a significant improvement over the design in Fig. 2a, it does not provide control over the state stored in the event-detection container. This means that it cannot yet fully satisfy the isolation goal **(G2)**. Concretely, the (untrusted) code running in the event-detection container could continuously eavesdrop, store captured information in the container, and exfiltrate this data to the application as soon as an event is detected.

4.3 Event-Detection Timeline

The shortcomings in the straw-man designs show that a more precise definition of the isolation goal **(G2)** is needed. Specifically, it must be defined which information may be made available to the main application once an event is detected.

To this end, we introduce the notion of a sensor *interaction*. Each sensor interaction corresponds to a time window during which data relevant for the device's legitimate operation is sampled by the always-standby sensor. For example, for the voice assistant interaction "Hey Kimya, what time is it", the time window would correspond to the period during which the user is uttering this sentence. For event-triggered sensors, interactions are initiated by an event. In the case of our example, the event is the utterance of "Hey Kimya".

Ideally, KIMYA would only grant unrestricted access to data sampled during the interaction period. However, doing so is impractical. As is illustrated in Fig. 3, an event can generally not be detected right at the start of an interaction (e.g., the utterance "Hey Kimya" can only be detected once it has been fully articulated). Similarly, immediately after the trigger event, it is unclear how long an interaction will last.

Therefore, we introduce the durations t_{lifetime} and $t_{\texttt{TRIGGERED}}$, as illustrated in Fig. 3. t_{lifetime} specifies how much data from before a trigger event can be released. In the case of a voice assistant, it should be set based on the expected maximum duration of the trigger phrase, plus any required *pre-roll*[3]. We discuss this in more detail in Section 9.1. Similarly, $t_{\texttt{TRIGGERED}}$ defines the duration for which data can be freely accessed after an event. Because interactions might last longer than $t_{\texttt{TRIGGERED}}$, a time extension mechanism must be foreseen. This can be achieved by allowing applications to generate spontaneous notifications, each of which resets $t_{\texttt{TRIGGERED}}$. Doing so effectively visualizes sensor-access behavior with a temporal granularity of $t_{\texttt{TRIGGERED}}$. Moreover, it facilitates interaction models where user input is expected at the end of a prompt.

Based on this model, we extend the straw-man design of Fig. 2b by additionally rendering the event detection container *amnestic*. That is, by ensuring that no data older than t_{lifetime} can be present in the container.

A naive approach to implement amnesia could be to periodically zero out the event-detection container's memory. However, doing so risks trigger events being segmented by a wipe. To illustrate this, consider again a wake-word detection engine that listens for the phrase "Hey Kimya". If a wipe event were to occur after a user has said "Hey", but before they said "Kimya", no trigger event would be detected. This

[3] *pre-roll* data refers to data captured just before a trigger-event occurred. It is often used to calibrate noise levels.

would be a violation of the non-restrictiveness goal **(G5)**.

In order to ensure data continuity while simultaneously limiting data age, KIMYA instead uses multiple buffers that are routinely wiped, and enforces an unidirectional information flow between these buffers. We present this design in more detail in the following section.

4.4 KIMYA Design

KIMYA provides two key features. First, it provides an isolated execution environment that has direct access to data from an always-standby sensor. Second, it ensures that this container is amnestic, i.e., it provides strong guarantees on the maximum age of sensor data (or information derived thereof).

In order to achieve these features, KIMYA segments its host MCU into five memory *regions*, and introduces four execution *phases*. Because on modern MCUs most peripherals are memory-mapped, these memory regions can also be considered to be *resource* regions. Concretely, the five memory regions, as depicted in Fig. 4a, are:

Sensor: A region containing the always-standby sensor to which access should be restricted.

Buffer A and Buffer B: Two memory regions forming a pair of alternating buffers to store sensor data.

Scratch: A memory region that can be used to store state for the event-detection algorithm.

All other memory: All memory-mapped resources not included in the other four regions. This includes General Purpose Input/Output (GPIO), timers, and communication peripherals.

KIMYA's four execution phases are listed below. Execution starts in the `IDLE` phase, and phase transitions are requested by the application code.

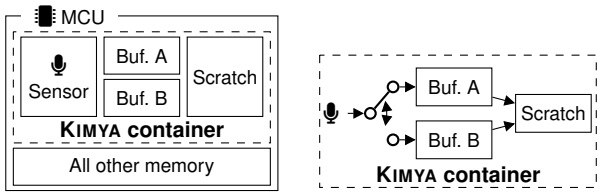
`IDLE`: As long as no trigger event has been detected, all application code that is not related to event detection is executed in the `IDLE` phase.

`ACQUIRE`: Used to acquire sensor data, to perform data preprocessing, and to store the result in the alternating buffer formed by the Buffer A and Buffer B memory regions.

`PROCESS`: After acquiring fresh sensor data, the event-detection is performed in the `PROCESS` phase. Event-detection state can be stored in the Scratch region.

`TRIGGERED`: When a trigger event has ocurred, the application code is executed in the `TRIGGERED` phase (instead of in the `IDLE` phase) for a preset duration $t_{\texttt{TRIGGERED}}$. Before the `TRIGGERED` phase can be entered, a notification (see Section 4.5) must be generated. Regenerating this notification while the `TRIGGERED` phase is active extends the duration of this phase to $t_{\texttt{TRIGGERED}}$ after the notification was generated.

Depending on the active execution phase, KIMYA restricts access to the various memory regions according to the permissions shown in Table 1. In the `IDLE` phase, no memory regions related to the always-standby sensor and event detection are accessible. During the `ACQUIRE` phase, sensor data can be sampled and written to one of the alternating buffers, but no other memory can be accessed. During the `PROCESS` phase, there is read-only access to both buffers and full access to the Scratch memory. Finally, during the `TRIGGERED` phase, all memory is fully accessible.



(a) Memory regions. (b) Data flow.

Figure 4: The KIMYA container.

Table 1: Memory access rights in the four KIMYA execution phases. '👁' indicates read access, '✎' indicates write access.

KIMYA phase	MCU memory region				
	Sensor	Buf. A	Buf. B	Scratch	All other mem.
`IDLE`	-	-	-	-	👁 ✎
`ACQUIRE (A)`	👁	👁 ✎	-	-	-
`ACQUIRE (B)`	👁	-	👁 ✎	-	-
`PROCESS`	-	👁	👁	👁 ✎	-
`TRIGGERED`	👁 ✎	👁 ✎	👁 ✎	👁 ✎	👁 ✎

Enforcing the Table 1 access map has two desirable effects. First, the `ACQUIRE` and `PROCESS` phase form an isolated container in which event detection can be performed. Second, a unidirectional data path, as illustrated in Fig. 4b is created.

In order to guarantee amnesia, KIMYA must enforce strong limits on the maximum age of the data inside the KIMYA container, that is, in the Buffer A, Buffer B, and Scratch memory regions.[4] To this end, the following buffer management schedule is executed every $t_{\text{lifetime}}/2$ seconds: (i) zero out the Buffer A or B that is not currently accessible from the `ACQUIRE` phase; (ii) zero out the Scratch memory region; and (iii) alternate Buffer A and Buffer B.

t_{lifetime} is a static value representing the permissible lifetime of sensor data. For most applications, this value will be on the order of seconds or less. Because Buffer A and B are not simultaneously erased, data continuity is provided.

Combining this buffer management schedule with the unidirectional data path, ensures that at any point in time, the data inside the KIMYA container cannot be older than t_{lifetime}, unless a notification is generated. Intuitively, this is the case because both the Scratch region and the alternating buffers are erased at least every t_{lifetime}, and the unidirectional data path ensures that no data can be transferred between Buffer A

[4] Data in the Sensor region is always fresh.

and Buffer B to circumvent the erasure schedule. A proof of this property is provided in Section 6.1.

4.5 Notification Design

KIMYA has no event-detection logic built-in. Therefore, KIMYA cannot rely on ground truth information to regulate access to the TRIGGERED phase. Instead, KIMYA places control over the active execution phase with the application itself. Concretely, execution starts in the IDLE phase, from which the application can request specific functions to be executed in the ACQUIRE or PROCESS phases. Upon returning, these functions indicate if they want to return the MCU to the IDLE phase, or, they can request a transition to the TRIGGERED phase. In the latter case KIMYA will generate a user-auditable notification before returning control back to the application code.

We do not prescribe a specific notification mechanism in this work. Instead, KIMYA provides a flexible platform upon which different notification mechanisms can be build. The design of an effective privacy notification mechanisms is orthogonal to our work, and has been studied before [48, 49].

To illustrate the flexibility KIMYA provides, we briefly discuss three types of notification below.

LED indicators. Similar to current smart-speaker products, a LED can be used to indicate when the device is in the TRIGGERED phase. KIMYA then provides strong guarantees that this indicator LED cannot be circumvented. Note that when using a visual indicator, a careful design is needed to ensure its effectiveness [36, 46].

Bluetooth beacons. Devices could broadcast bluetooth beacons containing information about the sensor that is being accessed. Other devices receiving these beacons could then visualize which information is being sampled from their surroundings.

Centralized logging. KIMYA could require a central server to be contacted before granting access to sensor data. This server can log all sensor activity and make it available for later auditing.

In order to guarantee the availability and integrity of the notification mechanism, it should be protected. This protection can be achieved through isolation or by using cryptographic techniques, depending on the notification mechanism that is in use. For example, an LED indicator can be efficiently protected by isolating the control over the GPIO pin to which it is connected. Conversely, a centralized logging scheme is best protected by establishing a cryptographic channel between the KIMYA gateway and logging server.

When notifications are generated in a machine-readable format (e.g., Bluetooth beacons or logs on a central server), a *privacy assistant* [12, 26] can be used to aid the user with the auditing of notifications.

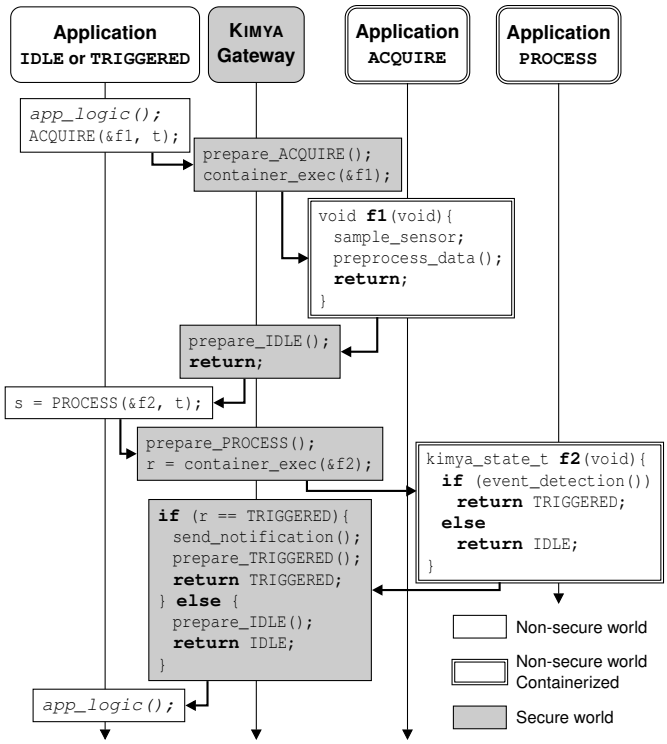


Figure 5: Simplified KIMYA execution flow. The parameter t is introduced in Section 5.4. The notification process is executed in the secure world.

5 Implementation on Cortex-M

We leverage TrustZone to implement KIMYA on a Cortex-M33 MCU. Specifically, we prototype our implementation on an STM NUCLEO-L552ZE-Q development board with an ultra-low-power STM32L552ZE MCU running at 110 MHz with MPU and a floating-point unit (FPU) [54].

We implement KIMYA as a gateway running in the secure world. All application code (including event detection) runs in the non-secure world and interacts with the secure-world gateway to request KIMYA phase transitions. We illustrate the basic KIMYA control flow in Fig. 5.

In order to transition to the ACQUIRE and PROCESS phases, the application makes a call to the gateway, passing a function pointer. The gateway function then transitions the MCU to the desired phase and executes the specified function in the non-secure world. When a function call to the ACQUIRE phase returns, the gateway transitions the MCU back to the IDLE phase and returns control to the non-secure world. Functions executing in the PROCESS phase can specify if the MCU should be transitioned to the IDLE or TRIGGERED phase before control is returned to the application. In the latter case the gateway will generate a notification (by calling send_notification(), Fig. 5) before executing the phase transition. In either case the non-secure world is informed about the currently active phase when control is returned to it.

5.1 Enforcing the KIMYA Access Map

The core of our KIMYA implementation is the enforcement of the access map in Table 1. We leverage a combination of the MPU, the TrustZone configuration of peripherals, and the TrustZone security gates (see Section 2) for this purpose. Concretely, during the `IDLE` phase, all KIMYA container-related memory regions are marked as secure in the TrustZone configuration, preventing the non-secure application from accessing them. Therefore, full MPU control can be granted to the application, ensuring compatibility with existing OSs.

When the application makes a call to the `ACQUIRE` or `PROCESS` phases, the KIMYA gateway calls `prepare_ACQUIRE()` or `prepare_PROCESS()` (Fig. 5) to take control of the MPU for the duration of that call. The required container resources are temporarily marked as non-secure, making them accessible to the application code. The MPU is then configured according to Table 1 to enforce the necessary access restrictions.

Before control is returned to the main application code, the gateway either transitions the MCU back to the `IDLE` configuration described above (by calling `prepare_IDLE()`, Fig. 5), or to the `TRIGGERED` configuration. In the latter case, the `prepare_TRIGGERED()` call marks all container-related memory regions as non-secure, and grants MPU control to the application. The application now has direct access to all resources needed to process the trigger event.

5.2 Enforcing Amnesia

In order to guarantee the amnestic property of the KIMYA container, the Buffers A/B and Scratch memory must be periodically erased and the one-way dataflow illustrated in Fig. 4b guaranteed. While this could be achieved using a secure timer-driven interrupt, our implementation instead checks if buffer maintenance is needed each time before entering the `ACQUIRE`, `PROCESS`, or `TRIGGERED` phase. This ensures that all data older than t_{lifetime} is erased before it can be accessed, while avoiding long interrupt routines.

We further extend the KIMYA API for the non-secure world with a `maintain_buffers(t)` call. This call prepones buffer maintenance by up to `t` time units. This allows the non-secure application to schedule buffer maintenance ahead of time, thereby eliminating the need for buffer maintenance to complete before the KIMYA container can be accessed and thereby reducing timing jitter.

5.3 Non-Secure Calls

Although we can rely on standard toolchain behavior to secure calls to the secure gateway (i.e., the application's `ACQUIRE()` and `PROCESS()` calls in Fig. 5), this is not the case for the gateway's calls into the non-secure world (i.e., the `container_exec()` calls in Fig. 5). The reason for this is twofold. First, upon entering the `ACQUIRE` or `PROCESS` phases, the MPU will be in a highly restrictive state. Because the main application's stack will not be accessible, a new stack must be set up for the containerized function. Second, the official toolchain requirements for TrustZone on Cortex-M [31] are designed to provide isolation between the secure and non-secure worlds, but do not isolate non-secure function calls made by the secure world from the main non-secure application thread. Concretely, standard toolchain behavior does not clear all non-secure world registers before and after non-secure function calls, and would thus allow for communication from the KIMYA container to the main application.

To address these issues, we implement `container_exec()` as an assembly function that performs the following tasks.

1. Push all general-purpose, special, and floating-point processor registers to the secure stack. In the case of banked registers, the non-secure register is pushed.
2. Clear all registers saved on the stack.
3. Move the non-secure stack pointer to a memory region that will be writeable in the container. When transitioning to the `ACQUIRE` phase, the stack pointer is moved to the top of the currently active Buffer A/B. For the `PROCESS` phase the top of the Scratch region is used.
4. Branch and link to the containerized function in the non-secure world.
5. Once the containerized function has returned, erase and restore all saved registers. This includes the non-secure stack pointer.

5.4 Ensuring Container Isolation

Beyond the core aspects described above, a number of other measures must be taken to ensure no data can be leaked from a KIMYA container. We discuss these practical measures here and present a more theoretical discussion in Section 6.2.

Container execution time. Using a timer, real-time clock (RTC), or similar peripheral, it is possible for the main application to measure the execution time of a KIMYA container. When no special care is taken, the event-detection code running in the KIMYA container could modulate its execution time to establish a uni-directional communication channel out of the container. This channel could have a capacity of up to $\log_2(f_{cpu})$ bps, where f_{cpu} is the core frequency in Hz.

In order to prevent this source of information leakage, we require the application to specify a desired execution time for each call to the `ACQUIRE` or `PROCESS` phases. The called function *must* return before the specified duration has passed. The gateway will then wait for the remainder of that duration before control is returned to the application. A read-only timer is made available in the container to allow the event-detection code to know how much execution time is left. This timer can also be used as a relative time base for event-detection tasks.

`TRIGGERED` execution time. When an event has been detected and a notification generated, the MCU is transitioned

to the TRIGGERED phase. By default the MCU can stay in this phase for up to $t_{\texttt{TRIGGERED}}$ seconds. $t_{\texttt{TRIGGERED}}$ is an implementation defined value. Before this duration has passed, the application *must* either (i) yield back to the IDLE phase, or (ii) request another notification to be generated, extending the yield deadline to $t_{\texttt{TRIGGERED}}$ seconds beyond that request. This deadline is enforced using an interrupt triggered by a secure timer. Setting the PRIS bit in the Application Interrupt and Reset Control Register (AIRCR) ensures that the non-secure application cannot mask this interrupt [32].

Caches. The STM32L552ZE MCU used for our implementation has a built-in instruction cache [54]. In order to prevent cache-timing attacks, the KIMYA gateway clears this cache whenever leaving the ACQUIRE or PROCESS phases. There are no other caches present on the STM32L552ZE.

DMA. As shown in Fig. 1, the MPU is part of the Cortex-M33 core, and therefore does not affect the operation of the DMA controller. Because KIMYA container resources are marked as non-secure during the ACQUIRE and PROCESS phases, this means that the application could preprogram the DMA controller to steal sensitive information while the MCU is in one of those phases. We propose two mechanisms to prevent this attack. First, the gateway can disable the DMA controller before moving container resources to the non-secure world. Second, the DMA controller can be moved to the secure world, removing the non-secure application's ability to program it directly. A thin secure-world DMA configuration shim can then verify that no DMA operations affect the KIMYA container memory regions. Our implementation uses the former strategy.

Peripheral use. In some cases, peripherals must be accessible in the KIMYA container. For example, our wake-word detection prototype (see Section 7) requires the cyclic redundancy check (CRC) peripheral to be available for intellectual property management.[5] In such cases, it must be ensured that (i) container isolation cannot be violated by using peripherals as communication channels with the main application, and (ii) that container amnesia cannot be violated by storing information in peripheral registers during memory wipe events. This can be achieved by setting all writeable peripheral memory locations to a well-known value after the function call from the ACQUIRE or PROCESS phase returns. In the case of our prototype, this is done by resetting the CRC peripheral.

Handling policy violations. Whenever event-detection or main application code violates a KIMYA policy (e.g., a container executing longer than was requested by the main application), a notification is generated. This ensures that misbehaving applications are detected and can be mended.

[5]The proprietary STM X-CUBE-AI neural network library uses the CRC peripheral to verify that it is running on STM hardware.

6 Security Analysis

KIMYA's security guarantees are derived from the two main properties it provides: amnesia and isolation. We provide a theoretical proof of KIMYA's amnesia property in Section 6.1. Isolation is discussed in Section 6.2.

6.1 Amnesia

In Section 4.4 we claimed that no data in the KIMYA container can be older than t_{lifetime}. We will now prove this property.

Proof model. In order to facilitate the proof, we introduce the variables T_{sensor}, $T_{\text{buffer A}}$, $T_{\text{buffer B}}$, T_{scratch}, which keep track of the genesis time of the oldest data that can be present in each memory region inside the KIMYA container. Because only the latest sample can be read of the sensor, we assume $T_{\text{sensor}} = t$ at all times t. At $t = 0$, all buffers are zeroed out, so we initialize the other variables as $T_{\text{buffer A}} = T_{\text{buffer B}} = T_{\text{scratch}} = 0$.

Whenever $t = n \cdot \frac{t_{\text{lifetime}}}{2}, n \in \mathbb{N}$, buffer maintenance is performed. This is modeled using the following sequential operations: $T'_{\text{buffer B}} = T_{\text{buffer A}}$, $T'_{\text{buffer A}} = T_{\text{scratch}} = t$, where we model the alternating buffer using a renaming operation to simplify notation. Additionally, at any point in time, the following two operations may be performed arbitrarily often: $\alpha : T'_{\text{buffer A}} = \min(T_{\text{buffer A}}, T_{\text{sensor}})$, and $\beta : T'_{\text{scratch}} = \min(T_{\text{buffer A}}, T_{\text{buffer B}}, T_{\text{scratch}})$. α and β model the ACQUIRE and PROCESS phase, respectively.

Proof. We now prove that at any time t, it holds that $T_{\text{min}} = \min(T_{\text{sensor}}, T_{\text{buffer A}}, T_{\text{buffer B}}, T_{\text{scratch}}) \geq t - t_{\text{lifetime}}$.

To this end, we first proof that neither operation α nor β can change T_{min}. This holds because both operations assign the minimum value from a subset of variables considered for T_{min} to a variable from that same set, thus not changing T_{min}.

Next, we show that neither α nor β can change the variables $T_{\text{buffer A}}$ and $T_{\text{buffer B}}$. For β this holds trivially, as it does not write to either of those variables. α writes only to $T_{\text{buffer A}}$, so cannot affect $T_{\text{buffer B}}$. Moreover, because time is monotonically increasing, we know that $T_{\text{buffer A}} \leq t$, so $T'_{\text{buffer A}} = \min(T_{\text{buffer A}}, T_{\text{sensor}}) = \min(T_{\text{buffer A}}, t) = T_{\text{buffer A}}$.

We know that $T_{\text{min}} \geq t - \frac{t_{\text{lifetime}}}{2}$ at $t = 0$ because all variables are initialized at 0. Now assume that $T_{\text{min}} \geq t - \frac{t_{\text{lifetime}}}{2}$ at $t = k \cdot \frac{t_{\text{lifetime}}}{2}, k \in \mathbb{N}$, then at $t = (k+1) \cdot \frac{t_{\text{lifetime}}}{2}$, after performing buffer maintenance it holds that (i) $T_{\text{sensor}} = t$ (by assumption), (ii) $T_{\text{buffer A}} = T_{\text{scratch}} = t$ (because of the wipe event), and (iii) $T_{\text{buffer B}} = T_{\text{buffer A}}(t = k \cdot \frac{t_{\text{lifetime}}}{2}) = k \cdot \frac{t_{\text{lifetime}}}{2}$ because $T_{\text{buffer A}}$ was set at $t = k \cdot \frac{t_{\text{lifetime}}}{2}$ and cannot have changed since then (see above). Thus, it holds that $T_{\text{min}} \geq t - \frac{t_{\text{lifetime}}}{2}$ at $t = (k+1) \cdot \frac{t_{\text{lifetime}}}{2}, k \in \mathbb{N}$. Therefore, by induction it holds that for all $n \in \mathbb{N} : T_{\text{min}} \geq t - \frac{t_{\text{lifetime}}}{2}$ at $t = n \cdot \frac{t_{\text{lifetime}}}{2}$.

Finally, because neither operations α nor β can change T_{min}, and because buffer maintenance is performed at least every $\frac{t_{\text{lifetime}}}{2}$, it must hold that $T_{\text{min}} \geq t - t_{\text{lifetime}}$ at any time t. □

6.2 Isolation

In order for KIMYA's isolation (and by extension amnesia) property to hold, it must be ensured that no covert channels exist. This section presents an overview of the considerations made during the design of KIMYA. As motivated in Section 3, we do not consider adversaries with physical access.

6.2.1 Storage Channels

Under storage channels we consider channels that transmit data by explicitly writing it to a storage location from which it can later be read back. Two types of storage are available on our target MCU: memory-mapped and register storage.

Memory-mapped storage. Memory-mapped storage comprises all storage locations that have a memory address and are accessed over the data bus using load or store instructions. As described in Section 5, our KIMYA implementation dynamically manages the MPU, TrustZone configuration, and the DMA controllers, and combines this with (re)setting memory location to well-known values to ensure that no communication is possible using these resources.

Processor registers. Some storage locations on the Cortex-M33 core are not memory-mapped, but can be directly accessed using dedicated instructions. To prevent covert channels that leverage these registers, we analyze the Cortex-M33 ISA to identify all writeable registers. As discussed in Section 5.3, we constructed the `container_exec()` function to ensure that all these registers are set to a well-known value after the containerized call returns.

6.2.2 Other Channels

Beside channels that directly write data to storage locations, other, indirect, channels must be considered as well. We inspect the architecture [33] and reference manual [55] of our target MCU to identify potential covert channel and present tailored defences below.

Timing channels. A containerized call could leak information by modulating its execution time. As discussed in Section 5.4, the application must therefore specify an exact execution time for each containerized function call.

Caches. Our target MCU has an instruction cache. Timing analysis on this cache could be used to establish a covert channel. As specified in Section 5.4 we flush the cache after every containerized call to prevent this.

Counters. Our target MCU has multiple debug and performance counters, e.g., a cache-miss counters. The containerized code could attempt to actively to influence these timer. We ensure that these counters are not readable by the non-secure world application.

Physical channels. Channels communicating using physical properties could be established. For example, the event-detection code could attempt to influence the temperature of the MCU package, which could then be measured using the MCU's built-in temperature sensor. We did not explicitly consider such channels, but if deemed necessary, they could be avoided by restricting non-secure access to specific resources, such as, the on-board temperature sensor.

Although the relative simplicity of our target MCU compared to high-end processors facilitates a covert-channel analysis, we consider strong claims about the total covert-channel capacity to be beyond the scope of this work. However, we do note that KIMYA's amnesia property strictly limits the attacker's window to exfiltrate information from the KIMYA container. This significantly reduces the utility of low-capacity covert channels, as any information not exfiltrated within at most t_{lifetime} from its genesis is lost to the attacker.

7 Prototype Application

In order to demonstrate KIMYA's practicality, we implement a proof-of-concept *keyword-spotting* pipeline on an ultra-low-power STM32L552ZE MCU, simulating the wake-word detection functionality of a voice assistant. We train the pipeline using recordings of one of the authors speaking the same words as used in the Google Speech Commands dataset [61]. The word "cat" is used as keyword. We then apply KIMYA to this detection engine.

7.1 Keyword-Spotting Pipeline

A typical keyword-spotting pipeline first extracts high-level speech features from the audio signal and then feeds these features to a neural network classifier [67]. Running inferences on speech features instead of raw audio considerably reduces the input dimensions of the classifier, thereby significantly simplifying the classifier's network and training process.

A commonly used feature set for speech processing are mel-frequency cepstral coefficients (MFCCs) making up the mel-frequency cepstrum [15]. Given the constrained nature of our target platform, we instead use the mel spectrum, a less processed version of the cepstrum. In order to obtain a mel spectrum of an audio segment the following steps must be taken. First, the audio segment is segmented into shorter *chunks* to which a fast Fourier transform (FFT) is applied. The resulting coefficients represent the Fourier spectrum of each chunk. Plotting these spectra against time results in the spectrogram of the audio segment. This spectrogram shows how the frequency components in the segment change over time. The frequency coefficients in each chunk's Fourier spectrum are then binned using the mel scale [53], a scale based on the sensitivity of the human ear. The end result is a mel

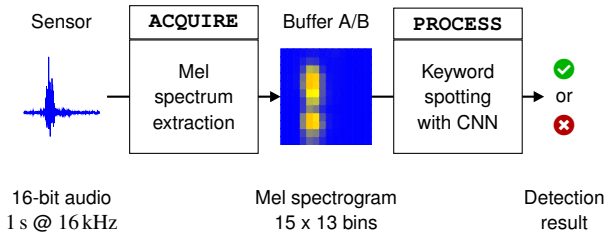


Figure 6: The keyword-spotting pipeline.

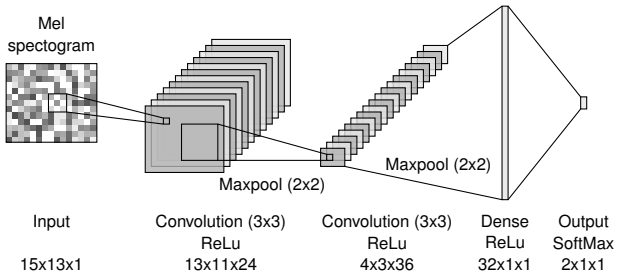


Figure 7: The CNN used for keyword spotting on the MCU.

spectrogram, showing how the human-perceived spectrum of the audio segment changes over time.

We implement the keyword spotting pipeline shown in Fig. 6. The input to the pipeline is 16-bit mono audio sampled at 16 kHz from a SPH0645LM4H microphone with digital Inter-IC Sound (I2S) output. The audio is processed in chunks of 1024 samples or 64 ms. These chunks are non-overlapping and a rectangular sampling window is used. Because the spectrum of each chunk is independent, each audio chunk must be processed only once, and the result can be appended to the previous mel spectrogram from which the oldest chunk is simultaneously dropped. From each chunk, a 13-coefficient mel spectrum is extracted. 15 such spectrums form the mel spectrogram that is used as input for the classifier network.

We use the convolutional neural network (CNN) shown in Fig. 7 to perform the keyword spotting task on the 15x13 mel spectrograms. The network contains two convolutional layers and one dense layer, totaling 10,454 trainable parameters.

The feature extraction is performed with the ARM CMSIS-DSP software library [29] using single precision floats. The neural network is executed in the STM X-CUBE-AI runtime [56], again using single precision floats.

7.2 KIMYA Integration

We implement the keyword-spotting pipeline with KIMYA as shown in Fig. 6. Mel-spectrum coefficients are calculated in the `ACQUIRE` phase and CNN inference is performed in the `PROCESS` phase.

In order to minimize the required number of KIMYA phase changes, we use the secure world to implement a *virtual sensor*. As the microphone is sampled at 16 kHz, and the MCU's I2S peripheral only has a 8-frame FIFO buffer, the event-detection code would have to read out this buffer at least every $\frac{8}{16\text{kHz}} = 500\,\mu\text{s}$. As each access of the I2S peripheral require a transition to the `ACQUIRE` phase, this would be inefficient. Instead, we permanently configure the I2S peripheral as a secure-world resource, and set up a secure-world DMA stream from the I2S FIFO buffer to an alternating 1024-frame (= 64 ms) memory buffer. This memory buffer is then treated as a virtual sensor, taking the place of the I2S peripheral in the Table 1 access map. Additionally, a `check_for_new_data()` API call is provided for the application to check if a new buffer frame is available.

The control flow of the keyword-spotting application follows the typical KIMYA flow illustrated in Fig. 5. If the application is in the `IDLE` phase, it checks if a new microphone frame is available. If so, it calls into the `ACQUIRE` phase, performs the feature extraction on the microphone data and stores the result in the alternating Buffer A/B. Performing this task in the `ACQUIRE` phase, ensures that it does not need to be repeated when the buffers are alternated and the Scratch memory is erased. Next, a call to the `PROCESS` phase is made. This call copies and orders the required data from the Buffers A/B to the Scratch memory to assemble the mel spectrogram for the most recent audio data and runs the neural network inference on it. If the keyword was detected, the MCU is transitioned to the `TRIGGERED` phase, and the main application thread starts streaming microphone data to a serial port. This simulates a voice assistant streaming microphone data to a cloud service for further processing.

The non-secure application must ensure that no interrupts that violate the KIMYA access map (Table 1) occur while the core is in the `ACQUIRE` or `PROCESS` phase. Therefore, non-secure interrupts are masked by setting the fault mask for the duration of the container calls. Further, all functions using the Buffer A, Buffer B, and Scratch memory regions must be aware that these memory regions can be erased between calls. To this end, a canary value is placed in each buffer. When this value reads as zero, the function knows the memory was emptied and re-initializes all necessary data structures.

Configuration values. We set $t_{\text{lifetime}} = 2\,\text{s}$ and $t_{\texttt{TRIGGERED}} = 5\,\text{s}$. Buffer A, Buffer B, and the Scratch memory are each 16 KiB. Two proof-of-concept notification mechanism are used simultaneously: a LED that is continuously lit when the `TRIGGERED` phase is active, and a LED that flashes upon entering the `TRIGGERED` phase or extending the $t_{\texttt{TRIGGERED}}$ deadline. We protect the notification mechanisms by limiting access to the GPIO pins that controls the LEDs to the secure world.

8 Evaluation

A basic evaluation of the on-MCU pipeline shows a precision of 100 % and a recall of 89 %. This evaluation was performed using 100 utterances of the keyword, and 100 utterances of

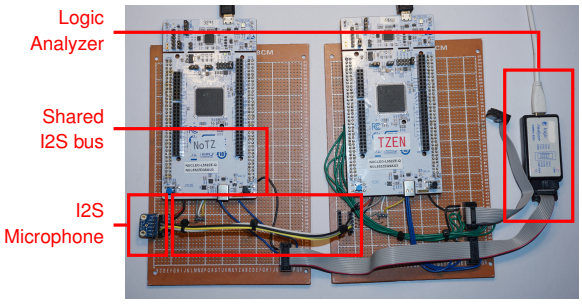


Figure 8: The evaluation setup. The reference is on the left and the KIMYA prototype on the right.

uniformly sampled other words in the dataset. Because evaluating the performance of keyword-spotting systems is a complex task with many variables, and because we consider it to be beyond the scope of our proof of concept, we did not perform a more detailed performance analysis of the pipeline. Instead, we focus on the performance differences between the pipeline running with and without KIMYA.

We use the setup shown in Fig. 8 to evaluate our KIMYA implementation. We use two identical STM NUCLEO-L552ZE-Q development boards. One board has TrustZone activated and is running the keyword-spotting pipeline inside a KIMYA container. The other board functions as a reference and has TrustZone disabled. The reference is thus running the entire keyword-spotting pipeline in the non-secure world, without KIMYA. Functionality that is needed for KIMYA compatibility (i.e., to reinitialize data structures or to copy and reorder data from the alternating Buffer A/B) is removed.

To facilitate an accurate comparison of the two boards, they are both connected to the same I2S microphone. The microphone is configured as an audio source, the MCUs as sinks. Only the KIMYA board generates a clock. This setup ensures that both boards receive identical microphone data.

Functional evaluation. As a preliminary evaluation, we verified that it is not possible to access microphone data without generating a notification: doing so results in a hard fault, stalling the MCU. We also logged the status of a button to the serial port, demonstrating that KIMYA does not prevent the transfer of data that did not originate in the KIMYA container.

8.1 Macro Benchmarks

We evaluate KIMYA using four macro benchmarks: output correlation with the reference implementation, pipeline latency, MCU duty cycle, and binary size.

8.1.1 Evaluation Setup

Each board updates the mel spectrogram and runs the neural network inference process each time a new chunk of 1024 microphone samples is available. This corresponds to one inference per 64 ms or 15.6 inferences per second. Each board exposes a `keyword_detected` signal on a GPIO pin which we sample at 16 MSamples/s using a logic analyzer. We stimulate the microphone using recordings of the keyword to generate 300 detection events per evaluation setting. This results in 600 measurable transitions of the `keyword_detected` signal. The boards also expose a signal indicating if the pipeline is running or if the MCU core is idle. We sample this signal to calculate the MCU duty cycle.

We run the macro benchmarks in three software settings. In the first setting, the application fully relies on the KIMYA gateway to perform buffer management. In the second setting, the application proactively makes calls to `maintain_buffers()` (see Section 5.2) to ensure that no buffer maintenance must be performed when running the pipeline. In the third setting, the application additionally proactively reinitializes the neural network after the Scratch memory was erased by executing a dedicated network initialization function in the `PROCESS` phase. All code was compiled using GCC and optimized for binary size (`-Osize`).

8.1.2 Results

Output correlation. When compensating for the additional delay introduced by KIMYA, both `keyword_detected` signals have a correlation coefficient of 1 in all settings. That is, they are identical. We confirm this in an additional experiment in which we allow both keyword-spotting pipelines to print their prediction and confidence score to a serial port. These values are also identical. This is expected and confirms that KIMYA does not introduce data loss and does not affect computational results.

Latency. Because voice commands do not have well-defined boundaries, the absolute latency of a keyword-spotting pipeline is ambiguous. Instead, we measure the relative latency of the KIMYA-enabled pipeline compared to the reference implementation. The results are shown in Fig. 9. We see that in the basic setting, a median 1.43 ms of additional delay is incurred because of the KIMYA containerization. In about 5 % of cases an additional 0.5 ms of delay is incurred on top of that. This corresponds to the pipeline runs where the gateway performs buffer maintenance before either call to the KIMYA container. In the setting where the application performs buffer maintenance proactively, this tail is not present. Finally, we see that when the application proactively reinitializes the neural network, the median delay is reduced to 1.19 ms. This improvement can be seen uniformly across all detection event runs, because when the application specifies a duration for a KIMYA container call, it must always assume a worst-case execution time. Thus, if the state of the neural network is unknown, the application must budget time for reinitialization during every `PROCESS` call.

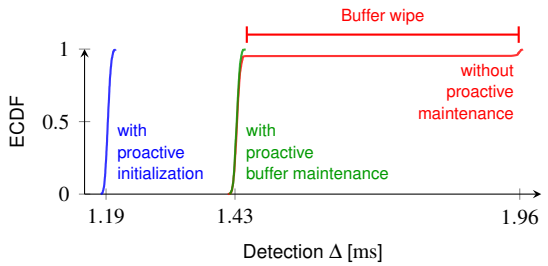


Figure 9: Detection delay of the KIMYA-enabled board compared to the reference implementation.

Table 2: Sizes of ROM and RAM sections of the benchmark binaries in KiB.

	Secure (Gw.)		**Non-secure** (App.)	
	ROM	RAM	ROM	RAM
With KIMYA	17.01	2.19	156.46	1.95
Reference	–	–	162.61	28.53

MCU duty cycle. The reference implementation runs at a duty cycle of 24.5 %. The KIMYA implementation without proactive buffer management has a duty cycle of 26.8 %. Proactively calling `maintain_buffers()` did not meaningfully change the duty cycle. However, when combined with proactive reinitializations, the duty cycle was slightly reduced to 26.5 %. This improvement stems from the fact that in the latter case, time to initialize the neural network must only be budgeted in the dedicated reinitialization calls, and not in all calls to the `PROCESS` phase.

Binary size. Table 2 shows the binary sizes of the KIMYA and reference implementations. There were no meaningful differences between the three KIMYA settings. We see that using KIMYA does not measurably increase the binary of the non-secure application. In fact, the reference binary is larger than the KIMYA-enabled application binary. We attribute the difference in the ROM sections to variations in compiler optimizations. The reference RAM region is larger because it includes statically allocated variables that are dynamically allocated in the Buffer A/B or Scratch regions in the KIMYA implementation. The small secure-world binary size shows that the KIMYA gateway has a small binary footprint.

8.2 Micro Benchmarks

To better understand the origin of the additional delay observed in the macro benchmarks, we perform micro benchmarks to measure the cost of individual KIMYA operations. These benchmarks use the proactive reinitialization setting.

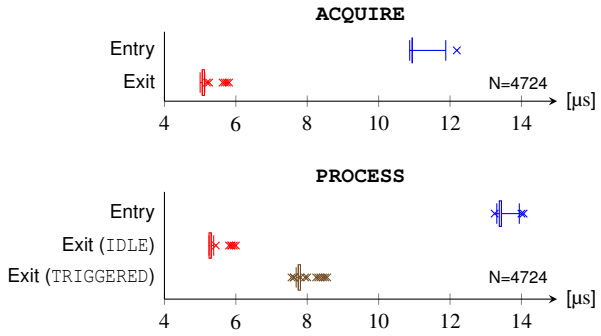


Figure 10: Boxplot showing the overhead of entering and exiting the KIMYA container. Exit timing excludes time spent waiting to reach the specified container execution time. Whiskers drawn at percentiles 1 and 99.

8.2.1 Container Operations

Container entry and exit. We instrument the code on the KIMYA-enabled board to write well-known values to a GPIO port at key points in the execution flow. This creates an 8-bit parallel signal that can be used to profile both the secure and non-secure worlds.

As shown in Fig. 10, entering the KIMYA container takes between 11 and 14 µs. Leaving the container to the `IDLE` phase takes around 5 µs. Due to the additional overhead required (i.e., buffer management, setting up the $t_{\texttt{TRIGGERED}}$ timer), leaving from `PROCESS` to the `TRIGGERED` phase takes around 8 µs. These times exclude any time spent waiting to reach the specified container execution time (see Section 5.4).

Figure 11 decomposes the operations shown in Fig. 10 into three suboperations: (i) switching from the non-secure world to the secure gateway, (ii) executing the gateway logic, and (iii) returning from the gateway to the non-secure world. We see a base cost of 1 to 2 µs to switch between the secure and non-secure worlds. KIMYA-specific logic (i.e., MPU configuration, buffer checks, timer management) adds around 10 µs per container call. Results for the `PROCESS` phase are similar and therefore not shown.

Buffer management. The timing values shown in Figs. 10 and 11 include buffer management logic, but because the application performs proactive buffer maintenance, it does not include buffer erasure overhead. We separately measure buffer erasure to take 261 µs per 16 KiB memory region, resulting in an average 0.06 % core load for buffer erasure.

8.2.2 Comparison to Macro Benchmarks

Summing up the median container entry and exit times with a median 33 µs spent waiting for the specified container execution time to be reached (not shown in Fig. 10), results in an overhead of 70 µs per pipeline run. This number is significantly lower than the 1.19 ms measured during the macro

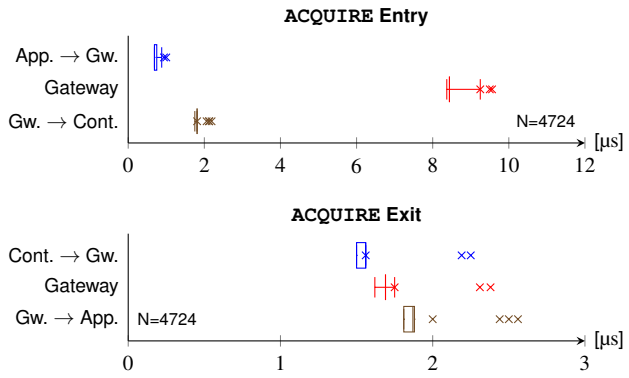


Figure 11: Boxplot showing KIMYA gateway overhead. Broken down into time required to enter the secure-world gateway, time spent in the gateway, and time required to return to the non-secure world. Gateway time excludes time spent waiting to reach the specified container execution time. Whiskers drawn at percentiles 1 and 99.

benchmarking. To understand the origin of this discrepancy, we perform further benchmarking on the application and container code. We observe that the inference call to the (precompiled) neural network library takes 15.50 ms in the KIMYA container instead of 14.46 ms in the reference implementation. This difference of 1.04 ms makes up the majority of the 1.12 ms of unapportioned overhead. We attribute this performance gap to the different memory access pattern in the KIMYA implementation, leading to more bus contention.

9 Discussion

9.1 Limitations

Data lifetime. Although KIMYA provides strong guarantees that no sensor data can be stored beyond t_{lifetime}, it requires t_{lifetime} to be configured at twice the actual useful data lifetime. For example, our prototype implementation uses $t_{\text{lifetime}} = 2\,\text{s}$, although only 1 s of data is fed into the CNN.

This is a direct consequence of using two buffers to form the alternating buffer A/B. Extending the alternating buffer to a ring buffer composed of n buffer elements, would allow t_{lifetime} to be reduced to $1 + \frac{1}{n-1}$ times the true data lifetime. Limiting factors are (i) the size of the chunks in which data is preprocessed, and (ii) the requirement to wipe the entire scratch region each time an element of the ring buffer is wiped.

Imperfect auditing. KIMYA relies on device users to perform notification auditing. Past work has shown that the design of notification mechanisms is critical to ensure the audit quality [46]. However, even with an optimal notification mechanism it is possible that a user would miss some false-trigger events. When designing a notification mechanism, it is important to ensure that systematic misbehavior will eventually be noticed by the user. If this is not the case, an adversary could simply keep the KIMYA device in the `TRIGGERED` phase at all times. Care must also be taken to ensure that other factors (e.g., the location of the device) do not hinder the notification mechanism (e.g., placing a device with a light-based notification mechanism in a closed closet).

Instead of generating a false-trigger event, an adversary could also artificially extend the the time during which the device stays in the `TRIGGERED` phase after detecting a true-trigger event. If kept within limits, we expect users to be more likely to attribute such behavior to non-malicious technical limitations. As an example, we consider an adversary that extends each trigger by 10 s. Assuming 18 smart-speaker interactions per day [5], this would allow an adversary to maliciously capture up to 3 min of superfluous audio a day without raising suspicion.

Alternative event detection. KIMYA does not place restrictions on which type of events the application code can detect. Therefore, the KIMYA mechanism by itself does not provide any privacy guarantees. Instead, the core KIMYA mechanism needs to be used together with suitable notification and auditing mechanisms.

For example, consider the case where an adversary writes code that detects specific information (e.g., by triggering on the keyword "password"). The attacker could then potentially export this data by generating only a small amount of low-frequency false-positive notifications. Depending on the notification and auditing mechanisms used, these notifications may go unnoticed.

The use of software attestation mechanisms to limit which code can be executing inside the KIMYA container could mitigate this attack, although at the cost of reduced system agility. Alternatively, the quality of the notifications and notification auditing could be improved, for example, through the use of privacy assistants [12, 26].

Multi-stage pipelines. For efficiency and accuracy reasons, event-detection pipelines often use multiple stages with decreasing false-positive rates and lazy evaluation [21, 25]. In such cases, our KIMYA implementation requires container execution time for all pipeline stages to be budgeted during each call. Modifications that allow the container execution time to be dependent on the result of the pipeline stages could be made, but would create a (small) information channel out of the container. Logging when the container execution time was extended could deter adversaries from abusing this channel. Alternatively, this channel can be eliminated by decoupling the KIMYA execution time from that of the main application, for example, by executing KIMYA on a dedicated core.

Some vendors offer cloud-based wake word verification services, intended to be used as the last stage of a keyword-spotting pipeline [23]. KIMYA requires a notification to be generated each time data is sent to such a service. If the on-device pipeline stages have a high false-positive rate, this can

dilute the value of KIMYA notifications. However, there is an ongoing trend to move voice assistant functionality from the cloud to end-user devices [8,22]. Therefore, we anticipate that cloud-based wake-word detection will disappear over time.

Reduced scheduling flexibility. Calls to the `ACQUIRE` and `PROCESS` phases are fixed-duration and non-preemtable. This results in a reduced scheduling flexibility, and in certain cases, might lead to a performance degradation. However, as embedded systems are designed assuming worst-case execution times, a KIMYA-enabled system should be able to meet the same deadlines as an equivalent non-KIMYA system.

TCB bloating. In our KIMYA implementation, all code running in the secure world has the same security level, and is thus able to affect KIMYA's properties. Therefore, all secure world code is part of KIMYA's TCB. If additional functionalities are implemented using TrustZone, this has the potential to bloat the TCB size. A secure-world OS could be used to limit the access permissions of secure-world code. TCB bloat affects all platforms of which the hardware provides a single-world secure environment.

9.2 Multiple Sensors

This work focuses on regulating access to a single event-triggered sensor. However, many devices have multiple sensors. In such cases the following KIMYA deployment models are possible:

Independent. The access permissions for each sensor are considered individually. Multiple sensors can be protected by creating multiple, independent, KIMYA containers.

Cross-sensor. The permissions for multiple sensors can be linked together, enabling opportunities for cross-sensor activations. Consider, for example, a smart display with both a camera and a microphone. In this setting, KIMYA could be configured to link access permissions for the camera to events detected on the microphone. This way, KIMYA can ensure that camera data cannot be accessed unless the user speaks an *activation phrase*, e.g., "Hey Kimya, turn on the camera".

Virtual sensors. The trigger output of a KIMYA enabled sensor can be used as a virtual sensor input for sensor access management or attack detection systems (e.g., 6thSense [51]).

9.3 Deployability

MCU requirements. Implementing KIMYA requires a mechanism that allows protected gateway code to restrict the resource access of application code. Our implementation on ARM Cortex-M uses a combination of TrustZone and the MPU for this purpose. Many modern, embedded, architectures provide similarly suitable mechanisms. These include: (i) TrustZone on ARM Cortex-A [45] in combination with a MPU or memory management unit (MMU); (ii) Physical Memory Protection (PMP) and machine mode on RISC-V [62]; and (iii) LX secure mode (XLS) on Xtensa LX [10] in combination with an MPU or MMU. In the absence of suitable hardware security extensions, KIMYA could be implemented as an OS service, although this would result in increased TCB size and significantly reduced robustness.

Additionally, KIMYA requires one timer, and a mechanism to ensure that control is periodically returned to the gateway. The latter can be established using interrupts generated by the timer, or through a gateway-controlled watchdog timer.

Hardware and software design. For KIMYA to be effective, the hardware design of KIMYA enabled products must ensure that the KIMYA gateway can restrict the application access to the event-triggered sensor. We expect this to be the case in most existing designs. Designers must implement base KIMYA functionality for their target platform analog to KIMYA for Cortex-M as described in Section 5. Additionally, a covert-channel analysis must be performed and countermeasures similar to those in Section 5.4 must be implemented.

To dimension the size of the Buffer A/B and Scratch memory regions, one must consider the amount of intermediary state that the `ACQUIRE` process will generate, and the required amount of temporary state used for signal processing. In our demo application, Buffer A and B each require 11 KiB of RAM. Of those 11 KiB, around 800 B is used to store the mel spectrograms, the remainder is used as temporary storage for signal processing. Similarly, our demo application requires a 8 KiB Scratch region, all of which is used as temporary storage for the CNN. If original sensor data is to be stored for reprocessing after a trigger event, sufficient space must be allocated for it in the Buffer A/B regions. For our demo application, this would correspond to 32 KiB per buffer.

Finally, a notification mechanism must be designed and implemented as discussed in Section 4.5. If cryptographically protected notifications are used, a key-establishment mechanism must also be implemented.

In-field retrofitting. If an existing hardware designs make it possible to isolate the sensor that is to be protected, KIMYA can—in principle—be retrofitted to these devices using a vendor-issued software update. However, depending on the existing product configuration (i.e., TrustZone setup, MCU fuses, and exposed interfaces), the complexity of the update process can vary greatly. Some devices could be retrofitted using an over-the-air software update, others might require physical contact, and yet others could be impossible to retrofit.

9.3.1 Compatibility with Popular Voice Assistants

We explore the compatibility of three popular devices with KIMYA: (i) the Amazon Echo Dot (3rd generation), (ii) the Google Nest Home Mini, and (iii) the Apple HomePod. We base this analysis on publicly available data.

Amazon Echo Dot 3rd generation. This device has four microphones, connected to two ADCs. The ADCs are con-

nected to the CPU using both an I2C bus (for configuration) and an I2S bus (for audio data) [13]. The ADCs are Texas Instruments TLV320ADC3101 chips. The CPU is a Mediatek MT8516. Given that the MT8516 supports TrustZone for Cortex-A [17], implementing KIMYA should be possible. However, special care must be taken because the ADCs feature a *miniDSP* [57] which has access to the microphone stream and has (limited) storage capabilities. Hence, the miniDSP could be abused to break KIMYA isolation. Further research to determine the maximum storage duration on the ADCs, or to determine if the memory on the ADCs can efficiently be flushed would be needed. Alternatively, it could be ensured that only signed code can be loaded onto the ADC. Given that the Echo Dot already uses trusted boot [17], and that we expect the ADC configuration to be static, this is a plausible strategy.

Google Nest Home Mini. Not much information is available about this device, only that it runs on a Synaptics AS370 SoC (ARM Cortex-A52) [58]. No explicit information about TrustZone support can be found, but we find it likely for it to be available. The device has 3 on-board microphones [9], but it is unclear how the audio signal is digitized. Based on the presence of a recent Cortex-A based CPU, we expect that KIMYA support would be possible on this device. Care must be taken to ensure that the SoC and ADC architectures do not break KIMYA isolation.

Apple HomePod. Apple's HomePod speaker runs on a custom Apple-designed APL1011 SoC [63]. The HomePod uses seven MEMS microphones which are digitized by a Conexant CX20810 ADC [63]. This ADC does not have an on-board DSP, and is therefore unproblematic. However, we were unable to find any documentation indicating that the APL1011 silicon features the hardware security features required to support KIMYA.

10 Related Work

There are a number of proposals to limit access to sensor data. We discuss the most relevant areas of research below.

Control of sensor access. SeCloak [28] uses TrustZone on Cortex-A to allow mobile device users to disable access to specific peripherals. Brasser et al. [6] propose a similar mechanism, but allow a third party to control access restrictions instead of the device owner. AWare [43] is targeted towards mobile devices, and provides an operating-level service that binds user interactions with specific user interface elements to sensor access rights. EnTrust [44] further generalizes this to other types of input events, and to cooperating applications.

Although they provide strong guarantees, none of these works is designed to capture the semantics of always-standby sensors. Therefore, sensor access must be permanently granted for event detection to work. Moreover, these works were not designed for constrained environments.

Work on trusted I/O paths (e.g., SGXIO [64] and Wimpy kernels [68]) can be used to provide secure I/O access to trusted execution environments (TEEs). Although TEEs provide isolation, they do no provide amnesia. It might be possible to implement KIMYA-like logic using multiple TEEs (e.g., multiple SGX enclaves) and message passing channels, but future work would be needed to confirm this.

Auditing of sensor access. Viola [41] provides guarantees that sensor notifications (e.g., LED indicators) are active when (and only when) a sensor is accessed. Ditio [40] securely logs sensor access for later auditing. Neither of these mechanisms is designed for always-standby sensors, and will mark an always-standby sensor as being continuously accessed. 6thSense [51] analyses sensor access patterns to detect malicious activity. However, it does not support always-standby sensors. Depending on the concrete sensor type, 6thSense would mark an always-standby sensor as always accessed, or assign it an access state that is independent from the event detection. KIMYA can be used to augment 6thSense as discussed in Section 9.2.

Trigger limitations. Mhaidli et al. [39] propose to use non-verbal communication cues, such as gaze and volume level as a pre-trigger for voice assistants, however they do not consider adversarial settings. Our work is orthogonal to their proposal. In fact, in Section 9 we discuss how their work, and more generally, *cross-sensor triggers*, can be included in KIMYA. EKOS [1] uses collaborative keyword-spotting to limit (adversarial) false positives. Contrary to KIMYA, EKOS requires keywords to be heard by multiple devices and does not consider on-device adversaries.

Information flow tracking. There is a large body of work on information-flow or *taint* tracking for mobile devices [3, 14, 18, 20]. FlowFence [16] provides taint tracking for Internet of Things (IoT) cloud platforms. These works rely either on static or dynamic code analysis. In the former case, they must be combined with software attestation or similar mechanisms. In the latter case, they are challenging to apply to constrained environments. Moreover, these mechanisms were not designed to support always-standby semantics, meaning that they do not provide guarantees on which historic data can be accessed when a trigger event occurs.

Skill behavior. A number of works [19, 50, 66] analyze the behavior of third-party voice assistant *skills*. This is comparable to analyzing the behavior of Android or smart-home apps, and focuses on individual functionality add-ons, rather than on the underlying system. Therefore, we believe these works to be synergetic to ours.

11 Conclusion

Despite their high popularity, voice assistants continue to prompt significant privacy concerns. These concerns often

focus on the always-standby nature of such assistants, and a perceived lack of transparency in how they operate.

KIMYA demonstrates that it is possible to unify the functionality of always-standby sensors with strong, low-level, guarantees on privacy. Moreover, our implementation for Cortex-M demonstrates that KIMYA introduces low overhead and is applicable to constrained environments.

Although we have implemented KIMYA for Cortex-M, its design is not platform specific, and can be implemented on other architectures (e.g., Cortex-A, RISC-V, Xtensa LX, ...) as well. Moreover, as KIMYA does not require hardware modifications, it can be retrofitted to existing systems. This reduces time to market, and makes it possible to bring significant privacy enhancements to millions of devices already deployed in people's homes.

References

[1] Shimaa Ahmed, Ilia Shumailov, Nicolas Papernot, and Kassem Fawaz. Towards more robust keyword spotting for voice assistants. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[2] Amazon.com. Alexa Voice Service (AVS) security requirements. https://developer.amazon.com/en-US/docs/alexa/alexa-voice-service/avs-security-reqs.html, January 2022.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android appls. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2014.

[4] BBC News. Smart speaker recordings reviewed by humans. https://www.bbc.com/news/technology-47893082, April 2019.

[5] Frank Bentley, Chris Luvogt, Max Silverman, Rushani Wirasinghe, Brooke White, and Danielle Lottridge. Understanding the long-term use of smart speaker assistants. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2(3), sep 2018.

[6] Ferdinand Brasser, Daeyoung Kim, Christopher Liebchen, Vinod Ganapathy, Liviu Iftode, and Ahmad-Reza Sadeghi. Regulating ARM TrustZone devices in restricted spaces. In *ACM International Conference on Mobile Systems, Applications, and Services*, June 2016.

[7] Fabian Bräunlein and Luise Frerichs. Smart spies: Alexa and Google Home expose users to vishing and eavesdropping. https://www.srlabs.de/bites/smart-spies, October 2019.

[8] Manuel Bronstein. Bringing you the next-generation Google assistant. https://blog.google/products/assistant/next-generation-google-assistant-io/, May 2019.

[9] Bureau Veritas. Construction photos of EUT H2C. https://fcc.report/FCC-ID/A4R-H2C/4401997, August 2019.

[10] Cadence Design Systems, Inc. Xtensa Instruction Set Architecture (ISA) summary. https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/ip/tensilica-ip/isa-summary.pdf, April 2022.

[11] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432, 2020.

[12] Anupam Das, Martin Degeling, Daniel Smullen, and Norman Sadeh. Personalized privacy assistants for the internet of things: Providing users with notice and choice. *IEEE Pervasive Computing*, 17(3), July 2018.

[13] Brian Dorey. Echo dot 3rd gen digging deeper. https://www.briandorey.com/post/echo-dot-3rd-gen-digging-deeper, August 2019.

[14] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 32(2), June 2014.

[15] Haytham M. Fayek. Speech processing for machine learning: Filter banks, mel-frequency cepstral coefficients (mfccs) and what's in-between, 2016.

[16] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. FlowFence: Practical data protection for emerging IoT application frameworks. In *USENIX Security Symposium*, August 2016.

[17] Dennis Giese and Guevara Noubir. Amazon echo dot or the reverberating secrets of IoT devices. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. ACM, jun 2021.

[18] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of android applications in DroidSafe. In *ISOC Network and Distributed System Security Symposium*, 2015.

[19] Zhixiu Guo, Zijin Lin, Pan Li, and Kai Chen. SkillExplorer: Understanding the behavior of skills in large scale. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, August 2020.

[20] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for. In *ACM Conference on Computer and communications security*, 2011.

[21] Apple Inc. Hey Siri: An on-device DNN-powered voice trigger for Apple's personal assistant. https://machinelearning.apple.com/research/hey-siri, October 2017.

[22] Apple Inc. Apple advances its privacy leadership with iOS 15, iPadOS 15, macOS Monterey, and watchOS 8. https://www.apple.com/newsroom/2021/06/apple-advances-its-privacy-leadership-with-ios-15-ipados-15-macos-monterey-and-watchos-8/, June 2021.

[23] Ted Karczewski. Cloud-based wake word verification improves "Alexa" wake word accuracy on your AVS products. https://developer.amazon.com/blogs/alexa/post/b136b3e7-0ba8-4589-aaf9-2a037fc4e9c9/cloud-based-wake-word-verification-improves-alexa-wake-word-accuracy-on-your-avs-products, May 2017.

[24] Deepak Kumar, Riccardo Paccagnella, Paul Murley, Eric Hennenfent, Joshua Mason, Adam Bates, and Michael Bailey. Skill squatting attacks on Amazon Alexa. In *USENIX Security Symposium*, August 2018.

[25] Rajath Kumar, Mike Rodehorst, Joe Wang, Jiacheng Gu, and Brian Kulis. Building a Robust Word-Level Wakeword Verification Network. In *Interspeech 2020*, 2020.

[26] Marc Langheinrich. A privacy awareness system for ubiquitous computing environments. In *ACM Conference on Ubiquitous Computing*. 2002.

[27] Josephine Lau, Benjamin Zimmerman, and Florian Schaub. Alexa, are you listening? privacy perceptions, concerns and privacy-seeking behavious with smart speakers. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW), November 2018.

[28] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. SeCloak: ARM trustzone-based mobile peripheral control. In *ACM International Conference on Mobile Systems, Applications, and Services*, June 2018.

[29] ARM Ltd. CMSIS DSP software library. https://github.com/ARM-software/CMSIS_5/tree/develop/CMSIS/DSP.

[30] ARM Ltd. Introduction to the ARMv8-M architecture. https://developer.arm.com/documentation/100688/0200, February 2017. Version 1.0.

[31] ARM Ltd. ARM v8-M security extensions: Requirements on development tools. https://developer.arm.com/documentation/ecm0359818/11/, November 2019. Version 1.1.

[32] ARM Ltd. ARM Cortex-M33 devices generic user guide. https://developer.arm.com/documentation/100235/0100, June 2020. Version r1p0.

[33] ARM Ltd. Armv8-M architecture reference manual. https://developer.arm.com/documentation/ddi0553/br, December 2021. Version B.r.

[34] Lan Luo, Yue Zhang, Clayton White, Brandon Keating, Bryan Pearson, Xinhui Shao, Zhen Ling, Haofei Yu, Cliff Zou, and Xinwen Fu. On security of trustzone-m-based iot systems. *IEEE Internet of Things Journal*, 9(12):9683–9699, 2022.

[35] Dorian Lynskey. 'Alexa, are you invading my privacy?' – the dark side of our voice assistants. https://www.theguardian.com/technology/2019/oct/09/alexa-are-you-invading-my-privacy-the-dark-side-of-our-voice-assistants, October 2019.

[36] Zongheng Ma, Saeed Mirzamohammadi, and Ardalan Amiri Sani. Understanding sensor notifications on mobile devices. In *ACM International Workshop on Mobile Computing Systems and Applications*, February 2017.

[37] Lydia Manikonda, Aditya Deotale, and Subbarao Kambhampati. What's up with privacy? user preferences and privacy concerns in intelligent personal assistants. In *AAAI/ACM Conference on AI, Ethics, and Society*, December 2018.

[38] Nicole Meng, Dilara Keküllüoğlu, and Kami Vaniea. Owning and sharing: Privacy perceptions of smart speaker users. *Proceedings of the ACM on Human-Computer Interaction*, 5(CSCW1), April 2021.

[39] Abraham Mhaidli, Manikandan Kandadai Venkatesh, Yixin Zou, and Florian Schaub. Listen only when spoken to: Interpersonal communication cues as smart speaker privacy controls. *Proceedings on Privacy Enhancing Technologies*, 2020(2), April 2020.

[40] Saeed Mirzamohammadi, Justin A. Chen, Ardalan Amiri Sani, Sharad Mehrotra, and Gene Tsudik. Ditio: Trustworthy auditing of sensor activities in mobile & iot devices. In *ACM Conference on Embedded Network Sensor Systems*, November 2017.

[41] Saeed Mirzamohammadi and Ardalan Amiri Sani. Viola: Trustworthy sensor notifications for enhanced privacy on mobile systems. *IEEE Transactions on Mobile Computing*, 17(11), November 2018.

[42] George Orwell. *Nineteen Eighty-Four*. Secker and Warburg, 1949.

[43] Giuseppe Petracca, Ahmad-Atamli Reineh, Yuqiong Sun, Jens Grossklags, and Trent Jaeger. AWare: Preventing abuse of Privacy-Sensitive sensors via operation bindings. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 379–396, Vancouver, BC, August 2017. USENIX Association.

[44] Giuseppe Petracca, Yuqiong Sun, Ahmad-Atamli Reineh, Patrick McDaniel, Jens Grossklags, and Trent Jaeger. EnTrust: Regulating sensor access by cooperating programs via delegation graphs. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 567–584, Santa Clara, CA, August 2019. USENIX Association.

[45] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Comput. Surv.*, 51(6), jan 2019.

[46] Rebecca S. Portnoff, Linda N. Lee, Serge Egelman, Pratyush Mishra, Derek Leung, and David Wagner. Somebody's watching me?: Assessing the effectiveness of webcam indicator lights. In *ACM Conference on Human Factors in Computing Systems*, April 2015.

[47] Qualcomm Technologies, Inc. Snapdragon 8 Gen 1 mobile platform. https://www.qualcomm.com/products/snapdragon-8-gen-1-mobile-platform, December 2021.

[48] Florian Schaub, Rebecca Balebako, and Lorrie Faith Cranor. Designing effective privacy notices and controls. *IEEE Internet Computing*, 2017.

[49] Florian Schaub, Rebecca Balebako, Adam L. Durity, and Lorrie Faith Cranor. A design space for effective privacy notices. In *USENIX Symposium On Usable Privacy and Security*, July 2015.

[50] Faysal Hossain Shezan, Hang Hu, Gang Wang, and Yuan Tian. VerHealth. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(4), dec 2020.

[51] Amit Kumar Sikder, Hidayet Aksu, and A. Selcuk Uluagac. A context-aware framework for detecting sensor-based threats on smart devices. *IEEE Transactions on Mobile Computing*, 19(2):245–261, 2020.

[52] Amit Kumar Sikder, Giuseppe Petracca, Hidayet Aksu, Trent Jaeger, and A. Selcuk Uluagac. A survey on sensor-based threats and attacks to smart devices and applications. *IEEE Communications Surveys & Tutorials*, 23(2):1125–1159, 2021.

[53] S. S. Stevens, J. Volkmann, and E. B. Newman. A scale for the measurement of the psychological magnitude pitch. *The Journal of the Acoustical Society of America*, 8(3), January 1937.

[54] STMicroelectronics. *R*, October 2020. DS12737 Rev 6.

[55] STMicroelectronics. *Reference manual: STM32L552xx and STM32L562xx advanced ARM-based 32-bit MCUs*, May 2020. RM0438 Rev 6.

[56] STMicroelectronics. DB3788: X-CUBE-AI databrief. https://www.st.com/en/embedded-software/x-cube-ai.html, September 2021.

[57] Texas Instruments. TLV320ADC3101 low-power stereo ADC with embedded miniDSP for wireless handsets and portable audio. Datasheet, August 2015.

[58] Various Authors. Google nest mini (H2C). https://wikidevi.wi-cat.ru/Google_Nest_Mini_(H2C), March 2022.

[59] Voicebot Research. U.S. smart speaker consumer adoption report 2021 (executive summary). https://research.voicebot.ai/report-list/smart-speaker-consumer-adoption-report-2021/, January 2021.

[60] Anran Wang, Dan Nguyen, Arun R. Sridhar, and Shyamnath Gollakota. Using smart speakers to contactlessly monitor heart rhythms. *Communications Biology*, 4(1), March 2021.

[61] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. April 2018.

[62] Andrew Waterman, Krste Asanović, and Joan Hauser. The risc-v instruction set manual: Volume ii: Privileged architecture. techreport, 2021.

[63] Stacy Wegner, Daniel Yang, and Kim Waterman. Apple homepod teardown and cost comparison. https://www.techinsights.com/blog/apple-homepod-teardown-and-cost-comparison, February 2018.

[64] Samuel Weiser and Mario Werner. SGXIO: Generic trusted i/o path for intel sgx. In *ACM Conference on Data and Application Security and Privacy*, March 2017.

[65] Joseph Yiu. White paper: Introduction to the ARM Cortex-M55 processor. Technical report, ARM Ltd., February 2020.

[66] Jeffrey Young, Song Liao, Long Cheng, Hongxin Hu, and Huixing Deng. SkillDetective: Automated Policy-Violation detection of voice assistant applications in the wild. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[67] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. Hello edge: Keyword spotting on microcontrollers. November 2017.

[68] Zongwei Zhou, Miao Yu, and Virgil D. Gligor. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *2014 IEEE Symposium on Security and Privacy*, pages 308–323, 2014.