SafePath: Encryption-less On-demand Input Path Protection for Mobile Devices

Xin Zhang and Yifan Zhang

Binghamton University
Binghamton, NY 13902, USA

Abstract. Sensitive data from input devices such as touchscreens and microphones is vulnerable to attackers with OS-level privileges. While prior work has primarily focused on protecting data after it reaches applications, the problem of securing the data path from the device to the application remains underexplored. Existing solutions either use encryption, which incurs significant performance overhead and complexity in key management, or bypass the OS entirely, which reduces compatibility and requires substantial changes to the application.

We present SafePath, a system that secures input data paths without encryption or OS bypass. SafePath leverages ARM memory virtualization to implement a minimal memory hypervisor that intercepts device input data at its entry point into the system, stores it securely, and transmits only lightweight data indices to applications. This design isolates sensitive data from the OS while preserving the native input path structure and requiring minimal modifications to applications. We prototype SafePath on ARM hardware and support six different input devices. Evaluation shows that SafePath blocks OS-level access to input data while preserving system performance. For example, under high system load, it reduces audio frame loss by over 45% compared to encryption-based approaches and maintains over 98% of native throughput across all tested devices. These results demonstrate that SafePath offers strong protection with low overhead and practical deployability.

1 Introduction

Mobile devices increasingly serve as the gateway to sensitive personal services, such as secure messaging, mobile banking, and biometric authentication. These services depend on input from hardware peripherals, such as touchscreens, microphones, fingerprint sensors, and cameras, that often carry highly sensitive data. Unfortunately, this input data is typically exposed to the operating system (OS) before reaching its destination application. On platforms like Android, input data traverses multiple layers and memory buffers before reaching applications, creating a wide attack surface for adversaries with OS-level privileges. As a result, attackers with kernel-level access can intercept or tamper with raw input data in transit, undermining the confidentiality and integrity of the entire interaction. For instance, touch coordinates from a touchscreen could allow an attacker to infer passwords [5, 29, 54], while audio data from a microphone could expose private conversations [37, 41, 48]. Even seemingly innocuous sensor readings, such as those from accelerometers, can be exploited to extract sensitive information like voice data [2, 7, 33, 53] or infer touchscreen taps [13, 34, 36, 50].

Existing efforts to protect input data typically fall into two categories. The first category assumes that a trusted input path already exists and focuses on shielding application memory after the data is received. Most existing works fall into this category. A smaller body of work belongs to the second category, which explicitly secures an input path between the data source and the data recipient. They achieve their goals by either *encrypting input data end-to-end* or *bypassing the untrusted OS*. However, both directions suffer from significant limitations: *Encryption-based approaches* introduce substantial performance overhead, which can be prohibitive for latency-sensitive tasks such as voice input or gesture recognition. They also expand the trusted computing base (TCB) to include complex cryptographic modules, which are themselves vulnerable to side-channel leakage. *OS-bypassing solutions*, on the other hand, often require invasive changes to applications and drivers, limit compatibility with standard input processing stacks, and prevent benign in-OS processing operations from being used. A more detailed review of prior work is provided in Section 2.

Securing the input path in a way that avoids both encryption and OS bypassing presents several challenges. *First*, the system must prevent a compromised OS from accessing sensitive input data, even though it controls the driver stack and shared memory. *Second*, it must preserve compatibility with the existing input flow, allowing input to pass through standard OS components, so that user applications and in-OS processing can function normally. *Third*, the solution should avoid the overhead and complexity of cryptographic solutions, which are burdensome on resource-constrained devices. *Finally*, the solution should allow applications to selectively enable protection for individual input sessions, without introducing significant development effort or performance cost.

We present SafePath, a system that enables secure input data paths on ARM-based mobile platforms without relying on encryption or bypassing the OS. SafePath uses ARM's hardware memory virtualization support [3, 17, 20] to construct a minimal hypervisor that intercepts and securely stores input data as it first enters system memory, typically in the driver's initial software-visible buffer. Rather than passing the raw input through the OS, SafePath replaces it with opaque indexes, which propagate through the system's existing input path. Applications later use a controlled hypercall interface to retrieve the original data from the hypervisor, ensuring that only the intended recipient can access the protected input. This index-based indirection model preserves the original data flow and allows in-OS input processing to continue as normal, while preventing even a fully compromised OS from accessing the underlying input data. SafePath supports on-demand protection that can be enabled per device or per session, and it minimizes trusted code by confining all sensitive logic to a small hypervisor with a narrow interface.

It is important to note that SafePath is not intended as a complete end-to-end solution for securing input data throughout its lifecycle. Rather, it complements existing mechanisms, such as trusted execution environments (TEEs) (e.g., Intel SGX [30], ARM TrustZone [42]), that can protect input after it reaches the application. This modular approach allows SafePath to focus on a specific but under-addressed segment of the input data flow: from the device hardware up to the application boundary.

We have implemented a prototype of SafePath on an ODROID-XU4 development board [35] running Android 4.4.4, supporting six types of input devices: touchscreen, microphone, camera, GPS receiver, fingerprint scanner, and

accelerometer. Our evaluation shows that SafePath prevents input data leakage in the presence of a fully compromised OS, while incurring negligible performance overhead and requiring minimal changes to applications. Compared to encryption-based approaches, SafePath performs significantly better under high I/O load, and it supports input scenarios, such as OS-mediated processing, that are infeasible for OS-bypassing systems.

In summary, we the following contributions:

- We identify limitations in prior approaches to securing device input paths and motivate the need for a non-encryption, OS-compatible solution.
- We design SafePath, a hypervisor-based system that securely intercepts and delivers input data without disrupting the original input path or requiring cryptographic protection.
- We prototype SafePath on real hardware and demonstrate its effectiveness across diverse input types, with low performance overhead and minimal developer effort.

2 Background and Related Work

Background. Securing device input paths against OS-level attackers is difficult due to the following two primary reasons.

(1) *System and hardware vulnerabilities*: OS-level attackers can exploit numerous classes of vulnerabilities to extract sensitive input data. For example, memory disclosure bugs allow unauthorized access to process and kernel memory [21,22,24,43,46]. Transient-execution attacks such as Spectre, Meltdown, and their variants can leak sensitive information by exploiting speculative behavior in modern processors [9, 11, 14, 31, 32, 40]. In addition, kernel-level malware such as rootkits can gain full access to kernel memory, including buffers that temporarily store raw input [12, 18, 27, 44, 45].

(2) *Wide attack surface in input data paths*: Our analysis of Android source code reveals that input devices have extensive attack surfaces along their input data paths. An *input path* refers to the sequence of memory locations (buffers) through which input device data travels before reaching the application. In Android, these buffers typically span three layers: the device driver layer, the hardware abstraction layer (HAL), and the Android framework layer. Table 1 illustrates the number of "travel stops" in the input paths of various devices. For example, the touchscreen input path has 14 travel stops, each representing a potential vulnerability point where attackers can intercept and misuse sensitive data.

Table 1: Number of "travel stops" in device input path.

	DD	HAL	AF	Total
Touchscreen	8	0	6	14
Microphone	1	1	3	5
Camera	1	3	1	5
Accelerometer	7	3	5	15
GPS receiver	2	4	2	8

DD: Device driver
HAL: Hardware Abstraction Layer
AF: Android Framework

Related work. Existing solutions fall into two categories depending on when and how they provide input protection. The *first category* assumes the input path is already trusted and focuses on protecting application memory after the input arrives. The *second category* protects the input path itself, ensuring that data can reach applications securely even in the presence of an untrusted OS.

Most prior work belongs to the first category above [1, 4, 6, 8, 10, 15, 16, 23, 25, 26, 28, 38, 47, 51, 52]. In contrast, only a few systems attempt to secure the path

between device and application directly [19,49,55,56]. These systems employ one of two main design philosophies: *OS-bypassing* and *encryption-based methods*.

OS-bypassing approaches establish new trusted channels that route input data around the OS entirely. For example, they may configure hardware devices to deliver data directly to a trusted VM or enclave, bypassing all software components in the untrusted OS [19, 55, 56]. While effective in principle, these systems suffer from poor compatibility and limited flexibility. They require non-trivial changes to applications and drivers and are unsuitable for use cases where input must be pre-processed by OS components, for instance, gesture detection in touchscreens or audio buffering in HALs.

Encryption-based approaches, on the other hand, protect data by encrypting it at the point of origin and decrypting it only at a trusted endpoint [19, 49]. While this allows the reuse of the original data path, it introduces two serious drawbacks. *First*, cryptographic key distribution and protection are non-trivial, especially in adversarial environments. As demonstrated in past work, attackers can extract encryption keys stored in memory via transient execution attacks, even in trusted computing environments like SGX [39]. *Second*, encryption and decryption introduce significant performance overhead, which grows with input volume. In our experiments, encrypting audio streams under system load led to substantial frame loss, in contrast to our optimized SafePath design that maintained performance near native levels.

3 SafePath Design

3.1 Threat model

SafePath addresses the threat of OS-level compromise on mobile devices. We assume an adversary who has full control over the operating system and can inspect, modify, and replay any data that passes through the OS. Our design makes the following trust and threat assumptions:

(A1) Trusted minimal core for input data protection: SafePath relies on a small, isolated component, introduced later, that operates independently of the compromised OS to manage sensitive input data. We assume this component correctly enforces memory isolation and controls access to protected data, with a minimal code base to facilitate verification.

(A2) Application trust post-data retrieval: SafePath protects input data until it is retrieved by the designated application. After retrieval, securing the data is the application's responsibility, potentially with support from trusted execution environment (TEE) techniques, such as Intel SGX [30] and ARM TrustZone [42]. SafePath does not protect against compromises within the application.

(A3) No physical or side-channel attacks: SafePath does not protect against physical attacks, such as direct hardware probing, nor against side-channel attacks like cache timing or speculative execution leakage. These threats are orthogonal and require complementary defenses.

(A4) Trusted secure boot: We assume the system boots into a trusted initial state, establishing integrity for SafePath's isolated component and preventing adversary control before SafePath's protections are activated.

(A5) Denial-of-service (DoS) attacks are out of scope: SafePath does not attempt to prevent DoS attacks, such as blocking or dropping input data before it reaches applications. Our focus is on protecting the confidentiality of input

data against a compromised OS. We assume adversaries prioritize stealth and data exfiltration over causing observable disruptions.

3.2 SafePath overview

The high level idea. We define a device input data path, or simply input path, as the sequence of memory locations (buffers) that hold input device data before it reaches an application. The first buffer in this path is the first memory location in the software stack that captures the data generated by hardware. We refer to this initial location as the **first stop buffer (FSB)**. Because attackers with OS-level privileges can freely access memory content, protecting an input path typically requires a **trusted software entity (TSE)**, which is a component operating at a higher privilege level than the OS, to intercept input data as it is written to the FSB. The TSE then either sends the data directly to the application, bypassing the OS, or encrypts the data and allows it to continue traveling along the original input path.

At first glance, encryption appears to be the only way to preserve the input path while ensuring data confidentiality. However, closer examination shows that encryption-based solutions rely on two roots of trust: the TSE that intercepts and encrypts the data, and the cryptographic algorithms and key management protecting it during transmission. If either is compromised, the confidentiality of input data is at risk.

SafePath adopts a different approach by relying solely on the TSE for data protection. Rather than encrypting and forwarding input data along the original path, the TSE securely stores the data and transmits only indexes, which are opaque references to protected data chunks, through the untrusted OS to the application. Upon receiving these indexes, the application can securely query the TSE to retrieve the original input data.

This design has two major advantages over encryption-based approaches. *First*, SafePath significantly reduces trust dependencies: it relies only on the TSE's memory isolation, eliminating the need to trust cryptographic primitives or key management against OS-level attackers. *Second*, by avoiding cryptographic operations, SafePath achieves substantially lower overhead compared to encryption-based solutions. We analyze the reasons behind these performance gains in Section 5 and present detailed evaluation results in Section 6.2.

Operations overview. Figure 1 illustrates the data flow in both the native system and the SafePath-enhanced system. We first describe the baseline operation before presenting the changes introduced by SafePath.

In the native system, when an input device generates data, the device driver writes the data into the first stop buffer (FSB) (Op. ①). The data then flows through multiple OS-managed buffers spanning the Linux kernel and Android framework (Op. ②) before reaching the application (Op. ③). Because these intermediate buffers are exposed to the OS, attackers with OS-level privilege can observe or tamper with the data.

When SafePath protection is enabled, the input flow is modified to secure data confidentiality while preserving OS compatibility. As input data arrives, the device driver attempts to write to the FSB as usual (Op. ❶). However, SafePath's trusted hypervisor, the **Tiny Memory Hypervisor (TMH)**, sets the FSB to be read-only for OS code, causing any driver write to trap to

the TMH (Op. ❷). The TMH intercepts the data, partitions it into chunks, assigns indexes, and securely stores the data in protected memory (Op. ❸). Instead of transmitting the original data, the TMH writes the generated indexes back into the FSB (Op. ❹), allowing the OS to continue handling input events as normal. The indexes propagate through the OS input stack (Op. ❺) and eventually reach the application (Op. ❻). Upon receiving an event containing indexes, the application calls the **Data Retrieval Interface (DRI)** provided by the TMH, supplying the device ID, the indexes, and a buffer to receive the original data (Op. ❼). After validating the request, the TMH retrieves the corresponding data from secure storage and returns it to the application (Op. ❽). Throughout this process, SafePath enforces a one-time-use policy for indexes: any attempt to misuse or replay an index is detected and triggers user alerts about potential breaches. This design secures input data end-to-end against a compromised OS while preserving compatibility with the native event-driven input model.

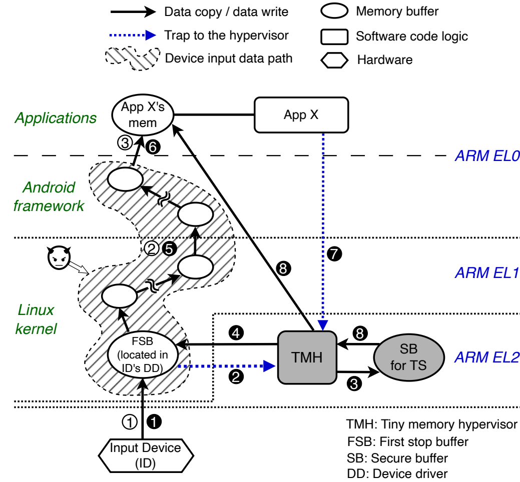


Fig. 1: SafePath operations overview. (Black numbers in white circles mark the sequence of ops when SafePath protection is disabled (i.e., the native system). White numbers in black circles mark the sequence of ops when SafePath protection is enabled.)

3.3 The tiny memory hypervisor (TMH)

SafePath's trusted software component, the Tiny Memory Hypervisor (TMH), secures input device data against a compromised OS. The TMH, operating at ARM Exception Level 2 (EL2), above the OS at EL1, realizes the Trusted Software Entity (TSE) introduced earlier and leverages ARM hardware virtualization to enforce memory isolation without disrupting OS execution.

The TMH's primary role is to intercept input data at the FSB (Section 3.2) before the OS can access it, securely store the data in protected memory, and mediate its retrieval by applications. To achieve this, the TMH configures the FSB memory pages to be read-only from the OS's perspective (EL1). Any attempt by a device driver to write to the FSB triggers a stage-2 permission fault, trapping execution to the TMH at EL2. Upon trapping, the TMH captures the input data, partitions it into chunks, assigns each chunk a unique index, and stores the data in memory regions protected by stage-2 translation control.

After securing the data, the TMH substitutes the original input in the FSB with the corresponding indexes and returns control to the device driver. From the OS's viewpoint, input event processing proceeds normally, but only opaque indexes traverse the input path instead of sensitive data.

The TMH also mediates application access to protected input data. When an application receives input events containing indexes, it interacts with the

TMH through a controlled retrieval interface to obtain the original data. To defend against misuse, the TMH enforces a one-time-use policy: each index can be resolved exactly once, and any attempts at reuse or tampering are detected.

Unlike traditional hypervisors, the TMH focuses solely on securing input paths. It does not manage device emulation, guest scheduling, or general I/O. This narrow scope keeps the TCB small and auditable. By relying on ARM's two-stage memory translation and privilege separation, the TMH provides robust isolation guarantees with minimal system impact.

3.4 Intercepting input data at the first stop buffer (FSB)

The FSB is the first memory location in software that holds device input after it leaves the hardware. Typically, the FSB resides in the device driver. For devices using direct memory access (DMA), input data initially lands in a DMA buffer, which could be treated as the FSB. For simplicity, we uniformly consider the driver's first buffer as the FSB, since SafePath's protection mechanism applies equally in either case. We discuss handling DMA devices later in Section 4.

Fig. 2: FSB redirection illustration: before redirection

```
1 char* ptrFSB = kmalloc(32, GFP_KERNEL);
2 ...
3 kfree(ptrFSB); //when driver is
      unloaded
```

To intercept input data at the FSB (Op. ❷ in Figure 1), the TMH configures the FSB's memory pages as read-only (R/O) for EL1 and EL0 code, using ARM's stage-2 translation control. This ensures that any write to the FSB by the device driver triggers a stage-2 permission fault, trapping execution to the TMH at EL2. However, a challenge arises because stage-2 permissions are enforced at page granularity, typically 4 KB, while an FSB is often much smaller (e.g., tens of bytes). Therefore, marking an entire page as R/O would inadvertently restrict access to unrelated memory contents, causing excessive trapping and performance degradation.

Fig. 3: FSB redirection illustration: after redirection

```
1 char* ptrFSB = PAB_alloc(32);
2 ......
3 PAB_free(ptrFSB);//when driver is
      unloaded
```

To address this challenge, our solution allocates FSBs from a special page-aligned buffer (PAB), which is a contiguous, page-aligned memory region reserved exclusively for storing FSBs of all protected input devices. Device drivers are modified to allocate and free FSBs through wrapper functions, `PAB_alloc()` and `PAB_free()`, replacing the standard `kmalloc()` and `kfree()` calls (Figures 2 and 3). Specifically, `PAB_alloc()` allocates a desired memory region from the PAB, and `PAB_free()` returns it. This redirec-

Fig. 4: Dynamic FSB redirection illustration

```
1  char * roFSB = ROPAB_alloc(32);
2  char * rwFSB = RWPAB_alloc(32);
3  char * ptrFSB;
4  ...
5  // Line 6-9 is called every time
       the FSB is about to receive
       data from the hardware.
6  if (protectionIsEnabled)
7      ptrFSB = roFSB;
8  else
9      ptrFSB = rwFSB;
10 ...
11 PAB_free(roFSB); //when driver is
       unloaded
12 PAB_free(rwFSB); //when driver is
       unloaded
```

tion confines FSBs to a easily protected memory region without affecting unrelated memory.

However, the initial PAB redirection has a drawback: since the PAB stores FSBs for all protected devices, enabling SafePath protection for one device also affects all other devices sharing the same PAB, even if protection is not desired. To support per-device selective protection, SafePath maintains two PABs of equal size: one always configured as read-only (R/O) for protected devices, and the other as read-write (R/W) for unprotected ones. At runtime, the device driver dynamically redirects FSB usage to the appropriate PAB by maintaining two pointers and selecting the correct one before each input transaction based on the user preference (Figure 4). This dynamic PAB redirection allows SafePath to selectively protect individual input devices with minimal overhead while preserving compatibility with existing driver logic.

3.5 Input data partitioning and indexing

When SafePath protection is enabled, any attempt to write input data to the FSB triggers a trap to the TMH. The TMH captures the original input data and saves it into a secure buffer (Op. ❸ in Figure 1), while placing indexes corresponding to the input chunks into the FSB (Op. ❹).

Partitioning and indexing are straightforward for devices that generate event-like inputs, such as touchscreen taps, GPS fixes, and motion sensor readings. These events have clear boundaries and consistent sizes, making it easy to partition input data along event boundaries. Each index written to the FSB occupies the same space as the original event, ensuring that the OS input pipeline remains undisturbed. For example, if a touchscreen event is 32 bytes, the TMH writes a 32-byte index into the FSB by padding as needed.

Handling stream input data, such as microphone or camera streams, is more complex because there are no natural boundaries in the incoming data. However, we observe that system components process stream data in well-defined batches. As shown in Figure 5(a), microphone input flows through the device driver using an 8 KB FSB, is polled by the hardware abstraction layer (HAL) in 4 KB units, and further processed by the `AudioFlinger` service in 1 KB chunks.

Based on this observation, the TMH partitions stream input based on the smallest batch size seen along the processing path. Figure 5(b) illustrates this approach: the TMH partitions incoming audio into 1 KB chunks, stores the chunks securely, and places corresponding indexes into the FSB. As with event-like inputs, each index occupies the same space as its original data chunk—in this case, 1 KB—to preserve compatibility with downstream software.

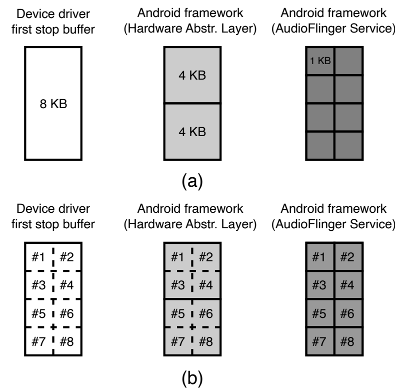


Fig. 5: Partitioning and indexing stream audio data from microphone. (a) In the native system, different software entities process the stream data in different sizes. (b) The tiny memory hypervisor partitions stream audio data based on the smallest processing sizes used in the native system.

3.6 The data retrieval interface (DRI)

The TMH provides the Data Retrieval Interface (DRI) for applications to fetch original input data after receiving input indexes (Op. ❼ in Figure 1). The DRI is implemented as a hypercall, a direct communication path between the caller and the TMH. However, user applications run at EL0 and cannot directly invoke hypercalls, which are accessible only from EL1 or higher. To bridge this gap, we design the DRI as a sys-hypercall, which is a standard system call that internally issues a hypercall—allowing applications to access TMH services indirectly.

The sys-hypercall DRI requires three input arguments: the device ID, an input data chunk index, and a memory buffer to receive the retrieved data. Upon invocation, the TMH uses the device ID to locate the correct secure buffer, retrieves the original data chunk based on the index, and writes it to the caller-provided buffer (Op. ❽ in Figure 1).

Detecting abusive usage of the DRI. Security concerns may arise because the sys-hypercall DRI does not require caller authentication. In principle, any process, including malicious ones, could invoke the DRI and attempt to retrieve protected input data.

SafePath mitigates this risk by enforcing a single-use policy for input indexes. For input data intended for one receiving application, such as touchscreen events delivered to a banking app, each index should be resolved exactly once. The TMH tracks usage of all input indexes: if an index is used more than once, the system triggers a security alert, warning the user of a potential compromise and suggesting that the affected input device should be disabled.

An attacker might attempt to bypass this defense by intercepting indexes and preventing them from reaching the legitimate application, ensuring each index is still used only once. However, this approach introduces a new weakness: the receiving application would observe missing inputs, easily detecting the attack. Given our threat model assumption (A3) that attackers seek to avoid detection, such denial-based strategies are unlikely.

Through single-use tracking and anomaly detection, SafePathsecures the DRI against unauthorized retrievals without relying on complex caller authentication mechanisms. While our current design enforces a strict single-use policy for each input index, we acknowledge that some advanced scenarios may require legitimate input multicasting to multiple receivers. We discuss how SafePath can be extended to support such cases later in Section 4.

3.7 Using SafePath by application developers

SafePath provides a switch interface that applications can invoke to enable or disable protection for specific input devices on demand. Like the data retrieval interface (DRI), the switch interface is implemented as a sys-hypercall.

From a developer's perspective, using SafePath is straightforward and requires minimal code changes. Developers define a security policy specifying which input devices to protect and under what conditions, then invoke the switch interface accordingly. For example, a banking app may enable SafePath protection for the touchscreen, accelerometer, and gyroscope while the user is entering a password, and disable it afterward to minimize overhead.

The switch interface is the only SafePath API intended for direct use by applications. Application developers are not required to invoke the DRI manually. Instead, when an application requests input through standard Android

services, such as the `InputManager`, the system automatically triggers the DRI if SafePath protection is enabled for the associated device. If protection is disabled, input handling proceeds normally. By integrating DRI invocation into the Android framework rather than exposing it to individual apps, we ensure broad compatibility and minimizes the development burden for application developers.

4 Discussion

4.1 Input data multicasting

SafePath's default one-time-use policy for input indexes prevents misuse and replay, assuming input data is intended for a single recipient. However, some scenarios require multicasting, where multiple trusted components need access to the same input.

To support this, SafePath can be extended to allow explicit multicast declarations at the DRI. Instead of consuming an index after one use, the caller can specify how many times it may be used (e.g., via a usage count or recipient list). The TMH enforces this limit, ensuring that input data is accessed only by authorized entities the intended number of times. This extension maintains SafePath's core security properties while providing flexibility for advanced use cases, with misuse still detectable through index overuse.

4.2 Dealing with input devices that use DMA

As discussed in Section 3.4, devices using Direct Memory Access (DMA) deliver input data to a dedicated memory buffer (typically allocated by the driver) before software processes it. In these cases, the DMA buffer should be treated as the first stop buffer (FSB). However, our current prototype does not yet support trapping writes to DMA buffers, because ARM's stage-2 translation only applies to CPU-initiated memory access; it cannot interpose on hardware-initiated DMA writes.

We identify two potential solutions to support DMA-based FSBs in SafePath:

(1) *IOMMU virtualization.* By configuring an IOMMU, the hypervisor could monitor and isolate DMA transactions at the device level. This would allow SafePath to trap or redirect input data at the DMA layer, similar to how CPU-accessed FSBs are handled today. While promising, this requires IOMMU support in both hardware and firmware, which is not always present on embedded/mobile platforms.

(2) *Trap-on-read via write-only mapping.* As an alternative, SafePath could mark the DMA buffer as write-only in stage-2 page tables. This would allow DMA writes to proceed unimpeded, but any CPU read, either by the driver or an attacker, would trigger a stage-2 permission fault. When trapped, the hypervisor can inspect the program counter (PC) to determine if the access originated from the legitimate device driver. If so, it emulates the read instruction, performs operations ❸ and ❹ from Figure 1, and continues SafePath's normal flow. Unauthorized reads from outside the driver would be blocked or logged.

This second approach offers a lightweight alternative to IOMMU-based mediation and may enable SafePath to protect DMA-driven inputs even on devices lacking IOMMU support. We leave the detailed implementation and evaluation of this mechanism to future work.

4.3 Supporting in-OS processing

A key advantage of SafePath over OS-bypassing input protection techniques is its ability to support in-OS input processing, which is often required in real-world workflows. Many systems rely on trusted OS components to preprocess or filter input data before it reaches user applications, such as gesture smoothing for touchscreen events, noise filtering in audio capture, or sensor fusion in motion tracking.

To support these cases, SafePath allows trusted in-OS components to act as intermediate recipients of protected input. Each such component invokes the same Data Retrieval Interface (DRI) used by applications to securely fetch input data from the TMH. After processing, the component can re-index the result for continued secure transmission. This design treats each legitimate consumer, whether in the OS or in user space, as an authorized participant in its secure input pipeline, requiring use of the DRI and re-indexing interface for access and forwarding.

4.4 Confining input writes from untrusted drivers

A potential concern in SafePath's threat model is that a compromised OS could instruct the device driver to bypass the FSB and write input data directly to arbitrary memory locations outside the protected path. Since SafePath's enforcement relies on trapping writes to the FSB via stage-2 permissions, it does not prevent such out-of-path memory writes.

However, this behavior would disrupt the expected input flow. Input written outside the FSB would not be processed and delivered to the application via SafePath's pipeline, resulting in dropped or unusable input. This breaks application functionality and violates the attacker's objective of remaining stealthy and undetected—an assumption central to our threat model. In other words, SafePath thus relies on the system's functional requirements to constrain attacker behavior: writing input elsewhere results in visible malfunction.

While our current prototype does not confine driver memory writes at runtime, stronger defenses, such as selectively making all driver-accessible memory read-only except for the FSB, could be implemented using stage-2 translation or driver isolation mechanisms. We leave this as future work.

5 System Implementation

We implemented a prototype SafePath system on an ODROID-XU4 development board [35], which features a Samsung Exynos5422 Cortex-A15/Cortex-A7 octa-core CPU and 2 GB of memory. The prototype supports six input devices: touchscreen, microphone, camera, GPS receiver, fingerprint scanner, and accelerometer. The first five are real devices connected via USB, while the accelerometer is a virtual device managed by a real driver. Several notable implementation practices and design strategies are discussed below.

5.1 Identifying the first stop buffer (FSB)

To support a new input device, SafePath must first identify the device's FSB within the driver and then apply the dynamic FSB redirection described in Section 3.4. Our implementation uses the following strategies to locate the FSB depending on the device's communication model.

For devices that use direct memory access (DMA), we first locate the DMA buffer in the driver source code. This is aided by inspecting common OS kernel functions that assign DMA buffers to bus systems. For example, USB devices typically allocate DMA buffers using the `usb_fill_bulk_urb()` function. Once the DMA buffer is identified, we trace the data flow from the DMA buffer into driver memory to pinpoint the FSB.

For devices that do not use DMA but communicate via a system bus, we examine how the driver reads input data using bus APIs. For instance, devices connected through I2C often use functions like `i2c_smbus_read_word_swapped()` to retrieve register values into driver memory. In this case, the destination memory region serves as the FSB.

For devices that do not use formal bus systems, such as those relying on GPIO or UART interfaces, we identify the FSB by analyzing how memory-mapped I/O (MMIO) is set up and used to transfer input data from hardware to driver memory.

5.2 Analyzing and emulating trapped write instructions

When SafePath protection is enabled, any attempt by a device driver to write to a device's FSB triggers a trap to the TMH. To handle each trap, the TMH must analyze the faulting instruction and obtain the data intended for the write.

This analysis begins by using the saved program counter (PC) from the OS context to locate the faulting machine instruction. The TMH then decodes the instruction according to the ARM architecture specification to determine the type of store operation and extract the data that was being written. The extracted data is then saved into a secure buffer maintained and accessible only by the TMH. To maintain execution consistency, the TMH emulates the original write instruction, but substitutes the real input with the corresponding index in the FSB. Our implementation supports analyzing and emulating all 24 ARM "STORE" instructions that write to memory, ensuring compatibility across diverse device drivers.

5.3 Reducing trap and retrieval overhead

Our original SafePath prototype strictly followed the design described in Section 3. However, experiments revealed significant performance overhead compared to the native system. After analysis, we identified two primary causes and implemented corresponding optimizations.

Performance degradation reason 1: trap overhead for large Writes. In the original implementation, every write to an R/O FSB triggered a trap to the TMH. This became costly for large memory operations, such as `memcpy()`, where copying a 1 KB buffer could result in hundreds of traps.

Optimization 1: bulk memory write Recognition and trap consolidation. To address this, the TMH attempts to detect if a trapped write originates from a `memcpy()` function. By using the saved program counter (PC), the TMH locates the faulting machine instruction and inspects a window of surrounding instructions. It matches the observed instruction sequence against known patterns generated by typical `memcpy()` implementations, such as loops of load-store operations with pointer increments. If a match is found, the TMH extracts the source address, destination address, and data size, and performs the entire copy operation in a single trap, significantly reducing overhead.

Performance degradation reason 2: syscall overhead at high input rates. We also observed poor performance on high-frequency input devices, such as microphones and cameras, especially under system load. When an application invoked the sys-hypercall DRI to retrieve a chunk of input data, there were often many data chunks already buffered and ready, but each DRI call retrieved only one chunk, leading to excessive syscalls.

Optimization 2: batch retrieval in DRI. To improve efficiency, we modified the DRI to return all ready data chunks in a single call, including the chunk corresponding to the requested index. This optimization significantly reduces syscall overhead, particularly when input data is generated rapidly.

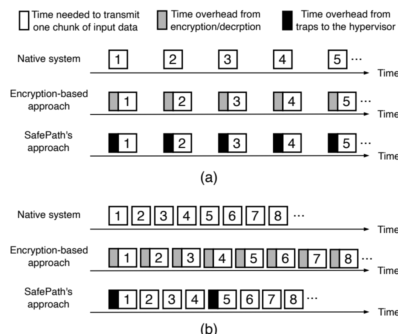


Fig. 6: Input data transmission time analysis. (a) An illustration for the case when input data generation is low. (b) An illustration for the case when input data generation frequency is high.

Notably, this optimization achieves a performance improvement that encryption-based input protection schemes cannot easily replicate. As illustrated in Figure 6, when input data rates are low, both encryption-based approaches and SafePath introduce comparable overheads (Figure 6(a)). However, as input rates increase, encryption-based approaches incur linearly growing encryption and decryption costs, while SafePath can amortize retrieval overhead by fetching multiple data chunks per trap (Figure 6(b)).

6 Evaluation

We conducted extensive experiments to evaluate the effectiveness and performance of our SafePath prototype. The evaluation covers six input devices supported by the system: touchscreen, microphone, camera, GPS receiver, fingerprint scanner, and accelerometer. For each device, we developed a corresponding application to collect input and analyze system behavior under SafePath protection. The accelerometer is a virtual device, but we injected real-world accelerometer traces into the driver to simulate realistic input behavior. These traces traverse the full software stack, following the same data path as inputs from a physical device. All experiments were conducted on Android 4.4.4, the latest version officially supported on the ODROID-XU4 platform used in our prototype.

Our evaluation focuses on two key aspects: (1) the security and functional correctness of SafePath's input protection mechanism, and (2) the performance overhead introduced by enabling SafePath.

6.1 Security and functional correctness

We evaluated SafePath's ability to secure the device input path from OS-level attackers by simulating adversaries with direct access to system memory. Specifically, we analyzed whether an attacker could extract meaningful input data from

Fig. 7: Effectiveness of secure microphone input path.

Fig. 8: Effectiveness of secure camera input path.

intermediate buffers (i.e., input path buffers or IPBs) when SafePath protection was enabled. We conducted experiments on three input devices carrying distinct data formats to visually assess confidentiality breaches: microphone (audio waveform), camera (video frame), and fingerprint scanner (image).

Microphone. Figure 7(a) shows the waveform rendered from data captured at an IPB when SafePath protection was disabled. The waveform clearly reflects a recorded human voice. In contrast, Figure7(b) illustrates the same buffer under protection: the waveform becomes flat and unrecognizable, confirming that SafePath prevented the attacker from recovering the original audio content.

Camera. Similarly, Figure 8(a) shows a sharp video frame captured from an unprotected IPB. With SafePath protection enabled (Figure8(b)), the frame turns into visual noise, and no useful image information can be extracted. This demonstrates that SafePath's protection also extends to high-bandwidth, frame-based inputs like video.

Fingerprint scanner. The fingerprint data is especially sensitive and often targeted in biometric attacks. In Figure 9(a), the unprotected fingerprint buffer reveals a complete and identifiable print. When protection is enabled (Figure 9(b)), the visual signature is eliminated, validating that no biometric pattern can be reconstructed from the intercepted memory.

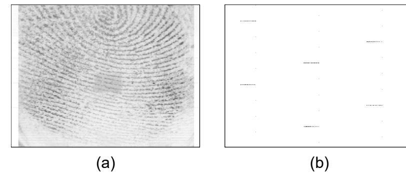
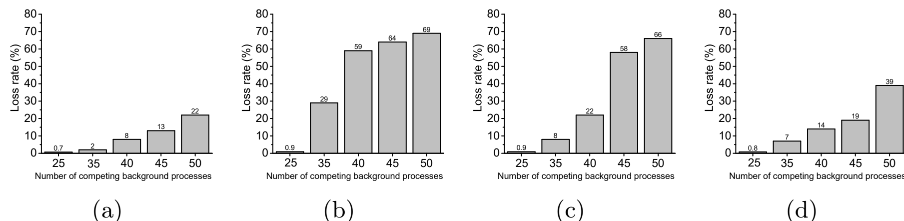


Fig. 9: Effectiveness of secure fingerprint scanner input path.

Across all three devices, these results confirm SafePath's ability to preserve data confidentiality in the presence of a fully compromised operating system. More importantly, the functional correctness of the input delivery is retained. Applications still receive and process user input correctly even though intermediate buffers are sanitized of the actual data.

6.2 Performance evaluation

We evaluated the runtime performance of the SafePath prototype system across six supported input devices: touchscreen, microphone, camera, GPS receiver, fingerprint scanner, and accelerometer. For each device, we built a simple application to collect input data and measured the performance of SafePath under varying workloads. We compared four configurations: the native system, an encryption-based system, the original SafePath implementation, and the optimized SafePath system.



(a) (b) (c) (d)

Fig. 10: Microphone input path overhead measured in terms of audio loss rate in percentage when different numbers of background processes competing for the CPU exist. (a) Native system. (b) Encryption-based approach. (c) SafePath Original implementation. (d) SafePath Optimized implementation.

Table 2: Touchscreen input path overhead.

	Touch event end-to-end transmission time (milliseconds)	Time to write a touch event to first stop buffer (microseconds)
Native system	225	1
Encryption-based	227	12
SafePath	228	32

Touchscreen. We evaluated two performance metrics for touchscreen input: (1) end-to-end latency and (2) FSB write time. The end-to-end latency measures the time from when the driver receives a hardware interrupt to when the application callback is invoked. Table 2 shows that SafePath (228ms) incurs only 3ms additional latency compared to the native system (225ms), on par with the encryption-based baseline (227ms).

For the second metric, we measured the time to write a 4-byte touch event to the FSB. SafePath incurred a higher overhead (32 µs) due to instruction emulation in EL2, but given that the total touch event transmission time operates at the scale of hundreds of milliseconds, this microsecond-level overhead is negligible and does not significantly impact application responsiveness.

Microphone. We evaluated audio recording performance under varying system loads by measuring the audio loss rate. The loss rate is calculated as

$$\frac{HCAF - URAF}{HCAF}, \tag{1}$$

where HCAF denotes the number of hardware-captured audio frames, and URAF denotes the number of user-application-received audio frames. Figure 10 presents the loss rates observed across four system configurations.

In the native system (Figure 10a), audio loss remained minimal even under heavy CPU load, demonstrating baseline system resilience. The encryption-based system (Figure 10b) showed severe degradation: the audio loss rate grew linearly with the number of background processes, reaching approximately 69% loss under 50 background tasks. This highlights the high runtime overhead introduced by per-frame encryption and decryption.

The original SafePath implementation (Figure 10c) performed better than the encryption-based system but still suffered noticeable degradation at higher loads, with a loss rate of around 39% at peak load. This overhead was primarily due to the frequent trapping and handling of small writes without optimization.

After applying our two optimizations, namely bulk memory write Recognition and trap consolidation and batch retrieval in DRI (Section 5.3), SafePath achieved performance comparable to the native system, maintaining low audio loss even as

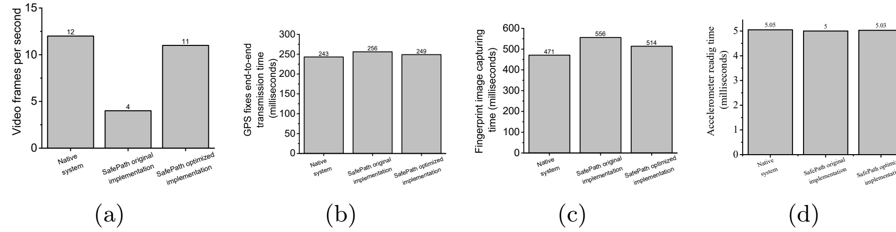


Fig. 11: Input path overhead for different devices: (a) Camera; (b) GPS receiver; (c) Fingerprint scanner; (d) Accelerometer.

background load increased (Figure 10d). Specifically, the optimized SafePath system exhibited only a 14% loss rate under 50 background processes, versus 8% for the native system. This confirms the effectiveness of both trap consolidation for `memcpy()` operations and batching multiple audio frames per DRI invocation in reducing runtime overhead under stress.

Camera. Figure 11a shows that optimized SafePath achieved nearly identical video frame rates as the native system. In contrast, the encryption-based approach introduced severe overhead that disrupted normal operation, preventing usable video recording. This highlights the practical advantage of SafePath's non-cryptographic design when handling continuous, high-throughput data streams.

GPS Receiver. We measured the end-to-end latency of receiving a GPS fix. All three tested systems (i.e., native, original SafePath, and optimized SafePath) performed nearly identically (Figure 11b), since GPS fix generation is relatively infrequent and latency-insensitive.

Fingerprint Scanner. As shown in Figure 11c, SafePath optimizations reduced fingerprint capture latency by about 10% over the original implementation, bringing performance in line with the native system. The benefit here stems mainly from faster FSB access management.

Accelerometer. For this virtual device, we measured the time to transmit a sensor reading. Figure 11d demonstrates that the native, original SafePath, and optimized SafePath systems performed almost identically, which is unsurprising given the device's low data rate and lightweight access pattern.

Summary. Our evaluation yields three key takeaways: (1) SafePath's performance optimizations are highly effective, bringing input path performance close to native levels, even for high-frequency devices like microphones and cameras. (2) Encryption-based solutions introduce substantial overhead under heavy load or for streaming inputs, which SafePath avoids by design. (3) For low-frequency or event-driven devices, all approaches perform similarly, reaffirming that SafePath's overhead remains minimal when traps are rare.

7 Conclusion

We presented SafePath, a system that secures device input paths without relying on encryption or OS bypassing. By combining a tiny memory hypervisor with an index-based transmission mechanism, SafePath protects input data while preserving OS compatibility. Our prototype, supporting six input devices, demonstrates that SafePath effectively blocks OS-level attacks with minimal overhead, outperforming encryption-based approaches under high load.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by NSF Award #1566375 and #1943269.

References

1. Ahmad, A., Schultz, A., Lee, B., Fonseca, P.: An extensible orchestration and protection framework for confidential cloud computing. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2023)
2. Anand, S.A., Saxena, N.: Speechless: Analyzing the threat to speech privacy from smartphone motion sensors. In: IEEE Symposium on Security and Privacy (S&P) (2018)
3. ARM Limited: Armv7-a virtualization extensions. https://developer.arm.com/documentation/den0013/d/Virtualization/ARMv7-A-Virtualization-Extensions/Types-of-virtualization (2011)
4. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M., Goltzsche, D., Eyers, D.M., Kapitza, R., Pietzuch, P.R., Fetzer, C.: SCONE: secure linux containers with intel SGX. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2016)
5. Aviv, A.J., Gibson, K.L., Mossop, E., Blaze, M., Smith, J.M.: Smudge attacks on smartphone touch screens. In: USENIX Workshop on Offensive Technologies (WOOT) (2010)
6. Azab, A.M., Ning, P., Shah, J., Chen, Q., Bhutkar, R., Ganesh, G., Ma, J., Shen, W.: Hypervision across worlds: Real-time kernel protection from the ARM trustzone secure world. In: ACM Conference on Computer and Communications Security (CCS) (2014)
7. Ba, Z., Zheng, T., Zhang, X., Qin, Z., Li, B., Liu, X., Ren, K.: Learning-based practical smartphone eavesdropping with built-in accelerometer. In: Network and Distributed System Security Symposium (NDSS) (2020)
8. Baumann, A., Peinado, M., Hunt, G.C.: Shielding applications from an untrusted cloud with haven. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2014)
9. Borrello, P., Kogler, A., Schwarzl, M., Lipp, M., Gruss, D., Schwarz, M.: Æpic leak: Architecturally leaking uninitialized data from the microarchitecture. In: USENIX Security Symposium (2022)
10. Brasser, F., Gens, D., Jauernig, P., Sadeghi, A., Stapf, E.: SANCTUARY: arming trustzone with user-space enclaves. In: Annual Network and Distributed System Security Symposium (NDSS) (2019)
11. Bulck, J.V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In: USENIX Security Symposium (2018)
12. Butler, J.: DKOM: Direct kernel object manipulation. Black Hat USA (2004)
13. Cai, L., Chen, H.: Touchlogger: Inferring keystrokes on touch screen from smartphone motion. In: USENIX Workshop on Hot Topics in Security (HotSec) (2011)
14. Canella, C., Bulck, J.V., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D.: A systematic evaluation of transient execution attacks and defenses. In: USENIX Security Symposium (2019)

15. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J.S., Ports, D.R.K.: Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In: ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2008)
16. Cheng, Y., Ding, X., Deng, R.H.: Efficient virtualization-based application protection against untrusted operating system. In: ACM Asia Conference on Computer and Communications Security (AsiaCCS) (2015)
17. Dall, C., Li, S., Lim, J.T., Nieh, J., Koloventzos, G.: ARM virtualization: Performance and architectural implications. In: ACM/IEEE Symposium on Computer Architecture (ISCA) (2016)
18. David, F.M., Chan, E., Carlyle, J.C., Campbell, R.H.: Cloaker: Hardware supported rootkit concealment. In: IEEE Symposium on Security and Privacy (S&P) (2008)
19. Dunn, A.M., Lee, M.Z., Jana, S., Kim, S., Silberstein, M., Xu, Y., Shmatikov, V., Witchel, E.: Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2012)
20. Fang, Y.: Introduction to the armv8 virtualization system. https://www.openeuler.org/en/blog/yorifang/2020-10-24-arm-virtualization-overview.html (2020)
21. Gionta, J., Enck, W., Ning, P.: Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In: ACM Conference on Data and Application Security and Privacy (CODASPY) (2015)
22. Guan, L., Lin, J., Luo, B., Jing, J., Wang, J.: Protecting private keys against memory disclosure attacks using hardware transactional memory. In: IEEE Symposium on Security and Privacy (S&P) (2015)
23. Guan, L., Liu, P., Xing, X., Ge, X., Zhang, S., Yu, M., Jaeger, T.: Trustshadow: Secure execution of unmodified applications with ARM trustzone. In: ACM Conference on Mobile Systems, Applications, and Services (MobiSys) (2017)
24. Harrison, K., Xu, S.: Protecting cryptographic keys from memory disclosure attacks. In: IEEE/IFIP Conference on Dependable Systems and Networks (DSN) (2007)
25. Hof, A.V., Nieh, J.: Blackbox: A container security monitor for protecting containers on untrusted operating systems. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2022)
26. Hofmann, O.S., Kim, S., Dunn, A.M., Lee, M.Z., Witchel, E.: Inktag: secure applications on an untrusted operating system. In: ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2013)
27. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In: USENIX Security Symposium (2009)
28. Hunt, T., Zhu, Z., Xu, Y., Peter, S., Witchel, E.: Ryoan: A distributed sandbox for untrusted computation on secret data. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2016)
29. Incel, Ö.D., Günay, S., Akan, Y., Barlas, Y., Basar, O.E., Alptekin, G.I., Isbilen, M.: DAKOTA: sensor and touch screen-based continuous authentication on a mobile banking application. IEEE Access **9**, 38943–38960 (2021)
30. Intel Corporation: Intel Software Guard Extensions. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html (nd)

31. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: IEEE Symposium on Security and Privacy (S&P) (2019)
32. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: USENIX Security Symposium (2018)
33. Michalevsky, Y., Boneh, D., Nakibly, G.: Gyrophone: Recognizing speech from gyroscope signals. In: USENIX Security Symposium (2014)
34. Miluzzo, E., Varshavsky, A., Balakrishnan, S., Choudhury, R.R.: Tapprints: your finger taps have fingerprints. In: ACM Conference on Mobile Systems, Applications, and Services (MobiSys) (2012)
35. ODROID Wiki: ODROID-XU4. https://wiki.odroid.com/odroid-xu4/odroid-xu4 (nd)
36. Owusu, E., Han, J., Das, S., Perrig, A., Zhang, J.: Accessory: password inference using accelerometers on smartphones. In: Workshop on Mobile Computing Systems and Applications (HotMobile) (2012)
37. Petracca, G., Sun, Y., Jaeger, T., Atamli, A.: Audroid: Preventing attacks on audio channels in mobile devices. In: Annual Computer Security Applications Conference (ACSAC) (2015)
38. Santos, N., Raj, H., Saroiu, S., Wolman, A.: Using ARM trustzone to build a trusted language runtime for mobile applications. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2014)
39. van Schaik, S., Kwong, A., Genkin, D., Yarom, Y.: SGAxe: How SGX fails in practice. https://sgaxeattack.com/ (2020)
40. van Schaik, S., Minkin, M., Kwong, A., Genkin, D., Yarom, Y.: Cacheout: Leaking data on intel cpus via cache evictions. In: IEEE Symposium on Security and Privacy (S&P) (2021)
41. Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: A stealthy and context-aware sound trojan for smartphones. In: Network and Distributed System Security Symposium (NDSS) (2011)
42. Scott Thornton: Arm TrustZone explained. https://www.microcontrollertips.com/embedded-security-brief-arm-trustzone-explained (nd)
43. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.: Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: IEEE Symposium on Security and Privacy (S&P) (2013)
44. Song, W., Choi, H., Kim, J., Kim, E., Kim, Y., Kim, J.: Pikit: A new kernel-independent processor-interconnect rootkit. In: USENIX Security Symposium (2016)
45. Sparks, S., Butler, J.: SHADOW WALKER: Raising the bar for rootkit raising the bar for rootkit detection detection. Black Hat Japan (2005)
46. Tang, A., Sethumadhavan, S., Stolfo, S.J.: Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In: ACM Conference on Computer and Communications Security (CCS) (2015)
47. Tsai, C., Porter, D.E., Vij, M.: Graphene-sgx: A practical library OS for unmodified applications on SGX. In: USENIX Annual Technical Conference (ATC) (2017)
48. Tung, Y., Shin, K.G.: Exploiting sound masking for audio privacy in smartphones. In: ACM Asia Conference on Computer and Communications Security (AsiaCCS) (2019)
49. Weiser, S., Werner, M.: SGXIO: generic trusted I/O path for intel SGX. In: ACM Conference on Data and Application Security and Privacy (CODASPY) (2017)

50. Xu, Z., Bai, K., Zhu, S.: Taplogger: inferring user inputs on smartphone touchscreens using on-board motion sensors. In: ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec) (2012)
51. Yun, M.H., Zhong, L.: Ginseng: Keeping secrets in registers when you distrust the operating system. In: Network and Distributed System Security Symposium (NDSS) (2019)
52. Zhang, F., Chen, J., Chen, H., Zang, B.: Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In: ACM Symposium on Operating Systems Principles (SOSP) (2011)
53. Zhang, L., Pathak, P.H., Wu, M., Zhao, Y., Mohapatra, P.: Accelword: Energy efficient hotword detection through accelerometer. In: ACM Conference on Mobile Systems, Applications, and Services (MobiSys) (2015)
54. Zhang, Y., Xia, P., Luo, J., Ling, Z., Liu, B., Fu, X.: Fingerprint attack against touch-enabled devices. In: Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM) (2012)
55. Zhou, Z., Gligor, V.D., Newsome, J., McCune, J.M.: Building verifiable trusted path on commodity x86 computers. In: IEEE Symposium on Security and Privacy (S&P) (2012)
56. Zhou, Z., Yu, M., Gligor, V.D.: Dancing with giants: Wimpy kernels for on-demand isolated I/O. In: 2014 IEEE Symposium on Security and Privacy (S&P) (2014)