BlueSWAT: A Lightweight State-Aware Security Framework for Bluetooth Low Energy

Xijia Che
INSC, Tsinghua University & BNRist
Beijing, China
cxj22@mails.tsinghua.edu.cn

Yi He
INSC, Tsinghua University
Beijing, China
heyi21@mails.tsinghua.edu.cn

Xuewei Feng
DCS, Tsinghua University
Beijing, China
fengxw06@126.com

Kun Sun
IST, George Mason University
Fairfax, USA
ksun3@gmu.edu

Ke Xu
DCS, Tsinghua University
Beijing, China
xuke@tsinghua.edu.cn

Qi Li
INSC, Tsinghua University
Beijing, China
qli01@tsinghua.edu.cn

Abstract

Bluetooth Low Energy (BLE) is a short-range wireless communication technology for resource-constrained IoT devices. Unfortunately, BLE is vulnerable to session-based attacks, where previous packets construct exploitable conditions for subsequent packets to compromise connections. Defending against session-based attacks is challenging because each step in the attack sequence is legitimate when inspected individually. In this paper, we present BlueSWAT, a lightweight state-aware security framework for protecting BLE devices. To perform inspection on the session level rather than individual packets, BlueSWAT leverages a finite state machine (FSM) to monitor sequential actions of connections at runtime. Patterns of session-based attacks are modeled as malicious transition paths in the FSM. To overcome the heterogeneous IoT environment, we develop a lightweight eBPF framework to facilitate universal patch distribution across different BLE architectures and stacks, without requiring device reboot. We implement BlueSWAT on 5 real-world devices with different chips and stacks to demonstrate its cross-device adaptability. On our dataset with 101 real-world BLE vulnerabilities, BlueSWAT can mitigate 76.1% of session-based attacks, outperforming other defense frameworks. In our end-to-end application evaluation, BlueSWAT patches introduce an average of 0.073% memory overhead and negligible latency.

CCS Concepts

• **Security and privacy → Mobile and wireless security**.

Keywords

Bluetooth Low Energy; State-aware Security; eBPF; Cross-Platform Defense

ACM Reference Format:
Xijia Che, Yi He, Xuewei Feng, Kun Sun, Ke Xu, and Qi Li. 2024. BlueSWAT: A Lightweight State-Aware Security Framework for Bluetooth Low Energy. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3658644.3670397

1 Introduction

Bluetooth is a widely used wireless communication protocol in the Internet of Things (IoT), allowing smart devices to exchange data in short range. In 2010, the Bluetooth Core Specification 4.0 [72] heralds the advent of BLE, an innovative wireless protocol designed specifically for resource-limited IoT devices. BLE, which focuses on low-power communication, has experienced substantial growth over the past decade. More than 5 million BLE-enabled smart devices are estimated to be in use by 2023, and total annual shipments of Bluetooth devices will reach 7.5 billion by 2028 [74].

With the proliferation of BLE-enabled devices in smart homes and wearables, a vast amount of sensitive information is being transmitted over BLE channels and the attacks continue to emerge, including the *packet-based attacks* and the *session-based attacks*. A packet-based attack [3–5, 9, 40, 82] manipulates one crafted packet to exploit the target BLE stack, which usually aims at implementation flaws such as missing bounds checks. It may lead to device DoS or even remote code execution (RCE) when carefully manipulated. By contrast, a session-based attack [34, 56, 64, 81, 85, 88, 89, 91] is performed by constructing complex BLE interaction processes that require multiple previous data packets to lay the groundwork for the final attack packet. Such sophisticated attacks can cause severe consequences, including Man-in-the-Middle (MITM) session hijacking and device impersonation. According to our dataset of 117 real-world CVEs, around 54% of BLE vulnerabilities are session-based, which forms a huge attack surface (as shown in Figure 8, Appendix E). Detecting BLE session-based attacks is challenging, as each packet in the attack sequence is legitimate when examined individually, and can be used in normal BLE connections. A session-based attack is only triggered when these packets are maliciously used by an attacker to form a specific attack sequence.

As a result, if the BLE security mechanism fails to perform inspections on the packet sequence (i.e., at the session level), it cannot verify the legitimacy of the entire session, thereby becoming prone

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670397

to session-based attacks. However, to the best of our knowledge, existing BLE defense frameworks are inefficient against session-based attacks, primarily due to the following two limitations:

Only Inspecting Individual Packets. LBM [79] leverages stateless filters on individual packets in Linux BlueZ stacks, which can only inspect the packet payloads one at a time. It ignores the forward messages before the target packet and therefore fails to detect session-based attacks without the knowledge of session context.

Long Patching Window. The mainstream countermeasure against session-based attacks is for product vendors to distribute software patches from chip manufacturers. However, the traditional firmware update mechanism has many drawbacks in a fragmented IoT environment, where vendors usually maintain many BLE devices with different chips. For vulnerabilities that affect multiple BLE stacks, such as Specification weaknesses, firmware updates usually result in a long vulnerable window (typically months or years) after vulnerability disclosure. Vendors have to wait for a Specification update and then for each manufacturer to develop a patch. After that, vendors must test it with their custom stacks, recompile the firmware, and update the corresponding user products. In addition, the traditional software update mechanism suffers from disruptive device reboots which are unacceptable for time-sensitive devices, and irreversible installations which exacerbate the risk of faulty patches. In the worst case, many weak IoT devices do not support firmware updates due to limited IO capability, leaving traditional software patches backward incompatible [86].

Other security frameworks do not support comprehensive BLE traffic filtering and are therefore inadequate for mitigating session-based attacks. BlueShield [86] deploys additional monitor devices to capture cyber-physical features of malicious advertising packets. However, it only defends against spoofing attacks and has to work in a stationary BLE network, overlooking a large number of mobile BLE products including wearable devices, mobile medical devices, etc. Other IoT security solutions [50, 57, 83] focus on smart homes and Cloud peripheral protocols and thus fail to study BLE in a fine-grained way. For instance, T2Pair [57] proposes a Universal Operation Sensing scheme for users to pair wireless devices securely. Unfortunately, many BLE attacks happen beyond the pairing phase.

In this paper, we present **BlueSWAT** (**Blue**tooth LE **S**tate-a**WA**re securi**T**y framework), a lightweight security solution for IoT devices, which can mitigate BLE session-based attacks. BlueSWAT records session context and monitors sequential actions of BLE connections. With collected session context, BlueSWAT is capable of performing a stateful inspection on the session level in addition to individual packets. Specifically, we leverage finite state machine (FSM) to detect session-based attacks. We capture the patterns of attack sequences and model them as state transition paths in FSM. During BLE connections, BlueSWAT updates the FSM based on session context and prevents session-based attacks from transiting along malicious paths. BlueSWAT is highly flexible for vendors to customize at runtime, which supports adding and removing security policies (namely, transition rules of FSM) to extend FSM for new attacks.

Although the concept of FSM has been involved in Bluetooth security by previous works, BlueSWAT serves distinct motivations and goals, and functions in a different manner. For example, L2Fuzz [65] utilized the FSM as a fuzzing component to increase code coverage. It focused on the L2CAP protocol of Bluetooth Classic, while BlueSWAT covers both controller and host layers of BLE. BLEdiff [51] abstracted FSM models from different stack implementations to compare and identify deviant non-compliant behaviors as potential vulnerabilities. Profactory [83] took user-defined FSMs to automatically generate memory and multiplex safe stack implementations. By contrast, BlueSWAT presents a novel FSM to model the general behavior of BLE standard and various attack patterns, which aims to capture malicious session sequences by monitoring FSM transitions.

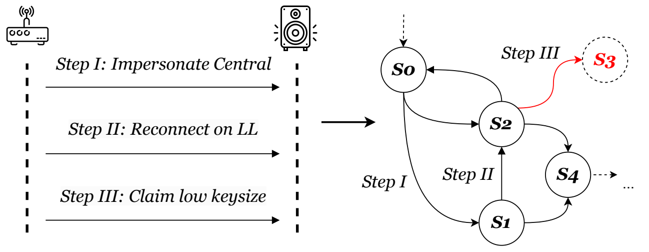


Figure 1: BLE KNOB attack [26] and FSM diagram. The pattern of the KNOB session is modeled as a malicious transition path in FSM. S3 is the exploiting state.

BlueSWAT FSM is abstracted from the Specification to monitor BLE connections, which is comprehensive to model various attack patterns. To capture session states and update FSM at runtime, BlueSWAT deploys a compact set of hooks at the Link Layer (LL) and Security Manager Protocol (SMP) of BLE stacks. Hooks at these layers can comprehensively capture session context for modeling various BLE attacks. This is achieved by leveraging the universal packet format of LL and SMP defined by Bluetooth Specification. In BLE connections, LL has access to all session data and SMP controls security procedures such as key sharing and authentication. Therefore, hooks on these two protocols are sufficient to monitor session progress and capture FSM events.

To overcome the disadvantages of the traditional patching mechanism, we develop a lightweight eBPF framework [39] to facilitate BlueSWAT patch distribution. Security patches are first written in C and then compiled into eBPF programs. For a vulnerability that affects multiple devices, vendors can create one common patch and deploy it across all victims. The insights here are (1) BLE stacks of different manufacturers share similar layers and procedures defined by Bluetooth Specification, (2) eBPF bytecode can be executed across different chip architectures. Therefore, vendors can build universal FSM models and transition rules for cross-device deployment, reducing the long vulnerable window after vulnerability disclosure. Vendors can transmit eBPF programs to victims via BLE and directly integrate them into BlueSWAT. The updating strategy brings two advantages: (1) It allows vendors to patch vulnerabilities for devices that do not support firmware updates, and (2) It avoids device reboot and firmware recompilation when installing new patches. Such convenience is essential to BLE devices in security and time-sensitive scenarios, such as manufacturing plants, and medical and health care.

We implement BlueSWAT on 5 embedded devices with different chips and BLE stacks to demonstrate its adaptability in heterogeneous IoT environments. We systematically investigate various BLE attacks and build an evaluation dataset with 101 real-world vulnerabilities. Our evaluation result shows that BlueSWAT is capable of mitigating 76.1% (35 / 46) of session-based attacks and 96.4% (53 / 55) of packet-based attacks, which outperforms existing defense frameworks like LBM [79]. In our end-to-end application test, BlueSWAT patches introduce an average of 0.073% flash overhead and negligible latency, which can be considered controllable in resource-constrained IoT devices. Our source code is available at https://github.com/RayCxggg/BlueSWAT.

We summarize our main contributions as follows:

- We design and implement BlueSWAT as the first state-aware security framework against BLE session-based attacks. It performs session-level inspection to monitor sequential actions of BLE connections, in addition to inspecting individual packets.
- We develop a lightweight eBPF framework to facilitate patch distribution of BlueSWAT. Compared with the traditional firmware patching mechanism, BlueSWAT transmits patches as eBPF programs via BLE, avoiding firmware recompilation and device reboot.
- For a vulnerability that affects multiple devices, vendors can develop one common eBPF patch and deploy it to all victims, reducing the long vulnerable window after vulnerability disclosure.
- We implement BlueSWAT on 5 real-world IoT devices with different BLE stacks and chip architectures. On our dataset with 101 real-world vulnerabilities, BlueSWAT can mitigate 76.1% of session-based attacks and 87.1% of all attacks. In our end-to-end application evaluation, BlueSWAT patches introduce less than 0.073% of memory overhead on average and negligible latency.

2 Background and Motivation

2.1 BLE Basics

In a BLE connection, two devices work in a central-peripheral scheme. During different phases of BLE sessions (as shown in Figure 6, Appendix C), two devices have different names according to the Specification, such as advertiser, scanner, and initiator. For simplicity, we use central and peripheral to distinguish peer BLE devices in this paper.

The BLE protocol stack consists of a controller and a host. The controller contains the physical layer and the logical link layer, which is typically implemented in Bluetooth chips with manufacturers customization to match the hardware. The host contains a Logical Link Control Adaptive Protocol (L2CAP) layer and some upper protocols to support application profiles. The Host Controller Interface (HCI) provides a coarse-grained communication channel between the host and the controller. Between an HCI command and an HCI event, there may be multiple actions in the lower controller layer. Therefore, unlike LBM, which hooks at HCI, BlueSWAT hooks at the lower link layer for comprehensive inspection of malicious connections. LL hooks capture the session context at runtime, where all raw incoming bytes are decoded for the first time and accessible as plaintext in the stack.

2.2 Extended Berkeley Packet Filter (eBPF)

eBPF is a virtual machine in the Linux kernel that allows users to load userspace code, known as eBPF programs, into kernel space for better interaction with the kernel. These programs can be written in C code and compiled using the LLVM toolchain. After verification, the eBPF program is compiled into native code, ensuring safe and high-performance execution inside the kernel. Because of its extensive and powerful capabilities, eBPF has become a popular choice for filtering network packets [35, 47, 63, 79] and implementing complex kernel extensions [37, 38, 41, 43–45, 53, 76].

Previous works [48, 54] demonstrate that eBPF can be ported to run efficiently in resource-constrained embedded devices, outperforming other script runtimes such as MircoPython [18], WASM [11], RIOTjs [20]. Since using scripting languages to define firewall policies offers greater flexibility and extensibility than traditional configuration files, it is promising to extend the existing embedded eBPF runtimes to support the BLE state machine and execute policies in the eBPF sandbox. Moreover, since eBPF programs can be separately compiled into bytecode in advance without recompiling the entire firmware, when developers need to add new policies, they can simply transmit the bytecode over BLE or USART which is compatible with millions of legacy embedded devices that may not even support firmware updates. RapidPatch [48] has leveraged the scalability of eBPF bytecode on multiple chip architectures to patch across RTOSes. However, it can only patch RTOSes with the same vulnerable libraries (e.g., OpenSSL). By contrast, BlueSWAT extends the patch scalability to different stack implementations based on our insight that BLE stacks of different manufacturers share similar layers and procedures defined by Bluetooth Specification.

2.3 A Motivating Example

Figure 1 presents the famous BLE KNOB attack [26] as a motivating example. The attacker first jams a benign connection and then launches a MITM attack. It impersonates the central and sends a pairing request with a low Maximum Keysize (7 bytes) to the peripheral, and vice versa. Two victims, who previously used a strong session key (16 bytes), are deceived and downgrade to use a weak encryption key. The attacker then eavesdrops on the ciphertext and brute-forces the low-entropy key, which breaks BLE encryption.

Challenges. To defend against KNOB attack, we mainly need to address two challenges in detection and mitigation:

(1) To detect the session-based KNOB attack, we need to track the entire attack session rather than individual packets. Since KNOB is a session-based attack, individual steps of the malicious session are benign. In other words, a peer device claiming low key entropy may not be a KNOB attacker, since weak IoT devices can use short keys for computing resource control. It is impractical to simply reject all pairing requests with low key entropy, which will introduce many false cases and break usability. Therefore, to be compatible with devices that do not support large-entropy keys, the Specification still allows keys smaller than 16 bytes.

(2) To mitigate KNOB, a Specification design weakness that affects all BLE devices, we need to address the vulnerable window between

attack disclosure and patch update. For compliant attacks such as KNOB and BlueMirror [34], the existing mitigation is to update the Specification [84]. However, for vendors that maintain multiple devices from different manufacturers, distributing Specification updates is extremely time-consuming because they have to wait for all manufacturers to develop their patches. It usually takes months or years for manufacturers to update BLE stacks with new Specifications and thus leaves a long vulnerable window after vulnerability disclosure. For example, as of January 2024, the BLE stack of Texas Instrument [22] has not yet been updated to Bluetooth v5.4 and supports Advertising Encryption, which was introduced in January 2023 [73]. Even though the Secure Connections Only mode can avoid KNOB, no general-purpose BLE devices are observed to use this mode by default [84]. Therefore, most devices are still vulnerable to the KNOB attack.

We use FSM to model BLE connection actions and capture malicious sequential behaviors. As shown in Figure 1, BlueSWAT captures the pattern of the KNOB session as a malicious transition path in the FSM. Each step of the attack triggers a transition and eventually drives the FSM to the exploiting state, i.e., from S0 via S1 and S2 to S3. A session that performs Step III without I and II will not transit to S3 because it will not enter S2 in the first place. Therefore, by modeling the state transitions of the KNOB session in the FSM, BlueSWAT performs an inspection of the connection context to detect session-based attacks and avoid false cases. Our eBPF framework allows vendors to develop one common eBPF patch for a compliant attack and deploy it to all victims, reducing the long vulnerable window after the attack is disclosed. We present design details of BlueSWAT in Section 4.

3 Threat Model

We use the Dolev-Yao model [36] as our threat model, which is compliant with previous works and the Bluetooth Specification. We assume the attacker is in the range of a BLE network that consists of two legitimate devices. The attacker is capable of eavesdropping on the BLE network to collect all plaintext information publicly broadcast by the victims, such as address, IO capability, and authentication requirements. Also, the attacker can jam the BLE spectrum to block benign connections and inject arbitrary packets into the network. The attacker cannot physically temper the devices.

We focus on the controller and host part of BLE stacks, which excludes attacks on the physical layer and application layer, such as side-channel analysis [42, 59, 90] and signal eavesdropping [28, 42]. Based on the observation that BLE connections are in the format of packets, we summarize BLE attacks as two types, namely, *packet-based attack* and *session-based attack*. A packet-based attack leverages one manipulated packet to exploit the BLE stack, which usually aims at the implementation faults such as bounds check missing and buffer overflow. By contrast, a session-based attack uses a series of crafted packets to construct a malicious attack sequence. In a malicious session, the preceding packets prepare the exploiting conditions for the subsequent packets to eventually compromise the victim. Note that each step of a session-based attack is legitimate when taken apart and can be commonly used in other benign sessions. A session-based attack is only triggered when these steps form a specific attack sequence.

In this paper, we focus on defending against session-based attacks. Besides, BlueSWAT is compatible with previous works in mitigating packet-based attacks by filtering packets. We evaluate BlueSWAT against both session-based and packet-based attacks in Section 6.

4 System Design

4.1 Design Goals and Overview

When mitigating BLE session-based attacks in IoT networks in real-time, we are destined to achieve five design goals as follows.
G1: Effectiveness and Extensibility. BlueSWAT should be effective in mitigating known BLE session-based attacks, and easy to extend for new vulnerabilities.
G2: Compatibility. BlueSWAT should be compatible with heterogeneous chip architectures and BLE protocol stacks in a fragmented IoT environment.
G3: Usability. For vendors, integrating BlueSWAT into BLE stacks and maintaining defense policies should be undemanding. BlueSWAT should be able to compensate for the inadequacy of the traditional firmware update mechanism in complex heterogeneous IoT scenarios, allowing vendors to patch vulnerabilities simply and efficiently.
G4: Comprehensive Mediation. BlueSWAT needs to guarantee that all traffic is inspected before handed over to the BLE stack. BlueSWAT should not be bypassed under any circumstances.
G5: Practicality. BlueSWAT should only introduce limited performance overhead, and thus be practical on resource-constrained IoT embedded devices.

Figure 2 shows the overall architecture of BlueSWAT that targets to achieve the above five design goals. As BLE connections proceed, BlueSWAT monitors session context in real-time, performing session-level inspection in addition to analyzing individual data packets. Based on the Bluetooth Specification, we build a general FSM to model BLE connections, which can describe sequential actions of different attacks. Specifically, we capture patterns of session-based attacks, including the attack steps and the final exploit packet, and model them as malicious transition paths in FSM. BlueSWAT prevents FSM from transiting through malicious paths into exploiting states, which indicates connection exploitation. For newly disclosed attacks, vendors can add corresponding transition rules in FSM for inspection. Therefore, we achieve the design goal of Effectiveness and Extensibility (G1).

To overcome the resource limitations of embedded devices and the fragmentation of the IoT environment, we develop a lightweight eBPF framework. It allows developers to write concise C policies for FSM and compile them into eBPF programs. The nature of eBPF bytecode allows BlueSWAT policies to be executed on different chips, which facilitates patch distribution across heterogeneous devices. BlueSWAT hooks at the shared layers of different stacks, which are the LL and SMP layers, to capture monitored attack patterns and state transitions in FSM. For a vulnerability that affects multiple different devices, vendors can develop one common patch and deploy it to all victims. Hence, we meet the design goal of Compatibility (G2). For patch distribution, vendors can transmit eBPF programs via BLE and directly integrate them into BlueSWAT, avoiding firmware recompilation and device reboot. Combined

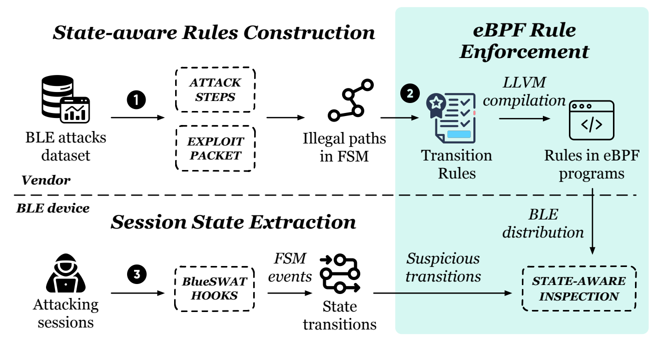


Figure 2: Workflow of BlueSWAT. ❶: Vendors abstract attack patterns and model them as illegal transition paths in FSM. ❷: Vendors compile transition rules into eBPF programs and distribute them to BLE devices. ❸: BlueSWAT captures session events and inspects FSM transitions at runtime.

with the cross-chip and cross-stack deployment capability, extending BlueSWAT for new attacks becomes more efficient and convenient than the traditional firmware update mechanism. The hooks of BlueSWAT are designed based on the Bluetooth Specification, which maintains a minimal dependency on specific stack implementations. Our compact set of hooks requires less than 10 lines of code insertion, which ensures BlueSWAT undemanding for vendors to integrate into both open-source and closed-source firmware. Hence, we achieve the design goal of Usability (G3).

The lower hooks of BlueSWAT are between the link layer and physical layer, where BLE traffic is first decoded and processed by the stack controller. By resolving input session data at the link layer, BlueSWAT performs an inspection on plaintext session packets before they reach higher layers and exploit potential vulnerabilities. Hence, we achieve the design goal of Comprehensive Mediation (G4). Our evaluation result shows that BlueSWAT introduces a controllable memory footprint and negligible runtime latency, meeting the design goal of practicability (G5).

4.2 State-aware Rules Construction

Based on the Bluetooth Specification, we build an FSM to model the BLE connection actions and capture BLE attacks. As shown in Figure 3, a session-based attack will transit along a malicious path in FSM, which eventually enters an exploiting state and is rejected by BlueSWAT transition rules.

4.2.1 States. The FSM state space consists of benign states and exploiting states. Legal sessions drive FSM in benign states while malicious sessions trigger transitions into exploiting states. The FSM is abstracted from the Bluetooth Specification, which is comprehensive to model the behaviors of different BLE stacks.

Benign States. Benign states describe the behaviors of legitimate BLE connections.

Standby. Standby state is the initiating state. In this state, BLE devices are active but not discoverable. They do not transmit or receive any packets. When a connection is finished or terminated by BlueSWAT, the BLE stack is reset to the Standby state.

Discovery. In this state, centrals scan for advertising peripherals. BLE advertising packets are unencrypted and contain device information such as device address and name. Many attacks leverage public messages in the Discovery state to perform impersonation and spoofing attacks. Peripherals can use the advertising encryption procedure (introduced in Bluetooth v5.4 [73]) with previously paired centrals. But to connect with new peer devices, exchanged messages in the Discovery phase have to be plaintext. After a peripheral response to the connection request from a central, two devices enter the LL Connection state.

LL Connection. In this state, an LL connection is established. Connection parameters, such as interval and channel map are exchanged. The messages in the LL Connection state are also plaintext. Hence, many attacks exploit firmware vulnerabilities with crafted malicious packets in this state. After a peripheral response to a pairing request from a central, two devices enter the Key Sharing state and initiate the pairing procedure. Two devices may skip pairing and exchange data in plaintext if they do not support LL encryption.

Key Sharing. Two devices perform the SMP pairing procedure in this state. They exchange device features and then generate and distribute a shared key. BLE devices can choose multiple pairing methods and association models based on their IO capability and authentication requirements. Most of the specification weaknesses are discovered in this state. They allow attackers to downgrade the security models into vulnerable ones to skip authentication, break encryption, etc. After two devices exchange the shared key, they enter the Data Exchange state.

Data Exchange. In this state, two devices establish an encrypted connection and transmit BLE data. They derive a session key from the shared key and then perform the encryption procedure. If the connection is correctly encrypted, BLE data transmission is relatively secure. However, nearly 80% of existing BLE devices communicate in plaintext [60], which leaves a huge attack surface. After the data transmission is over, the BLE stack returns to the Standby state and waits for future connections.

Exploiting States. Malicious BLE sessions will eventually drive the FSM into exploiting states and trigger BlueSWAT alerts.

Discovery Error. Few attacks compromise the Discovery state because two devices are not connected. In practice, we only observe one vulnerability (i.e., CVE-2017-13211) caused by repeated scan requests. Therefore, the Discovery Error state is not presented in Figure 3 for simplicity. Many attacks leverage the plaintext messages in the Discovery state to exploit other connection states.

Connection Break. After the LL connection establishment, attackers can invoke some BLE procedures and exploit victim stacks. Since the connection is not encrypted, attackers can inject malicious plaintext packets with invalid fields to exploit implementation faults, which cause buffer overflow, device crash, etc.

Pairing Exploitation. The pairing procedure is a critical security procedure of BLE for authentication and encryption. Hence, many attacks aim to compromise device pairing, such as association model and shared key compromise.

Encryption Failure. If two devices correctly perform encryption, the Data Exchange state is relatively secure. However, some implementation faults of manufacturer stacks allow attackers to cause encryption failures and compromise BLE confidentiality.

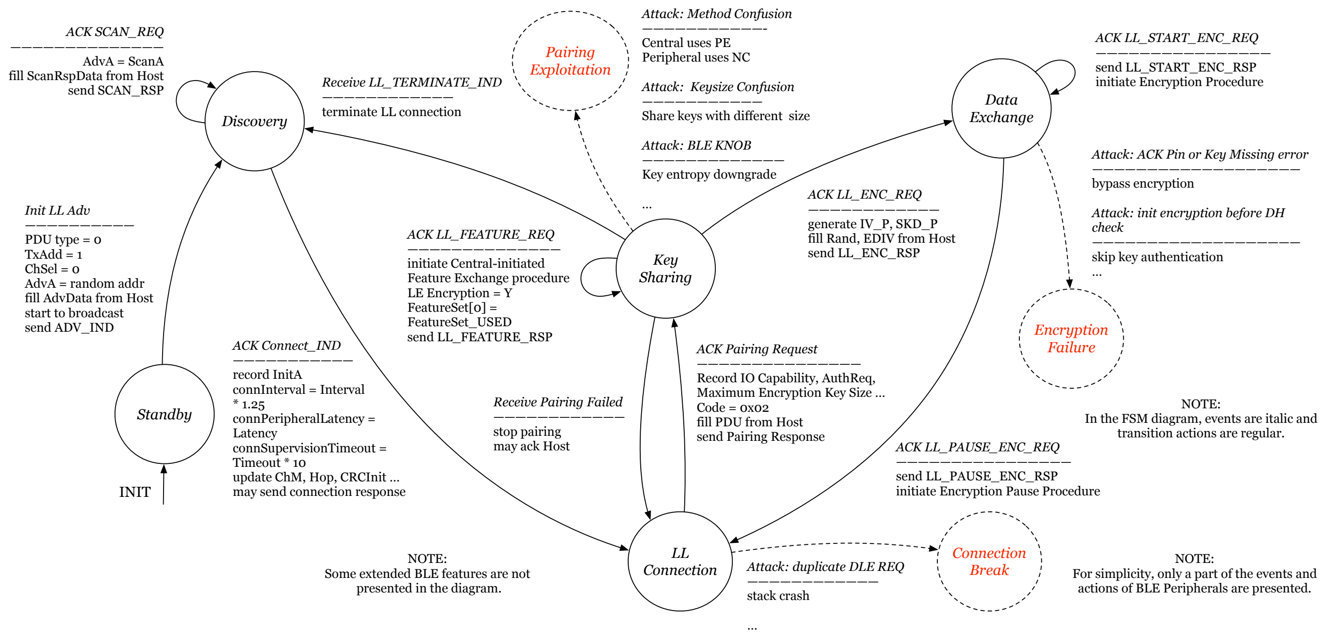


Figure 3: BlueSWAT FSM.

4.2.2 *Events.*

BlueSWAT FSM events are compliant with the Specification and comprehensive to model all BLE connections. When BlueSWAT captures these events, it automatically updates the FSM to follow the session progress. Since BLE centrals and peripherals are identical in security-related protocols and peripherals are more pervasive [51], we describe the events from the perspective of peripherals.

(1) *Advertising Packets.* When LL starts to broadcast advertising packets such as `ADV_IND` and `SCAN_REQ` PDU, FSM transits from the Standby to the Discovery state.

(2) *Connection Indicators.* In the Discovery state, when the peripheral receives a `CONNECT_IND` PDU and responds with a positive Connection Response, FSM transits to the LL Connection state.

(3) *Pairing Indicators.* If two devices are not bonded, they need to perform the SMP pairing procedure. When the peripheral receives a Pairing Request and responds with a positive Pairing Response, FSM transits from the LL Connection to the Key Sharing state.

(4) *Encryption Indicators.* In the Key Sharing and the LL Connection state, when the peripheral receives a `LL_ENC_REQ` PDU and responds with a positive `LL_ENC_RSP` PDU, FSM transits to the Data Exchange state.

(5) *BlueSWAT Alerts.* When a session enters an exploiting state, BlueSWAT sends an alert to FSM and terminates the connection. FSM transits to the Standby state whenever an alert is accepted.

4.2.3 *Variables.*

Apart from tracking session states, BlueSWAT records a few security-related variables used in previous benign connections. The variables help BlueSWAT in recognizing attacks with session history.

(1) *Device address.* BlueSWAT records MAC addresses to recognize paired devices. If a peer device uses RMA, BlueSWAT records the real address resolved by the stack.

(2) *SMP parameters.* BlueSWAT records security critical SMP parameters used in benign connections, such as session key entropy, pairing methods, and association models. Together with the device address, BlueSWAT can prevent an impersonator or MITM from downgrading the SMP security level.

4.2.4 *Malicious Paths.*

The sequential actions of session-based attacks are modeled as malicious transition paths in FSM. Malicious paths are driven by attack-specific events and eventually enter the exploiting states. As sessions proceed, BlueSWAT collects monitored states and updates FSM at runtime. A session triggers a BlueSWAT alert when its transition path from the initiating state covers a malicious path. Paths with parts of the malicious paths are considered benign. Therefore, BlueSWAT can avoid false cases while tracking session sequences.

4.2.5 *Transition Rules.*

Transition rules check whether FSM events are in accord with attack patterns. If an event sequence transits along a malicious FSM path, the monitored session will drive the FSM into an exploiting state and trigger a BlueSWAT alert. The rules are specific to attacks, which can be added after new vulnerability disclosure.

4.2.6 *An Example: BLE KNOB attack.*

We use the famous BLE KNOB attack [26] as an example to explain our defense strategy in detail. Specifically, KNOB is represented as a malicious path in FSM with the following sequence of (state, event) pairs:

(**State**: `Standby`, **Event**: Device init.)
(**State**: `Discovery`, **Event**: `ADV_IND` sent)

(**State**: `LL Connection`, **Event**: `CONN_REQ` received and `CONN_RSP` sent, , hit a paired record in BlueSWAT)
(**State**: `Key Sharing`, **Event**: Pairing Request received and Pairing Response sent)
(**State**: `Pairing Expoitation`, **Event**: smaller `Keysize` than previously used one)

When the victim pairs with the peer for the first time, BlueSWAT records the peer device identity and the agreed key entropy. In a future reconnection, a KNOB attacker hijacks the session after an LL connection is established. At this time, the FSM has transited from *Standby* through *Discovery* to *LL Connection*. When the attacker impersonates the peer and claims a smaller `Maximum Keysize` in Pairing Request, BlueSWAT compares the new Keysize with the previously used one and detects a key entropy reduction. It violates BlueSWAT transition rules and drives the FSM into the Pairing Exploitation state. Therefore, the KNOB session transits along a malicious path in FSM and will be terminated by BlueSWAT.

In real-world scenarios, a short key may be necessary if the device is too weak or the BLE service needs more usability than security. In this case, BlueSWAT allows two devices to use a weak key if they pair for the first time. In other words, a session that transits along a part of the malicious path is allowed by BlueSWAT. Only when a session covers the entire malicious path will it be considered as a session-based attack. In general, mitigation of BlueSWAT does not impact weak devices or add restrictions to the encryption procedure. It only forbids reducing key size rather than using a weak one by default. Unfortunately, for packet-level defending methods like LBM, they have to drop any Pairing Request packets with field `Maximum Encryption Key Size` equal to 7. Such methods could severely impact or disable pairing for weak BLE devices that require short session keys. Compared with the mitigation of Bluetooth SIG, our mitigation does not require updating the Specification and breaking backward compatibility.

4.3 Session State Extraction

BlueSWAT deploys a compact set of hooks in BLE stacks to extract session states in real-time. The lightweight hooks require less than 10 lines of code modification on BLE stacks, which facilitate the integration of BlueSWAT into both open-source and closed-source embedded firmware.

4.3.1 BlueSWAT Hooks. Although BLE stacks are separately implemented by manufacturers with various solutions, their basic architecture and behaviors are similar, such as BLE packet formats, HCI command formats, pairing procedures, etc. This similarity determined by the Bluetooth Specification allows us to design a set of universal hooks at certain layers of the stacks, regardless of implementation details of manufacturers. Such design significantly reduces the engineering effort required to integrate BlueSWAT with the existing BLE stacks, while providing BlueSWAT policies with the capability of cross-stack defense.

Specifically, BlueSWAT focuses on two security critical protocols in BLE, which are the Link Layer (LL) protocol and the Security Manager Protocol (SMP). LL is responsible for managing BLE connection states and directly interacts with the physical layer. It also controls the encryption and decryption procedure of BLE, which means LL is where plaintext data can be accessed in the stack for the first time. Security Manager Protocol (SMP) serves as the core architecture of BLE security mechanisms, such as device pairing and encryption. Hence, monitoring the behaviors of SMP is crucial for BLE security. Bluetooth Specification stipulates the packet formats for SMP, therefore we can universally resolve the SMP traffic at the entrance. Specifically, the SMP parser hook acquires important session parameters for the FSM models, such as encryption keys and pairing association models, which are broadly manipulated by session-based attacks [25, 26, 34, 46, 71, 81, 88, 91].

To ensure that all BLE traffic is subject to BlueSWAT inspection, we integrate hooks into the link layer which serves as the entry point for traffic into the BLE stack. As shown in Figure 4(a), the LL RX hook resolves the traffic where the packet data is accessible in plaintext for the first time after it is received by the BLE controller. It monitors the session packets and collects the state transitions that need to be verified. Simply monitoring the RX path is not sufficient to infer connection states, since the device may require data retransmission to renegotiate parameters or fix error packets. In this case, packets may be accepted by BlueSWAT but rejected by the stack, which should not be allowed to trigger transitions. Therefore, we design the LL TX hook to monitor the output traffic from the stack, collaborating with the RX hook to determine the states of the sessions. When the stack responds with error messages to indicate retransmission or disconnection, BlueSWAT will not update the new states of the illegal input to FSM, even if they pass the verification. BlueSWAT FSM operates at a higher level than inner FSMs of the stack (e.g., L2CAP FSM). It does not share common states with inner FSMs or temper with their transitions. When a session triggers an event of BlueSWAT FSM, BlueSWAT waits for the inner FSMs to finish transitions and transits along with the response (TX) messages.

```
// nimble/controller/src/ble_ll_conn.c:
void ble_ll_conn_rx_data_pdu(struct os_mbuf *rxpdu,
struct ble_mbuf_hdr *hdr){
  ...
  if (IFW_DC_LL_CTRL_PARSER(connsm, rxpdu)){
    goto conn_rx_data_pdu_end;
 }
  ...
 }
```

(a) LL RX parser for control PDUs.

```
// nimble/host/src/ble_sm.c:
int ble_sm_rx(struct ble_l2cap_chan *chan){
  ...
    if (IFW_SMP_PARSER(chan)){
        return BLE_HS_EUNKNOWN;
    }
  ...
}
```

(b) SMP RX parser.

Figure 4: BlueSWAT hooks in Mynewt NimBLE.

4.3.2 State Extraction. With the hooks monitoring session traffic, BlueSWAT uses eBPF programs to capture the inspected FSM events based on transition rules. Once a state transition is triggered by the captured event, BlueSWAT invokes corresponding transition rules based on the state and event, and performs a validity check. BlueSWAT policies are mapped to corresponding states and events and are only invoked when target states are changed. This scheme ensures that BlueSWAT introduces less performance overhead in comparison to existing solutions such as LBM, which requires the traversal of all the filters for every packet check.

4.4 eBPF-based Rule Enforcement

We develop a lightweight eBPF framework for IoT platforms based on the Linux uBPF library [23]. The transition rules are written in C and compiled into eBPF bytecode in advance. When a session triggers an FSM transition, BlueSWAT loads the eBPF programs from the policy cache for verification. Compared with implementing software patches in C firmware, our eBPF framework provides three advantages: (1) eBPF programs can be transmitted via BLE and dynamically loaded by BlueSWAT. Hence, patch update does not require firmware recompilation and device reboot. (2) eBPF bytecode can be executed across different chips regardless of their architectures, which ensures that BlueSWAT is compatible in the fragmented IoT environment. (3) eBPF programs consume limited memory resources and introduce negligible runtime overhead as shown in Section 7, and thus provide BlueSWAT with real-world practicality across resource-constrained devices.

4.4.1 Patch Compilation and Transmission. We use the bpf LLVM backend to compile C patches into eBPF bytecode. Instead of deploying new patches through traditional software update mechanisms (which encounter many drawbacks as mentioned in Section 1), we transport the eBPF bytecode to BlueSWAT via BLE. Specifically, we implement a new BLE service and characteristic for patch transmission. In the service, BlueSWAT stores new patches locally for future execution. With our eBPF framework, updating policies does not need to recompile firmware and reboot devices because eBPF programs can be directly loaded into the kernel for execution. For weak IoT devices that do not support firmware updates, transmitting patches via BLE enables them to accept security updates.

4.4.2 Cross Stack and Architecture Deployment. Since all different BLE stacks are implemented following the Bluetooth Specification, BlueSWAT is designed to maintain minimal dependency on specific implementation details of the vendor stacks. Specifically, the only part of BlueSWAT that is related to specific stack implementation details are the hooks. For example, LL entries of the NimBLE and Zephyr stacks are different, and so are the data structures storing packet payload. BlueSWAT hooks on the data structures and extracts states for FSM, where states are universal to all BLE stacks. The FSM model and security policies are all in accord with the universal FSM states. Since we implement multiple instruction sets for embedded devices, the eBPF bytecode can be directly executed in eBPF virtual machines across different chips. Therefore, for a vulnerability that affects all stacks, vendors can develop one patch with our universal states and deploy it to all end devices.

4.4.3 Defense Extension. BlueSWAT is highly flexible for vendors to customize, which supports dynamically adding and removing policies. To this end, we introduce BlueSWAT *specifications* which leverage the eBPF map to maintain FSM security policies. Specifically, specifications maintain two sets of (key, value) pairs in the eBPF map: (1) the hooks and the monitored events, and (2) the events and the security policies. Therefore, to modify events and policies, developers only need to change the eBPF map entries with specifications. For example, to remove the transition rule `MYNEWT_INTERVAL` on the connection interval value, developers can simply remove the map entry of (interval, `MYNEWT_INTERVAL`). Then, BlueSWAT will stop inspecting the interval value and remove the policy. Since specifications are also eBPF programs transmitted to BlueSWAT via BLE, the extension operations of BlueSWAT share the advantages of patch distribution, such as avoiding firmware recompilation and introducing limited performance overhead.

5 Implementation

We implement BlueSWAT on 5 real-world devices with mainstream BLE stacks and architectures as shown in Table 1. The open-source ZephyrOS [62] and Apache Mynewt [12] are popular small-footprint kernels designed for resource-constrained and embedded systems, which have been widely adopted by many commercial products [14]. The other stacks are also broadly used in real-world products, which are produced by famous manufacturers including TI [21], Espressif [17] and Bouffalo [13]. The evaluation boards run on mainstream IoT architectures, including ARMv7-M of Cortex-M chips, Xtensa of LX6, and RISC-V of BL618.

We implement BlueSWAT as a kernel extension and install several hooks in the BLE protocol stacks. Due to different manufacturer implementations of the stacks, the number of BlueSWAT hooks slightly changes to meet the demand for comprehensively processing BLE traffic. As shown in Table 2, we hook BlueSWAT into 5 different BLE stacks, among which 3 are closed-source. In general, BlueSWAT needs no more than 5 hooks and requires inserting less than 10 lines of code. In most cases, we need 2 LL RX hooks and 1 LL TX hooks to fully monitor the input traffic because the stacks separately process LL advertising PDUs and data PDUs with L2CAP payload. The reverse engineering process is relatively simple for closed-source stacks (i.e., TI SimpleLink [22] and ESP-IDF [16]) since we only need to locate the LL and SMP entrance functions. We do not need to reverse-engineer the packet payload formats because they are uniform to the Bluetooth Specification. The compact design of hooks significantly reduces the engineering effort to integrate BlueSWAT into the fragmented IoT environment.

Table 1: Real-world devices used in evaluation. Stacks with * are partly closed-source.

Device	Manufacturer	Processor	Architecture	BLE Stack
nRF51833 DK	Nordic.	Cortex-M0	ARMv7-M	NimBLE
CC2640R2	TI.	Cortex-M3	ARMv7-M	SimpleLink*
nRF52840 DK	Nordic.	Cortex-M4	ARMv7-M	Zephyr
ESP32	Espressif.	Xtensa LX6	Xtensa	ESP-IDF*
Sipeed M0P	Bouffalo	BL618	RSIC-V	Bouffalo*

In terms of the eBPF framework, we use several functions from Newlib [10] that are developed as a replacement for libc for embedded platforms. Considering that most embedded platforms do not encompass MMU to offer kernel and user space partitioning similar to the Linux kernel, our eBPF maps eliminate the necessity of copying map data from userspace to kernel. Moreover, we provide Python assembly, disassembly, and Clang compilation tools to support developers in testing and verifying their eBPF programs. We develop a compatible JIT compiler for Cortex-M3+ MCU with the ARM Thumb-2 instruction set. We utilize two 32-bit registers as a singular 64-bit eBPF register to counter the length disparity between Thumb-2 and eBPF instruction sets. Each eBPF instruction is translated into at least two Thumb-2 instructions. For the safety and accuracy of BlueSWAT, we incorporate the SFI mechanism for VM interpretation and JIT mode. It allows us to verify instruction numbers, behaviors, and loop iteration times while in interpretation mode. It enables us to prevent dubious C function calls and malicious out-of-bound memory writing within JIT mode as well.

Table 2: Hooks and lines of code inserted into the stacks.

BLE Stack	# LL Hooks	# SMP Hooks	LoC Inserted
NimBLE	3	2	8
SimpleLink	3	2	8
Zephyr	3	3	9
ESP-IDF	3	2	8
Bouffalo	2	2	6

Changes were made to the existing eBPF VM implementations [48, 54] on embedded devices, adding approximately 1.2k LOC for state machine augmentation. Overall, BlueSWAT encompasses around 2k lines of C code and 1k lines of Python code.

6 Security Analysis

In this section, we evaluate the defense effectiveness of BlueSWAT on real-world attacks and compare it with previous frameworks.
Dataset. We systematically collect 117 BLE vulnerabilities in the dataset by November 2023 (as shown in Figure 8, Appendix E). Around 54% of them are session-based, which is left unstudied by previous research. Specifically, we investigate a wide range of vulnerabilities which include 94 CVEs marked with "ble" and "bluetooth low energy" from the CVE database [15], 16 documented CVEs of open-source BLE stacks [19, 61], and vulnerabilities disclosed by previous publications [26, 34, 40, 59, 71, 75, 81, 85, 89–91]. In the testing dataset, we exclude 16 records for lacking details and threat model mismatch. Specifically, attacks [30, 55, 59, 89, 90] on the BLE physical layer and application layer, such as signal injection and device tracking, are excluded from the testing dataset. In general, the testing dataset contains 101 real-world records, including 46 session-based and 55 packet-based vulnerabilities. We do not claim that our dataset includes all BLE session-based and packet-based attacks, as there may be vulnerabilities that are not publicly studied or assigned CVE records.

6.1 Defense Effectiveness

Table 4 shows the defense effectiveness of BlueSWAT and LBM on our CVE dataset with 101 real-world vulnerabilities. We categorize the vulnerabilities into *Design Flaw* of the Bluetooth Specification and implementation faults of vendor stacks, which include *Function Error* and *Runtime Error*.

BlueSWAT can successfully mitigate 87.1% of them, including 76.1% of session-based and 96.4% of packet-based attacks, which outperforms LBM. Since alarms are only triggered when the FSM transits into malicious states, indicating session exploitation, BlueSWAT introduces no FP on the defendable vulnerabilities. For new attacks, vendors can develop new patches with minimal engineering effort and use BlueSWAT eBPF verifier to check policy correctness and eliminate FN. The unresolvable vulnerabilities are mainly implementation faults of the BLE stacks, such as using weak authentication and no encryption. These problems cannot be fixed unless the manufacturers correctly implement their stacks.

6.1.1 Session-based Attacks Mitigation. Among the 46 session-based CVEs in our dataset, BlueSWAT can effectively mitigate 35 of them. We present the details of defendable session-based attacks in Table 3. The attacks on Specification weaknesses mostly target the BLE pairing procedure to compromise the pairing association models by MITM and impersonation. BlueSWAT successfully mitigates 12 out of 15 Design Flaws that have unique attack patterns. For example, Downgrade attacks [91] always leverage a "Pin or Key Missing" error and a read/write attribute to downgrade the session into using plaintext. We add transition rules to the Key Sharing state, which capture corresponding malicious session patterns and transit FSM to the Pairing Exploitation state. To prevent a BLESA attack, BlueSWAT records the security level of the vulnerable attribute used in previous connections. If a remote device with the address of the server downgrades the security level, BlueSWAT will reject the reconnection, discard the shared key, and perform the pairing procedure again.

In summary, by monitoring the sequential actions with FSM, BlueSWAT can detect session-based attacks based on their corresponding connection patterns. CVE-2019-19194 [2] and CVE-2020-13595 [6] initiate BLE procedures in unexpected orders that maliciously compromise the encryption. With the FSM regulating connection progress, BlueSWAT does not allow out-of-order session behaviors because the transition from the Key Sharing state to the Data Exchange state is not allowed except for no encryption. For replay attacks, BlueSWAT can set a threshold for repeating request and response messages based on specific attack patterns. Repetitive and reflected messages are rarely seen in our observation of benign real-world BLE sessions, although they are not explicitly forbidden by the Specification. For attacks with sequential malicious packets, such as CVE-2019-19192 [1] and CVE-2020-17520 [7], the attacking sessions transit within one benign state and eventually enter exploiting states when they violate corresponding transition rules. Compared to packet-based attacks which we inspect payload fields, preventing session-based attacks needs to model attack patterns as transition paths in FSM and check against predefined transition rules. Caching session states and running additional user-defined rules for verification may invoke performance overhead, but our eBPF framework ensures the introduced memory consumption and runtime latency at a limited level as shown in Section 7.

6.1.2 Packet-based Attacks Mitigation. Among the 55 packet-based CVEs in our dataset, BlueSWAT can effectively mitigate 53 of them as shown in Table 4. Especially, it successfully defends against all BLE bounds check missing and buffer overflow errors, which are the most common vulnerabilities of embedded firmware. Most of these

Table 3: Defense effectiveness of BlueSWAT on session-based BLE attacks. Attacks are grouped into cause and impact. Chip: Affecting multiple chip architectures. Stack: Affecting multiple BLE stacks. D: Discovery, LL: LL Connection, KS: Key Sharing, DE: Data Exchange.

Category	Impact	Vulnerability	Cross-device		Monitered Attack Patterns	Affected States				Mitigation
			Chip	Stack		D	LL	KS	DE	
Design Flaw	Pairing Compromise	BLE KNOB [26]	■	■	Claim smaller key entropy			●	◐	12 / 15 (80.0%)
		BlueMirror [34]	■	■	*BLE-A*: use legacy pairing & reflect commitment scheme			●	◐	
			■	■	*PE-A1*: force PE & reflect Key Exchange & reflect Auth-1&2			●		
			■	■	*PE-A2*: force PE & reflect Key Exchange & reflect Auth-1 & initiate Auth-2			●		
		Keysize Confusion [71]	■	■	Claim different Keysize from history bonds			●	◐	
	Illegal Service Access	BLESA [85]	■	■	*Reactive*: downgrade to no authentication and encryption			●	●	
		Downgrade Attacks [91]	■	■	*Attack I,II,V*: "Pin or Key Missing" error & R/W attributes		◐	●	●	
			■	■	*Attack III,IV,VI*: "Pin or Key Missing" error & IO capability downgrade		◐	●	●	
Function Error	Authentication Bypass	CVE-2022-45190			Bypass passkey entry in legacy pairing			●		19 / 23 (82.6%)
		CVE-2020-12860			Role switch & access identification		◐	●		
		CVE-2023-34625			Duplicate unlocking messages			●	●	
		CVE-2018-16242	■		Replay ciphertext based on predictable nonce				●	
		CVE-2023-26979	■	■	Overwrite Privacy Flag and Reconnection handles		●			
		CVE-2019-13953			Specific exploiting commands		●	●		
		CVE-2019-12500	■		"suddenly accelerate" commands		●	◐	●	
	Key Compromise	CVE-2020-16630	■		Use Just Works & device impersonation	◐		●		
		CVE-2019-19192	■		Sequential Attribute Protocol requests			●		
		CVE-2019-17520			Sequential SM Public Key packets			●		
		CVE-2021-3436	■	■	Maliciously pair & Overwrite existing bond			●		
	Encryption Failure	CVE-2019-19194	■		Out of order LL_ENC_REQ & zero LTK		●	●		
		CVE-2020-13593	■		Link Layer encryption setup before DH checking		●	●	◐	
	Denial of Service	CVE-2020-10068	■	■	Duplicate DLE request packets		●			
		CVE-2021-3430	■	■	Duplicate LL_CONNECTION_PARAM_REQ		●			
		CVE-2021-3431	■	■	Duplicate LL_FEATURE_REQ		●			
		CVE-2018-20957			Replay pairing requests		●	●		
		CVE-2020-27269	■		Replay communication sequences		●	●	◐	
		CVE-2017-13211	■		Repeated BLE scan results	●				
Runtime Error	Bounds Check Missing	CVE-2019-16518			Illegal large power or voltage values				●	4 / 8 (50.0%)
	Buffer Overflow	CVE-2023-23609	■	■	Sequential L2CAP packets		●	◐	●	
	Logic Error	CVE-2022-2993	■	■	Pairing & Not qualified SMP keys		●	●		
		CVE-2020-13595			Discovery & wrong return numbers of completed packets	●	◐	◐		

■: Affecting range of an attack. ●: Target state of an attack. ◐: The state that is not targeted by an attack but is affected at the same time.

errors are packet-based because they can be triggered by one packet with a malformed field. To defend against these attacks, BlueSWAT can easily filter the potential invalid or overflow fields and drop illegal packets. For example, CVE-2021-3432 [8] triggers a division by zero error when the `Interval` field of `CONNECT_IND` PDU is set to 0. BlueSWAT defends against it by applying a sanity check on the `Interval` field and drops malformed packets. For the other packet-based vulnerabilities, BlueSWAT can add the specific exploiting packet payload in security policies and block them when monitoring session context. Compared with LBM which can only filter Bluetooth HCI and L2CAP packets, BlueSWAT extends the defense range to BLE controller which provides additional cover for link layer attacks [5, 8, 9, 82].

6.1.3 Insufficient Fix. There are 13 vulnerabilities in our dataset that BlueSWAT cannot fully mitigate, and we present detailed descriptions in Table 5. The main reason is that these CVEs result from incorrect implementations of the Bluetooth Specification which directly makes BLE functionality not-compliant or incomplete. Specifically, 10 of them are caused by using weak authentication or no authentication and no encryption on sessions. Under the circumstances, BlueSWAT cannot implement authentication and encryption procedures for the stacks. 3 vulnerabilities are caused by improper reconnection handling and attribute permissions misuse of BLE services. The only way to fix these functionality errors is for manufacturers to correctly implement their stacks and replace the buggy firmware.

6.2 Comparison With LBM

We compare the defense effectiveness of BlueSWAT with LBM against both session-based and packet-based attacks. As shown in Table 4, BlueSWAT outperforms LBM in defending against both session-based and packet-based attacks. LBM is vulnerable to all session-based attacks because it cannot capture the sequential patterns of packet sequences. For example, LBM cannot mitigate CVE-2019-19194 and CVE-2020-13593 because it cannot detect out-of-order malicious sequences. Without recording the history of previous connections, LBM cannot detect BLE KNOB because it does not know the previously used key entropy value. When a reduced key entropy or regeneration request is received, LBM cannot compare it to previous ones and considers it a legitimate request if the packet payload is benign. For replay attacks that send duplicate messages, LBM considers them legitimate because it is not aware of the sequential actions in connections. It individually inspects each message and thus can be deceived by replay attacks. Note that BlueSWAT rules operate on FSMs, examining both packet fields and FSM state transitions, while LBM only checks packet fields. Therefore, to mitigate session-based attacks, LBM has to define rules to drop every packet of one kind which will result in a high FP, or even disable normal BLE functions.

Table 4: Comparison of BlueSWAT and LBM for mitigating real-world BLE vulnerabilities in our dataset.

Category	Impact	LBM		BlueSWAT	
		S	P	S	P
Design Flaw	Pairing Compromise	0 / 5	0 / 0	5 / 5	0 / 0
	Illegal Service Access	0 / 10	0 / 0	7 / 10	0 / 0
Function Error	Authentication Bypass	0 / 10	1 / 2	7 / 10	2 / 2
	Key Compromise	0 / 4	0 / 1	4 / 4	1 / 1
	Encryption Failure	0 / 3	0 / 1	2 / 3	1 / 1
	Denial of Service	0 / 6	0 / 0	6 / 6	0 / 0
Runtime Error	Bounds Check Missing	0 / 1	15 / 24	1 / 1	24 / 24
	Buffer Overflow	0 / 1	9 / 18	1 / 1	18 / 18
	Logic Error	0 / 6	5 / 9	2 / 6	7 / 9
Overall	-	0 / 46	31 / 55	35 / 46	53 / 55
Proportion	-	0	56.4%	76.1%	96.4%

S: Session-based vulnerabilities. P: Packet-based vulnerabilities.

In addition, LBM can only mitigate 56.4% of packet-based attacks because it only inspects L2CAP and HCI packet payloads, ignoring the lower controller part of BLE stacks. We note that many attacks (e.g., CVE-2020-10069, CVE-2021-3433, and CVE-2021-3581) exploit LL implementation faults because it is easy to craft plaintext LL packets. Since the pairing and encryption procedures of BLE security are not initiated in the early stages of connections, the LL is completely exposed to a huge attack surface. By contrast, BlueSWAT places hooks in the LL to inspect lower-layer packet payloads. All data to the upper protocols is packed into the LL PDUs, allowing BlueSWAT to universally monitor all session traffic at the entrance of the BLE stacks. We present some descriptions of the packet-based mitigation comparison in Table 7, Appendix A.

In general, LBM can only mitigate 30.7% of all vulnerabilities in our real-world dataset, including 56.4% of packet-based vulnerabilities and none of session-based vulnerabilities. The defense effectiveness of BlueSWAT significantly outperforms LBM. BlueSWAT can successfully defend against 87.1% of all attacks in our dataset, which includes 76.1% of session-based attacks and 96.4% of packet-based attacks.

6.3 Defense Scalability

We articulate the process of deploying BlueSWAT for new devices and its adaptability on various stacks and architectures. We manually inject 3 vulnerabilities (i.e., CVE-2020-10069, CVE-2021-3430, and BLE KNOB) in the 5 evaluation boards described in Table 1.

Firstly, we add BlueSWAT as a component to the kernels and insert the hooks into the stacks. New components can be added to the firmware by modifying a few lines of the project compilation configurations and hooking the stacks requires minimal engineering efforts as demonstrated in Section 5. Secondly, we analyze the sequential patterns of the attacks in terms of FSM events and transition paths. Based on the attack patterns, we write C filter rules and use BlueSWAT compilation tools to compile them into eBPF programs. Then, the eBPF programs are uploaded to the vulnerable devices via our BLE services. The eBPF verifier ensures the new programs do not introduce new vulnerabilities. Finally, we test the effectiveness of the policies by reproducing the attacks and observing the attacking result.

In the end-to-end evaluation, the same set of eBPF security policies successfully functions on different processors and BLE stacks. The cross-architecture adaptability of BlueSWAT is achieved by packaging defense policies in eBPF bytecode, which can universally run on ARMv7-M, RSIC-V, and Xtensa SoCs. The cross-stack deployment is achieved by our universal hooks which block stack implementation details and capture universal session states for policies to inspect.

In a nutshell, with minimal engineering efforts, BlueSWAT can be easily integrated into vendor devices, tested on targeting attacks, and extended across different stacks and chip architectures. Intuitively, vendors need to ensure the policy *effectiveness* (i.e., the policy actually works on the targeting attack). Meanwhile, BlueSWAT eBPF mechanism can automatically check the policy *correctness* (i.e., the policy does not introduce new vulnerabilities).

7 Performance Evaluations

In this section, we perform an evaluation of the runtime performance of BlueSWAT, including memory consumption, runtime overhead and power performance on real-world applications. We conduct performance evaluations with the Zephyr stack [61] on a Nordic nRF52840 DK [70], which has a Cortex-M4 SoC running at 64 MHz, 1 MB Flash, and 256 KB SRAM. We build the firmware, compile transition rules into eBPF bytecode, and run Python scripts on Ubuntu 20.04. We further use Nordic Connect for Desktop [69] to customize the nRF52840 dongle and evaluate two real-world Bluetooth applications.

7.1 Memory Consumption

We use a bare BLE peripheral sample as the experiment baseline. It is provided by the Zephyr project as a barebone for vendors to develop their BLE applications, which takes 182.6 KB Flash and 43.1 KB SRAM with a kernel and bare BLE stack. Compared to real-world IoT firmware, the BLE baseline has a smaller size because it does not employ other components such as LED and display. When BlueSWAT is integrated with no inspection rules, it takes around 12.8 KB Flash memory and 720 B SRAM. The eBPF framework (which includes a VM environment, JIT compiler, verifier, etc.) takes up most of the memory consumption and the FSM is relatively small. When BlueSWAT is integrated with 10 eBPF programs, the size of the firmware increases by 1372 B in Flash, which brings a 0.75% overhead. On average, one eBPF program takes around 137.2 bytes and brings an overhead of around 0.073%. For eBPF program execution, BlueSWAT needs to allocate additional dynamic memory space for code interpretation. When JIT is disabled, BlueSWAT allocates 24 B to VM for each program. When JIT is enabled, BlueSWAT

Table 5: Vulnerabilities that BlueSWAT cannot mitigate.

Category	Vulnerability	State S	State P	Description	Why BlueSWAT cannot fully mitigate
Design Flaw	BLESA Proactive [85]	✔		Reconnection fault.	Cannot fix reconnection procedure.
	Downgrade VII, VIII [91]	✔		Misused attribute permissions.	Cannot modify attribute settings.
Function Error	CVE-2020-12730	✔		Lack BLE encryption.	Cannot implement encryption mechanism.
	CVE-2020-27276	✔		Lack of authentication on the communicating entities.	Cannot implement authentication procedure.
	CVE-2020-27270	✔		Lack of adequate measures to protect encryption keys in transit.	Cannot enforce channel protection.
	CVE-2017-18642	✔		Receive RGB parameters over cleartext.	Cannot implement encryption mechanism.
Runtime Error	CVE-2018-10825		✔	No authentication and encryption.	Cannot implement authentication and encryption.
	CVE-2017-17436		✔	No encryption.	Cannot implement encryption mechanism.
	CVE-2017-17435	✔		Fake authentication.	Cannot implement authentication mechanism.
	CVE-2020-27264	✔		Use deterministic keys.	Cannot fix usage of static keys.
	CVE-2018-20958	✔		Rely on public information for private operations.	Cannot hide MAC address.
	CVE-2020-11957	✔		Low entropy key.	Cannot fix the key generation procedure.

S: Session-based vulnerabilities. P: Packet-based vulnerabilities.

allocates an additional 217.8 B on average for each program as shown in Figure 10, Appendix F. Note that the dynamic memory is allocated when the program is executed and destroyed when finished. Hence, the dynamic memory consumption is controllable even if more programs are further added.

On average, one ePBF program takes up 137.2 B Flash memory (less than 0.08% overhead) and 217.8 B dynamic memory, which can be considered controllable.

7.2 Runtime Latency

7.2.1 Micro benchmark. We first load one rule (conn_chan_map) in BlueSWAT and generate 1k packets on the Bluetooth RX path to evaluate runtime latency. For the scalability test, we further load 10 different rules in BlueSWAT and generate 1k packets for evaluation. Both benchmarks are tested under VM interpretation mode and Just-In-Time (JIT) mode. As shown in Table 6, when one rule is loaded, the average latency is 1.219 μs (78 MCU cycles) under interpretation mode and 1.172 μs (75 MCU cycles) under JIT mode. The latency variance is 0.44 under interpretation mode and drops to 0.31 when JIT is on. The maximum overhead is controlled under 5 μs. In the scalability test, we add 10 different rules to BlueSWAT and the average latency in interpretation mode slightly rises to 1.266 μs (81 MCU cycles) while with JIT it drops to 1.094 μs (70 MCU cycles). In summary, BlueSWAT introduces negligible latency and maintains the runtime performance when scaling up.

Table 6: The runtime latency (us) introduced by BlueSWAT.

Benchmark	Runtime Latency (us) Min.	Max.	Med.	Avg.	Var.
BlueSWAT-1	0.844	4.969	1.031	**1.219**	0.44
BlueSWAT-1-JIT	1.031	4.953	1.031	**1.172**	0.31
BlueSWAT-10	0.906	4.891	0.922	**1.266**	0.89
BlueSWAT-10-JIT	0.906	4.984	0.922	**1.094**	0.45

We further use l2ping [58] on Linux to perform end-to-end evaluation on BlueSWAT. We use l2ping to send 1K random L2CAP packets to our nRF52840 DK and evaluate the round-trip time (RTT) of each request. As shown in Figure 11, Appendix F, the average RTT of 1K packets in the baseline is 2.099 ms, which is significantly larger than the latency introduced by BlueSWAT at the microsecond level. In the VM interpretation and JIT mode, the overall distribution of request RTT is similar to the baseline while the average RTT drops. The fluctuation of request RTT almost reaches 4 ms which indicates the latency introduced by different payloads impacts the time consumption more than BlueSWAT on each request. Since not all requests would trigger BlueSWAT inspection and the maximum latency of each rule is less than 5 μs, the fluctuation at the millisecond level demonstrates that BlueSWAT barely impacts the runtime performance of the stack in end-to-end connection. When we add 10 transition rules, the time consumption of 1K requests further drops in both VM interpretation and JIT mode. It is because the first rule we load in BlueSWAT-1 inspects the link layer packet header field which can be invoked by most packets. The rest of the rules used in BlueSWAT-10 aim at complicated session-based attacks, such as inspecting redundant SMP keys for the same peer address, which are triggered less frequently than the previous one. Hence, BlueSWAT maintains a stable latency performance when scales to more transition rules against newly disclosed attacks. In all cases, the interquartile range lies above 1.3 ms and the minimum RTT is higher than 400 μs. Combined with the above analysis results, the micro benchmark demonstrates that BlueSWAT introduces a negligible latency overhead in end-to-end communications.

7.2.2 Macro benchmark. We test the performance of BlueSWAT in two real-world BLE applications which are Battery Level Service (BAS) and Heart Rate Service (HRS). We use the Nordic Connect for Desktop [69] to invoke the services 1K times and record the real-time RTT of the applications. Our test conditions are stricter than real-world scenarios because normally HRS on a smartwatch does not need to update the data frequently. It may take the average heart rate of the user in a 10-second interval while we let the applications continuously refresh and update the values in a 3.5 ms interval. A longer interval in a real-world scenario will relatively reduce the latency impact of BlueSWAT.

We load 10 different eBPF programs for both applications and test under VM interpretation and JIT mode. The CDF diagrams of the evaluation results are shown in Figure 5, Appendix B. For BAS, the average RTT in the baseline is 3489.7 μs and P99.5 of RTT is

3740.62 μs. The baseline time consumption is significantly larger than the latency of BlueSWAT shown in Table 6, which indicates that BlueSWAT introduces a negligible latency. Under VM interpretation and JIT mode, the average RTT of BAS increases 12 μs (0.32%) which roughly matches the time consumption increase of 10 rules. Running eBPF programs in JIT mode does not explicitly improve the latency performance but requires additional space consumption as shown in Figure 10, Appendix F. Hence, VM interpretation is enough for simple transition rules that we deploy in our prototype. For HRS, the average RTT in the baseline is 3487.82 μs and the P99.5 of RTT is 3736.62 μs. The time consumption in the baseline is also significantly larger than the latency of BlueSWAT shown in Table 6, which indicates a minimal impact on HRS. Under VM interpretation and JIT mode, the average RTT of HRS increases around 18.6 μs (0.50%) and 10.8 μs (0.29%). In this case, JIT mode saves around 7.8 μs on average application RTT, which roughly matches the time difference in BlueSWAT-10 and BlueSWAT-10-JIT shown in Table 6. In summary, the runtime latency that BlueSWAT introduces to real-world applications is limited to less than 18.6 μs (overhead of 0.50%) and can be considered negligible.

7.3 Power Performance

We access the power and energy performance of BlueSWAT over a 120-second window, encompassing four phases: 20s of connection, 40s of BAS, another 20s of connection, and 40s of HRS. To measure real-time processing power, energy use, and capacity of BlueSWAT, we employed a ChargerLAB POWER-Z KT002 device [31] with a sampling rate of 2.5 times per second. We use a bare Zephyr BLE project as the evaluation baseline. For BlueSWAT, we load 10 security policies and enable the JIT compiler. The evaluation results are presented in Figure 9, Appendix F.

The evaluation reveals that the average power consumption of the baseline is 0.3935 W. BlueSWAT introduces an average of 0.0009 W more power than the baseline, representing a 2.29% increase. This additional power draw arises from enhanced functionalities of BlueSWAT. Since the baseline only runs fundamental BLE functions, the power overhead of BlueSWAT will decrease as vendors deploy their real-world applications with a wider range of BLE features. As shown in Figure 9(b) and 9(c), Appendix F, the energy and capacity consumption rate of BlueSWAT is almost the same as the baseline. In the 120-second evaluation window, the baseline consumes 13.09 mVh of energy and 2.59 mAh of battery capacity. After deploying BlueSWAT, the device consumes 13.12 mVh of energy and 2.59 mAh of battery capacity. Nordic 52840 DK uses a Renata CR2032.MFR battery [66] with a capacity of 260 mAh. Under the circumstances, the battery can support the Nordic board running BlueSWAT for approximately 3.35 hours straight. In a nutshell, BlueSWAT introduces a 0.03 mWh (0.23%) of energy consumption overhead and minimal overhead on battery capacity usage. Therefore, the impact on device performance and battery life of BlueSWAT is negligible.

8 Discussion

Application on Bluetooth Classic. BlueSWAT is potentially effective in Bluetooth Classic (BC) defense because BC and BLE share similar protocol architecture and security procedures, such as the link layer (Link Manager in BC) and the pairing procedure. Currently, we focus on the BLE protocol because there is no available open-source BC stack on embedded devices for evaluation. Many IoT devices use tailored Linux kernels with BlueZ stack [27], but they do not openly share their code. Research about BC attacks can overcome non-public protocol stack issues because verifying attacking performance does not need firmware source code. But for a defense framework like BlueSWAT, it requires heavy and tedious efforts to inject vulnerabilities to perform defense evaluation. Nevertheless, we plan to expand BlueSWAT to BC once there are available open-source stacks.

Policy Correctness and Effectiveness. As explained in Section 6.3, a successful mitigation with BlueSWAT demands the *correctness* and *effectiveness* of the policy. Correctness denotes that the policy itself does not include vulnerabilities that will lead to new attacks. Effectiveness signifies that the policy can actually mitigate the targeting attack if it is successfully deployed to the device.

Vendors have the responsibility to make sure that their systems are correct and vulnerability-free. When new vulnerabilities are disclosed, BlueSWAT allows the vendors to rapidly patch it. Therefore, vendors should verify the correctness of patches and ensure the correctness of BlueSWAT policies, which is consistent with existing studies [32, 48, 79]. Otherwise, if the policy has fundamentally misinterpreted the attack, the defense is destined to fail. Then, vendors can use our automatic compiler to compile C policies into eBPF bytecode, while BlueSWAT automatically guarantees the correctness of the policies as mentioned in Section 5.

A high-level Domain Specific Language (DSL) can potentially facilitate the rule creation process of BlueSWAT, as shown in Appendix D. After vendors identify the transition paths of the attacks, they write C transition rules with the API of BlueSWAT FSM, which provides access to FSM events, states, and variables. The policies of BlueSWAT now contain 20 LOC on average and mostly in a similar format. In Figure 7(b), we present a DSL based on Lua [67] which can further restrict the policy format and facilitate the development progress. The Lua scripts can be compiled into C with Python first and then compiled into eBPF programs with our compilation tools automatically, which will ensure policy correctness.

9 Related work

BLE Attacks. The research on BLE attacks mainly falls into two categories based on the target: firmware compromise [68, 75] and Specification exploitation [25, 29, 34, 81, 91]. SweynTooth [40] built a general fuzzing framework for BLE implementations and found 24 new vulnerabilities across multiple vendor products. BLEEDINGBIT [56] unveiled a set of critical chip-level vulnerabilities affecting BLE chips manufactured by Texas Instruments. It is a typical session-based attack that sends benign packets to store malicious code on the victim device in advance, then overflows the stack with an exploiting packet. In recent years, the community also revealed some BLE Specification design flaws that threaten millions of standard-compliant products. Antonioli et al. [26] demonstrated the effectiveness of low key entropy attack on standard compliant BLE devices which is similar to the well-known KNOB [24] attack on Bluetooth Classic devices. Relay attacks were found capable of circumventing latency bounding or encryption on the BLE link layer and compromising Tesla cars [52, 89]. BLESA [85] introduced two spoofing attacks targeting the BLE link layer, where an attacker impersonates a previously paired device and feeds forged data to the victim. BLE physical layer was also found vulnerable to signal eavesdropping, injection, and device tracking attacks [30, 42, 59, 77, 78, 90]. Many of the BLE attacks, especially those targeting Specification flaws, are session-based and cannot be detected by existing stateless packet filtering systems.

BLE Security. The research on BLE security mainly falls into two categories, which are protocol analysis and defense solutions.

Formal analysis methods [49, 71, 80, 88] built various verification models of the protocol, which mostly focused on the pairing procedure, and uncovered several Specification design flaws. However, their proposed mitigation mostly required revising the Specification which resulted in a long patching window, and some were deprecated by Bluetooth SIG due to backward compatibility [84, 88]. Besides, the traditional software update mechanism demands device reboot and firmware recompilation, which is sometimes impractical for resource-constrained IoT devices. BLEDiff [51] proposed a protocol differential testing framework that can identify deviant non-compliant behaviors between multiple BLE implementations. By contrast, we aim to mitigate attacks after vulnerability disclosure via cross-platform instant deployment and stateful traffic inspection.

Existing defense solutions are not as comprehensive as attacking works because a defense framework must consider many restrictions, such as source code availability, backward compatibility, performance overhead, and demand significant engineering workloads. The most comparable solution to BlueSWAT is LBM [79], a Linux subsystem hardening peripheral protocols (e.g., USB, Bluetooth, NFC) by traffic filtering. Though LBM and BlueSWAT share partly different scopes, in terms of BLE defense, LBM is not capable of monitoring sequential actions of session context, and hence vulnerable to BLE session-based attacks. It also overlooks the link layer which controls the fundamental procedures of BLE connections. Apart from LBM, LightBlue [87] reduced the attack surface of Bluetooth stacks by debloating unneeded code. However, the remaining firmware is still exploitable without comprehensively monitoring the connection messages. ProFactory [83] can automatically generate secure low-level protocol implementations for IoT protocols in Linux kernels from protocol specifications. However, it only eliminates basic implementation bugs such as memory safety bugs (i.e., buffer overflow, memory leakage, etc.) and concurrency control vulnerabilities (i.e., race and deadlock). The security-critical procedures of BLE, such as pairing and encryption, are left unstudied. BlueShield [86] proposed a defense strategy for spoofing attacks based on physical patterns of BLE packets. Sadly, it needs to deploy additional hardware and must work in a stationary network, which overlooks numerous mobile BLE devices such as wearables.

IoT Peripheral Security. Several works [33, 57, 79, 83] focus on IoT security and take BLE as one of the communication protocols that peripherals may choose, such as USB, WiFi, and Bluetooth. These solutions usually consider the common features of different wireless protocols and hence fail to study BLE security in a fine-grained way. For instance, T2Pair [57] proposed the Universal Operation Sensing which allows IoT users to pair their devices without inertial sensors. However, it ignored the fact that a large amount of BLE attacks happen beyond the pairing phase.

10 Conclusion

We propose BlueSWAT, a lightweight state-aware security framework for IoT devices, which can mitigate BLE session-based attacks. BlueSWAT captures the patterns of session-based attacks and models them as malicious FSM transition paths. During BLE connections, BlueSWAT monitors session sequential actions at runtime, performing an inspection on the session level in addition to separate packets. We develop a lightweight eBPF mechanism to support resource-constrained embedded devices across heterogeneous IoT architectures. For a vulnerability that affects multiple devices, vendors can develop one common patch and distribute it to all victims. Besides, BlueSWAT patches are transmitted as eBPF programs via BLE, which avoids firmware recompilation and device reboot. We implement BlueSWAT on 5 real-world devices with different stacks and chips. On our dataset with 101 real-world BLE vulnerabilities, BlueSWAT can mitigate 76.1% of session-based attacks and 96.4% of packet-based attacks, which outperforms existing stateless frameworks. In our end-to-end application evaluation, BlueSWAT patches introduce less than 0.08% of memory overhead on average and negligible latency.

Acknowledgments

We appreciate the valuable feedback of anonymous reviewers. This research is supported in part by NSFC under Grant 62425201. Qi Li is the corresponding author of this paper.

References

[1] 2019. CVE-2019-19192. https://nvd.nist.gov/vuln/detail/CVE-2019-19192. (2019).
[2] 2019. CVE-2019-19194. https://nvd.nist.gov/vuln/detail/CVE-2019-19194. (2019).
[3] 2020. CVE-2020-10065. https://nvd.nist.gov/vuln/detail/CVE-2020-10065. (2020).
[4] 2020. CVE-2020-10066. https://nvd.nist.gov/vuln/detail/CVE-2020-10066. (2020).
[5] 2020. CVE-2020-10069. https://nvd.nist.gov/vuln/detail/CVE-2020-10069. (2020).
[6] 2020. CVE-2020-13595. https://nvd.nist.gov/vuln/detail/CVE-2020-13595. (2020).
[7] 2020. CVE-2020-17520. https://nvd.nist.gov/vuln/detail/CVE-2020-17520. (2020).
[8] 2021. CVE-2021-3432. https://nvd.nist.gov/vuln/detail/CVE-2021-3432. (2021).
[9] 2021. CVE-2021-3433. https://nvd.nist.gov/vuln/detail/CVE-2021-3433. (2021).
[10] 2021. Newlib: C library intended for use on embedded systems. https://sourceware.org/newlib/. (2021).
[11] 2022. WebAssembly. https://webassembly.org/. (2022).
[12] 2023. Apache MynewtOS. https://mynewt.apache.org. (2023).
[13] 2023. Bouffalo Lab. https://en.bouffalolab.com. (2023).
[14] 2023. Commercial products using ZephyrOS. https://www.zephyrproject.org/products-running-zephyr/. (2023).
[15] 2023. CVE Database. https://cve.mitre.org. (2023).
[16] 2023. ESP-IDF. https://github.com/espressif/esp-idf. (2023).
[17] 2023. Espressif. https://www.espressif.com. (2023).
[18] 2023. MicroPython: Python for microcontrollers. https://micropython.org/. (2023).
[19] 2023. Mynewt Nimble. https://mynewt.apache.org/latest/network/. (2023).
[20] 2023. Riot.js: Simple and elegant component-based UI library. https://riot.js.org/. (2023).
[21] 2023. Texas Instrument. https://www.ti.com. (2023).
[22] 2023. Texas Instrument BLE Stack. https://www.ti.com/tool/BLE-STACK. (2023).
[23] 2023. Userspace eBPF VM. https://github.com/iovisor/ubpf. (2023).
[24] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Bonne Rasmussen. 2019. The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR. In *USENIX Security Symposium*.
[25] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Bonne Rasmussen. 2020. BLURtooth: Exploiting Cross-Transport Key Derivation in Bluetooth Classic and Bluetooth Low Energy. *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security* (2020).

[26] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Bonne Rasmussen. 2020. Key Negotiation Downgrade Attacks on Bluetooth and Bluetooth Low Energy. *ACM Transactions on Privacy and Security (TOPS)* 23 (2020), 1 – 28.

[27] The Linux Kernel Archives. 2023. Linux BlueZ stack. http://www.bluez.org/. (2023).

[28] Johannes K. Becker, David Li, and David Starobinski. 2019. Tracking Anonymized Bluetooth Devices. *Proceedings on Privacy Enhancing Technologies* 2019 (2019), 50 – 65.

[29] Eli Biham and Lior Neumann. 2019. Breaking the Bluetooth Pairing – The Fixed Coordinate Invalid Curve Attack. In *Selected Areas in Cryptography – SAC 2019: 26th International Conference, Waterloo, ON, Canada, August 12–16, 2019, Revised Selected Papers.* 250–273.

[30] Romain Cayre, Florent Galtier, and Guillaume Auriol. 2021. InjectaBLE: Injecting malicious traffic into established Bluetooth Low Energy connections. *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2021), 388–399.

[31] ChargerLab. 2021. PowerZ KT002. https://www.chargerlab.com/category/power-z/power-z-kt002/. (2021).

[32] Yaohui Chen, Yuping Li, and Long Lu. 2018. InstaGuard: Instantly Deployable Hot-patches for Vulnerable System Programs on Android. In *Network and Distributed System Security Symposium.*

[33] Haotian Chi, Qiang Zeng, and Xiaojiang Du. 2021. PFirewall: Semantics-Aware Customizable Data Flow Control for Smart Home Privacy Protection. *ArXiv* abs/2101.10522 (2021).

[34] Tristan Claverie and José Lopes-Esteves. 2021. BlueMirror: Reflections on Bluetooth Pairing and Provisioning Protocols. *2021 IEEE Security and Privacy Workshops (SPW)* (2021), 339–351.

[35] Nicholas DeMarinis, Kent Williams-King, and Di Jin. 2020. sysfilter: Automated System Call Filtering for Commodity Software. In *International Symposium on Recent Advances in Intrusion Detection.*

[36] Danny Dolev and Andrew Chi-Chih Yao. 1983. On the security of public key protocols. *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)* (1983).

[37] W. Findlay, David Barrera, and Anil Somayaji. 2021. BPFContain: Fixing the Soft Underbelly of Container Security. *ArXiv* abs/2102.06972 (2021).

[38] W. Findlay, Anil Somayaji, and David Barrera. 2020. bpfbox: Simple Precise Process Confinement with eBPF. *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop* (2020).

[39] The Linux Foundation. 2022. Dynamically program the kernel for efficient networking, observability, tracing, and security. https://ebpf.io/. (2022).

[40] Matheus E. Garbelini, Chundong Wang, and Sudipta Chattopadhyay. 2020. Sweyn-Tooth: Unleashing Mayhem over Bluetooth Low Energy. In *USENIX Annual Technical Conference.*

[41] Seyedhamed Ghavamnia, Tapti Palit, and Shachee Mishra. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *USENIX Security Symposium.*

[42] Hadi Givehchian, Nishant Bhaskar, and Herrera. 2022. Evaluating Physical-Layer BLE Location Tracking Attacks on Mobile Devices. In *2022 IEEE Symposium on Security and Privacy (SP).*

[43] Sasha Goldshtein. 2016. The Next Linux Superpower: eBPF Primer. USENIX Association, Dublin.

[44] Brendan Gregg. 2017. Performance Superpowers with Enhanced BPF. USENIX Association, Santa Clara, CA.

[45] Brendan Gregg. 2019. BPF Performance Tools. https://www.brendangregg.com/bpf-performance-tools-book.html. (2019).

[46] Keijo Haataja and Pekka Toivanen. 2010. Two practical man-in-the-middle attacks on Bluetooth secure simple pairing and countermeasures. *IEEE Transactions on Wireless Communications* 9 (2010).

[47] Yi He, Roland Guo, and Yunlong Xing. 2023. Cross Container Attacks: The Bewildered eBPF on Clouds. In *32nd USENIX Security Symposium (USENIX Security 23).*

[48] Yi He, Zhenhua Zou, and Kun Sun. 2022. RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices. In *31st USENIX Security Symposium (USENIX Security 22).* 2225–2242.

[49] Mohit Kumar Jangid, Yue Zhang, and Zhiqiang Lin. 2023. Extrapolating Formal Analysis to Uncover Attacks in Bluetooth Passkey Entry Pairing. In *Proceedings 2023 Network and Distributed System Security Symposium.*

[50] Wenqiang Jin, Ming Li, and Srinivasan Murali. 2020. Harnessing the Ambient Radio Frequency Noise for Wearable Device Pairing. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (2020).

[51] I. Karim, A. Ishtiaq, and S. Hussain. 2023. BLEDiff : Scalable and Property-Agnostic Noncompliance Checking for BLE Implementations. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP).*

[52] Sultan Khan. 2022. Tesla BLE Phone-as-a-Key Passive Entry Vulnerable to Relay Attacks. https://research.nccgroup.com/2022/05/15/technical-advisory-tesla-ble-phone-as-a-key-passive-entry-vulnerable-to-relay-attacks/. (2022).

[53] Taesoo Kim and Nickolai Zeldovich. 2013. Practical and Effective Sandboxing for Non-root Users. In *2013 USENIX Annual Technical Conference (USENIX ATC 13).* USENIX Association, 139–144.

[54] Zandberg Koen, Baccelli Emmanuel, and Yuan Shenghao. 2022. Femto-Containers: Lightweight Virtualization and Fault Isolation for Small Software Functions on Low-Power IoT Microcontrollers. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference.* 161–173.

[55] Changseok Koh, Jonghoon Kwon, and Junbeom Hur. 2022. BLAP: Bluetooth Link Key Extraction and Page Blocking Attacks. *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2022), 227–238.

[56] Armis Lab. 2018. BLEEDINGBIT vulnerabilities. https://www.armis.com/research/bleedingbit/. (2018).

[57] Xiaopeng Li, Qiang Zeng, and Lannan Luo. 2020. T2Pair: Secure and Usable Pairing for Heterogeneous IoT Devices. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (2020).

[58] Linux Command Library. 2023. L2ping man. https://linuxcommandlibrary.com/man/l2ping. (2023).

[59] Norbert Ludant, Tien D. Vo-Huu, and Sashank Narain. 2021. Linking Bluetooth LE & Classic and Implications for Privacy-Preserving Bluetooth-Based Protocols. In *2021 IEEE Symposium on Security and Privacy (SP).*

[60] Tal Melamed. 2017. BLE Application Hacking. https://owasp.org/www-pdf-archive/OWASP2017_HackingBLEApplications_TalMelamed.pdf. (2017).

[61] Zephyr Project members and individual contributors. 2023. Zephyr Bluetooth. https://docs.zephyrproject.org/latest/connectivity/bluetooth/index.html. (2023).

[62] Zephyr Project members and individual contributors. 2023. Zephyr Project. https://www.zephyrproject.org. (2023).

[63] Luke Nelson, Jacob Van Geffen, and Emina Torlak. 2020. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *USENIX Symposium on Operating Systems Design and Implementation.*

[64] Trung Nguyen and Jean Leneutre. 2014. Formal Analysis of Secure Device Pairing Protocols. *2014 IEEE 13th International Symposium on Network Computing and Applications* (2014), 291–295.

[65] Haram Park and Carlos Kayembe Nkuba. 2022. L2Fuzz: Discovering Bluetooth L2CAP Vulnerabilities Using Stateful Fuzz Testing. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).*

[66] renata. 2024. CR2032.MFR Battery. https://www.renata.com/en-us/products/lithium-batteries/cr2032.mfr-/. (2024).

[67] PUC RIO. 2024. Lua The Programming Language. https://www.lua.org/. (2024).

[68] Jan Ruge, Jiska Classen, and Francesco Gringoli. 2020. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. *ArXiv* (2020).

[69] Nordic Semiconductor. 2023. nRF Connect for Desktop. https://www.nordicsemi.com/Products/Development-tools/nrf-connect-for-desktop. (2023).

[70] Nordic Semiconductor. 2023. nRF52840 DK. https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dk. (2023).

[71] Min Shi, Jing Chen, Haoran Zhao, Menglin Jia, and Ruiying Du. 2023. Formal Analysis and Patching of BLE-SC Pairing. In *USENIX Security Symposium.*

[72] Bluetooth SIG. 2023. Bluetooth Core Specification 4.0. https://www.bluetooth.com/specifications/specs/core-specification-4-0/. (2023).

[73] Bluetooth SIG. 2023. Bluetooth Core Specification v5.4. https://www.bluetooth.com/specifications/specs/core-specification-5-4/. (2023).

[74] Bluetooth SIG. 2024. 2024 Bluetooth Market Update. https://www.bluetooth.com/2024-market-update/. (2024).

[75] Pallavi Sivakumaran and Jorge Blasco. 2018. A Study of the Feasibility of Co-located App Attacks against BLE and a Large-Scale Analysis of the Current Application-Layer Security Landscape. In *USENIX Security Symposium.*

[76] Chengyu Song, Chao Zhang, and Tielei Wang. 2015. Exploiting and Protecting Dynamic Code Generation. In *Network and Distributed System Security Symposium.*

[77] Milan Stute, A. Heinrich, and Jan-Hendrik Lorenz. 2021. Disrupting Continuity of Apple's Wireless Ecosystem Security: New Tracking, DoS, and MitM Attacks on iOS and macOS Through Bluetooth Low Energy, AWDL, and Wi-Fi. In *USENIX Security Symposium.*

[78] Milan Stute, Sashank Narain, and Alex Mariotto. 2019. A Billion Open Interfaces for Eve and Mallory: MitM, DoS, and Tracking Attacks on iOS and macOS Through Apple Wireless Direct Link. In *USENIX Security Symposium.*

[79] Dave Jing Tian, Grant Hernandez, and Joseph I. Choi. 2019. LBM: A Security Framework for Peripherals within the Linux Kernel. *2019 IEEE Symposium on Security and Privacy (SP)* (2019), 967–984.

[80] Michael Troncoso and Britta Hale. 2021. The Bluetooth CYBORG: Analysis of the Full Human-Machine Passkey Entry AKE Protocol. In *Proceedings 2021 Network and Distributed System Security Symposium.*

[81] Maximilian von Tschirschnitz, Ludwig Peuckert, and Fabian Franzen. 2021. Method Confusion Attack on Bluetooth Pairing. *2021 IEEE Symposium on Security and Privacy (SP)* (2021), 1332–1347.

[82] SwenyTooth BLE Vulnerabilities. 2020. CVE-2020-10061. https://nvd.nist.gov/vuln/detail/CVE-2020-10061. (2020).

[83] Fei Wang, Jian-Liang Wu, and X. Zhang. 2022. ProFactory: Improving IoT Security via Formalized Protocol Customization. In *USENIX Security Symposium.*

[84] J. Wu, R. Wu, D. Xu, and D. Tian. 2024. SoK: The Long Journey of Exploiting and Defending the Legacy of King Harald Bluetooth. In *2024 IEEE Symposium on Security and Privacy (SP)*. 23–23.
[85] Jian-Liang Wu, Yuhong Nan, and Vireshwar Kumar. 2020. BLESA: Spoofing Attacks against Reconnections in Bluetooth Low Energy. In *WOOT @ USENIX Security Symposium*.
[86] Jian-Liang Wu, Yuhong Nan, and Vireshwar Kumar. 2020. BlueShield: Detecting Spoofing Attacks in Bluetooth Low Energy Networks. In *International Symposium on Recent Advances in Intrusion Detection*.
[87] Jian-Liang Wu, Ruoyu Wu, and Daniele Antonioli. 2021. LIGHTBLUE: Automatic Profile-Aware Debloating of Bluetooth Stacks. In *USENIX Security Symposium*.
[88] Jian-Liang Wu, Ruoyu Wu, and Dongyan Xu. 2022. Formal Model-Driven Discovery of Bluetooth Protocol Design Vulnerabilities. *2022 IEEE Symposium on Security and Privacy (SP)* (2022), 2285–2303.
[89] Xinyi Xie, Kun Jiang, Rui Dai, Jun Lu, Lihui Wang, Qing Li, and Jun Yu. 2023. Access Your Tesla without Your Awareness: Compromising Keyless Entry System of Model 3. *Proceedings 2023 Network and Distributed System Security Symposium* (2023). https://api.semanticscholar.org/CorpusID:257502883
[90] Yue Zhang and Zhiqiang Lin. 2022. When Good Becomes Evil: Tracking Bluetooth Low Energy Devices via Allowlist-Based Side Channel and Its Countermeasure. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 3181–3194.
[91] Yue Zhang, Jian Weng, and Rajib Dey. 2020. Breaking Secure Pairing of Bluetooth Low Energy Using Downgrade Attacks. In *USENIX Security Symposium*.

Appendix

A Additional Comparison result

B Real-world Benchmark Result

C BLE Connection

In this section, we provide some background information about BLE connections. To establish a secure connection, two devices must go through the following phases.

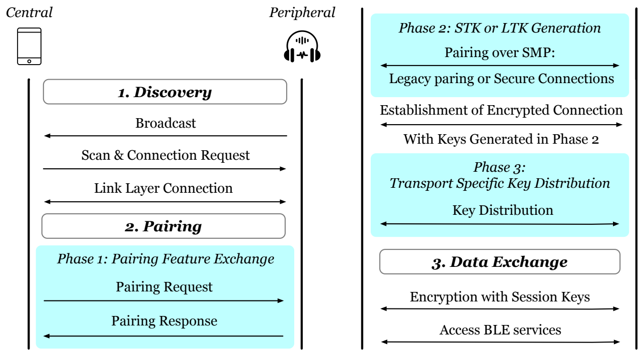

Central
Peripheral
1. Discovery
Broadcast
Scan & Connection Request
Link Layer Connection
2. Pairing
Phase 1: Pairing Feature Exchange
Pairing Request
Pairing Response
Phase 2: STK or LTK Generation
Pairing over SMP:
Legacy paring or Secure Connections
Establishment of Encrypted Connection
With Keys Generated in Phase 2
Phase 3: Transport Specific Key Distribution
Key Distribution
3. Data Exchange
Encryption with Session Keys
Access BLE services

Figure 6: BLE connection process.

Discovery. When the Bluetooth function is enabled, the device first goes into the discovery phase. During discovery, the peripheral (e.g., a headset) starts to send advertising packets, which contain the device address, protocol version, etc. The central (e.g., a smartphone) scans for advertising packets in radio range, and sends out connection requests to target devices. In the discovery phase, the two devices exchange several supporting features and establish a link layer connection at last.

Pairing. After establishing 6 a LL connection, the central and peripheral try to negotiate and distribute a shared secret key with the SMP pairing procedure. The pairing procedure includes three phases, which are Pairing Feature Exchange (Phase 1), Short Term Key (STK) or Long Term Key (LTK) Generation (Phase 2), and Transport Specific Key Distribution (Phase 3).

In Phase 1, the devices first exchange authentication requirements and IO capabilities to determine the association models and pairing methods used in Phase 2. OOB authentication data availability, key size requirements, and which transport specific keys to distribute are also exchanged in Phase 1.

In Phase 2, the Bluetooth Specification provides two pairing protocols, which are the *Secure Connections Pairing* (SCP) and *legacy pairing*. Released in Bluetooth v4.0, the legacy pairing protocol generates two keys, which are Temporary Key and STK, to encrypt the connection. Since it does not use the Diffie-Hellman Key Exchange (DHKE) protocol during key sharing, legacy pairing is vulnerable to passive attacks. In Bluetooth v4.2, SCP is introduced to mitigate the weakness by adopting the DHKE protocol. There are four pairing methods to use during pairing: *Just Works*, *Out of Band*, *Passkey Entry*, and *Numeric Comparison*. The methods with higher security levels involve user interaction to enhance the authentication procedure, which can mitigate eavesdropping and the MITM attack. SCP is found vulnerable to reused passkey attacks and reflection attacks and receives mitigations through Specification updates. At the end of Phase 2, two devices establish an encrypted connection with the shared STK or LTK.

In phase 3, two devices may distribute transport specific keys, such as the Identity Resolving Key (IRK) value and Identity Address information. Phase 1 and Phase 3 are identical regardless of the method used in Phase 2.

Data Exchange. Once the central and peripheral exchange the sharing keys, devices can start the encryption and authentication procedure. To derive session keys from Link Keys, BLE performs a proactive or reactive authentication procedure. The session keys are then used for connection encryption and are refreshed every time the devices reconnect.

D Domain Specific Language

E BLE vulnerabilities

In this section, we provide the investigation result of BLE vulnerabilities. We systematically collect 117 BLE vulnerabilities in our CVE dataset by November 2023 (as shown in Figure 8, Appendix E). Specifically, we investigate a wide range of vulnerabilities which include 94 CVEs marked with "ble" and "bluetooth low energy" from the CVE database [15], 16 documented CVEs of open-source BLE stacks [19, 61] and the vulnerabilities disclosed by previous publications [26, 34, 40, 59, 71, 75, 81, 85, 89–91]. We aim to protect the firmware layer and host layer of the BLE protocol [84]. Attacks [30, 55, 59, 89, 90] on the BLE physical layer and application layer, such as signal injection and device tracking, are out of the scope.

Around 54% of the vulnerabilities are session-based, which is left unstudied by previous research. Specifically, all Specification weaknesses are session-based because they need complicated interactions to be exploited. These weaknesses cause severe consequences, such as MITM session hijacks and encryption breaks, which threaten all BLE devices. Besides, Function Errors and Access Control Flaws that are caused by implementation faults are usually session-based. To exploit these vulnerabilities, the attackers need to carefully craft malicious message sequences. Replay attacks are

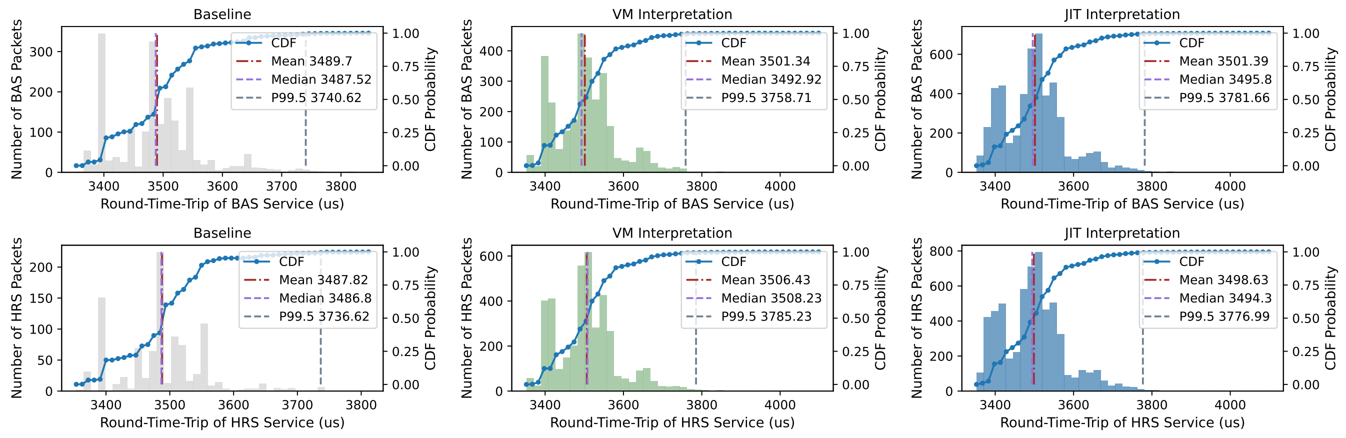


Figure 5: RTT (ms) in real-world applications macro benchmark.

Table 7: Comparison between BlueSWAT and LBM on mitigating packet-based attacks.

Category	CVE	Attack Patterns	LBM	BlueSWAT
Bounds Check Missing	CVE-2020-10069	Invalid Channel Map field	✗	✓
Buffer Overflow	CVE-2020-10065	Overflow HCI_ACL header field	✓	✓
Function Error	CVE-2021-3433	Invalid Hop Map field	✗	✓
Weak Configuration	CVE-2019-2102	Hardcoded Long Term Key	✗	✓
Corrupted Pointer	CVE-2022-41972	Invalid L2CAP channel ID field	✓	✓
Other (Lack of length check)	CVE-2021-3581	Overflow SCAN_REQ payload	✗	✓

```
1   #define BT_KEYS_LTK_P256 32
2   #define BT_KEYS_LTK 4
3   #define BT_KEYS_AUTHENTICATED 1
4
5   uint64_t zephyr_filter(uint8_t *newState)
6   {
7         struct FsmState *fsm = (struct FsmState *)
              newState;
8
9         // Check that if a new pairing procedure with
              an existing bond will not lower the
              established security level of the bond.
10
11        if (!(fsm->dc_param[SMP_KEYS] & (
              BT_KEYS_LTK_P256 | BT_KEYS_LTK))) {
12          return IFW_OPERATION_PASS;
13        }
14
15        if (fsm->dc_param[SMP_ENC_SIZE] > fsm->dc_param
              [SMP_ENC_SIZE_PREV]) {
16          return IFW_OPERATION_REJECT;
17        }
18
19        if ((fsm->dc_param[SMP_KEYS_FLAGS] &
              BT_KEYS_AUTHENTICATED) &&
20           fsm->dc_param[SMP_METHOD_PREV] == JUST_WORKS
                ) {
21          return IFW_OPERATION_REJECT;
22        }
23
24        return IFW_OPERATION_PASS;
25  }
```

(a) An example of BlueSWAT C policy.

```
1   rule1 = (bit.band(DC[SMP_KEYS] or 0, KEYS_LTK_P256 |
          KEYS_LTK) == 0)
2   rule2 = (DC[SMP_ENC_SIZE] <= DC[SMP_ENC_SIZE_PREV])
3   rule3_not_just_work = (bit.band(DC[SMP_KEYS_FLAGS] or
           0, KEYS_AUTHENTICATED) == 0 or
4       DC[SMP_METHOD_PREV] ~= "JUST_WORKS")
5
6   return rule1 and rule2 and rule3_not_just_work
```

(b) Policy in Lua. Lines 1-3 are customized user variables and Lines 9-14 are user-defined rules.

Figure 7: BlueSWAT C policy and Lua policy.

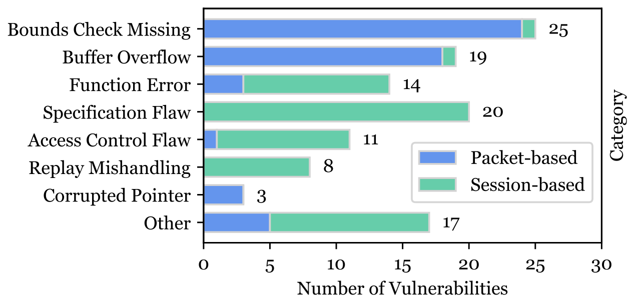
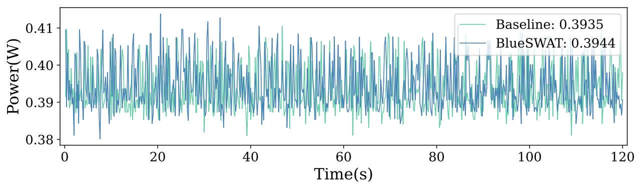
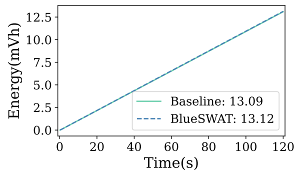


Figure 8: Category of BLE Vulnerabilities. Vulnerabilities under the "Other" category are strongly related to specific coding errors.

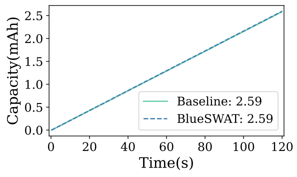
obviously session-based because they manipulate duplicate messages. Stateless packet filter mechanisms like LBM cannot detect these attacks since the individual message is usually benign.



(a) Processing power.



(b) Energy usage.



(c) Capacity usage.

Figure 9: Evaluation on power performance.

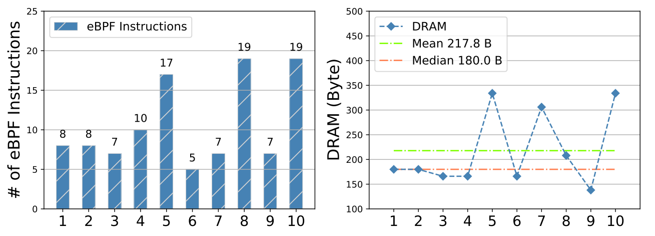


Figure 10: Number of eBPF instructions and dynamic memory (DRAM) consumption (Byte) of 10 different rules when JIT enabled.

F Evaluation

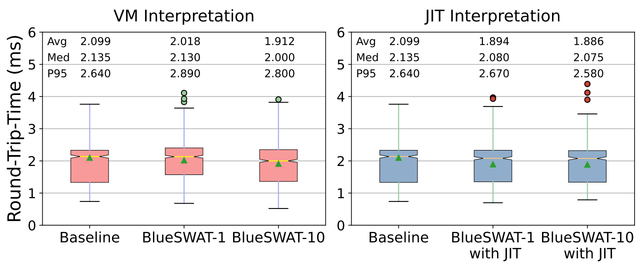


Figure 11: RTT (ms) in end-to-end micro benchmark.