Formalization, Implementation, and Verification of the Bluetooth L2CAP State Machine

Tan Khang Le, Mohammad Omidvar Tehrani, Yuepeng Wang,
Jianliang Wu, Steven Y. Ko
Simon Fraser University, Canada
{khang_le,m_omidvart,yuepeng,wujl,steveyko}@sfu.ca

Abstract

The Logical Link Control and Adaptation Protocol (L2CAP) is a core Bluetooth component, and verifying its correctness is crucial for reliable and secure connectivity. However, verification can be challenging due to the complexity and ambiguities in its natural language (English) specification. In this paper, we present a *formally verified implementation* of the L2CAP state machine. Our approach introduces the Specification State Machine (SSM) to formalize the L2CAP state machine in the specification and the Operational State Machine (OSM) as an abstraction of the implementation. We then formally prove that (i) OSM refines SSM, and (ii) our implementation semantically conforms to OSM. By combining these two proofs, we verify that our implementation complies with our formalization of the specification. Furthermore, we define critical safety and liveness properties and formally prove that our implementation satisfies these guarantees. To ensure practicality, we implement the L2CAP state machine in Dafny and integrate it into Android's Fluoride Bluetooth stack. Our evaluation demonstrates that our formally verified implementation maintains competitive performance while ensuring formal correctness.

CCS Concepts

• **Networks** → **Protocol testing and verification**; **Formal specifications**.

Keywords

Formal Verification, State Machine, L2CAP, Bluetooth, Dafny

ACM Reference Format:
Tan Khang Le, Mohammad Omidvar Tehrani, Yuepeng Wang,, Jianliang Wu, Steven Y. Ko. 2025. Formalization, Implementation, and Verification of the Bluetooth L2CAP State Machine. In *The 31st Annual International Conference on Mobile Computing and Networking (ACM MOBICOM '25), November 4–8, 2025, Hong Kong, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3680207.3765254

ACM MOBICOM '25, November 4–8, 2025, Hong Kong, China

ACM ISBN 979-8-4007-1129-9/25/11
https://doi.org/10.1145/3680207.3765254

1 Introduction

Bluetooth technology is widely used for short-range wireless communication in various applications, including consumer electronics and healthcare devices. The Bluetooth protocol stack comprises multiple layers that work together to establish and manage reliable communication between devices. Among these layers, the Logical Link Control and Adaptation Protocol (L2CAP) plays a crucial role in facilitating data exchange by providing logical channel multiplexing, segmentation and reassembly, and Quality of Service support [4]. To manage these operations, L2CAP employs a dedicated state machine that governs the behaviors of logical channels.

However, recent research has revealed that an incorrect L2CAP state machine implementation, such as channels remaining in the Open state even after security check failures, can lead to critical problems such as exploitable vulnerabilities [34, 36, 40]. These findings highlight the need for rigorous testing and verification to ensure the *behavioral correctness* of an L2CAP state machine implementation—that is, its adherence to the specification. While traditional testing methods, such as fuzzing and unit testing, are effective at identifying specific implementation flaws such as memory safety issues, they often fail to provide exhaustive guarantees about an implementation's adherence to the specification. In contrast, formal verification offers a systematic approach to proving the behavioral correctness of a system with respect to its specification under all possible scenarios [3].

Motivated by the need for behavioral correctness, we present a *formally verified* implementation of the L2CAP state machine. Our verification aims to satisfy three key goals for our L2CAP state machine implementation: (i) verification of compliance with the Bluetooth specification, (ii) proof of strong guarantees (safety and liveness), and (iii) integration with Android's Bluetooth stack, demonstrating practicality. To achieve (i), we begin with the formalization of the Bluetooth specification by constructing a model called the Specification State Machine (SSM). We then derive an implementation-level model called the Operational State Machine (OSM) and use it to formally prove that OSM correctly

refines SSM, meaning that *every possible behavior in OSM is allowed by SSM*. Then, we implement the L2CAP state machine and formally verify its *semantic conformance* to OSM, meaning that *the state machine behavior of our implementation precisely matches that of OSM*. Since OSM is proven to refine SSM, verifying that our implementation semantically conforms to OSM guarantees its compliance with SSM, our formalization of the specification.

To achieve (ii), our work formally proves both *safety* and *liveness* properties of the L2CAP state machine implementation. Specifically, our safety properties ensure that the state machine behaves correctly upon termination, during error conditions, and when encountering unmodeled or invalid events. Additionally, our liveness property guarantees that the Bluetooth connection will eventually be established or terminated, a crucial requirement for real-world applications.

To achieve (iii), we develop a fully functional L2CAP state machine that is not only compliant with the Bluetooth specification but also integrated with Android's Bluetooth stack. We further conduct experiments to evaluate its performance in real-world scenarios, assessing both its correctness and efficiency. Our evaluation shows that while our verified implementation functions correctly, it does not introduce noticeable overhead in user experience.

In summary, our work[1] makes the following contributions:

- **Formalization of the L2CAP State Machine**: We provide a rigorous formalization of both the specification and implementation of the L2CAP state machine.
- **Verification of Safety and Liveness Properties**: We formally define and prove crucial safety and liveness properties, which guarantee that the L2CAP state machine avoids incorrect transitions and eventually reaches the desired states.
- **Verification Methodology**: We demonstrate that a structured, layer-based verification methodology [12, 13, 15, 16] that uses state machine refinement and semantic conformance can effectively prove that our implementation adheres to our formalization of the L2CAP state machine specification.
- **Formally Verified Implementation**: We develop and integrate a formally verified L2CAP state machine into Android's Bluetooth stack, demonstrating its correctness, security, and real-world applicability.

2 Background

In this section, we present the background for the L2CAP specification and verification techniques.

2.1 Bluetooth and L2CAP Specification

The Bluetooth specification [4] defines the protocol stack for Bluetooth Low Energy (BLE) and Bluetooth Classic. At a high level, the Bluetooth protocol stack consists of two components: the Bluetooth host (upper layer) and the Bluetooth controller (lower layer) [4]. The Bluetooth host is implemented in software and includes higher-layer protocols like L2CAP, which is the focus of our work. It also includes Bluetooth profiles that define specific use cases, such as audio device control, file transfer, network tethering, etc. On the other hand, the Bluetooth controller is typically a hardware module, and handles low-level tasks like radio transmission, baseband processing, and link control.

L2CAP plays an important role in the entire Bluetooth protocol stack by shielding the upper-layer protocols from the complex details of the lower-layer protocols. It multiplexes between various logical connections made by the upper layers. The L2CAP state machine governs the behavior of logical channels such as connection establishment and configuration, through defined states, events/conditions, and actions:

- **States**: Stages of a channel's lifecycle, such as Closed, Config, and Open. These states determine the operational status of the connection.
- **Events/Conditions**: External or internal triggers that prompt state transitions. Examples include connection requests and data packet exchanges.
- **Actions**: Operations executed during state transitions, such as sending signaling packets and processing data units, such as the Service Data Unit.

```
method increase(x: int) returns (y: int)
  requires x > 0
  ensures y > x
{
  y := x * 2;
}
```

Figure 1: Floyd–Hoare verification example in Dafny.

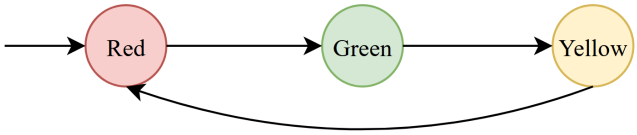
2.2 Dafny and Floyd-Hoare Logic

In our work, we use Dafny [24], a "verification-aware" programming language that allows a developer to write both a program and its formal specification together. Dafny's static verifier automates many of the tasks for verifying whether the program is correct according to the formal specification. This is done with the help of Z3 SMT Solver [10]. Once a program is verified, Dafny can translate it into various target languages, such as C++, Java, and Python.

Dafny uses Flyod-Hoare logic [11, 18] as its proof framework. Using this, a developer annotates their program's functions with pre- and post-conditions about the program's state when entering/leaving the function. Dafny's static verifier then checks that these claims hold for all possible inputs and outputs. Figure 1 shows an example program that asserts a

[1]Source code is available at: https://github.com/sfu-rsl/bv

(a) High-level abstract state machine capturing basic transition logic.



(b) Low-level concrete state machine incorporating implementation-specific detail about traffic light colors.

Figure 2: Example of state machine refinement for a traffic light system, from abstract transition logic to detailed transitions.

condition about its input via a pre-condition and asserts a condition about its output via a post-condition.

However, many proposition classes are not decidable in general, i.e., many types of logical statements cannot always be automatically decided as true or false. Thus, Dafny (or more specifically, the Z3 [10] solver it relies on) uses heuristics. For example, propositions involving universal quantifiers (∀) and existential quantifiers (∃) are not decidable. Thus, it is possible to write correct code in Dafny that the solver nevertheless cannot prove automatically. Because of this, a developer sometimes needs to insert annotations to guide Dafny's verifier. For instance, to prove the liveness properties of concurrent systems, many *trigger* and *assert* clauses are needed to guide the verifier [25].

2.3 State Machine Refinement and Semantic Conformance

State machine refinement [1, 22] is a powerful tool to prove that a lower-level implementation correctly implements a higher-level specification. A high-level, abstract state machine models the desired behavior of a system based on a specification. For example, it can specify the behavior of a key-value map (e.g., it should support key-value pair insertion) without detailing exactly how to implement it. A concrete state machine then models an implementation with detailed, low-level behavior, showing one way to instantiate the abstract state machine. For example, it can model a hash map as a way to implement the key-value map. State machine refinement then makes it possible to prove that the concrete state machine *refines* the abstract state machine, meaning that every execution of the concrete state machine can be mapped to a possible execution of the abstract one. Thus, proving refinement guarantees that the low-level implementation is compliant with the high-level specification.

For example, Figure 2 presents an abstract state machine and a concrete one for a traffic light system. The abstract state machine only has two states, Red and ¬Red, meaning that it only models a red light and non-red lights. The concrete state machine refines this behavior by adding more details for ¬Red with Green and Yellow, demonstrating one possible instance of implementation of the abstract state machine. The refinement between the two state machines can be trivially proved using the following refinement mapping:

$$\text{Red} \sim \text{Red}, \quad \text{Green} \sim \neg\text{Red}, \quad \text{Yellow} \sim \neg\text{Red}$$

This mapping guarantees that every execution of the concrete state machine corresponds to a valid execution in the abstract state machine:

- The initial states match (Red).
- When the concrete machine moves from Red to Green, the abstract machine moves from Red to ¬Red.
- When the concrete machine moves from Green to Yellow, the abstract machines remains in ¬Red.
- When the concrete machine moves from Yellow to Red, the abstract machine transitions back to Red.

This proves that the concrete state machine complies with the abstract state machine.

Semantic conformance [31, 32] follows the same fundamental principles as refinement but extends them by ensuring that the behavior of one state machine precisely matches with another. Here, "behavior" refers to the state transitions and any observable side effect of each transition (e.g., *actions* in the L2CAP state machine as described in Section 2.1). "Match" means that for every valid execution in the concrete implementation, there exists a corresponding execution in the abstract model with identical observable effects.

3 Verification Methodology

Figure 3 provides an overview of our layer-based verification methodology, using the hierarchical verification approach [12, 13, 15, 16]. At the very top, we have the Bluetooth specification from the Bluetooth SIG that defines high-level system behavior in informal natural language. We then have the Specification State Machine (SSM) that formalizes the L2CAP state machine according to the Bluetooth specification. As we explain in Section 3.2, our SSM removes ambiguities and omissions of informal natural language, providing a well-defined baseline for verification. Next, we have the Operational State Machine (OSM) in the middle layer that *refines* the abstract states in SSM into more detailed states that capture implementation details, such as handling run-time errors, security checks, and timeouts. After that, we have the concrete system implementation that *semantically conforms* to OSM, meaning that the state machine behavior of our implementation precisely matches that of OSM. Since OSM is proven to refine SSM, verifying that our implementation

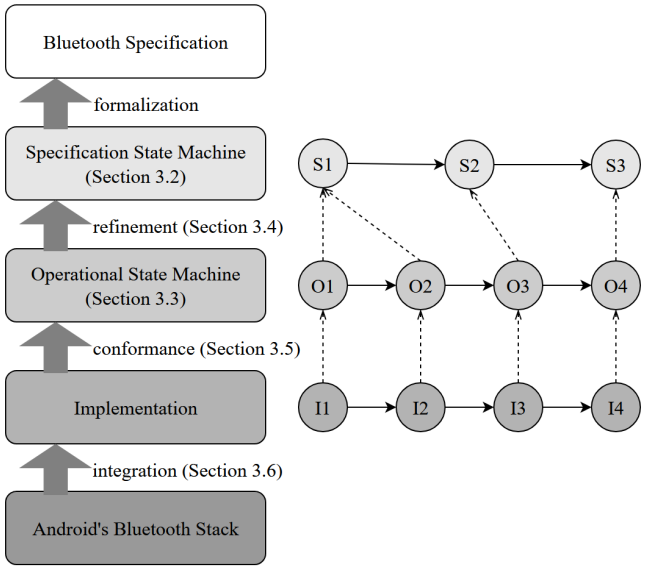


Figure 3: An overview of our layer-based verification methodology with multiple levels of abstraction. On the right-hand side, $S1 - S3$ are states in SSM, $O1 - O4$ are states in OSM, and $I1 - I4$ are states in the implementation. Dashed arrows indicate how lower layers adhere to higher-level abstractions, ensuring consistency across layers, while solid arrows represent state transitions within the same layer.

semantically conforms to OSM guarantees its compliance with the specification. Moreover, our implementation is integrated with Android, demonstrating the real-world applicability of our approach. As mentioned in Section 2.2, we use Dafny for the entire process—formalization, verification, and implementation.

3.1 Assumptions

The guarantees of our formally verified L2CAP implementation rely on a set of assumptions. First, we assume that the specification [4] is "correct," i.e., it is not our goal to analyze the specification for potential design flaws, security vulnerabilities, or other issues. Previous work has extensively focused on modeling the specification to uncover such problems [14, 37–39]. In contrast, our focus is on formalizing the specification and proving that our implementation is compliant to it.

Second, we assume the correctness of the entire toolchain, which consists of Dafny's program verifier, Dafny's built-in functionality for translating verified code into C++, and the C++ compiler and runtime. Prior research has demonstrated that each component in the toolchain can also be verified [8, 20, 26, 35], though we do not conduct such verification ourselves.

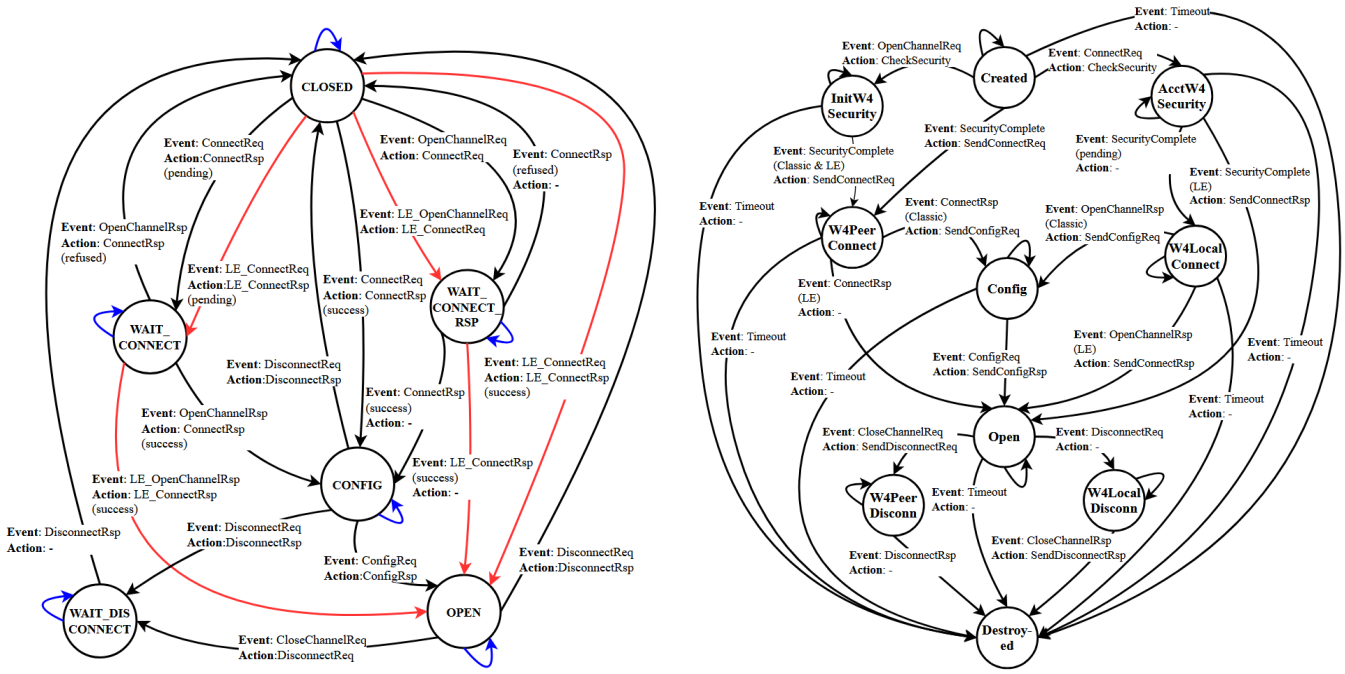
```
datatype SsmState = Closed | WaitConnectRsp
datatype SsmEvent = OpenChannelReq
datatype SsmCondition = None
datatype SsmAction = SendConnectReq | Ignore

type SsmEventCondition = (SsmEvent, SsmCondition)

function ExecuteSsmModel(
  state: SsmState, event_condition: SsmEventCondition
): (SsmAction, SsmState) {
  if (state == Closed)
     && (event_condition == (OpenChannelReq, None)) then
    (SendConnectReq, WaitConnectRsp)
  else
    (Ignore, state)
}
```

Figure 4: A simplified modeling example of the Specification State Machine (SSM) in Dafny

3.2 Formalization and Augmentation of the L2CAP Specification

Formally verifying an implementation of the L2CAP state machine first requires formalizing its specification. We achieve this through a model called the Specification State Machine (SSM), a rigorous, machine-checkable model formalized in Dafny. As we describe below, formalizing a specification is not simply a matter of translating natural language into formal language. We start with the state transition tables and diagrams in the specification [7], but also combine it with our analysis of all L2CAP signaling packet formats [6] and the general procedures of L2CAP [5]. In the end, SSM captures event-driven state transitions and the actions described in various places in the specification. Figure 4 shows a simplified modeling example in Dafny for a single state transition from the `Closed` state to the `WaitConnectRsp` state. The transition is triggered by an event `OpenChannelReq`, and the action `SendConnectReq` is produced as the output. Figure 5a shows our final state machine diagram for SSM, which we explain further later in this section.

Our formalization addresses two key challenges in using the Bluetooth specification for verification. First, the specification describes the L2CAP state machine in natural language (English), which is imprecise at times. This imprecision leads to ambiguities that are subject to interpretation. For example, on page 1092 of the specification, it states that "*The CONFIG state is left for the OPEN state if both the initiator and acceptor paths complete successfully.*" This statement is ambiguous because it does not explicitly define what it means for "both the initiator and acceptor paths [to] complete successfully." Thus, it is up to the developer to interpret it and define what the correct behavior is. In fact, successful



(a) High-level SSM capturing the fundamental transition logic based on the Bluetooth specification (with augmentations highlighted in red and blue).

(b) Low-level OSM incorporating implementation-specific details of L2CAP behavior in Android.

Figure 5: L2CAP state machines at different levels of abstraction: (a) the high-level SSM derived from the Bluetooth specification and (b) the low-level OSM representing the concrete implementation in Android's Bluetooth stack.

path completion could mean either that configuration is complete so the state machine could leave the Config state for the Open state, or that a response is received for a connection request so the state machine could enter the Open state. The specification does not describe this behavior anywhere else, making it difficult to understand what to do exactly.

To resolve such ambiguities, we adopt *one* possible interpretation of each unclear statement and apply it to SSM. While we do not claim that it is *the only* correct interpretation, our approach eliminates imprecision and potential misinterpretation. By explicitly formalizing every aspect of the L2CAP state machine precisely, we provide a clear and verifiable foundation that other developers can examine and agree upon. We note that this approach poses a risk of not accepting other possibly valid interpretations. This is inevitable when there are multiple competing interpretations. However, we posit that formalizing non-competing interpretations together may be possible, though we did not pursue it in our research.

The second challenge that our formalization addresses is that, the L2CAP state machine specification does not cover all possible scenarios and is, therefore, incomplete. The specification itself acknowledges this limitation as well and notes that "*The state machine does not necessarily represent all possible scenarios.*" on page 1089. Some scenarios are covered elsewhere in the document, while other scenarios are completely omitted. The most significant omission is that the specification does not specify how Bluetooth Low-Energy (BLE) signaling packets should be handled within the state machine. Due to this incompleteness, developers have to come up with new states in their Bluetooth implementation to handle such scenarios [9]. This particular omission illustrates another difficulty for formalization—omissions make it impossible to formally prove the compliance of an implementation with the specification. The reason is that an implementation *does* need to handle omitted events, such as BLE signaling packets for correct operation, but there is no matching behavior described in the L2CAP state machine specification. Hence, compliance is impossible to prove.

To address incompleteness, we take the L2CAP state machine diagrams and descriptions from the specification and selectively augment it with additional details. Figure 5a highlights how we augment the original state machine. This includes, (i) events that should be present but are omitted in the specification, such as BLE signaling packets (labeled in red), and (ii) events that should trigger a self-transition to the

same state but are not explicitly listed as such in the specification (labeled in blue). All states have such self-transition that the specification does not discuss, e.g., events that the state machine can safely ignore. As a result, our SSM precisely defines all aspects of the L2CAP state machine behavior.

3.3 Implementation Modeling

Though SSM formalizes the specification, we cannot directly use it to prove that an implementation complies with it. This is because SSM, as any formal specification does, operates at a high level of abstraction that does not provide guidance on how an implementation should handle run-time behaviors, such as errors, timeouts, and other operational aspects.

Therefore, we formalize an implementation-level model called the Operational State Machine (OSM) in Dafny as an intermediate abstraction between our implementation and SSM. We use this OSM to prove that it refines SSM, i.e., every behavior in OSM is allowed by SSM (Section 3.4). We also use this OSM to prove that our implementation conforms to it semantically, i.e., every behavior in our implementation precisely matches that of OSM (Section 3.5). Combining these two, we prove that our implementation complies with SSM.

While we have the freedom to design an implementation-level model any way we want, we choose to design a model based on an existing implementation—specifically, Android's L2CAP state machine implementation [2]. This choice originates from our desire to make our implementation practical and deployable on real devices. Android is the most popular mobile OS, and making our models and implementation compatible with Android's implementation paves a way for us to ensure real-world applicability.

Figure 5b presents the state transition diagram for OSM. Compared to SSM, OSM refines the `Closed` state by decomposing it into two distinct states: `Created` and `Destroyed`. This distinction allows us to reason about safety and liveness properties of the state machine, which we formally address in Section 4. In addition, OSM introduces two new states to handle implementation-level security checks: `InitW4Security` and `AcctW4Security`, upon receiving connection requests. These security checks perform authentication, authorization, and encryption for logical channels, before proceeding to other connection states. Lastly, OSM defines additional implementation-specific events like `Timeout` and `InternalFailure` to handle run-time errors.

Figure 6 illustrates a simplified modeling example for a subset of possible OSM transitions involving a security check. The state machine makes a transition from the `Created` state to the `InitW4Security` state upon receiving the `OpenChannelReq` event, and subsequently to the `W4PeerConnect` state following the `SecurityComplete` event.

```
datatype OsmState = Created | InitW4Security | W4PeerConnect
      | Destroyed
datatype OsmEvent = OpenChannelReq | SecurityComplete
datatype OsmAction = CheckSecurity | SendConnectReq | None

function ExecuteOsmModel(state: OsmState,
    event: OsmEvent): (OsmAction, OsmState) {
  if (state == Created) && (event == OpenChannelReq) then
    (CheckSecurity, InitW4Security)
  else if (state == InitW4Security)
       && (event == SecurityComplete) then
    (SendConnectReq, W4PeerConnect)
  else
    (None, state)
}
```

Figure 6: A simplified modeling example of the Operational State Machine (OSM) in Dafny

3.4 Refining Operation to Specification

A key step in our verification methodology is to prove that OSM, which models our implementation, *refines* SSM, which models the specification. We prove this by demonstrating that every behavior in OSM, as shown in Figure 5b, corresponds to an allowable behavior in SSM, as shown in Figure 5a. We use the standard refinement mapping approach [1] to prove the refinement theorem (Theorem 1).

Informally speaking, we map each state and event in OSM to a corresponding state and event, respectively, in SSM. Given this mapping, our refinement theorem states that every state transition in OSM corresponds to a state transition in SSM. Formally, we first define SSM and OSM as below. They consist of (i) *states* for each logical channel, e.g., channel open and channel closed, (ii) *events* that represent the input to the state machine, e.g., events triggered by packet received from the peer device and command received from upper layers, (iii) *actions* that represent the output from the state machine, e.g., packet sent to the peer device and command sent to the lower layers, and (iv) a *function* that maps each pair of (state, event) to an action, i.e., the action that should be produced when an event is received in a state.

Definition 1 (**SSM**). *The Specification State Machine (SSM) is a tuple* $\tilde{\mathcal{L}} = (\tilde{S}, \tilde{E}, \tilde{A}, \rightsquigarrow, \tilde{F}, \tilde{s_0})$ *where*

- $\tilde{S}$ *is a set of states;*
- $\tilde{E}$ *is a set of events;*
- $\tilde{A}$ *is a set of actions;*
- $\rightsquigarrow \subseteq \tilde{S} \times \tilde{E} \times \tilde{S}$ *is the state transition relation;*
- $\tilde{F} : \tilde{S} \times \tilde{E} \rightarrow \tilde{A}$ *is the action function;*
- $\tilde{s_0} \in \tilde{S}$ *is the initial state.*

Definition 2 (**OSM**). *The Operation State Machine (OSM) is a tuple* $\mathcal{L} = (S, E, A, \rightarrow, F, s_0)$ *where:*

- S *is a set of states;*

- E *is a set of events;*
- A *is a set of actions;*
- $\rightarrow \subseteq S \times E \times S$ *is the state transition relation;*
- $F : S \times E \rightarrow A$ *is the action function;*
- $s_0 \in S$ *is the initial state.*

By convention, we use the notation $s_i \xrightarrow{e} s_{i+1}$ to represent a transition from state s_i to state s_{i+1} given event e. Then, we formalize the refinement relation between OSM and SSM using the following theorem:

Theorem 1 (**Refinement**). *Let $\tilde{\mathcal{L}} = (\tilde{S}, \tilde{E}, \tilde{A}, \rightsquigarrow, \tilde{F}, \tilde{s}_0)$ be the SSM state machine and $\mathcal{L} = (S, E, A, \rightarrow, F, s_0)$ be the OSM state machine. It holds that $\mathcal{L}$ refines $\tilde{\mathcal{L}}$, i.e., for every transition in $\mathcal{L}$, there is a corresponding transition in $\tilde{\mathcal{L}}$. Formally,*

$$\begin{aligned}&\forall s_i, s_{i+1} \in S.\ \forall e \in E.\ (s_i \xrightarrow{e} s_{i+1}) \Rightarrow \\ &\quad \exists \tilde{s}_i, \tilde{s}_{i+1} \in \tilde{S}.\ \exists \tilde{e} \in \tilde{E}.\ (\tilde{s}_i \overset{\tilde{e}}{\rightsquigarrow} \tilde{s}_{i+1} \wedge \texttt{AbsState}(s_i) = \tilde{s}_i \\ &\qquad \wedge \texttt{AbsState}(s_{i+1}) = \tilde{s}_{i+1} \wedge \texttt{AbsEvent}(e) = \tilde{e})\end{aligned}$$

where:

- `AbsState` *is the state abstraction function that maps an OSM state to its corresponding state in SSM*
- `AbsEvent` *is the event abstraction function that maps an OSM event to its corresponding event in SSM*

Our refinement proof uses Floyd-Hoare triples of the form $\{\phi\}\ S\ \{\psi\}$, indicating that any execution of statement S starting in a state satisfying ϕ results in a state ψ (if the execution of S terminates) [11, 18]. Using this, we prove Theorem 1 by induction and use Dafny for automated theorem proving of the following cases:

- **Initiation**: $\texttt{AbsState}(s_0) = \tilde{s}_0$
- **Preservation**: For any states $s_i, s_{i+1} \in S,\ \tilde{s}_i, \tilde{s}_{i+1} \in \tilde{S}$, events $e \in E,\ \tilde{e} \in \tilde{E}$, the following Floyed-Hoare triple is valid:

$$\begin{aligned}&\{\texttt{AbsState}(s_i) = \tilde{s}_i \wedge \texttt{AbsEvent}(e) = \tilde{e}\} \\ &\qquad s_i \xrightarrow{e} s_{i+1};\ \tilde{s}_i \overset{\tilde{e}}{\rightsquigarrow} \tilde{s}_{i+1} \\ &\{\texttt{AbsState}(s_{i+1}) = \tilde{s}_{i+1}\}\end{aligned}$$

3.5 Connecting Implementation to Operation

Our implementation is a fully functional L2CAP state machine written in Dafny that is integrated with Android. This means that it compiles and operates with the rest of Android. We use Dafny's built-in support for Dafny-to-C++ translation, which allows us to statically verify our implementation in Dafny before translating it to C++. We discuss our Android integration further in Section 3.6.

With our implementation, we prove that it semantically conforms to OSM. Informally speaking, we show that given the same input (events), our implementation makes the exact same state transitions and takes the exact same actions as OSM. We prove this in two steps. First, we define an internal state machine for our implementation that is identical to OSM, ensuring that our implementation's state transitions never deviate from those of OSM. Second, whenever our implementation performs an action, we record it and verify that it precisely matches the correct action described in OSM. To achieve this, we define pre- and post-conditions for each function in our implementation (using Floyd-Hoare triples, as described below), in a way that allows Dafny to verify that our implementation's recorded actions precisely match those of OSM. This proof is entirely static and does not pose any run-time overhead as we use so-called *ghost variables* in Dafny, which are only used for verification and compiled away for the final binary.

Formally, we first define the implementation state machine and then state the semantic conformance theorem.

Definition 3 (**Implementation State Machine**). *The implementation state machine is a tuple $\mathcal{L}' = (S', E', A', \rightarrow', F', s'_0)$ where:*

- S' *is a set of states;*
- E' *is a set of events;*
- A' *is a set of actions;*
- $\rightarrow' \subseteq S' \times E' \times S'$ *is the state transition relation;*
- $F' : S' \times E' \rightarrow A'$ *is the action function;*
- $s'_0 \in S'$ *is the initial state.*

Theorem 2 (**Semantic Conformance**). *Let $\mathcal{L} = (S, E, A, \rightarrow, F, s_0)$ be the OSM, $\mathcal{L}' = (S', E', A', \rightarrow', F', s'_0)$ be the implementation state machine. It holds that $\mathcal{L}'$ semantically conforms to $\mathcal{L}$, i.e., for every transition in $\mathcal{L}'$, there is a corresponding transition in $\mathcal{L}$, and they output the same action. Formally,*

$$\begin{aligned}&\forall s'_i, s'_{i+1} \in S'.\ \forall e' \in E'.\ (s'_i \overset{e'}{\rightarrow'} s'_{i+1}) \Rightarrow \\ &\quad \exists s_i, s_{i+1} \in S.\ \exists e \in E.\ (\texttt{OpsState}(s'_i) = s_i \\ &\qquad \wedge \texttt{OpsState}(s'_{i+1}) = s_{i+1} \wedge \texttt{OpsEvent}(e') = e \\ &\qquad \wedge s_i \xrightarrow{e} s_{i+1} \wedge \texttt{OpsAction}(F'(s'_i, e')) = F(s_i, e))\end{aligned}$$

where:

- `OpsState` *is the bijective function that maps a state in the implementation to its corresponding OSM state*
- `OpsEvent` *is the bijective function that maps an event in the implementation to its corresponding OSM event*
- `OpsAction` *is the bijective function that maps an action in the implementation to its corresponding OSM action*

To prove Theorem 2, we perform induction and prove the following cases:

- **Initiation**: $\texttt{OpsState}(s'_0) = s_0$
- **Preservation**: For any states $s_i, s_{i+1} \in S,\ s'_i, s'_{i+1} \in S'$, events $e \in E,\ e' \in E'$, the following Floyd-Hoare triple

is valid:

$$\{\texttt{OpsState}(s_i') = s_i \wedge \texttt{OpsEvent}(e') = e\}$$
$$s_i \xrightarrow{e} s_{i+1};\ s_i' \xrightarrow{e'}{}' s_{i+1}'$$
$$\{\texttt{OpsState}(s_{i+1}') = s_{i+1} \wedge \texttt{OpsAction}(F'(s_i', e')) = F(s_i, e)\}$$

3.6 Integrating Implementation into Android's Bluetooth Stack

To integrate our implementation with Android, we introduce a shim that essentially works as a foreign function interface (FFI) between our Dafny code and Android. Our shim defines a clear interface that consists of (i) entry points to our Dafny code from Android, (ii) Android functions that our Dafny code uses (which we call external functions), and (iii) data structures shared between our Dafny code and Android.

More importantly, we define pre- and post-conditions for all external Android functions by carefully examining their behavior. This is necessary for verification—since our semantic conformance proof uses Floyd-Hoare triples (Section 3.5), we need to have pre- and post-conditions for all functions that we implement or use in our Dafny code, which include external Android functions. However, this does not mean that we verify external Android functions. Rather, their pre- and post-conditions serve as the *contracts* that we expect from Android.

In addition, we check the pre- and post-conditions of external functions at run time through assertions. It turns out that this effort was worthwhile as we discovered that there is an external function that sometimes modifies the state machine's internal state in an unexpected fashion. According to this finding, we adjusted our implementation.

4 Verification of State Machine Properties

By proving refinement and semantic conformance as described in Section 3.4 and Section 3.5, respectively, we show that the implementation is compliant with the specification, i.e., our implementation does not deviate from the allowable behaviors defined in the specification. However, it is equally important to verify the desirable properties that the implementation should guarantee. As such, we define safety and liveness properties for OSM and formalize them in Dafny for automated theorem proving. Given that the implementation semantically conforms to OSM, these properties are inherently upheld by our implementation.

4.1 Safety

Informally speaking, guaranteeing safety properties means guaranteeing that nothing bad happens. In our work, we define and prove three key safety properties for OSM. The first property states that once the connection is terminated, the state machine remains in the `Destroyed` state and does not transition to any other state. This should be the expected behavior of a terminated connection and we prove that the property holds in our state machine. The second property states that, in the event of errors, such as a timeout, a security check failure, etc., the state machine transitions to the `Destroyed` state. This ensures that failures lead to a proper termination. The third property states that, when the state machine encounters unexpected or unmodeled events, it ignores them without triggering any state transition or additional processing. This should also be an expected behavior, and we prove that the property holds in our state machine. The formal descriptions of these safety properties are as follows:

Theorem 3 (**Safety**). *Let $s_i \xrightarrow{e} s_{i+1}$ be a state transition in OSM, the following properties must hold upon every transition in the state machine:*

(1) *Destroyed:*

$$s_i = \texttt{Destroyed} \Rightarrow s_{i+1} = \texttt{Destroyed}$$

(2) *Error:*

$$\texttt{IsErrorEvent}(s_i,\ e) \Rightarrow s_{i+1} = \texttt{Destroyed}$$

(3) *Unmodeled:*

$$\neg\texttt{IsModeledEvent}(s_i,\ e) \Rightarrow s_{i+1} = s_i$$

where `IsErrorEvent` *and* `IsModeledEvent` *are functions used to determine whether an event is classified as an error or is modeled, respectively, based on the specification.*

To prove Theorem 3, we use Dafny for the automatic verification of the following induction steps:

- **Initiation:** For the initial state, when the channel is created ($s_0 = \texttt{Created}$), the safety properties are satisfied:
  (1) $s_0 = \texttt{Destroyed} \Rightarrow s_1 = \texttt{Destroyed}$
  (2) $\texttt{IsErrorEvent}(s_0,\ e) \Rightarrow s_1 = \texttt{Destroyed}$
  (3) $\neg\texttt{IsModeledEvent}(s_0,\ e) \Rightarrow s_1 = s_0$
- **Preservation:** Assume that the safety properties hold for some state s_i. The safety properties still hold for state s_{i+1}:
  (1) $s_{i+1} = \texttt{Destroyed} \Rightarrow s_{i+2} = \texttt{Destroyed}$
  (2) $\texttt{IsErrorEvent}(s_{i+1},\ e) \Rightarrow s_{i+2} = \texttt{Destroyed}$
  (3) $\neg\texttt{IsModeledEvent}(s_{i+1},\ e) \Rightarrow s_{i+2} = s_{i+1}$

4.2 Liveness

A liveness property guarantees that an operation will complete within a finite time [21]. For our implementation, we define and prove a key liveness property, which is that the state machine eventually reaches either the `Open` state or the `Destroyed` state. This liveness property ensures that the Bluetooth connection will eventually be either established or terminated, a desired behavior in real-world applications.

```
predicate Next(s: OsmState, s': OsmState) {
  exists e: OsmEvent :: ExecuteOsmModel(s, e).state == s'
}

predicate Fairness(trace: Trace) {
  forall i :: trace(i) != Destroyed
  ==> exists j :: j > i && trace(j) != trace(i) &&
    trace(i) == trace(j - 1) && Next(trace(i), trace(j))
}

predicate IsFairTrace(trace: Trace) {
  && trace(0) == Created
  && (forall i: nat :: Next(trace(i), trace(i + 1)))
  && Fairness(trace)
}
```

Figure 7: Fairness assumption for the L2CAP state machine in Dafny.

Liveness properties are considered more difficult to prove because (i) their formalization requires a notion of time, and (ii) a state machine may explicitly allow a loop, which prevents the state machine from making progress. There are separate techniques that exist to address these, which we use and detail below.

4.2.1 Notion of Time Our formalization captures system behavior as a sequence of states, inspired by the Temporal Logic of Actions (TLA) [22]. Specifically, we define a mapping called `trace` that maps a natural number to a state, where the natural number is the sequence index (representing the timestamp). For example, `trace(0)` is the initial state, and `trace(i)` is the *i*th state in the sequence. Using this formalization, we prove that for every possible "fair" sequence of state transitions, there exists an n such that `trace(n)` (the nth state) is either `Open` or `Destroyed`. The definition of "fair" is important and we describe it next.

4.2.2 Progress Liveness properties typically depend on *fairness assumptions*, that is, the underlying environment will enable progress [23]. In our case, a fairness assumption is necessary since the Bluetooth protocol allows that, under certain circumstances, the state machine can stay in the same state indefinitely. One such example is when the state machine continuously receives pending responses from the peer device when it is in the `W4PeerConnect` state. Thus, we assume fairness by assuming that *the state machine will eventually transition to a new state other than the current state, unless it is in the* `Destroyed` *state.* Figure 7 illustrates how we encode this fairness condition for OSM in Dafny.

4.2.3 Liveness Theorem Now, we are ready to formalize the liveness theorem, which states that the state machine eventually reaches the `Open` or `Destroyed` state for every possible "fair" sequence of state transitions.

Theorem 4 (**Liveness**). *For every possible sequence of state transitions (*`trace`*) in the L2CAP state machine, under the fairness assumption (represented by the predicate* `IsFairTrace`*), the following properties must hold:*

$$\texttt{IsFairTrace(trace)} \Rightarrow \exists n.(\texttt{trace}(n) = \texttt{Open} \vee \texttt{trace}(n) = \texttt{Destroyed})$$

The following examples illustrate two paths from OSM that demonstrate liveness:

- `trace = [Created, InitW4Security, W4PeerConnect, Config, Open]` demonstrates a path where the state machine successfully reaches the `Open` state.
- `trace = [Created, InitW4Security, Destroyed]` shows a path where the state machine transitions to the `Destroyed` state.

To prove this theorem, we again formalize it in Dafny and guide its verifier to verify that our implementation guarantees continuous progression.

5 Evaluation

Our evaluation mainly focuses on the implementation and integration with Android. Since the L2CAP state machine is only involved in connection management and configuration, it is not a significant bottleneck in the overall operation of Bluetooth. Nevertheless, we quantify how our implementation performs compared to the original Android in this section.

5.1 Evaluation Setup

The Bluetooth SIG defines Bluetooth *profiles* that support various application scenarios, such as audio device control, streaming, network tethering, etc. Since real-world Bluetooth applications rely on these profiles, they are ideal for our evaluation benchmark. Among all the profiles, we choose six that (i) are supported by Android, (ii) involve the state machine, and (iii) represent common real-world scenarios. Then, we define scenarios that correspond to them to evaluate our implementation.

We use a Google Pixel 6 as our test device running Android 12 (AOSP compiled by us). The peer companion devices are: an Android phone (Google Pixel 7) running regular, non-AOSP Android 15, and a pair of wireless headphones (Soundcore Life P3) for audio-based scenarios. We repeat each scenario six times. We start all scenarios by manually connecting the test device to the peer device. Then, based on the scenario, we perform either some manual actions on the devices or run an automation script to perform the desired actions. Table 1 provides descriptions for the scenarios. Our manual actions are as follows. In scenario 1, we request sending the file from the peer device and accepts it on the test device. In scenario 2, we perform clicks on the peer device by tapping on the test device's screen. In scenario 4, we play

#	Profile	Scenario
1	OBEX	The user sends a file from the peer device, then accepts it on the test device and waits until the file is completely received.
2	HID	The test device connects to the peer device as a mouse, then the user performs a left-click for 10 times.
3	PAN	The test device connects to the peer device's tethering network over Bluetooth. Then it sends an HTTP request (fetching `www.example.com`) and receives its response for 5 times.
4	A2DP	The test device connects to the peer device. Then the user plays a short song for 5 times.
5	AVRCP	The test device connects to the peer device, then increases the volume level for 12 times.
6	HFP	The test device connects to the peer device, then it increases the volume level for 5 times during a Zoom call.

Table 1: Experiment scenarios and their target profiles.

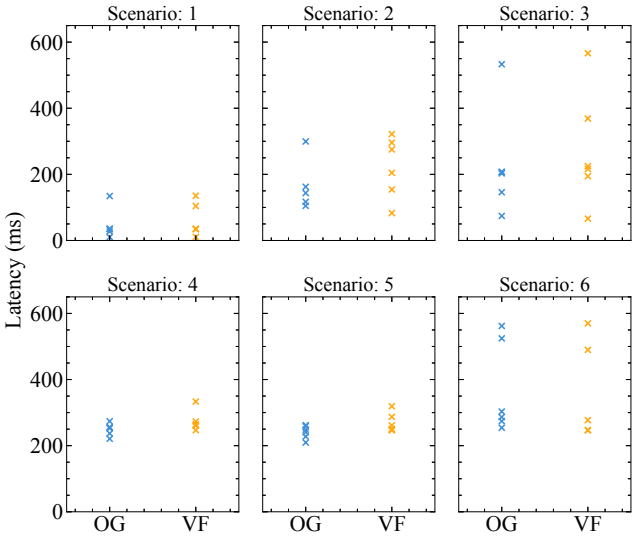


Figure 8: Distribution of time to reach the Open state after creation for the channels established in scenarios (OG: original Android, VF: our verified implementation).

a short audio, wait until the streaming is suspended and then repeat. The actions in the rest of the scenarios are performed programmatically using the Android Debug Bridge.

5.2 Cumulative Channel Setup Latency

We first evaluate how long it takes to establish L2CAP channels, i.e., to reach the Open state, after creation. Scenarios 2, 4, 5, and 6 establish multiple channels, and we sum up all the latency together to show cumulative channel setup latency. Figure 8 shows the distribution of the results, in which OG refers to the original Android and VF refers to our verified implementation. As shown, the latency varies widely, and there are no meaningful differences between our implementation and Android's. The wide variation in latency, especially at the millisecond scale of our experiments, is expected as we purposefully conduct our experiments in real-world conditions. In other words, we do not control the environment and run the scenarios as we would in real life. Since connection establishment involves multiple rounds of message exchange between devices, environmental factors such as signal interference can affect latency. Table 4 in Appendix A presents a breakdown of the experimental results, including averages and standard deviations.

5.3 Overhead

We measure the cumulative CPU time spent in our implementation compared to Android's using *perf*, a popular profiling tool. We measure this while running each scenario, which means that the results show how much CPU time is spent

Scenario	OG (ms)		VF (ms)	
1	3.02	±0.58	10.36	±0.55
2	0.10	±0.14	0.70	±0.33
3	0.38	±0.28	1.35	±0.42
4	0.87	±0.32	3.57	±0.65
5	0.87	±0.27	3.52	±0.13
6	0.84	±0.29	3.74	±0.77

Table 2: Cumulative CPU time spent in the execution of the L2CAP state machine, averaged over the runs.

overall in the L2CAP state machine in each run of each scenario. We use these results to analyze the general trend, as individual numbers depend on the exact behavior of a particular run, which is inherently non-deterministic.

As shown in Table 2, each run generally spends more CPU time in the L2CAP state machine in our implementation than in the original version. This is unsurprising as our implementation mainly aims for verification and is not optimized for performance. Further, the translated Dafny code does not benefit from compiler optimizations (e.g., inlining) as the native implementation does. Previous studies report similar or higher overhead for verified implementations [16, 17]. Ultimately, the L2CAP state machine is not a performance-critical module of L2CAP as mentioned earlier, and this increase does not noticeably affect overall performance as discussed in Section 5.2 and Section 5.4.

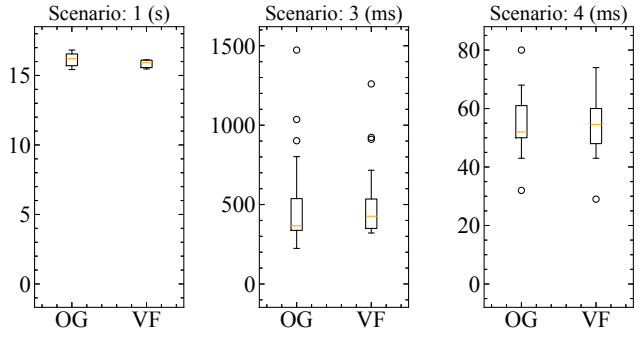


Figure 9: Distribution of the target round-trip time in scenarios.

Category	Spec	Impl	Proof
	(source lines of code)		
SSM	336	-	-
OSM	334	-	-
L2CAP state machine	-	2,107	174
Refinement	-	-	1,874
Conformance	-	-	406
Safety	-	-	217
Liveness	-	-	79
Total	670	2,107	2,750

Table 3: Dafny code size breakdown across specification, implementation, and proof.

5.4 End-to-End Performance

To evaluate the overall performance, we choose scenarios 1, 3, and 4 that allow us to concretely measure the performance of end-to-end, user-facing functions. For scenario 1, we measure the file transfer completion time. For scenario 3, we measure the round-trip time of the HTTP requests. For scenario 4, we measure the round-trip time between the stream suspension command initiation and response.

Similar to our latency results, our results in Figure 9 show a wide variation with no meaningful differences. As mentioned earlier, this is expected as we conduct our experiments in real-world conditions and do not control the environment. For example, Scenario 4 involves fetching a webpage from `www.example.com`, so the time is affected by factors such as server response time, network conditions, etc. Despite this, the results show that our verified implementation does not cause any noticeable difference in user experience. Table 5 in Appendix A provides average values and standard deviations of the results.

5.5 Security Analysis

Our verified implementation guarantees the compliance (free of bugs related to the state transition of the L2CAP state machine) of the implemented state machine. Additionally, verification of the safety properties ensure that our implementation can correctly handle any errors or unexpected events. Combining the compliance and safety properties, our implementation is fundamentally immune to discovered and potential attacks that exploit errors in the implementation of the L2CAP state machine.

For example, one of the reasons enabling several previous impactful attacks [34, 36, 40] is an incorrect implementation of the state machine. These attacks involve the process of reconnection—after two devices are paired, when they reconnect, they use a shared secret key generated during pairing for authentication and encryption. However, an implementation error introduces a vulnerability where the state machine stays in the `Open` state instead of transiting to the `Destroyed` state even if the authentication and encryption fail in a reconnection. Exploiting this vulnerability, an attacker can send spoofed data to the victim device, as it remains in the `Open` state and continues to accept incoming packets.

Our verified implementation mitigates these attacks as Theorem 3 guarantees that the state machine transitions to the `Destroyed` state if any error occurs. Thus, it prevents an attacker from sending spoofed data to the victim device.

5.6 Verification Effort

Table 3 presents our verification effort in terms of source lines of code (SLOC). Specifically, we classify all non-spec, non-executable code as proof annotations, which include constructs for pre- and post-conditions such as `requires`, `modifies` and `ensures` clauses, as well as assertions. Notably, the high-level trusted specification for SSM comprises only 336 SLOC, making it feasible for manual inspection for correctness. Overall, our ratio of proof annotation to implementation is relatively low (approximately 1.3 to 1), which we attribute to our verification methodology and techniques, as detailed in Section 3 and Section 4.

However, the raw SLOC count does not capture our formalization effort for SSM and OSM. Despite SSM comprising only 336 SLOC, its formalization was not trivial and took a few months with multiple rounds of revision. This process involved carefully reading and analyzing the informal, English-based specification, cross-referencing external sources, and resolving ambiguities and gaps in the documentation. Similarly, the formalization of OSM required a deep examination of Android's existing L2CAP implementation, involving extensive code analysis of state transitions to capture its behavior accurately. Such effort goes beyond what SLOC can measure but is crucial as it forms the foundation to ensure the correctness of our verification.

6 Related Work

Testing and Verification for Mobile and IoT Devices. In recent years, various studies have focused on verifying and testing wireless protocols and mobile applications, including Bluetooth. InternalBlue, developed by Mantz et al. [28], is a run-time framework designed to uncover both design flaws in the Bluetooth specification and implementation vulnerabilities. It enables the detection of security risks such as privilege escalation, remote code execution, cryptographic weaknesses, and spoofing attacks. Unlike InternalBlue's runtime testing and reverse engineering, our work uses formal verification to mathematically prove the correctness of an implementation against the Bluetooth specification. Additionally, Pek and Bogunovic [30] used NuSMV to model check parts of the L2CAP protocol, focusing on configuration and CTL specifications, but their work was limited to abstract models without a verified implementation. Our work goes further by proving refinement, safety, and liveness in Dafny and providing a verified implementation integrated into Android's Bluetooth stack, offering stronger and more practical guarantees.

In another study, to address the incompleteness of standardized LTE tests, Raza and Lu [33] introduced a systematic testing approach to validate LTE protocol operations based on a formalized conformance testing specification. More recently, Hou et al. [19] used formal methods on cellular networks to systematically explore the availability and security pitfalls in cellular emergency call systems. Their work also introduced a systematic approach for developing generalized formal models that can be applied to protocol-driven systems.

Given the difficulty in testing and verifying mobile and IoT devices, advanced techniques such as fuzzing have become an attractive option for discovering vulnerabilities. Liang et al. [27] proposed Caiipa, a testing framework that utilizes contextual fuzzing to test mobile apps over an expanded mobile context space in a scalable way. Similarly, L2Fuzz [29], a stateful fuzzer for Bluetooth host stack, used two key techniques: *state guiding* and *core field mutating*, to successfully discover five zero-day vulnerabilities in the L2CAP layer from eight real-world Bluetooth devices.

While these studies primarily focus on uncovering security flaws or validating implementations through systematic testing like fuzzing, our approach differs by emphasizing formal verification. Instead of relying on empirical testing, which may miss edge cases, formal verification uses mathematical proofs to establish strong correctness guarantees for an implementation against a specification.

Verified State Machines in Low-Level Systems State machines are fundamental to many types of systems. In the domain of operating systems, CertiKOS [12] verifies the correctness of a concurrent microkernel implementation using a refinement-based approach to state machines, expressed in a side-effect-free subset of C. To facilitate reasoning about concurrent state machines independently, Gu et al. [13] introduced the concept of contextual refinement.

In storage systems, VeriBetrKV [15] uses state machine refinement techniques to verify storage components, proving that low-level implementations align with high-level specifications. Their approach guarantees correctness in file systems and key-value stores while addressing the challenges of concurrency and crash recovery.

Similarly, IronFleet [16] demonstrates the feasibility of verifying complex distributed systems by combining refinement-based verification with automated theorem proving. In contrast to CertiKOS and VeriBetrKV, which do not prove any liveness properties for their implementation, IronFleet offers a comprehensive verification framework that ensures both liveness guarantees and state machine refinement.

These studies defined their own specifications of the protocols before building and verifying an implementation against them. In contrast, our approach focuses on formalizing an existing specification, written in English, rather than creating a new one. This distinction allows us to introduce a rigorous methodology for verifying the correctness of an implementation directly against an established specification.

7 Conclusion

Our work presents a formal verification methodology for the L2CAP state machine, ensuring compliance with the Bluetooth specification, strong correctness guarantees (safety and liveness), and seamless integration with Android's Bluetooth stack. We address the challenges posed by ambiguities and omissions in the specification by formalizing it into a Specification State Machine (SSM) and deriving an Operational State Machine (OSM) referenced from Android's existing implementation. Through state machine refinement and semantic conformance checking, we prove that our implementation correctly adheres to the specification. Additionally, our formal verification process guarantees that the state machine operates correctly under all scenarios, eliminating potential vulnerabilities that traditional testing methods might overlook. Our evaluation further demonstrates that our verified L2CAP implementation functions correctly in real-world scenarios without introducing noticeable performance overhead. This work not only advances Bluetooth verification efforts but also provides a structured approach to formally verifying protocol implementations against existing specifications.

References

[1] Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theoretical Computer Science* 82, 2 (1991), 253–284.

[2] Android Open Source Project. 2025. *l2c_csm.cc*. Google. https://android.googlesource.com/platform/system/bt/+/refs/heads/android12-release/stack/l2cap/l2c_csm.cc

[3] Per Bjesse. 2005. What is formal verification? *ACM Sigda Newsletter* 35, 24 (2005), 1–es.

[4] Bluetooth SIG. 2025. *Bluetooth Core Specification 5.4*. Bluetooth SIG, Inc. https://www.bluetooth.com/specifications/specs/core-specification-5-4/

[5] Bluetooth SIG. 2025. *L2CAP General Procedure - Vol. 3, Part A, Section 7*. Bluetooth SIG, Inc. https://www.bluetooth.com/specifications/specs/core-specification-5-4/

[6] Bluetooth SIG. 2025. *L2CAP Signaling Packet Formats - Vol. 3, Part A, Section 4*. Bluetooth SIG, Inc. https://www.bluetooth.com/specifications/specs/core-specification-5-4/

[7] Bluetooth SIG. 2025. *L2CAP State Machine - Vol. 3, Part A, Section 6*. Bluetooth SIG, Inc. https://www.bluetooth.com/specifications/specs/core-specification-5-4/

[8] Sascha Böhme and Tjark Weber. 2010. Fast LCF-style proof reconstruction for Z3. In *Interactive Theorem Proving: First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings 1*. Springer, Berlin, Heidelberg, 179–194.

[9] Bumble. 2025. Insights into States of LE Credit-Based Channel in L2CAP. https://github.com/google/bumble/discussions/627. [Online; accessed 15-Jan-2025].

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, Berlin, Heidelberg, 337–340.

[11] Robert W Floyd. 1993. Assigning meanings to programs. In *Program Verification: Fundamental Issues in Computer Science*. Springer, Berlin, Heidelberg, 65–81.

[12] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep specifications and certified abstraction layers. *ACM SIGPLAN Notices* 50, 1 (2015), 595–608.

[13] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 653–669.

[14] Creighton T Hager and Scott F MidKiff. 2003. An analysis of Bluetooth security vulnerabilities. In *2003 IEEE Wireless Communications and Networking, 2003. WCNC 2003.*, Vol. 3. IEEE, New York, NY, USA, 1825–1831.

[15] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. 2020. Storage Systems are Distributed Systems (So Verify Them That Way!). In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Savannah, GA, 99–115.

[16] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, USA, 1–17.

[17] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 165–181. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel

[18] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[19] Kaiyu Hou, You Li, Yinbo Yu, Yan Chen, and Hai Zhou. 2021. Discovering emergency call pitfalls for cellular networks with formal methods. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, New York, NY, USA, 296–309.

[20] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamarić, Neha Rungta, and Emina Torlak. 2022. Testing Dafny (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, New York, NY, USA, 556–567.

[21] Leslie Lamport. 1977. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering* SE-3, 2 (1977), 125–143.

[22] Leslie Lamport. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (1994), 872–923.

[23] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., USA.

[24] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning (LPAR)*. Springer, Berlin, Heidelberg, 348–370.

[25] K Rustan M Leino. 2018. Modeling concurrency in Dafny. In *Engineering Trustworthy Software Systems: Third International School, SETSS 2017, Chongqing, China, April 17–22, 2017, Tutorial Lectures 3*. Springer, Berlin, Heidelberg, 115–142.

[26] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[27] Chieh-Jan Mike Liang, Nicholas D Lane, Niels Brouwers, Li Zhang, Börje F Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, et al. 2014. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th annual international conference on Mobile computing and networking (MobiCom)*. ACM, New York, NY, USA, 519–530.

[28] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. 2019. InternalBlue-Bluetooth binary patching and experimentation framework. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, New York, NY, USA, 79–90.

[29] Haram Park, Carlos Kayembe Nkuba, Seunghoon Woo, and Heejo Lee. 2022. L2Fuzz: Discovering Bluetooth L2CAP vulnerabilities using stateful fuzz testing. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, New York, NY, USA, 343–354.

[30] Edgar Pek and Nikola Bogunovic. 2004. Formal verification of logical link control and adaptation protocol. In *Proceedings of the 12th IEEE Mediterranean Electrotechnical Conference (IEEE Cat. No. 04CH37521)*, Vol. 2. IEEE, New York, NY, USA, 583–586.

[31] Alexandre Petrenko, Gregor von Bochmann, and Rachida Dssouli. 1993. Conformance Relations and Test Derivation.. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test Systems VI*. ACM, New York, NY, USA, 157–178.

[32] Alexandre Petrenko, Nina Yevtushenko, Alexandre Lebedev, and Anindya Das. 1993. Nondeterministic state machines in protocol conformance testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test Systems VI*. ACM, New York, NY, USA, 363–378.

[33] Muhammad Taqi Raza and Songwu Lu. 2019. A Systematic Way to LTE Testing. In *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM, New York, NY, USA, 1–15.

[34] Jiliang Wang, Feng Hu, Ye Zhou, Yunhao Liu, Hanyi Zhang, and Zhe Liu. 2020. BlueDoor: breaking the secure information flow via BLE vulnerability. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, New York, NY, USA, 286–298.

[35] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on programming language design and implementation (PLDI)*. ACM, New York, NY, USA, 718–730.

[36] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Dave Jing Tian, Antonio Bianchi, Mathias Payer, and Dongyan Xu. 2020. BLESA: Spoofing attacks against reconnections in bluetooth low energy. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Savannah, GA, 1–12.

[37] Jianliang Wu, Ruoyu Wu, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. 2022. Formal model-driven discovery of bluetooth protocol design vulnerabilities. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, New York, NY, USA, 2285–2303.

[38] Fenghao Xu, Wenrui Diao, Zhou Li, Jiongyi Chen, and Kehuan Zhang. 2019. Badbluetooth: Breaking android security mechanisms via malicious bluetooth peripherals.. In *NDSS*. NDSS, USA, 1–15.

[39] 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 (CCS)*. ACM, New York, NY, USA, 3181–3194.

[40] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. 2020. Breaking secure pairing of bluetooth low energy using downgrade attacks. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Savannah, GA, 37–54.

A Appendix

Scenario	OG (ms)		VF (ms)	
1	25.68	±13.63	33.20	±21.17
2	165.36	±78.27	222.42	±92.14
3	228.42	±158.09	272.75	±173.05
4	249.24	±18.15	273.29	±30.59
5	241.12	±19.92	268.75	±28.96
6	367.45	±137.60	365.99	±152.61

Table 4: Cumulative setup time the channels established in scenarios averaged over the runs (OG: original Android, VF: our verified implementation).

Scenario	OG (ms)		VF (ms)	
1	16143.00	±562.72	15839.50	±304.65
3	483.90	±269.43	501.83	±220.95
4	54.74	±9.36	54.57	±9.42

Table 5: End-to-end performance for three scenarios.