Beyond Conventional Triggers: Auto-Contextualized Covert Triggers for Android Logic Bombs

Ye Wang, Bo Luo, and Fengjun Li
Department of Electrical Engineering and Computer Science, Institute for Information Sciences
The University of Kansas, Lawrence, KS, USA
{yeah_wong, bluo, fli}@ku.edu

***Abstract*—Recent advances in static analysis, fuzzing, and learning-based detection have substantially improved the defense against trigger-based malware; however, these approaches mostly assume that trigger conditions are semantically explicit or distinguishable from normal application logic. In this paper, we present *SensorBomb*, a novel logic-bomb framework that exploits this assumption through auto-contextualized triggers and onboard sensor-actuator covert channels. Instead of relying on obscure or rare trigger conditions, SensorBomb constructs triggers tightly aligned with the host app's legitimate sensor usage, actuator behaviors, and functional context so that they appear indistinguishable from benign behavior. To do so, SensorBomb automatically analyzes the host app to select context-compatible sensors, actuators, and sensitive operations, constructs covert trigger channels, and dynamically adapts trigger patterns to evade static analysis, fuzzing, sensor state anomaly detection, and user suspicion. We implement three representative prototypes of such triggers and evaluate them across diverse devices and environments. Our results show that SensorBomb consistently evades state-of-the-art detection techniques and achieves high trigger reliability with zero false positives. Large-scale injection experiments on real-world APKs further demonstrate that SensorBomb can be deployed without affecting normal app functionality. This work reveals a critical and previously underexplored attack surface in mobile malware defenses and calls for more advanced detection mechanisms.**

I. Introduction

Being a straightforward yet effective method for hiding themselves from detection, logic bombs execute hidden functions only when specific conditions are met, while remaining dormant otherwise. They have played significant roles in real-world security incidents against critical infrastructure such as banks and payroll systems [1], industrial control systems [2], [3], media outlets [4], and national security. As the most popular mobile platform, Android has also been heavily targeted by logic bombs, which are also known as passive backdoors [5], evasion malware [6], trigger-based malware [7], hidden sensitive operations (HSO) [8], [9], or hidden behaviors [10]. They commonly leverage triggers based on temporal conditions, specific events, system states, user interactions, or combinations thereof [11], [10], as summarized in Table I. Moreover, the unique hardware and software stack of mobile devices allows attackers to detect simulated or emulated test environments [12] and better conceal logic bombs from detection systems [6].

To counter such threats, researchers have developed various static and dynamic detection mechanisms. Dynamic analysis has advanced from simple trigger activation using fixed inputs [13] to sophisticated fuzzing techniques aimed at expanding behavioral coverage [14], [15]. Meanwhile, static analysis has matured significantly, with modern tools leveraging machine learning to detect both known and novel logic bombs, even those triggered by obscure or previously unseen conditions [7], [8], [10], [9], [16]. They offer a complementary and increasingly effective defense against mobile logic bombs. As shown in Table I, most of the commonly used triggers can be effectively detected by both static and dynamic detection approaches. Consequently, the real-world impact of Android logic bombs has been significantly reduced, and academic attention to the topic has declined.

Despite these advances, we observe a critical blind spot in existing detection frameworks, particularly those based on static analysis, which often overlook sensor-based triggers. For example, detectors like TriggerScope [7] and HSOMiner [8] do not incorporate sensor input into their analysis pipelines. Difuzer [9] in theory can detect arbitrary trigger types, but its implementation does not account for sensor data. Similarly, the widely used open-source TriggerZoo dataset [17], which catalogs thousands of logic bombs and benign trigger-based behaviors for detector training, lacks instances involving sensor data. As a result, sensor-based logic bombs such as *Cerberus* [18], *Currency Converter*, and *BatterySaverMobi* [19] have successfully bypassed state-of-the-art (SOTA) detectors and caused damage before public disclosure. Later, sensor-based covert channels have been explored that could activate malware via out-of-band signals [20]. Unlike conventional sensor-activated triggers, these covert channels exploit complex patterns that are difficult to reproduce through dynamic fuzzing, showing a promising potential to evade current detection techniques.

However, despite their potential, no existing work systematically investigates how such sensor-based triggers can be leveraged to construct practical logic bombs or assesses their effectiveness in evading current detection systems. As existing Android fuzzing frameworks can be extended to fuzz sensor APIs, we experimentally tested a sensor-based trigger using

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240348
www.ndss-symposium.org

TABLE I: Trigger types and detection mechanisms for known Android logic bombs.

Trigger Type	Description	Static	Dynamic		Anomaly Detection
			Fuzzing	Others	
Time-based [7], [8], [9], [23], [24]	At a specific time or date, or after a certain interval	×	×	-	-
Event-based [8], [9], [25], [11]	System events (e.g., `BOOT_COMPLETED`, screen unlock)	×	×	-	-
Context-aware [8], [9], [25], [11], [12]	Contextual data from the device's environment (GPS, battery level)	×	-	×	-
System property [8], [9], [25], [11]	System properties or configurations	×	×	-	-
Communication/Signal [8], [9], [10]	External triggers (SMS, HTTP, user input)	×	✓	×	-
Sensor status or value [11], [26], [27]	physical characteristics for the execution environment, and specific values to trigger	○	×	×	-
Covert channels [21], [22]	Sensor-based covert channels to transmit trigger commands from external emitters	○	✓	-	×
SensorBomb (ours)	Onboard sensor-based covert channels with auto-contextualized embedding	✓	✓	✓	✓

Note: ✓ and × denote can and cannot evade the detection mechanism, respectively. ○ means not being fully evaluated, and - denotes not applicable.

step count [18] (§VII-D) and found that sensor-value-based triggers face fundamental limitations: because sensor readings operate within narrow and predictable physical ranges, fuzzing such values is computationally expensive yet feasible (§VIII-C). To maintain stealth, attackers often sacrifice success rate by employing wide-net strategies such as using rare but naturally occurring conditions, e.g., specific GPS locations [7] or unusually high step counts [18]. These attacks are usually one-shot: once discovered, they can be easily defeated through targeted fuzzing. On the other hand, sensor-based covert channels offer a more intricate triggering mechanism using timing-sensitive or modulated sensor inputs, while maintaining a high attack success rate. However, these channels require interference isolation for reliable transmission, making them susceptible to anomaly detection through sensor behavior monitoring [21], [22]. Their practicality is limited since they often depend on customized hardware and close physical proximity to the target device [20]. Moreover, attackers lack visibility to the device's real-time status, which undermines the reliability of trigger delivery in practice (§VII-B). Therefore, we argue that, while sensor-based triggers and covert channels show potential, they cannot be naively adopted to construct practical logic bombs that evade all existing defenses.

In this work, we present SensorBomb, a novel framework to overcome these limitations by constructing logic bombs using ***auto-contextualized*** triggers and onboard sensor-actuator covert channels. These triggers align with the host app's sensor usage patterns and behavioral context, ensuring robust activation and resistance to detection. During embedding, SensorBomb automatically generates code that aligns sensor-based triggers, hidden payloads, and actuator behaviors with the host app's context. As a result, the injected logic can bypass static analysis, evade sensor anomaly detectors, avoid user suspicion, and remain resilient against fuzzing tests. At runtime, it leverages the app's own behavior and sensor characteristics to refine trigger patterns, thereby maximizing attack success rates and guaranteeing zero false triggers.

To support practical deployment, we propose a lightweight architecture and demonstrate its effectiveness through three prototypes. Our evaluation shows that SensorBomb is the first logic bomb framework capable of simultaneously evading state-of-the-art static analysis, fuzzing tests, and sensor-state-based anomaly detection. It eliminates key real-world limitations by removing the need for external emitters, increasing attack success rate from 70% to 99%, and reducing false-trigger rate to zero. Large-scale experiments also demonstrate that SensorBomb can be injected into existing APK files without affecting their normal functionality. We summarize our contributions as follows:

- We are among the first to introduce the concept of auto-contextualization into logic bomb design, shifting the focus from exploring new trigger conditions to embedding triggers within the application's context. This enables simultaneous evasion of all current detection mechanisms.
- We develop a framework to systematically implement auto-contextualized logic bombs, leveraging novel onboard covert channels. Our design eliminates the need for external emitters, minimizes computational overhead, and requires little engineering effort, making it practical for mobile platforms.
- We validate our approach through three real-world prototypes and large-scale injection experiments, demonstrating its effectiveness, improved practicality, strong evasion capabilities, and broad applicability.
- We conduct one of the first comprehensive analyses of sensor-based logic bomb detection methods, identifying key blind spots and offering insights to guide the development of more robust future defenses.

The remainder of the paper is organized as follows: Section II introduces the background; Section III presents the problem and threat model, followed by the feasibility study in Section IV; Section V presents the design of the SensorBomb framework and Section VI covers three prototypes; Sections VII and VIII evaluate attack performance and detection evasion, respectively; Section IX discusses defenses and limitations; and Section X concludes the paper.

II. Background and Related Works

Mobile Logic Bombs. *Trigger* is a piece of code that activates operations under certain conditions [13]. It consists of a triggering condition and a guarded code with a `true` branch and a `false` branch. As shown in Figure 1(B), the trigger is defined as a triplet $t = (c, T_c, Q_c)$, where c is the entry point, and the true branch T_c and false branch Q_c are invoked when the trigger conditions are met or unmet, respectively.

When T_c involves a security-sensitive operation, it is a hidden sensitive operation (HSO) [8]. PScount [28] compiles a list of sensitive APIs based on Android's permission specifications and scrutinizes a set of sensitive operations. An HSO is called a suspicious hidden sensitive operation (SHSO) [9] or logic bomb [7] if it contains suspicious or malicious operations. In this paper, we use the terms HSO, SHSO, and logic bomb interchangeably.

On the Android platform, resource APIs are often used to construct trigger conditions through an entry-point function $f(x)$. It may use the API type invoked by an application (i.e.,

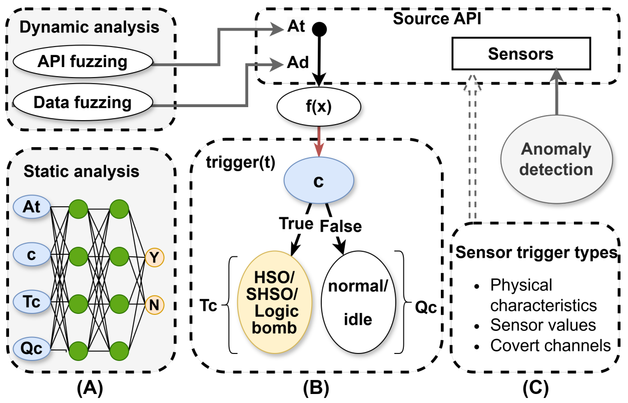


Fig. 1: (A) Conventional detection methods. (B) A typical trigger structure. (C) Sensor-based logic bombs and detection.

A_t) and/or the associated API data (i.e., A_d). Sensors can serve as source APIs and therefore be exploited to trigger logic bombs, as shown in Figure 1(C). A malicious payload may first probe onboard sensors to detect virtualized or emulated environments (e.g., by identifying unrealistic accelerometer noise or static gyroscope outputs [29]). Once a real device is confirmed, it can monitor specific sensor readings, such as step counts [18] or ambient light levels [21], to activate the trigger. Consequently, even carefully crafted analysis environments may fail to activate the logic bomb.

Finally, covert channels have been explored as transmission paths for logic-bomb triggers to evade fuzzing-based detection [20]. Prior work primarily adapted external-emitter covert channels to mobile logic bombs and evaluated evasion only against traditional malware detectors. Their resilience against modern logic-bomb detection frameworks remains unexplored.

Logic Bombs Detection. Logic bombs typically exhibit simple structures and therefore often leverage newly introduced APIs as triggers. As a result, detection has evolved into a cat-and-mouse game driven by the continual adaptation of both API types and API usage patterns. Automated malware detection consequently relies on static analysis and fuzzing to identify logic bombs, as illustrated in Figure 1(A).

To counter logic bombs that evade dynamic analysis using rare triggers, detectors employ fuzzing to expose hidden behavior. Early logic bombs relied on time- or event-based triggers exposed by known inputs [13]. Later, environment-aware triggers (e.g., virtual-machine checks) allowed malware to remain dormant in emulators until bare-metal analysis systems emerged [12], [30], [31]. As trigger mechanisms diversified, random fuzzing was adopted to improve coverage [14], [15], [32]. However, as triggers evolved to include communication signals and complex user interactions, the fuzzing search space expanded dramatically, reducing overall effectiveness.

IntelliDroid [11] mitigated this limitation by combining dynamic analysis with symbolic execution to generate targeted inputs. While no tools focus exclusively on sensor-value fuzzing, frameworks such as FuzzDroid [25] and IntelliDroid support sensor APIs and can inject synthetic readings to explore sensor-driven paths and expose hidden logic bombs.

Static analysis applies taint tracking to identify triggers by examining source APIs, trigger conditions, and branch behaviors, with efforts focused on widening detectable A_t and refining feature extraction for better accuracy. Early logic bombs with limited triggers were easily caught by symbolic execution and precise source analysis [7].

As logic bombs began exploiting diverse system events, detectors extended coverage from specific APIs to all source APIs. Tools such as HSOMiner [8] use features from trigger conditions, branch behaviors, and their relationships to train classifiers for capturing HSOs. To further improve detection effectiveness, Difuzer [9] included all potential trigger APIs by extracting a broad range of features from guarded code. Its upgraded version, Difuzer++ [16], adopts content-aware analysis to achieve high detection accuracy. Finally, user-action triggers add complexity: InputScope [10] uses static taint analysis to trace direct user commands and detect backdoors.

Defenses against Sensor-based Threats. The pervasive integration of sensors in smartphones has driven extensive research on sensor-based threat detection [33]. As shown in Figure 1(C), sensor-based anomaly detection identifies anomalies in sensor data/status that may result from covert-channel activity. For instance, the 6thSense framework [34] introduces context-aware intrusion detection for smart devices by monitoring sensor states associated with each user activity and constructing a contextual model using Markov, Naive Bayes, and LMT classifiers. More recently, SBTDDL [22] employs LSTM models and augments its training data with malicious activities crafted to closely resemble benign behavior.

III. The Problem and Threat Model

In this work, we consider an attacker who aims to design logic bombs that are both stealthy and reliable, while remaining resilient against evolving defenses. This requires satisfying two core requirements: comprehensive evasion and practicality. Comprehensive evasion requires the logic bomb to evade all relevant detection mechanisms, including static analysis, dynamic fuzzing, sensor anomaly detectors, online sandboxes, and app-store vetting systems. Beyond initial deployment, the logic bomb must also exhibit long-term robustness so that it remains effective even as detection systems evolve. Finally, it should maintain stealth on mobile platforms. Since users are highly sensitive to abnormal actuator behavior on their devices, the logic bomb must operate persistently without arousing suspicion or degrading user experience.

Practicality, on the other hand, demands precise control and reliability. The logic bomb must trigger only under specific, attacker-defined conditions, ensuring exact timing and context for payload activation. Once triggered, it must execute its hidden behavior consistently across devices and environments, achieving a high success rate. Meantime, it must avoid any false triggers, as unintended activations could cause system conflicts or raise alarms with users or automated defenses.

We propose a novel onboard covert channel triggered logic bomb framework, called SensorBomb, incorporating ***auto-contextualization*** capabilities to achieve all the goals.

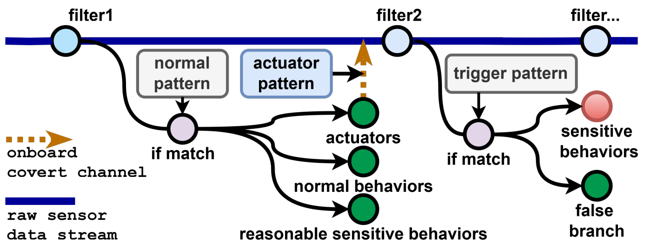


Fig. 2: Comparison between benign sensor-based branch (after filter 1) and malicious branch (after filter 2).

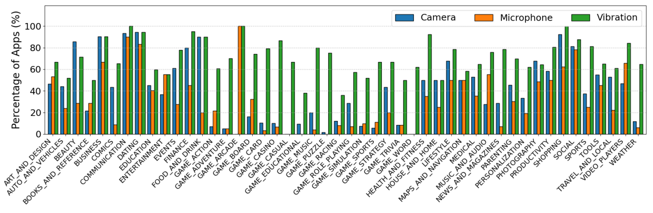


Fig. 3: Microphones, cameras, and vibrators permissions used by top-downloaded apps in each category.

SensorBomb automatically leverages the host app's contextual information to select optimal hidden behaviors that are intrinsically resistant to static analysis, and cannot be mitigated through simple updates. It then establishes a novel onboard covert channel through the app's native actuator–sensor pair to deliver triggers, thereby eliminating the need for any external emitter while keeping precise control. Furthermore, SensorBomb injects or configures actuator behaviors to align with the host app's context, effectively evading anomaly detection and remaining unsuspicious to end users. Finally, by exploiting real-time sensor data, it optimizes trigger timing to maximize the attack success rate and employs long-term statistical analysis to guarantee *zero* false triggers.

Threat Model. We adopt a common threat model that is widely used in Android malware research [35]. First, we assume the adversary could embed a SensorBomb into a host application and trick victim users into installing it from app stores or using sideloaded APK files. Host applications may be either custom-developed legitimate apps or repackaged versions of existing published apps, allowing for updates or modifications. We also assume the user's mobile device is equipped with sensors and actuators required to construct the onboard physical covert channel for the SensorBomb. These sensors and actuators are commonly available on most devices, as discussed in § IV-B.

The embedded APK file could bypass existing SOTA malware detection mechanisms, including open-source logic bomb detection tools and online malware scanners. Our evasion strategies and evaluations are discussed in §VIII. The SensorBomb acquire the same set of permissions as the host app to access Android sensor and actuator APIs. It does not request high-level permissions nor access sensor APIs at anomalous, high sampling rates.

Finally, we assume that the device is connected to the Internet, allowing it to receive trigger commands from the attacker's server via standard web APIs and multimedia channels, and to transmit trigger timing information and the exclusion set back to the server. All communications rely on standard protocols and transparent content, rendering the app's network behavior indistinguishable from that of benign applications.

IV. Attack Rationale and Feasibility

A. Attack Rationale

Let us first illustrate the workflow of sensor-based apps. As shown in Figure 2, after raw data acquisition, it is essential to mitigate noise and interference that could obscure the signal of interest. Digital filtering techniques, such as low-pass and band-pass filters, are commonly applied to smooth the sensor data and remove unwanted frequency components. These filters are universal and can be configured with different parameters to achieve various frequency passes. In legitimate applications, the functions using sensor data as the input typically fulfill three roles: (1) they perform data collection and statistical analysis (e.g., counting steps). (2) They make intelligent decisions on behalf of the user for sensitive operations (e.g., making emergency phone calls when a fall is detected). And (3) they trigger actuators to alert or provide feedback to the user when specific events occur (e.g., vibration notification when meeting an exercise goal). They correspond to the three branches activated once the normal pattern is satisfied, as shown in Figure 2.

In modern smartphones, sensors facilitate intelligent decision-making and enable automated execution [36]. In other words, sensor data can substitute for the user's role in both judgment and action. Consequently, many sensitive operations activated via other APIs might appear unreasonable in static analysis, but they seem perfectly plausible when based on sensor data. Consequently, SensorBomb can automatically analyze the host app, and then embed the malicious branch into the host app as shown in Figure 2 after the trigger pattern is satisfied. It leverages the same sensor data, and the sensitive operation is "reasonable" to evade static analysis.

Next, SensorBomb needs to devise a way to transmit that trigger through the sensors. Covert transmission channels are a promising approach, but existing schemes rely on external emitters and lack awareness of other sensor states, so they can only modulate a single, predetermined sensor. As a result, sensor anomaly detection like 6thSense [21] and SBTDDL [22] can reliably detect the covert channels. Instead, we introduce novel onboard covert channels that (1) eliminate any need for external hardware or physical proximity and (2) embed trigger signals within legitimate actuator behaviors. As shown in the onboard covert channel in Figure 2, the trigger pattern is transmitted by the legitimate actuator pattern. Consequently, in both anomaly detectors and end users' perception, the sensor activity generated by the channels is indistinguishable from normal actuator-driven events.

Finally, sensors continuously and passively collect data, making them vulnerable to noise and interference. Although the filter mitigates much of this noise, external factors (e.g., user behaviors) can still overwhelm the trigger signal or induce

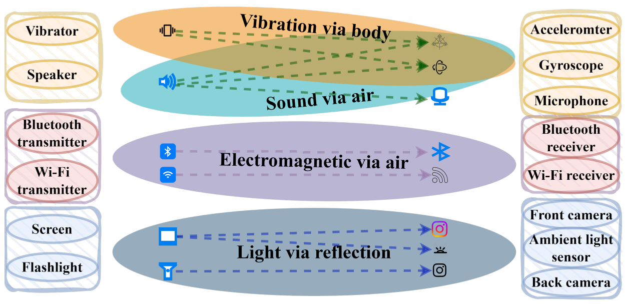


Fig. 4: Available actuator-sensor pairs forming onboard covert channels across various media.

false activations. To address this, SensorBomb leverages the false branch of the malicious branch to infer the sensor's current status in real time. This provides a precise timing reference for triggering, which boosts success rates, while maintaining a statistical collection of normal sensor behavior to filter out, thereby achieving 0 false triggers.

B. *Feasibility Study*

Sensors and Actuators Usage. The motion sensors have been widely used in various apps such as fitness and health tracking, gaming, gesture control, and navigation [37]. Besides, we also analyze the usage of other popular sensors by referencing the 49 categories listed in the Android Play Console [38]. Using the *google_play_scraper* tool [39], we retrieved the top 30 (Google's maximum, but may fluctuate) apps in each category and identified the permissions required by each app. As shown in Figure 3, *vibrator*, *camera*, and *microphone* are the top three sensors and actuators across a broad range of app categories. Notably, the vibrator is used in an average of 71% of the apps, and in three specific categories, its usage is universal (100%).

Web APIs are the most prevalent mechanism for achieving the desired actuator control. For example, JavaScript's Navigator object [40] offers a suite of APIs that enable the control of actuators through the use of predefined patterns. e.g., *Navigator.vibrate()* is compatible with Chrome, Firefox, Opera, and Webview on Android [41]. Some actuators, such as the screen and speakers, can be manipulated through media content, like adjusting sound volume or displaying different images. Currently, these web APIs and media manipulations have no restrictions.

Onboard Physical Covert Channel. Physical covert channels are primarily studied to transmit information between devices or inter-apps. Consequently, external-emitter covert channels have been studied extensively [42], [33]. Due to Android's strict background management [43], few onboard covert channels for inter-app communication have been explored. Because it requires one app to run in the background, where sensor access and actuator control are heavily restricted. To date, only the vibration–accelerometer pairing has been studied [44].

We expand this scope by identifying a diverse set of onboard actuator–sensor combinations that enable a wider range of SensorBomb instances. As shown in Figure 4, we build on four primary physical media for covert channels. Besides vibration, sound-based channels have been validated by motion-sensor eavesdropping attacks [45], and we introduce a novel light-reflection channel whose feasibility is demonstrated by our prototype (§VII-C). Finally, although electromagnetic channels are well studied, their physical properties, such as power levels and frequency-modulation capabilities, still present a promising avenue for covert communication.

Filter and Pattern Checking. As illustrated in Figure 2, the embedded payload includes not only the trigger and hidden operations but also a filter (e.g., Filter2) and a trigger-pattern matching algorithm. Both the filters and the pattern-checking are built on common algorithms utilized by the same sensor API. For example, Table II compares the filter parameters and pattern-matching algorithms for common accelerometer-based functions and those in our SensorBomb implementation. Because SensorBomb simply reuses well-tested, optimized algorithms with only minor parameter adjustments, it remains indistinguishable from benign code by signature-based detectors or resource-usage anomaly analysis.

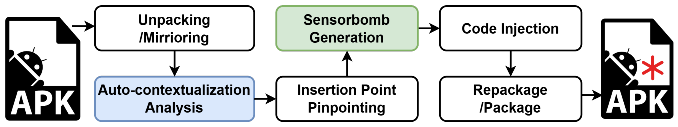


Fig. 5: The injection workflow of SensorBomb

V. The SensorBomb Framework

A. *Overview*

The SensorBomb framework injects guarded code containing hidden sensitive operations into a host application, which is activated when the target sensor's data stream matches a pre-determined trigger pattern. Figure 5 depicts the injection workflow of SensorBomb. Given a host app, the framework unpacks the APK and performs ***static auto-contextualization analysis*** (§V-B): (***i***) it first gathers information about the types of sensors and actuators used by the host app and identifies potential onboard covert channels that can be established and exploited by the SensorBomb. (***ii***) It then selects one covert channel and retrieves the physical specifications of the target sensor and actuator, such as their operating frequencies, to guide sensor data-stream extraction. (***iii***) Finally, it analyzes the source code to extract features such as the sensitive operations invoked and use them to guide context-aware guarded-code generation. In addition, the framework extracts metadata about the app, such as its name and category, to retrieve relevant usage data from the app-behavior knowledge base.

Next, the framework generates the code of the SensorBomb. As shown in Figure 6, an embedded SensorBomb consists of four components: ① sensor data filtering (§V-C1), ② guarded code (§V-C2), ③ trigger-pattern generator (§V-C3), and ④ a communication interface with the remote server (§V-C4). It is worth noting that the guarded code and trigger-pattern generator are designed to perform *dynamic auto-contextualization analysis* once the sensor bomb is embedded and running within the host app. Using the live sensor data stream, the false branch of the guarded code

TABLE II: Components of legitimate sensor data-based applications and SensorBombs

App Type	Frequency	Filter	Function	Threshold Decision	Checking Algorithms
Step Count	lower ($< 3Hz$)	low-pass	peak detection	calibrate with empirical data	timing, pattern matching
Fall Detection	full (0-200Hz)	no	peak detection	predefined with empirical data	followed by a period of inactivity, or ML
Accelerometer Bomb	vibrator ($80 - 100Hz$)	band-pass	peak detection	predefined with empirical data	pattern matching, or ML

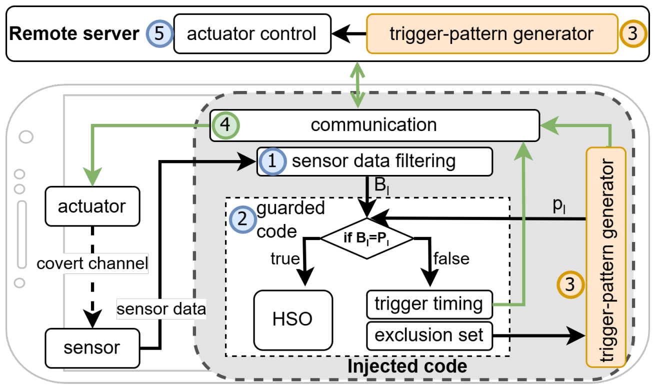


Fig. 6: SensorBomb design overview

evaluates the robustness of the covert channel and informs the remote server of the optimal activation timing, while the trigger-pattern generator extracts exclusion patterns reflecting normal usage scenarios to avoid false activations.

To activate the sensor bomb, the remote server deploys a trigger-pattern generator ③, synchronized with its counterpart on the device, along with an actuator controller ⑤, which issues the actuation commands and simulates cover behavior such as a notification event (§V-D).

Finally, the framework locates the insertion point where the sensor data is used, performs the code injection, and repackages the app with the embedded bomb. In this work, we follow standard mobile-app instrumentation for unpacking, insertion-point discovery, and repackaging, using existing tools [17]. So, we focus on auto-contextualization analysis and context-aware sensor-bomb generation in this section.

B. Static Auto-contextualization Analysis

Given a host app, the SensorBomb framework first uses Androguard [46] to unpack the APK and inspects the bytecode to extract contextual attributes such as the app's package name and the types of sensors and actuators used. It then retrieves the official app-category metadata by querying the Google Play Store using the extracted package name as the identifier via the *google_play_scraper* library.

Next, it applies Soot [47] to identify entry points across the package following prior studies [8], [9], and constructs a control-flow graph (CFG) by exploring all reachable code. Using this CFG, we identify sensitive methods invoked along the sensor-data processing path. We refer to the sensitive-API list in PScout [28], which contains approximately 70 permissions, and identify a subset of 10 permissions that are commonly used in sensor-driven events, such as *send_SMS*, *read_contact*, and *internet*. For each app, we record all sensitive methods requiring these permissions, as well as those invoked within the sensor-processing logic, to guide guarded-code generation.

C. Context-aware Sensor Bomb Construction

A sensor bomb is constructed based on the selected covert channel and embedded into the host app. Next, we describe the design of its four components.

1) Sensor Data Filtering: Given a covert channel, we first determine the filtering frequency for extracting the corresponding digital bitstream from the raw sensor waveform. Each sensor or actuator operates within its own sampling or activation frequency range. For example, accelerometers typically sample at 50Hz by default and operate across 10-200Hz, while vibrators typically operate between 10 and 200 Hz, with a common resonant frequency of 80 to 200 Hz. The chosen filtering frequency f should fall within the intersection of these ranges to ensure that the actuator's modulation can be reliably captured by the sensor and is further constrained by the Android's sampling-rate limits.

Next, we isolate the filtering frequency band associated with the covert channel using a band-pass filter (BPF), which passes frequencies within the target band while attenuating frequencies outside that band. For a raw sensor signal $x(t)$, we compute the magnitude of the filtered signal and smooth it using a windowed average:

$$X(t) = |BPF(x(t))| \times W(\Delta t) \tag{1}$$

where the time window $W(\Delta t)$ must have a minimum width of $2/f$ to capture a sufficient portion of the carrier signal for reliable bit extraction. Finally, we apply a threshold θ to output the binary bitstream $B(t) = \{b(t)\}$, where $b(t) = 1$, if $X(t) \geq \theta$, and $b(t) = 0$ otherwise. The threshold value is determined by the signal conditions of the covert channel. When signals are strong and well separated, we use a fixed threshold determined through preliminary calibration. In environments with high variability, low signal-to-noise ratios, or device-specific inconsistencies, we apply the classical Ridler–Calvard method [48] to iteratively compute an optimal threshold that separates the distributions corresponding to the actuator-off and actuator-on states.

2) Guarded-code Generation: The guarded code consists of a trigger condition, a true branch T_c, and a false branch Q_c. In the code-generation phase, we introduce two novel schemes: one to evade static analysis and the other to capture exclusion patterns and estimate the optimal triggering time through dynamic auto-contextualization.

Static Detection Evasion. The trigger condition is constructed as an *if statement* that matches B_l, the last l bits of the sensor datastream $B(t)$, and the current trigger pattern P_l (§V-C3). To construct T_c and Q_c for evading static detection, we focus on the selection of hidden sensitive operations (HSOs) for T_c, the alignment between the trigger condition and the operations along T_c, and the alignment between the two branches.

TABLE III: Features used in HSOMiner [8] and Difuzer [9]

Feature	Binary Evasion Indicator				Condition-path Relation						Behavior Difference	Trigger Condition
Notion	N	D	R	B	P	S	M_1	S_1	*DF*	*IR*	J, *AD, DD*	*SI*
Meaning	is native code used	is dynamic loading used	is reflection used	are background tasks used	are condition's parameters used	# of sensitive methods invoked	# of methods invoked only in GC	# of sensitive methods invoked only in GC	data flow	implicit relation	behavior difference between branches	condition related to system input

Note: Features defined in Difuzer and HSOMiner are denoted by single letters and italicized double letters, respectively. # denotes count.

Intuitively, detection is performed by a classifier $\mathcal{C}$, typically a deep neural network, which outputs a binary decision $\hat{y} = \mathcal{C}(\mathcal{F}(C))$, where $\mathcal{F}$ is a feature extraction function and C denotes the host app's code (e.g., the global CFG). Hence, our goal is to generate guarded code that, once embedded, preserves the classifier's original prediction.

We consider two SOTA static detectors, HSOMiner [8] and Difuzer [9], and summarize the features they rely on in Table III. In our design, we intentionally avoid common evasion techniques in the guarded code, such as native code ("N"), dynamic loading ("D"), reflection-based code ("R"), and background tasks ("B"), which aim to shift critical logic outside the detector's scope or conceal execution path until runtime. As a result, our guarded code does not exhibit these features. Since we adopt the same sensor used by the host app in the trigger condition, the presence of the system-input (SI) feature, which reflects the use of sensor input, does not introduce any anomaly.

Regarding the conditional-path relation, our strategy is to select sensitive operations that align with the sensor used in the trigger condition to embed into T_c. To support this, we analyze a large corpus of real-world APKs and construct a knowledge base (see Table IX in Appendix A) that records sensitive methods commonly used in legitimate scenarios for three types of sensors, i.e., accelerometers, camera sensors, and motion sensors. We select sensitive methods based on the sensor type and whether they are invoked in the host app as HSOs (obtained during static auto-contextualization), ensuring that the resulting "S_1" remains consistent with values observed in other guarded-code instances. We also constrain the number of HSOs in T_c so that it does not exceed the average number of sensitive methods observed across all guarded-code instances (the average "S"). We deliberately include a code logic that uses the sensor input to ensure the feature "P" remains true. Since the inserted guarded code is small, the feature "M_1", "DF", and "IR" provide little discriminatory power.

Lastly, for the behavior difference features, we ensure that the sensitive methods align with legitimate uses of the selected sensor and that the number of app methods invoked in both branches remains consistent. As a result, the distance features "J", "AD", and "DD" appear normal.

Dynamic Auto-contextualization for Trigger-pattern Enhancement. In the false branch Q_c, we additionally record each l-bit sensor stream B_l that fails to match the trigger pattern. These bitstreams and their corresponding sensor readings reflect legitimate user behavior and must be excluded from valid trigger patterns to guarantee a 0% false-trigger rate. In implementation, we maintain an exclusion-pattern set $\mathcal{P}_E$ initialized with patterns derived from legitimate sensor-data profiles in our real-world APK knowledge base and refined by continuously appending newly observed exclusion patterns.

Dynamic Auto-contextualization for Optimal Trigger Timing. The attack success rate relies on the robustness of the actuator–sensor covert channel, that is, how reliably the sensor can recover the actuator's signal in the presence of channel noise. Channel quality depends on the actuator's emission strength, the sensor's sensitivity, and the level of environmental interference. If the actuator emits $A(t)$ at time t, the sensor observes $S(t) = A(t) + I(t)$, where $I(t)$ captures interference and noise. Intuitively, we aim to identify intervals with a large signal-to-interference ratio $A(t)/I(t)$.

For the target sensor, after converting raw sensor readings into a binary stream, a feasible trigger opportunity corresponds to periods when all observed bits are zero. To ensure stability, we require that every sample within a sliding window of length δt remains zero, that is, for any $t' \in [t, t + \delta t]$, $B(t') = 0$. Only intervals satisfying this condition are considered the optimal timing, as they indicate a stable state without transient fluctuations. Because the decision is made after observing the full window, the effective trigger time occurs at $(t + \delta t)$.

3) Trigger-Pattern Generation: The trigger pattern is an l-bit binary stream matched in the trigger condition to HSO activation. Our design addresses three key tasks: (i) determining an optimal pattern length l that balances transmission errors and false triggers while increasing fuzzing difficulty; (ii) introducing dynamic pattern generation to further reduce the effective search window for fuzzing; and (iii) embedding trigger patterns within legitimate actuator behavior to evade abnormal detection.

Pattern Length. As shown in Figure 6, the bomb extracts B_l from the raw sensor data over the covert channel, while the trigger pattern P_l is output by the trigger-pattern generator. With the increase of l, the probability of transmission errors grows following $1-(1-p)^l$, where p is the channel BER, while the probability of a false trigger decreases exponentially as $e^{-\alpha l}$, where α is the exponential decay rate that varies across sensors (e.g., we adopt $\alpha = 0.2$ for common accelerometers and $\alpha = 0.45$ for camera sensors). Meanwhile, increasing l enlarges the effective search space, reducing the effectiveness of dynamic fuzzing by approximately $l \ln 2$. We should consider all three factors to determine the optimal pattern length:

$$l^* = \arg\max_{l \in \mathcal{L}} \; w_d \, (l \ln 2) - w_f \, e^{-\alpha l} - w_e \big[1 - (1-p)^l\big] \quad (2)$$

where w_d, w_f, and w_e are weights, and $\mathcal{L} = [l_{min}, l_{max}]$ denotes the soft boundary that reflects typical user-actuator interaction intervals. For example, selfie countdowns commonly last 3, 5, or 10 seconds, which yields $l_{min} = 3$ and $l_{max} = 10$. Our experiment shows that for a highly stable channel (e.g.,

$p = 0.1\%$), the optimal trigger pattern length should exceed 5 for camera-based channels and 11 for accelerometer-based channels to keep the false-trigger rate below 10%.

Dynamic Pattern Generation. Dynamic detectors such as IntelliDroid [11] record trigger behaviors to guide random fuzzing. To counter this, we design a dynamic pattern-generation scheme in which the trigger pattern changes over time or with an internal counter. We generate the patterns using a keyed hash function (e.g., HMAC-SHA256), ensuring that successive trigger patterns differ and cannot be predicted. Each new pattern is checked against the exclusion-pattern set before use. Finally, we deploy an identical trigger-pattern generator on the server side, and let the local generator periodically share its exclusion-pattern set to keep both generators synchronized. This enables the server to drive the actuator in a coordinated manner, producing the sensor-data changes required for reliable activation.

Anomaly Detection Evasion. Anomaly detectors based on sensor states, such as 6thSense [21] and SBTDDL [22], learn normal multi-sensor transitions using statistical or deep learning models. Each state is represented as a vector $X \in \mathbb{R}^n$, and the classifier C_θ is trained on labeled examples $(X^{(j)}, y^{(j)})$.

To evade detection, the state vector produced by the covert channel must preserve the same features as those produced by legitimate actuator behavior, ensuring $C_\theta(X_{\text{covert}}) = C_\theta(X_{\text{legitimate}})$. Consequently, the actuator behavior induced by the trigger pattern must mimic that observed in normal usage so that the resulting sensor states appear benign.

For continuously sampled sensors, such as accelerometers, state transitions are determined by comparing $\bar{a} = \frac{1}{T}\int_{t_0}^{t_0+T} a(t)\,dt$, the average reading over a time window T, against a threshold τ. As a result, the sensor-state changes caused by a SensorBomb depend on the proportion of bits '1' in the trigger pattern relative to this detection threshold. Given τ, the number of '1's required to induce a state change is $\tau l/(\Delta t \cdot I_a)$, where I_a is the accelerometer reading under continuous vibration and Δt is the sampling window. Therefore, the trigger pattern must be crafted to mimic legitimate actuator behavior so that the resulting sensor states satisfy the anomaly-evasion condition.

4) Communication with Remote Server: In the sensor bomb, we embed a lightweight communication layer to support interaction with the remote server. In one direction, the bomb sends the optimal triggering time and the latest exclusion trigger patterns to the server to synchronize its trigger patterns generator with the local one. We implement this by hard-coding the server's URL. Because this URL differs from the host app's backend endpoint, it may raise suspicion. Although techniques such as dynamic link redirection or URL masking (e.g., abusing in-app pop-ups) could conceal such modifications, we do not employ them here, as they fall out of the scope. In the other direction, the server sends actuator-control commands, generated based on the current trigger pattern, to operate the target actuator on the mobile device. Many Android applications embed a WebView [46], an official Android system component that provides a built-in browser engine for rendering HTML and JavaScript content within native applications. Therefore, we leverage the WebView API to send commands to the actuator.

Algorithm 1: Inserted code of a sensor bomb

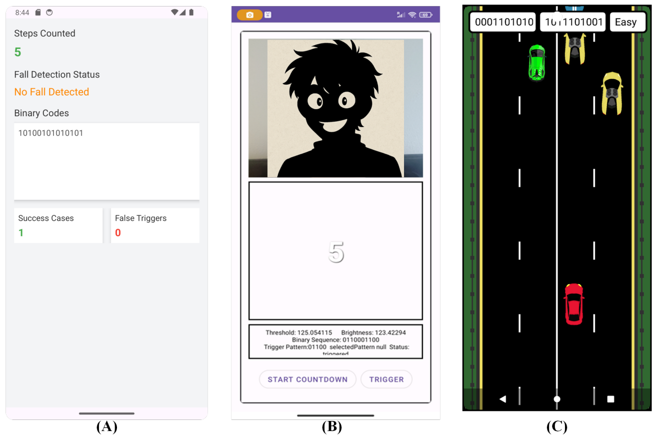
```
Data: Raw sensor data stream $S_t$; filter params
      $(f_2, \Delta t_2, \theta_2)$; selected HSO; pattern lengh $l$
1:  webView.loadUrl(https://server.com)
2:  Function SensorDataProcessingFunction():
3:      data ← getSensorValues($S_t$)
4:      filter1 ← applyFilter($f_1, \Delta t_1, \theta_1$)
5:      normSig ← filter1(data)
6:      if normSig.match(normalPat) :
7:          benignBehavior()
8:      filter2 ← applyFilter($f_2, \Delta t_2, \theta_2$)
9:      trigStream ← filter2(data)
10:     triggerSig ← getLastBits(trigStream, $l$)
11:     if trigSig.match(triggerPat) :
12:         HSO()
13:     else:
14:         excludedPatSet.append(trigSig)
15:         inferTiming()
16: Function GenTriggerPattern():
17:     triggerPat ← DynamicPattern($l$)
18:     assert triggerPat $\notin$ excludedPatSet
19:     return triggerPat
20: Function GenNormalPattern():
21:     normalPat ← NormalPattern()
22:     return normalPat
```

5) Sensor Bomb Implementation: We illustrate the sensor bomb's code logic in Algorithm 1, where the colored segments denote the code injected into the host app's sensor-data processing module. It adapts the app's sensor data filter with parameters for the covert channel, e.g., f_2, Δt_2, and θ_2, to convert the raw sensor readings into a binary stream (Lines 8-9), and extracts the last l bits to construct the activation pattern $triggerSig$ (Line 10). The function $GenTriggerPattern$ generates dynamic patterns (Line 17), which are checked against the $excludedPatSet$ (Line 18), and then matched with the activation pattern in the trigger condition (Line 11). In the false branch, the $triggerSig$ are recorded as exclusion patterns into $excludedPatSet$ (Line 14). The trigger-timing inference notifies the server if *trigSig* is all zeros (Line 15). In the true branch, the HSO is activated.

The sensor bomb, implemented in Java using a compressed style and well-packaged utility functions, is lightweight. For example, the Java snippet from our accelerometer-based prototype in Appendix B consists of roughly four lines for the trigger-pattern generator, four lines for the guarded logic injected into *onSensorChanged*, one line for a potential HTTPS transmission, and one to four lines for field initialization.

D. Actuator Controller

The actuator controller sends commands to activate the actuator, which may raise users' suspicions, as they are sensitive to the context in which an actuator is activated. Therefore, the sensor bomb must simulate actuator behavior that is both contextually appropriate and indistinguishable



8:44

Steps Counted
5

Fall Detection Status
No Fall Detected

Binary Codes
10100101010101

Success Cases
1

False Triggers
0

5

Threshold: 125.054115 Brightness: 123.42294
Binary Sequence: 0110001100
Trigger Pattern:01100 selectedPattern null Status: triggered

0001101010
1011101001

Fig. 7: The Android app interface of (A) Accelerometer Bomb, (B) Camera Bomb, (C) User Behavior Bomb.

from normal usage. We consider three factors affecting the expected actuator behavior: (i) *contextual alignment* C_a, which requires the sensor–actuator pair to be commonly used within the host app's category. For example, fitness apps routinely use vibration notifications together with accelerometer readings. If the pair appears in our knowledge base, we set C_a to 1. (ii) *Behavior alignment* C_b, which requires the actuator action to remain meaningful within the app's functional context. For example, when accelerometer readings stay near-zero for 30 minutes, a fitness app may remind the user to move. Triggering a vibration with such a message appears reasonable. If the injected code provides state information to preserve this meaning, we set C_b to 1. And (iii) *controlled variation* C_v, which requires the actuator pattern to remain within the range of normal variations learned from legitimate usage. Because sensor-based anomaly detection is stricter than user perception, we set C_v to 1 when the anomaly-evasion condition is satisfied. We aim to meet all three goals with information obtained from static auto-contextualization and our knowledge base. An example of the vibration-usage knowledge base is provided in Table X in Appendix D.

VI. SensorBomb Prototypes

The Android malware ecosystem is largely dominated by mass injection and repackaging techniques, which attackers employ for efficient propagation [49]. In contrast, highly stealthy threats are typically delivered through custom-built malware, allowing adversaries to evade signature-based systems and maintain a minimal footprint in public threat datasets [50]. To examine both ends of this spectrum, we developed three custom SensorBomb prototypes to model stealthy, hand-crafted malware and conducted large-scale experiments to evaluate the robustness of our injection mechanism under mass-injection scenarios (see §VII-F). In this section, we present three prototypes. The Accelerometer Bomb prototype represents the broadest class of attacks, as accelerometers are used across virtually all categories of Android applications. The User Bomb prototype enables direct comparison with existing real-world sensor-based logic bombs that rely on user behaviors as triggers. Finally, the Camera Bomb prototype validates our newly discovered onboard covert channel.

Accelerometer Bomb. We developed a host app in the health and fitness category, which exhibits nearly 100% accelerometer usage, to enable fair comparison of attack efficacy against existing logic-bomb implementations. As shown in Figure 4, the corresponding actuators for this category are the vibration motor and the speaker. Moreover, Figure 3 indicates that over 90% of health and fitness apps use the vibration motor. Accordingly, we select it as our actuator. The host app provides step-counting and fall-detection features (Figure 7(A)). Our trigger pattern aligns with the app's normal workflow: when prolonged inactivity is detected, the app reports this condition to the server, which in turn issues a vibration-based reminder prompting the user to exercise.

Camera Bomb. As shown in Figure 3, over 80% of beauty apps use the camera, and among the evaluated light-channel actuator–sensor pairs in Figure 4, the screen–front-camera combination provides the highest robustness. We therefore embed the SensorBomb into a selfie app, leveraging the typical user behavior of holding the phone at a relatively fixed distance from the face, which ensures stable reflection of screen-emitted light into the front camera. In this design, modulated screen colors serve as trigger signals: the light reflected from the user's face is captured by the front camera in real-time. The app monitors facial brightness and, when illumination drops below a predefined threshold, automatically increases screen brightness. This behavior mirrors common practices in commercial selfie apps (e.g., B612), which adjust the screen brightness under low-light conditions to enhance facial illumination and improve photo quality. As shown in Figure 7(B), the upper half of the interface displays the live selfie preview, while the lower half shows the countdown timer, whose background color varies with the trigger pattern.

User Bomb. Prior real-world attacks have relied on uncontrolled user behaviors (e.g., step counts) as trigger conditions, resulting in low and highly variable success rates. In contrast, SensorBomb treats the user as a programmable actuator, and action-control games offer a controlled environment for directing user motions. Our third prototype embeds SensorBomb into a car-racing game: by designing in-game tasks (Figure 7(C)) that lead the player through a predetermined sequence of left and right turns, the hidden sensitive operation is reliably triggered once the induced motion pattern matches the trigger pattern. The detailed configurations and parameters for all three prototypes are summarized in Table IV.

Injecting Code into Existing APKs. When injecting sensitive operations, SensorBomb must add any missing permissions required by the embedded logic. However, introducing new permissions increases the risk of detection, as a simple difference between the original and modified APK can reveal added privileges and expose the injected payload. A practical mitigation is to choose host apps that already request the necessary sensor, actuator, and sensitive operation permissions, thereby reducing the visibility of injected changes. Additional techniques for evading repackaging-detection systems exist but

TABLE IV: Host app, logic, and parameters of three SensorBombs

SensorBomb	Host App	Sensor Data Filtering			Actuator Logic		Trigger Enhancement	
		f	Δt	θ	actuator	behavior logic	l	timing
Accelerometer bomb	step count & fall detection	$80-100Hz$	$>20ms$	fixed	vibrator	notification	$<17bits$	stable
Camera bomb	selfie	$1Hz$	$1s$	optimal	screen	counting down	3, 5, 10	low ambient light
User bomb	car racing game	$50Hz$	on difficulty level	automatic	user	user playing	unlimited	playing

TABLE V: Characteristics of tested Android smartphones

Category	Brand	Model	API ver.	Max screen brightness	Screen size	Ratio	Vibrator axis
Low-end	Xiaomi	Mi 8	29	600 nits	6.2 in	2.08	Z-axis
	Samsung	S9	28	630 nits	5.8 in	2.06	Z-axis
	OnePlus	10T	30	950 nits	6.7 in	2.23	X-axis
	Motorola	Edge+	32	1000 nits	6.7 in	2.22	X-axis
Flagship	Samsung	S23+	33	1750 nits	6.6 in	2.17	X-axis
	Samsung	S24+	34	2600 nits	6.7 in	2.17	X-axis
	Vivo	x200	35	4500 nits	6.7 in	2.22	X-axis

fall outside the scope of this work.

On the Android platform, sensor readings are first stored in a system-managed buffer and then delivered to applications as independent event objects. Multiple components can access these readings concurrently, and the data flow is strictly unidirectional, from the sensor driver to the application. As a result, injecting the guarded code does not interfere with other functionalities in the host app; the injected logic simply receives a copy of each sensor event and processes it locally.

VII. Attack Performance Evaluations

A. Experimental Settings

Experimental Platforms. We evaluate our prototypes on seven Android smartphones with diverse configurations, as summarized in Table V. These devices span multiple popular brands and Android versions. The actuators include two types of vibration motors, one operating along the x-axis and one along the z-axis, and the screen sizes and maximum brightness levels are representative of low- to high-end smartphones.

Dataset. For the Accelerometer Bomb, we use the *Human Activity Recognition Dataset* (KU-HAR) [51], which contains accelerometer readings for 18 activities collected from 90 participants, including 1,945 raw activity samples and 20,750 subsamples. To evaluate the User Behavior Bomb, we recruited 15 volunteers from the University of Kansas to act as users and recorded the number of cars that passed in the car-racing game. For the Camera Bomb, we collected data under five representative illumination conditions.

Evaluation Matrix. We define a trigger pattern generated by the actuator that successfully activates the malicious function as a true positive (TP), and an unsuccessful activation as a false negative (FN). Cases where no actuator trigger is produced but the attack is activated are counted as false positives (FP), while cases with neither trigger generation nor activation are true negatives (TN). To evaluate attack performance, we use the true positive rate (TPR) and false positive rate (FPR), where $TPR = \frac{TP}{TP+FN}$ and $FPR = \frac{FP}{FP+TN}$.

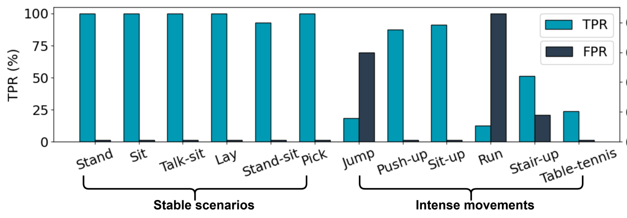


Fig. 8: TPR and FPR of an external emitter logic bomb.

In sensor bombs, the FPR is generally low due to the large number of TNs, but to avoid unintended activations that could crash the app or alert the user, the FPR must be kept at 0%. In contrast, occasional FNs do not harm system functionality, so a perfect TPR is desirable but not strictly required. There is an inherent trade-off: achieving 100% TPR demands more restrictive trigger timing, which attackers can tune based on their operational goals.

B. Evaluations of the Accelerometer Bomb

We compare our method with the vibration-channel approach of [20]. To reduce transmission loss and environmental noise, we adopt an optimized version of their setup in which a smartphone emitter is placed in direct contact with the target device, ensuring strong mechanical coupling. For fair comparison, we fix L=14 and Δt=0.07, to guarantee robust performance across both external and onboard covert channels.

Prior vibration–accelerometer logic bombs report high TPRs under stable conditions, consistent with our results in Figure 8. However, in realistic usage, such stability makes the induced vibrations noticeable, forcing attackers to operate in naturally noisy environments to mask them. Under these dynamic conditions, the TPR drops sharply (Figure 8) because vigorous motion produces accelerometer readings that overwhelm the vibration signal, an inherent limitation that cannot be compensated for. In contrast, the Accelerometer Bomb monitors real-time sensor states and triggers only when the device is stable, producing no activations during movement. This maintains a high TPR, avoids user suspicion, and leverages legitimate actuator behavior for stealth.

The same observation holds for FPR. In stable conditions, both the traditional attack and the Accelerometer Bomb achieve a 0% FPR (Figure 8). Traditional schemes avoid using simple patterns (e.g., all-zeros or all-ones) because they are prone to false triggers. However, in motion-heavy scenarios, accelerometer readings fluctuate dramatically, causing the traditional approach to exhibit a sharply increased false-trigger probability, for example, an FPR of 0.4% while running (Figure 8). In contrast, the Accelerometer Bomb consistently maintains a 0% FPR.

TABLE VI: TPR and FPR of Camera Bomb

Scenarios		Dim bedroom	Normal light bedroom	Bright living room	Daylight office	Sunlight
Times/TPR	Low-end	120/100%	108/95.4%	8/87.5%	0/-	0/-
	Flagship	10/100%	10/100%	0/-	0/-	0/-
FPR	All	0%	0%	0%	0%	0%

Note: dim bedroom: no lights or a night light; normal light bedroom: one floor lamp; bright living room: overhead brighter bulb and TV screens.

C. Evaluations of the Camera Bomb

Light-channel covert communication is rarely practical because it typically requires strict control over the external light source's angle, distance, and intensity [20]. Prior work shows that effective transmission demands angles above $45°$ (ideally $90°$) and a distance tuned to the source's brightness and orientation. However, selfie use naturally satisfies these constraints: our experiments show that users often hold the phone at roughly $90°$ to their face, which occupies more than 50% of the camera frame at a distance of $\sim 20 - 40cm$. This stable geometry enables Camera Bomb to reliably build the covert channel and identify optimal trigger windows.

We evaluated the Camera Bomb under five typical lighting conditions (Table VI). Each participant performed two trials on every test device, producing 120 total tests on low-end phones. In dim-bedroom conditions, every moment fell within the optimal trigger window, yielding 120 attempts and a 100% TPR. In normal nighttime bedroom lighting, 12 attempts were outside the optimal window, resulting in a 95.6% TPR. All misses occurred on two devices with low maximum brightness when volunteers stood near external lights, reducing reflected screen illumination. No triggers were attempted under office lighting or direct sunlight, where ambient illumination overwhelms the screen's output, making activation infeasible.

Because users typically remain still during selfies and the background is stable, the resulting sensor patterns tend to be entirely '1's or '0's. To prevent false triggers caused by sudden background changes, we proactively add all binary patterns containing exactly one contiguous run of '1's with all other bits set to '0' to the exclusion set. In our experiments, the FTP remained at 0%. These excluded patterns represent only 5.37% of all possible patterns, so the reduction in the usable trigger-pattern space is minimal.

D. Evaluations of the User Bomb

We compare our user behavior bomb with the Cerberus bomb [18], which uses a step-count trigger to activate SMS control and contact harvesting. Cerberus uses a simple policy: if the device is used by a real person, step counts will eventually increase until a preset threshold is reached. However, this creates a fundamental trade-off between success rate and fuzzing time. As shown in Figure 9(A), increasing the step threshold tightens the trigger condition but sharply reduces the fraction of users who naturally meet it. According to [52], the global average daily step count is 4,961 with a standard deviation of 2,684. Setting the trigger at 5,000 steps yields a 50% success rate with only 0.46 hours of fuzzing time.

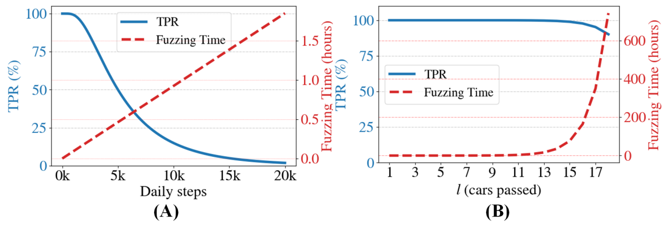


Fig. 9: TPR comparison between User Bomb and Cerberus.

Extending the fuzzing time to 1.85 hours requires raising the threshold to 20,000 steps, a level reached by fewer than 1% of users, which dramatically lowers the success rate.

In contrast, the User Bomb uses a sequence of user actions as its trigger, dramatically increasing the time required for successful fuzzing while still maintaining a high TPR. In this setting, performance is driven largely by user engagement, so the app must be interactive enough to guide users through the required actions. To strengthen engagement, we allow users to select their preferred difficulty level. As shown in Figure 9(B), with l=16, the TPR remains above 95% while the required fuzzing time reaches 163.8 hours. The fuzzing cost grows exponentially with longer action sequences: increasing l by just 2 reduces TPR slightly to 90% but pushes fuzzing time beyond 700 hours. This design provides strong resistance to fuzzing attacks while consistently achieving a high TPR.

Unlike Cerberus, which ignores FPR and accepts any trigger due to its inherently low success rate, our user bomb explicitly controls for false positives. The probability that an l-bit trigger pattern matches a random l-bit segment in a binary stream is $(1/2)^l$. For l=16, this is only 0.0015%. With an average of three cars per second, a mistrigger occurs roughly once every 3.03 hours. To eliminate these FPs, the server applies an l-bit sliding window and accepts a trigger only when the most recent l bits exactly match the current pattern, or match it with the last bit flipped (to accommodate a player's last-bit error). In a 20-hour simulation, this approach achieved a 0% FPR.

E. Threats to Validity

In the main experiments, we intentionally used low-end phones, as SensorBomb is expected to be less effective on them due to their weaker actuators, providing a conservative assessment of attack effectiveness. The feasibility of the vibrator-accelerometer and user-accelerometer channels across diverse hardware has been validated in prior research [44], [53] and deployment [18]. Our experiments with three flagship devices (Samsung S23+, Samsung S24+, and Vivo X200, as summarized in Table V) also confirmed that the vibrator-accelerometer and user-accelerometer channels behave similarly as the low-end phones. However, the screen-camera channel has not been studied/tested on diverse devices.

We further evaluated the screen-camera channel on three new flagship devices. Note that the advertised peak brightness occurs only under highly restricted conditions, such as HDR boosts with less than 5% active screen area, whereas the sustained full-screen luminance in daily use is much lower (typically 1,000–2,000 nits) [54]. Meanwhile, the CameraX

API [55] introduces a higher level of abstraction, and recent flagship phones use advanced AE (auto-exposure) and AWB (auto–white balance) for front cameras to normalize brightness. These system-level optimizations reduce the observable impact of controlled screen-luminance modulation.

As reported in Table VI, SensorBomb achieved 100% ASR in a dim bedroom and maintained high ASR in a normally lit bedroom on all devices. In the normal bedroom, the measured brightness of a black screen was higher than that of a white screen due to aggressive AE/AWB compensation. Our threshold-adaptation mechanism (§V-C) correctly handled this as a valid trigger opportunity and automatically adapted to the compensation. Meanwhile, in bright living rooms, AE and AWB effectively stabilize brightness so that screen-brightness modulation no longer produces observable changes in camera input. Flagship devices fail to establish a feasible attack window. The success cases came from the low-end device, whose lack of advanced front-camera AE/AWB algorithms allowed the screen-brightness variations to remain detectable.

F. Large-scale Injection

We identify a large set of Android apps, inject SensorBomb, and evaluate its impact on their normal operation.

Data Preprocessing. We first performed static analysis to identify APKs from AndroZoo [56] that meet the requirements of both SensorBomb and the AndroBomb injection tool [57]: (1) Apps must employ the sensors and actuators used by SensorBomb. (2) Apps must target Android API levels above 22, as the Soot framework [47] used by AndroBomb cannot reliably handle lower API levels. (3) The sensor-data processing classes must use fully qualified names consistent with the app's package name, as strictly required by AndroBomb. (4) Some methods invoked by AndroBomb are incompatible with different Android API versions. We eliminated incompatible APKs. We randomly selected 5,000 APKs and enforced five filters with static analysis using Androguard [46]. 1,729 APKs remained for SensorBomb injection.

Results. SensorBomb was successfully injected into 1,403 out of 1,729 APKs. The failures were primarily caused by unexpected exceptions thrown by the AndroBomb tool, most notably buffer-underflow crashes in ManifestEditor. For comparison, AndroBomb reported a 54.5% overall failure rate [57], including apps targeting API levels below 22 ($<$ 30% in our dataset and eliminated in static analysis) and crashes in the ManifestEditor (18.9% in our dataset). To confirm that SensorBomb did not introduce additional failures, we tested all 326 failed APKs with the original AndroBomb. They all exhibited the same crash, confirming that the failures stem from unexpected exceptions in AndroBomb.

App Operations. We developed a lightweight automated testing pipeline to evaluate the 1,403 APKs with SensorBomb injected. The pipeline uses ADB-based batch operations to install each APK, launch the app, wait 45 seconds for initialization, and then collect logs, exit codes, and device-state information to determine whether installation and execution succeeded [58]. A small UI script is used to dismiss permission

TABLE VII: Difuzer evasion testing results

HSO	contacts	calls	photos	location	microphone	network	bluetooth	SMS
Acc bomb	Y ∣ ✓	Y ∣ ✓	Y ∣ ✓	Y ∣ ✓	Y ∣ ✓	Y ∣ ✓	Y ∣ ✓	Y ∣ ✓
Cam bomb	Y ∣ ✓	N ∣ ×	Y ∣ ✓	Y ∣ ✓	Y ∣ ✓	Y ∣ ✓	N ∣ ×	N ∣ ×
User bomb	N ∣ ×	N ∣ ×	Y ∣ ✓	N ∣ ×	Y ∣ ✓	Y ∣ ✓	Y ∣ ✓	N ∣ ×

Note: Y means selected by SensorBomb, while N means not; ✓ means successfully evade Difuzer and × means not.

dialog (clicking "OK") and uninstall the app after testing. In this automated evaluation, 100% of the injected APKs ran normally, indicating no impact on baseline app behavior.

Usability. Following AndroBomb [57], we design a manual evaluation process to assess SensorBomb's impact on host-app usability. We randomly sampled 149 out of 1,403 infected apps (with 99% confidence level at 10% Margin of Error). We first confirmed that they were correctly infected and the trigger could be activated. To systematically evaluate the usability of heterogeneous apps, we categorized them into a smaller set of buckets based on their interaction patterns, and developed a manual evaluation protocol for each bucket of apps (refer to Appendix C for details). Four graduate students, including three not involved in this paper, collectively evaluated all 149 apps. Each app was examined by all four students, with 5–6 minutes spent per app per evaluator. Manual evaluation confirmed that all SensorBomb-infected apps installed and ran normally without any observable issues.

VIII. SensorBomb against Detectors

A. Static Analysis

Evaluation Tools. To our best knowledge, there are three static analysis detectors for Android logic bombs: TriggerScope [7], HSOMiner [8], and Difuzer [9]. HSOMiner did not release its code. TriggerScope [59] cannot detect SensorBomb because it is limited to a predefined set of trigger conditions (time, location, and SMS). We extended TriggerScope to additional sensor-based triggers and noticed markedly high false-positive rates, which is consistent with the observations in [60]. Therefore, we used Difuzer in our evaluation.

Results. We evaluated 8 common sensitive operations across 3 prototypes. We use SensorBomb to select operations to embed. Each infected app was analyzed with Difuzer for HSO. As shown in Table VII, every HSO chosen by SensorBomb successfully evaded Difuzer. The Accelerometer Bomb bypassed all 8 checks. The User Bomb's motion-based trigger is unsuitable for 3 operations, and the Camera Bomb is unsuitable for 2. These results demonstrate that naively using sensor data as triggers does not guarantee full evasion or detection: outcomes depend on both app context and HSO type, whereas SensorBomb consistently identifies and embeds only those operations that evade static analysis.

Next, we attempt to inject multiple HSOs into the host app. Results show that: (1) when 2 HSOs were injected, Difuzer cannot identify any malicious behaviors; (2) the User Bomb and Camera Bomb become detectable when 3 operations were injected; and (3) the Accelerometer Bomb is detected when 4 HSOs were injected. This suggests that triggering multiple sensitive behaviors substantially increases the risk of detection.

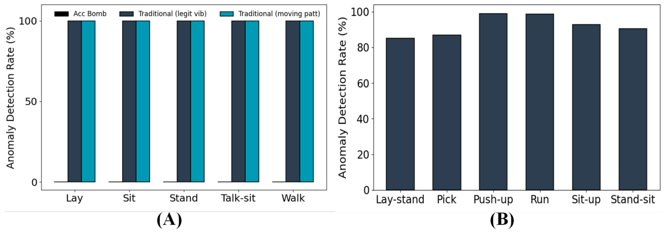


Fig. 10: Anomaly detection rate against (A) Accelerometer bomb and traditional accelerometer-based logic bomb in stable scenarios, and (B) the traditional accelerometer-based logic bomb in moving scenarios.

B. Anomaly Detection

Evaluation Tools. Both 6thSense [21] and SBTDDL [22] report effective detection of sensor-based logic bombs. However, neither of them was open-sourced. Hence, we followed their designs to collect data and train detectors from scratch.

Data Collection. We followed the protocol in SBTDDL [22] for data collection: 31 activity sessions of 5 minutes each; continuous sensors sampled at 1 Hz and state-change sensors sampled at 5 Hz. We recruited five volunteers and collected 9,300 minutes of sensor data across four smartphones. While neither 6thSense or SBTDDL specified whether vibration artifacts were used, we still incorporated them into our dataset. The predefined vibration patterns employed in our experiments are listed in Table X in Appendix D.

Results. We ran 100 trials for each SensorBomb variant, and in every case, the detector classified the actuator behavior as benign. The sensor status related to Camera Bomb is whether the camera is on or off, and the varied screen-color patterns have no effect on that state; consequently, the Camera Bomb never triggers the anomaly detection. The User Bomb's activation simply replicates legitimate user interactions, which are inherently labeled as benign. Consequently, we concentrate on the Accelerometer bomb. Following 6thSense [21], we collect 1 second of accelerometer data and compute the state change according to §V-C. Thanks to the wide variety of predefined vibration patterns, when converted to binary according to SensorBomb's parameters, predefined vibration patterns exhibit a wide range of "1" proportions: from the minimal single-bit pulses of a 2 ms "tick" or 6 ms "click" up to sustained bursts of 300, 500, or even 1000 ms (i.e., sequences of l consecutive "1" bits). In our experiments, although random trigger patterns produced varying densities of "1" bits, none were detected as anomalies by the sensor-anomaly detector, as shown in Figure 10(A).

In contrast, logic bombs relying on external emitters cannot manipulate the full sensor context. For example, legitimate vibrations typically coincide with changes in screen-on state, so even if an external device replays a legitimate vibration pattern, the mismatch with other sensors exposes the attack and yields 100% detection in stable conditions (Figure 10 (A)). Although an external emitter can also mimic the accelerometer signature of a locked phone in motion to evade anomaly detection during vigorous movement, replaying these patterns when the device is stationary creates inconsistencies with other sensor readings (e.g., the gyroscope) and is flagged as malicious. Furthermore, executing the attack while the phone is in motion may sometimes bypass anomaly detectors as shown in Figure 10(B). However, as shown in §VII-B, the TPR declines sharply, rendering the attack impractical.

Moreover, anomaly detection cannot rely on pattern-based heuristics, since attackers are free to craft arbitrary vibration sequences. As a result, it is impossible to assemble a comprehensive dataset that clearly separates benign from malicious patterns. Even a coarse classifier would be vulnerable to denial-of-service attacks, whereby an adversary floods the detector with custom vibration sequences to overwhelm it.

TABLE VIII: Theoretical fuzzing time for different Bombs.

SensorBomb	n_θ	n_f	l (bits)	Δt (s)	T_{max}
Acc bomb	fixed	fixed	13–16	0.02–0.06	104.2 hours
Cam bomb	2	20	3, 5, 10	1	115.8 hours
User bomb	fixed	fixed	16	0.5	145.6 hours
Sensor value	range		scale	sampling rate	T_{max}
Ambient light [61]	0–188,000 lux		0.045 lux	50 Hz	2.32 hours
Step count [18]	0–20,000 steps		1 step	3 steps / s	1.85 hours

C. Sensor Data Fuzzing Evasion

Methodology. While no off-the-shelf tool exists to directly measure the fuzzing time of sensor-based logic bombs, the principles in [25], [11] can be used to fuzz sensor data. Meanwhile, pure black-box fuzzing over all possible sensor readings is computationally impractical [20]. Therefore, we assume gray-box fuzzing: the defender knows the number of thresholds (n_θ) and their approximate values, e.g., if a $0.06 m^2/s$ threshold is used to binarize accelerometer readings, the defender fuzzes around it with values such as $0.04, 0.07$.

The SensorBomb derives its trigger from a dynamically generated bitstring (trigger pattern) extracted from the sensor stream. As these patterns appear to be pseudorandom, heuristics-guided data generation becomes ineffective. We adopt a data-sweeping strategy to cover the full parameter space defined by frequency f, time window width Δt, threshold θ, and trigger pattern length l. In theory, the time required to exhaustively cover this sensor-stream space is $T = n_\theta \cdot n_f \cdot \sum_{l=l_{min}}^{l_{max}} \sum_{k=1}^{i} 2^l \cdot L \Delta t_k$. Meanwhile, if we adopt the same data-sweeping strategy to fuzz existing logic bombs that trigger on a single sensor value, the required time is $T = \frac{R}{s f_s}$, where R is the sensor's value range, s is the scaling granularity, and f_s is the sampling rate f_s.

Theoretical Estimation. Table VIII shows the theoretical maximum fuzzing time T_{max} for the SensorBombs and 2 sensor-value-based logic bombs, assuming that the dynamic triggers are not updated during fuzzing, i.e., the trigger updating cycle t_x is larger than T_{max}. The maximum fuzzing time for the two sensor-value bombs are approximately 2 hours, while that of SensorBomb variants all exceed 100 hours.

Experimental Results. For single–sensor-value logic bombs, we randomly selected a sensor as the trigger, fuzzed it with non-repeating random values within its valid range, recorded

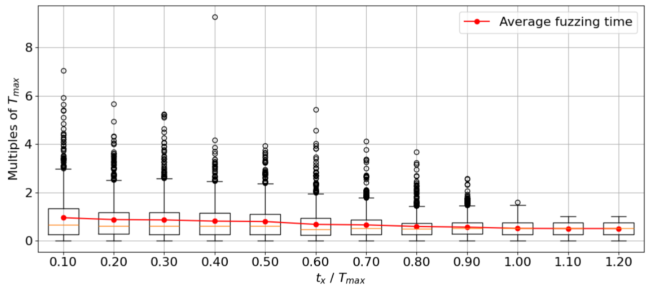


Fig. 11: Smaller ratio of t_x and T_{max} increase fuzzing time

the number of iterations required to hit the trigger, and multiplied this by $1/f_s$ to estimate the actual fuzzing time with real hardware sensors. With 10,000 experiments, we found that the average fuzzing time to trigger the logic bomb is about half of the theoretical maximum. For SensorBomb, we fuzz the app by repeatedly feeding it with non-repeating random sensor readings. When $t_x \geq T_{max}$, the average fuzzing time to trigger the SensorBomb is about half of the theoretical maximum shown in Table VIII. Meanwhile, when $t_x=0.1 \times T_{max}$, about half of the bombs were never triggered at the end of the fuzz (all valid sensor readings are covered). This is caused by the dynamic triggers: if the trigger switches to patterns that have already been fuzzed, the fuzzing process may never reach the new pattern within a single fuzzing pass.

To further examine the impact of dynamic triggers in fuzzing, we first accelerate the process by injecting binarized trigger patterns into the infected app at a higher frequency, instead of feeding real sensor readings at the regular hardware sampling rate. We then swept t_x from $0.1T_{\max}$ to $1.2T_{\max}$ in steps of $0.1T_{\max}$, and repeated the fuzzing process for each t_x for multiple passes until the trigger is eventually activated. We repeat the experiment for 10,000 trials. As shown in Figure 11, when $t_x/T_{\max} \geq 1$, the fuzzer, on average, activates the trigger after testing half of the patterns, i.e., the average fuzzing time stabilizes at $T_{\max}/2$. However, when $t_x/T_{\max} < 1$, the average fuzzing time increases, and some triggers are not activated in the first fuzzing pass. As t_x decreases, an increasing fraction of triggers requires multiple fuzzing passes. When $t_x/T_{max} = 0.1$, the average fuzzing time reached T_{max}, effectively doubling the expected fuzzing time.

IX. Potential Defense

From the evasion analysis, we can observe that static analysis is unlikely to detect SensorBomb, since the guarded code uses only common language features, operations, and APIs that are well adopted by benign apps.

A potential defense is to construct a whitelist to allow only actuator patterns found in legitimate apps. In an initial exploration, we examined app-embedded vibration behaviors using static analysis on 2,000 APKs. 1,396 apps invoked vibrations, and 98% adopted patterns consistent with Table X. Hence, there is potential to derive and enforce a whitelist of commonly used patterns. If the attacker wants to adapt to an OS-level filter/firewall of vibration patterns, they could only use common, whitelisted patterns as the trigger. Consequently, the zero-FPR guarantee is likely sacrificed, and fuzzing (as the number of whitelisted patterns is limited) may be employed to detect the trigger. However, this defense suffers from the potential false-blocking of benign vibration patterns, while the attacker may still combine multiple benign patterns as a trigger. Meanwhile, it is challenging to establish pattern-based whitelists for screen-based actuators, due to the variability and semantic flexibility of multimedia content. AE/AWB may offset brightness changes and compensate for low lights; hence, they have the potential to be employed in SensorBomb defense. However, designing effective and robust defenses that do not interfere with normal app usage still requires extensive research, which we leave for future work.

X. Conclusion

In this paper, we proposed SensorBomb , a practical and stealthy Android logic bomb. Our approach enables logic bombs to bypass all existing detection techniques and targeted updates, achieving a high TPR with 0% FTR. The efficacy and feasibility of SensorBomb are validated through 3 prototypes built on an easy-to-implement framework. In addition, large-scale injection experiments demonstrate that SensorBomb can be injected into existing applications without impacting their normal functionality. Finally, we argue that Sensor-Bomb can be extended to other platforms, potentially giving rise to multiple variants, underscoring the need for proactive defenses to mitigate potential real-world risks.

Acknowledgment

Ye Wang, Bo Luo, and Fengjun Li were supported in part by NSF IIS-2014552, DGE-1565570, and the Ripple University Blockchain Research Initiative. The authors would like to thank the anonymous reviewers and the shepherd for their valuable comments and suggestions, and the volunteers in the user study.

References

[1] U.S. Attorney's Office, Eastern District of North Carolina, "Georgia man sentenced for compromising u.s. army computer program," Sep. 2018, accessed: 2024-01-12.

[2] D. Kushner, "The real story of stuxnet," *IEEE Spectrum*, vol. 53, no. 3, p. 48, 2013.

[3] D. Fiser, "Attacking the supply chain: Developer," https://www.trendmicro.com/en_us/research/23/a/attacking-the-supply-chain-developer.html, Jan. 2023, trend Micro Research.

[4] National Association of Broadcasters, "Analysis of the south korean malware attack," https://www.nab.org/xert/scitech/pdfs/20130328_South%20Korean%20Malware.pdf, National Association of Broadcasters, Tech. Rep., 2013, accessed: 2024-01.

[5] Y. Yao, L. Zhu, and H. Wang, "Real-time detection of passive backdoor behaviors on android system," in *2018 IEEE Conference on Communications and Network Security (CNS)*, 2018, pp. 1–9.

[6] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste, "Malware dynamic analysis evasion techniques: A survey," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–28, 2019.

[7] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 377–396.

[8] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps." in *NDSS*, 2017.

[9] J. Samhi, L. Li, T. F. Bissyandé, and J. Klein, "Difuzer: Uncovering suspicious hidden sensitive operations in android apps," in *International Conference on Software Engineering*, 2022.

[10] Q. Zhao, C. Zuo, B. Dolan-Gavitt, G. Pellegrino, and Z. Lin, "Automatic uncovering of hidden behaviors from input validation in mobile apps," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[11] M. Y. Wong and D. Lie, "Intellidroid: a targeted input generator for the dynamic analysis of android malware." in *NDSS*, 2016.

[12] D. Kirat, G. Vigna, and C. Kruegel, "{BareCloud}: Bare-metal analysis-based evasive malware detection," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 287–301.

[13] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," *Botnet Detection: Countering the Largest Security Threat*, pp. 65–88, 2008.

[14] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.

[15] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, 2013.

[16] M. Alecci, J. Samhi, L. Li, T. F. Bissyandé, and J. Klein, "Improving logic bomb identification in android apps via context-aware anomaly detection," *IEEE Trans. on Dependable and Secure Computing*, 2024.

[17] J. Samhi, T. F. Bissyandé, and J. Klein, "Triggerzoo: a dataset of android applications automatically infected with logic bombs," in *International Conference on Mining Software Repositories*, 2022.

[18] ThreatFabric, "Cerberus - a new banking trojan from the underworld," https://www.threatfabric.com/blogs/cerberus-a-new-banking-trojan-from-the-underworld, Aug 2019.

[19] A. Martinez, "Android malware that triggers only when it detects motion," https://thethreatreport.com/android-malware-that-triggers-only-when-it-detects-motion/, Feb. 2019, accessed: 2025-07-20.

[20] R. Hasan, N. Saxena, T. Haleviz, S. Zawoad, and D. Rinehart, "Sensing-enabled channels for hard-to-detect command and control of mobile devices," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, 2013, pp. 469–480.

[21] A. K. Sikder, H. Aksu, and A. S. Uluagac, "{6thSense}: A context-aware sensor-based attack detector for smart devices," in *26th USENIX Security Symposium*, 2017, pp. 397–414.

[22] S. Manimaran, V. Sastry, and N. Gopalan, "Sbtddl: A novel framework for sensor-based threats detection on android smartphones using deep learning," *Computers & Security*, vol. 118, p. 102729, 2022.

[23] X. Wang, S. Zhu, D. Zhou, and Y. Yang, "Droid-antirm: Taming control flow anti-analysis to support automated dynamic analysis of android malware," in *ACSAC*, 2017.

[24] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques." in *NDSS*, 2016.

[25] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making malory behave maliciously: Targeted fuzzing of android execution environments," in *International Conference on Software Engineering (ICSE)*, 2017.

[26] E. B. Yi, A. Maji, and S. Bagchi, "How reliable is my wearable: A fuzz testing-based study," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018.

[27] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra *et al.*, "Caiipa: Automated large-scale mobile app testing through contextual fuzzing," in *Annual international conference on Mobile computing and networking*, 2014.

[28] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *ACM conference on Computer and communications security*, 2012.

[29] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 447–458.

[30] L. Guan, S. Jia, B. Chen, F. Zhang, B. Luo, J. Lin, P. Liu, X. Xing, and L. Xia, "Supporting transparent snapshot for bare-metal malware analysis on mobile devices," in *Annual computer security applications conference (ACSAC)*, 2017.

[31] W. Li, L. Guan, J. Lin, J. Shi, and F. Li, "From library portability to para-rehosting: Natively executing microcontroller software on commodity hardware," in *Network and Distributed System Security Symposium, NDSS*, 2021.

[32] W. Li, J. Shi, F. Li, J. Lin, W. Wang, and L. Guan, "μafl: non-intrusive feedback-driven fuzzing for microcontroller firmware," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1–12.

[33] A. K. Sikder, G. Petracca, H. Aksu, T. Jaeger, and A. S. Uluagac, "A survey on sensor-based threats and attacks to smart devices and applications," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 1125–1159, 2021.

[34] A. K. Sikder, H. Aksu, and A. S. Uluagac, "A context-aware framework for detecting sensor-based threats on smart devices," *IEEE Transactions on Mobile Computing*, vol. 19, no. 2, pp. 245–261, 2019.

[35] P. Hu, H. Zhuang, P. S. Santhalingam, R. Spolaor, P. Pathak, G. Zhang, and X. Cheng, "Accear: Accelerometer acoustic eavesdropping with unconstrained vocabulary," in *IEEE S&P*, 2022.

[36] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell, "A survey of mobile phone sensing," *IEEE Communications magazine*, vol. 48, no. 9, pp. 140–150, 2010.

[37] G. Grouios, E. Ziagkas, A. Loukovitis, K. Chatzinikolaou, and E. Koidou, "Accelerometers in our pocket: Does smartphone accelerometer technology provide accurate data?" *Sensors*, vol. 23, no. 1, 2022.

[38] Google Play Help, "Data safety section in google play - google play console help," 2024, accessed: 2024-10-26. [Online]. Available: https://support.google.com/googleplay/android-developer/answer/9859673?hl=en#zippy=

[39] JoMingyu, "google-play-scraper: Python api to crawl google play store data," Jun. 2024, mIT License. [Online]. Available: https://pypi.org/project/google-play-scraper/

[40] MDN Web Docs, "Navigator - web apis | mdn," 2024, accessed: 2024-11-03. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Navigator

[41] MDN Contributors, "Navigator: vibrate() method - web apis," 2023, accessed: 2023-12-16. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Navigator/vibrate

[42] B. Carrara and C. Adams, "Out-of-band covert channels—a survey," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, pp. 1–36, 2016.

[43] Android Developers, "Background work restrictions," https://developer.android.com/develop/background-work/background-tasks/bg-work-restrictions, 2024, accessed: 2024-11-03.

[44] A. Al-Haiqi, M. Ismail, and R. Nordin, "A new sensors-based covert channel on android," *The Scientific World Journal*, vol. 2014, no. 1, p. 969628, 2014.

[45] C. Bolton, Y. Long, J. Han, J. Hester, and K. Fu, "Characterizing and mitigating touchtone eavesdropping in smartphone motion sensors," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 164–178.

[46] Androguard Project, "Androguard: Reverse engineering and analysis of android applications," https://androguard.readthedocs.io/, accessed: 2025-11-20.

[47] P. Lam, O. Lhoták, L. Hendren *et al.*, "The soot framework for java program analysis: A retrospective," in *CETUS Users and Compiler Infastructure Workshop*, 2011.

[48] T. W. Ridler, S. Calvard *et al.*, "Picture thresholding using an iterative selection method," *IEEE Trans. Syst. Man Cybern*, vol. 8, no. 8, pp. 630–632, 1978.

[49] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 95–109.

[50] E. M. Rudd, A. Rozsa, M. Günther, and T. E. Boult, "A survey of stealth malware attacks, mitigation measures, and steps toward autonomous open world solutions," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 2, pp. 1145–1172, 2016.

[51] N. Sikder and A.-A. Nahid, "Ku-har: An open dataset for heterogeneous human activity recognition," *Pattern Recognition Letters*, vol. 146, pp. 46–54, 2021.

[52] T. Althoff, R. Sosič, J. L. Hicks, A. C. King, S. L. Delp, and J. Leskovec, "Large-scale physical activity data reveal worldwide activity inequality," *Nature*, vol. 547, no. 7663, pp. 336–339, 2017.

[53] I. Hwang, J. Cho, and S. Oh, "Vibecomm: Radio-free wireless communication for smart devices using vibration," *Sensors*, vol. 14, no. 11, pp. 21 151–21 173, 2014.

[54] N. TECH, "What you need to know about smartphone screen brightness — and why incredible peak brightness is mostly marketing," *NEWS.am TECH - Innovations and science*, Nov 2025, accessed: 2025-11-13. [Online]. Available: https://tech.news.am/eng/news/6322/

what-you-need-to-know-about-smartphone-screen-brightness-%E2%80%94-and-why-incredible-peak-brightness-is-mostly-marketing.html

[55] Android Developers, "CameraX Overview," https://developer.android.com/media/camera/camerax, 2023, accessed: 2024-01-06.

[56] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *IEEE/ACM Working Conference on Mining Software Repositories (MSR)*, 2016.

[57] J. Samhi, "AndroBomb (1.0)," 2022. [Online]. Available: https://doi.org/10.5281/zenodo.5907924

[58] Google, "Android debug bridge (adb)," 2025. [Online]. Available: https://developer.android.com/tools/adb

[59] JordanSamhi, "Tsopen," https://github.com/JordanSamhi/TSOpen, 2021, accessed: 2023-12-12.

[60] J. Samhi and A. Bartel, "On the (in) effectiveness of static logic bomb detection for android apps," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 6, 2021.

[61] Analog Devices, *MAX44009: Ambient Light Sensor Data Sheet*, Analog Devices, Norwood, MA, Aug 2013, accessed: 2025-04-12. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/MAX44009.pdf

Appendix

A. Knowledge Base of Sensitive Operations Driven by Sensors

Sensor-driven sensitive operations are numerous and diverse. We list a representative subset in Table IX, focusing on operations commonly used with accelerometers, cameras, and motion sensors in gaming applications.

B. Code Snippet of Accelerometer Bomb

Please refer to Code Snippet 1 for the injected SensorBomb source code in Java. Its size is negligible relative to the codebase of a typical Android app, which usually contains 10–20 lines of code.

C. Manual Testing Protocol

Since AndroZoo does not provide metadata on app categories, we retrieved the official app category information by querying the Google Play Store using the *google_play_scraper* library, based on each app's package name. For apps not found on Google Play, we manually assigned categories by exploring their functions. We have designed the following app buckets and the manual evaluation protocol for each bucket.

Games

- **Example Categories:** Arcade, Puzzle, Action, Casual
- **Primary Interaction Pattern:** Continuous gameplay loop
- **Manual Evaluation:** Play each game for 10–12 minutes; test start/stop, pause/resume, settings; observe stability, ads/IAP behavior, and UI consistency.

Health & Fitness

- **Example Categories:** Fitness, Step Counter, Sleep, Heart Rate
- **Primary Interaction Pattern:** Sensor-based measurement; activity logging
- **Manual Evaluation:** Complete onboarding process; run ≥3 measurement sessions; perform physical activities; confirm metrics update and history logs.

Content Creation & Editing

- **Example Categories:** Camera, Audio Recorder, Drawing, Doc Editors
- **Primary Interaction Pattern:** Create, edit, save artifact
- **Manual Evaluation:** Create a document; apply multiple edits; save/export; reopen to verify persistence; test one share/export path; repeat editing.

Content Consumption

- **Example Categories:** News, Video, Music, Books
- **Primary Interaction Pattern:** Feed browsing; media playback
- **Manual Evaluation:** Load the main feed; open ≥3 items; scroll/navigate; play ≥3 minutes of media; test pause/seek/search.

Communication & Social

- **Example Categories:** Chat, Email, Forums, Social Media
- **Primary Interaction Pattern:** Messaging, posting, commenting, etc.
- **Manual Evaluation:** Create and sign in dummy account; send multiple messages/posts; confirm delivery; test reply/attach/refresh; delete dummy posts.

Shopping

- **Example Categories:** Shopping, Finance
- **Primary Interaction Pattern:** Browse, search, product details, cart, checkout
- **Manual Evaluation:** Browse catalog; open details; add to cart; proceed to final payment (no actual purchase); check navigation and loading.

Utilities & Device Control

- **Example Categories:** File Manager, Cleaner, Launcher, SmartHome
- **Primary Interaction Pattern:** Execute a primary device control function
- **Manual Evaluation:** Run main utility action; verify visible output or logs; reopen app to confirm persistent state.

Education

- **Example Categories:** Language Learning, Kids Learning, Dictionary
- **Primary Interaction Pattern:** Lesson/lookup loop
- **Manual Evaluation:** Start lesson or lookup; complete a short unit; verify feedback/progress; test search or next-step navigation.

Maps, Travel, & Local Services

- **Example Categories:** Navigation, Travel, Local Business
- **Primary Interaction Pattern:** Location-based search and routing
- **Manual Evaluation:** Allow location access; browse/search POI; open details; start route preview or booking flow; confirm stable map loading.

Hybrid, Other

- **Example Categories:** Multi-functional or ambiguous apps
- **Primary Interaction Pattern:** Two or more primary functions with different flows
- **Manual Evaluation:** Identify ≥2 flows; complete both within 12–15 minutes; evaluate navigation, stability, and sensor/permission behavior.

D. Knowledge Base of Vibration Usage

Table X presents the vibration-usage knowledge base that we constructed across Android app categories. The upper section summarizes common vibration trigger conditions, mechanisms, and semantic intents observed across different categories of apps, while the lower section lists the patterns of common, predefined vibration sequences. In Android, vibration patterns are defined as alternating delay–vibration durations, where the first element specifies the initial delay. Accordingly, these patterns typically begin with 0 to indicate immediate activation.

TABLE IX: Legitimate sensor-involved sensitive operations and required permissions

Legitimate Use Context	Sensitive Operation Triggered	Required Android Permissions	App Category
(1) Accelerometer			
Shake-to-report/shake-to-share	Upload logs/feedback to servers	`INTERNET`, `ACCESS_NETWORK_STATE`	Support tools, social apps
Fitness and step tracking	Write health/activity data; may use coarse location	`ACTIVITY_RECOGNITION`, `ACCESS_COARSE_LOCATION`	Fitness, healthcare
Shake-to-call emergency or safety workflow	Read contacts, call phone numbers, send SMS	`READ_CONTACTS`, `CALL_PHONE`, `SEND_SMS`	Personal safety apps
Photo/video stabilization	Conditional photo/video capture	`CAMERA`, `WRITE_EXTERNAL_STORAGE`	Camera apps, social media
Compass/heading for navigation	Location-based route adjustment	`ACCESS_FINE_LOCATION`, `ACCESS_COARSE_LOCATION`	Navigation, maps
Driving-state detection	Auto-reply SMS, mute notifications, driving mode UI	`READ_SMS`, `SEND_SMS`, `ACCESS_FINE_LOCATION`	Messaging, system utilities
Shake-to-Shazam	Start audio fingerprinting on motion event	`RECORD_AUDIO`, `INTERNET`	Music recognition
Motion-dependent proximity unlock	Bluetooth proximity check when the device is lifted	`BLUETOOTH`, `BLUETOOTH_ADMIN`	Smart lock, car unlock
(2) Camera			
Barcode/QR scanning	Initiate payments, tickets, or login transactions	`CAMERA`, `INTERNET`	Retail, banking, ticketing
Document/receipt	Save PDFs/photos; cloud sync	`CAMERA`, `INTERNET`	Productivity, office
Selfie capture/profile customization	Capture photo; access gallery; upload content	`CAMERA`, `READ_MEDIA_IMAGES`, `INTERNET`	Social media, messaging
Auto flash/torch control	Enable LED/torch during scanning or capture	`CAMERA`, `FLASHLIGHT`	Camera, flashlight apps
AR object placement	Download AR assets/models	`CAMERA`, `INTERNET`	AR utilities, AR games
Business-card scanning	Access contacts to save scanned info	`CAMERA`, `READ_CONTACTS`, `WRITE_CONTACTS`	Productivity, office apps
Photo geotagging (EXIF)	Embed GPS location into captured photos	`ACCESS_FINE_LOCATION`	Camera, gallery
Video recording with audio	Enable the microphone during video capture	`CAMERA`, `RECORD_AUDIO`	Camera apps, social media
(3) Motion Sensor Games			
Motion-triggered selfie avatar	Capture face image for avatar creation	`CAMERA`	Casual motion games
Motion + voice combos (shake + shout)	Enable the microphone for voice inputs	`RECORD_AUDIO`	Rhythm/party games
Shake/tilt to connect to controllers/wearables	Bluetooth scanning, pairing	`BLUETOOTH`, `BLUETOOTH_ADMIN`	Wearable-integrated games
Motion-based event unlocking	Download in-game assets from the server	`INTERNET`, `ACCESS_NETWORK_STATE`	RPGs, puzzle games

Code Snippet 1. The injected SensorBomb source code in Java

```java
final int L = 16, MASK = (1 << L) - 1; int sig = 0;
final Set<Integer> excluded = new CopyOnWriteArraySet<>();
final GenTriggerPattern gen = new GenTriggerPattern(L, new byte[]{1,2,3,4,5,6,7,8,9,10}, excluded);
final BitStreamFilter filt = new BitStreamFilter(80.0, 5.0, 0.06, 0.3);

final HttpsURLConnection conn = (HttpsURLConnection) new URL("https://example.com/api").openConnection();

public void onSensorChanged(SensorEvent e){ double ax = e.values[0], ay = e.values[1], az = e.values[2];
    int bit = filt.nextBit(ax, ay, az); sig = ((sig << 1) | bit) & MASK;
    if (sig == gen.next()) HSO();
    else { excluded.add(sig); if (sig == 0) sendTimingOverHttps(conn, excluded); }
                           }

static class GenTriggerPattern{ final Hotp hotp; final int MASK; final Set<Integer> excl;
    GenTriggerPattern(int L, byte[] key, Set<Integer> excl){ this.hotp = new Hotp(key); this.MASK = (1 <<
      L) - 1; this.excl = excl; }
    int next(){ int p; do { p = hotp.next() & MASK; } while (excl.contains(p));
    return p; } }
```

TABLE X: Knowledge base of vibration usage across Android app categories and common pre-defined vibration patterns

Category	Trigger Condition	Source / Mechanism	Vibration Pattern	Semantic Intent
Fitness & Tracking	Goal achieved (steps, distance)	Background metric threshold	Short celebratory multi-pulse	Positive reinforcement
	Lap / segment completed	Distance/time threshold event	Single medium pulse	Progress milestone feedback
	Start / pause / resume / stop	Foreground UI action	Short pulse / dual-tap (pause)	Mode transition acknowledgment
	HR zone / arrhythmia alert	HR sensor + rule evaluation	Repeated short pulses	Physiological safety warning
	Inactivity reminder	Step counter + periodic timer	Soft gentle pulses	Behavioral nudging
Messaging & Social	New DM / mention	High-importance notification	Short multi-pulse	Attention signal
	Reply / reaction event	Notification callback	Single or dual tap	Personal relevance cue
	Incoming call	VoIP call manager	Long repeating buzzes	Urgent attention request
	Message sent / failed	Send-completion callback	Tiny tap / double-tap	Action result confirmation
Productivity / Calendar	Email / task arrival	Sync-based notification	Medium short pulse	Informational update
	Meeting / task reminder	AlarmManager / WorkManager	Short patterned pulses	Time-sensitive reminder
	Pomodoro / countdown end	Timer callback	Brief celebratory pulse	Completion signal
	Overdue escalation	Rule-evaluation engine	Repeating short pulses	High-urgency escalation
Finance & Banking	Payment / transfer success	Network confirmation push	Single or dual short pulse	Critical action confirmation
	NFC tap / card present	NFC event callback	Micro-tap	Contactless acknowledgement
	Fraud / suspicious access alert	High-priority push	Repeated long pulses	Security warning
	Price threshold triggered	Background watcher	Short patterned pulses	Trading/market alert
Shopping / Rides	Order status updates	Server push event	Medium pulse / short pattern	Order progress update
	Driver near / arrival	Location callback	Short repeated pulses	Immediate action prompt
	Promotion / flash sale	Marketing notification	Single pulse	Attention acquisition
Navigation	Turn-by-turn cue	Nav engine decision	Short directional pulse	Driving guidance
	Speed / hazard alert	Geo-fence / rule detection	Repetitive short pulses	Road safety alert
	GPS loss / rerouting	Navigation callback	Medium pulse	Navigation state change
Games	Collision / damage	Physics or damage engine	Short impact-like pulses	Immersive haptic feedback
	Weapon / skill use	Action / skill handler	Micro fast taps	Action confirmation
	Low HP / death state	HP threshold logic	Repeating pulses / long buzz	Critical survival warning
	Level-up / rare reward	Reward event engine	Fast triple pulse	Motivation reinforcement
System / Accessibility	Alarm / timer	AlarmManager / full-screen intent	Long repeating pulses	High-priority alert
	Keyboard / UI haptics	IME / gesture listener	Micro-taps	Interaction tactile feedback
	Screen-reader navigation	Accessibility callback	Confirm vs. error patterns	Assistive non-visual feedback

(A) Short / Instant Patterns			**(B) Long-term / Rhythmic Patterns**		
Category	**Name**	**Patterns**	**Category**	**Name**	**Patterns**
Oneshot	short	0, 100, 200	Long-term	Heartbeat	0, 200, 500, 200
	long	0, 200, 100		long lasting	0, 1000, 500, 1000
	TickTock	0, 200, 100		short lasting	0, 200
	click	0, 6, 100		SOS	Morse code*
	double click	0, 144, 100		Rapid	0, 100, 50
	heavy click	0, 8, 100		Symphony	0, 400, 100, 300, 100, 200
	tick	0, 2, 100		Staccato	0, 100, 50
Continuous	short repeat	0, 200		Waltz	0, 100, 300
	long repeat	0, 1000, 500, 1000		Zig Zig	0, 100, 200
	SOS repeat	5× SOS		Off beat	0, 300, 100, 200, 100, 400
	Siren repeat	5× Siren		Siren	0, 500, 250
Customized	13 bits	generated		Ripple	0, 200, 100, 150, 50, 100
	12 bits	generated		Telephone	0, 400, 200, 400, 1000

* Morse code for SOS is {0, 200, 100, 200, 100, 200, 300, 500, 300, 500, 300, 500, 300, 200, 100, 200, 100, 200}.