PROFACTORY: Improving IoT Security via Formalized Protocol Customization

Fei Wang, *Purdue University*
Jianliang Wu, *Purdue University*
Yuhong Nan, *Purdue University*
Yousra Aafer, *University of Waterloo*
Xiangyu Zhang, *Purdue University*
Dongyan Xu, *Purdue University*
Mathias Payer, *EPFL*

Abstract

As IoT applications gain widespread adoption, it becomes important to design and implement IoT protocols with security. Existing research in protocol security reveals that the majority of disclosed protocol vulnerabilities are caused by incorrectly implemented message parsing and network state machines. Instead of testing and fixing those bugs after development, which is extremely expensive, we would like to avert them upfront. For this purpose, we propose PROFACTORY which formally and unambiguously models a protocol, checks model correctness, and generates a secure protocol implementation. We leverage PROFACTORY to generate a group of IoT protocols in the Bluetooth and Zigbee families and the evaluation demonstrates that 82 known vulnerabilities are averted. PROFACTORY will be publicly available [1].

1 Introduction

As a pillar for smart living, the scale of IoT (Internet of Things) has recently experienced unprecedented growth in both the number of devices and their complexity. According to reports from Statista [11] and Forbes [10], about 26.6 billion IoT devices were installed and connected worldwide in 2019 and they project to exceed 42 billion devices in 2022, lifting the global IoT market to more than 1.2 trillion dollars. Such a growth prospect and tremendous investment are continuously motivating efforts to develop innovative IoT wireless techniques, such as Bluetooth which has effective connectivity, low hardware cost and well-maintained development community [21, 49]. In particular, Bluetooth has enabled bridging interfaces between applications and wireless peripherals including keyboards/mice, headsets/speakers, smart watches, fitness trackers, medical recorders, smart home appliances, and hands-free systems [21, 49]. It was reported that 4.2 billion Bluetooth devices were shipped in 2019, notching a 8.8% compound annual growth rate (CAGR) over 5 years [16]. Zigbee is another popular IoT protocol which also witnesses a 8.0% CAGR in market growth, and its market value is expected to reach 4.3 billion by 2023 [14].

However, accompanied with the ubiquitous adoption, security has inevitably become a critical issue in IoT protocols. According to the latest Bluetooth market update [16], there were 14 member groups working on 80 active protocol projects during 2019. Actually, most of these protocol projects were built from scratch [13,16,56,64] and this repeated procedure is considered onerous, tedious, and error-prone [32,33,36,50,60,63], leading to lots of vulnerabilities in protocol implementations. As reported by previous research on protocol security [37,45], the majority of disclosed protocol vulnerabilities are due to incorrectly-implemented message parsing, where parsing errors can result from the lack of sanity checks (especially for the sizes of message fields) and parsing ambiguities (caused by diverting understanding of specification across developers). Bluetooth implementations are such examples, as reflected by newly reported Bluetooth-related CVEs (Common Vulnerabilities and Exposures) [56]. In addition, protocol implementation vulnerabilities are also present in state machine components [35, 44, 61]. Although protocol-specific fuzzing techniques [23, 34, 60] are helpful in exposing those implementation bugs, difficulties in stateful fuzzing, specification encoding, and achieving complete coverage make exposing all defects infeasible in practice. Therefore, rather than imposing a time-consuming postmortem bug finding procedure in protocol engineering, we propose to address the problem at the very beginning of the development life-cycle. Specifically, protocol developers ought to be relieved from the error-prone low-level implementation efforts. Instead, they should focus on the design of essential pieces of protocol specifications (e.g., message format and finite state machine) and the corresponding protocol implementations will be automatically generated in a manner that ensures security.

The recent research shows that formalizing protocol specifications is a promising approach to addressing this issue [12, 22, 43, 45, 55, 57]. In particular, message formats are customized in a DSL (Domain Specific Language) and secure protocol implementations are emitted accordingly. For example, EverParse [55] devises a DSL describing tag-length-value message formats to produce zero-copy parsers and corresponding serializers. In contrast with existing data serialization/deserialization tools (e.g., Protobuf [18] and Thrift [20]), parsers generated by EverParse are formally verified and their security is guaranteed. Johnson et al. [45] propose a similar DSL for generating additional sanity checks to harden USB (Universal Serial Bus) message parsers. Nevertheless, we observe that existing protocol formalization efforts mostly focus on message formats but many fall short in modeling some dynamic functionalities such as multiplexing and state transitions. Consequently, these DSLs require substantial extension to fit low-level and kernel-oriented IoT protocols.

Our Solution To this end, we elect to develop a new unified

DSL whose syntax can specify both protocol message formats and dynamic behaviors. We propose PROFACTORY to automatically generate secure low-level protocol implementations for Linux kernels from protocol specifications written in the DSL. It aims at facilitating protocol development, and eliminating message-parsing vulnerabilities and fundamental state-transition errors in protocol implementations. Currently, PROFACTORY targets code generation for IoT protocols in the Linux kernel including those in the Bluetooth and Zigbee families. It can be easily adapted for other protocols or production kernels as long as the platform-dependent interfaces and settings are available.

Specifically, PROFACTORY works as follows. First, a protocol is modeled/customized in our DSL. Then, symbolic model checking is performed to verify the model correctness (e.g., network state transitions are not vulnerable). After passing model checking, the customized protocol model is fed to the code generation engine to produce kernel-oriented protocol implementation which provides guarantees of being free from memory safety vulnerabilities (i.e., buffer overflow, invalid pointer dereference, memory leakage, use after free and double free) in message parsing and from concurrency control vulnerabilities (i.e., race and deadlock) in message multiplexing. Such guarantees are provided by the automatically generated sanity checking code, such as bound checks and input validation checks, and by applying automated verification tools to the generated code. Only if the protocol passes the model and implementation verification, should the implementation be integrated into the production kernel on both of the communication peers. Finally, the peers communicate through the customized protocol. We highlight our contributions in the following.

- We propose PROFACTORY, a novel system that realizes efficient and secure protocol customization. In PROFACTORY, developers formally and unambiguously model protocols in a DSL instead of natural languages and the models lead to the production of vulnerability-free implementations.
- We develop a type-based DSL that is closely coupled with protocol semantics. In our DSL, various protocol specifications such as message format, finite state machine and connection multiplexing can be well expressed by a number of abstract types.
- We develop a code generation engine, emitting kernel code without message-parsing errors according to the DSL-defined protocol model, where concurrency correctness and memory access safety of generated C codes are formally verified using VCC [2, 52] and Frama-C [17].
- We develop a symbolic model checker to capture potential bugs residing in protocol state transitions. Those bugs are abstracted as protocol property violations.
- We build a prototype of PROFACTORY and generate 8 protocols in Bluetooth and Zigbee. The evaluation demonstrates that PROFACTORY can help to avert 82 known vulnerabilities with low overhead in generated implementations.

2 Motivation

The increasing number of security issues in IoT protocol implementation motivates secure protocol customization. Generally, those issues can be divided into two categories, message-parsing vulnerabilities and state-transition errors. Next, we use two examples (one for each category) to illustrate how PROFACTORY can help avert them.

Motivating Example #1. This vulnerability (CVE-2017-1000251) [8] resides in the L2CAP implementation of Linux BlueZ (kernel 4.13.1 and older) and it was disclosed in the Blueborne report [56], allowing a malicious Bluetooth user to launch a denial-of-service or remote-execution attack. In L2CAP, before data transmission, the two peers are required to negotiate a group of connection options (or parameters) and the negotiation is accomplished by exchanging two kinds of messages, *configuration request* and *configuration response*. If a configuration request from a peer cannot be accepted, the peer has to send a second request based on the response contents (e.g., copying the configurations indicated in the response to the second request).

Figure 1 elaborates the buggy code and its official patch (in blue) [9], where `l2cap_config_rsp` handles response events and `l2cap_parse_conf_rsp` is for response parsing. At Line 4171, a local 64-byte buffer `buf` is allocated to hold the second request (as the previous request was pending, i.e., not accepted) and a loop (Line 3527–3537) is used to extract information from the response and emit the request body, where the parser invokes `l2cap_add_conf_opt` to append a request option to `buf`. Before patching, since neither the response size nor the buffer boundary is checked, an attacker can craft a long response such that the buffer is overflowed when the large number of configuration options in the response are copied. From the perspective of protocol specification, the reason for such a buffer-overflow vulnerability is that the protocol is actually under-specified. In particular, the developer allocates 64 bytes (a magic number of bytes) for the request, indicating that there must exist an upper bound for the response size which is not explicitly respected by the emission loop, causing the vulnerability. Fortunately, such specification confusions can be eliminated by PROFACTORY in the protocol modeling stage.

In our DSL, the option group is modeled as a parameter list (see modeling details in Section 4), for which a maximum size must be specified. Hence, we have the upper-bound check for the option group size. In addition, instead of manipulating bare buffers, PROFACTORY performs all the message-related operations on the well protected socket buffer data structure `sk_buff`. This data structure allows convenient field appending or truncating through `skb_pull` and `skb_put`/`skb_push`, and the kernel intrinsically performs all the needed boundary checks. Furthermore, the structure always maintains the current data length and hence the size of each message segment (e.g., the option group) can be strictly

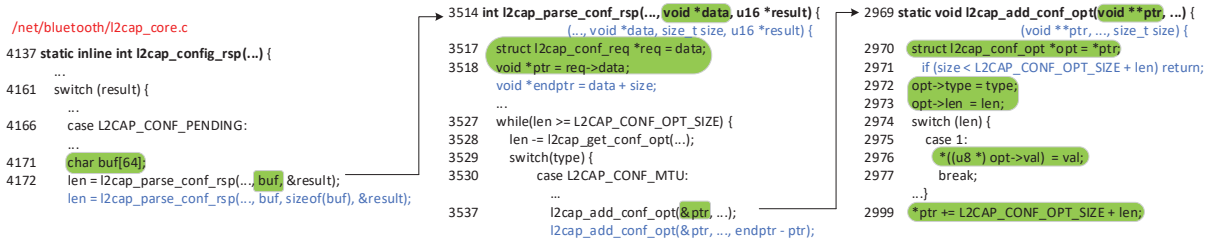

Figure 1: L2CAP configuration buffer overflow in BlueZ implementation

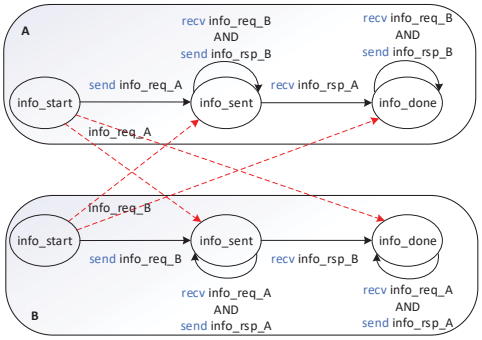

Figure 2: Message loss in L2CAP information exchange

validated when unpacked. Overall, PROFACTORY rejects any unspecified/invalid messages and offers secure message parsing/construction.

Motivating Example #2. This example demonstrates a typical transition error in an asynchronous state machine. In L2CAP, at the beginning of a connection, each peer can request information from the remote side in order to set up a connection with the supported functionalities. The textual specification of L2CAP simply mentions "*L2CAP implementations shall respond to a valid information request with an information response*" [15] without much detail. Figure 2 presents two buggy asynchronous state machines for two communicating devices A and B, respectively. In the state machines, the solid edges denote state transitions and the dashed edges denote messages (to the other party). The problem lies in that a device may undesirably drop requests when it is at the `info_start` state. The buggy state machine requires the device to first send a request before it can properly receive requests from the other party. Initially, assume both devices are in the `info_start` state and they both send out a request in the same time. However, the two sends form a race. It is hence possible that when the request from a remote peer arrives, the local device is not in the next state that can receive the message. This causes undesirable loss of messages. The user may encounter difficulties in establishing a Bluetooth connection due to the bug. This error is averted by performing the race-free property checking in PROFACTORY (Section 6). The fix is to allow devices to receive messages in `info_start`.

3 Approach Overview

Figure 3 presents the overall workflow of PROFACTORY. PROFACTORY executes in three main phases: *Protocol Modeling*, *Code Generation* and *Automated Verification*.

Protocol Modeling Phase. When creating a customized protocol, developers first need to model the protocol in PROFACTORY's DSL as the system input. The DSL, in a sense, is de-facto a translator of protocol specifications. In this DSL, all the protocol elements are abstracted as hierarchical data types and each of those types would instruct PROFACTORY to emit a unique set of concrete data structures and code blocks in the following code generation phase. Therefore, modeling a protocol in PROFACTORY corresponds to assembling definition instances of those types. To enrich the language to support various protocols, the design of the DSL syntax is closely coupled with protocol semantics including message format, FSM (finite state machine) and other protocol-specific features. The DSL is embedded in Haskell, which offers an easy-to-use development environment that allows manipulating its own syntax and facilitates implementing a DSL. We will discuss the complete DSL design in Section 4.

Code Generation Phase. In this phase, PROFACTORY automatically produces C code for a production kernel, according to the input protocol model. The generated code consists of type-based code blocks (i.e., protocol message data structure definitions and message parsing/construction procedures) and kernel shims which can assist seamless code insertion or replacement in a production kernel. In particular, the type-based code blocks perform both message recognition and state transition, while kernel shims prepare standard socket interfaces and the accesses to the underlying platform or lower-layer protocols, requiring the hardcoded platform-dependent interfaces. As demonstrated in the first motivating example, the lack of sanity checks is an important contributor to protocol vulnerabilities, PROFACTORY hence enforces validity verification on corresponding message fields when emitting code for message parsing/building. The details of code generation are elaborated in Section 5.

Automated Verification Phase. Protocol security issues include not only vulnerabilities incurred by field mishandling in message parsing and problematic concurrent accesses to shared information, but also correctness problems in the underlying FSM. Therefore, PROFACTORY verifies the generated implementation through VCC [2, 52] (free from race and deadlock) and Frama-C [17] (free from buffer overflow, invalid pointer dereference, memory leakage, use after free and double free). In Frama-C verification, limited manual

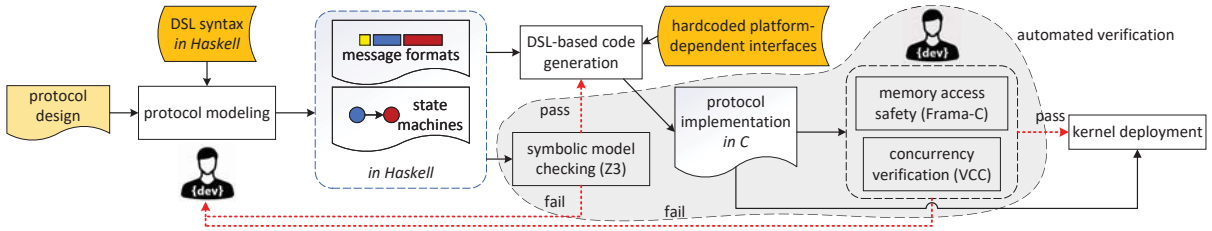

Figure 3: The overall workflow of PROFACTORY (black arrow denotes data flow and red arrow denotes control flow)

intervention may be required to assist constructing proofs. In addition, PROFACTORY also performs model checking for the protocol state machine. We encode the protocol model and perform property checking using the Z3 SMT solver [42]. We devise a set of general properties (i.e., transition, security and customization properties) that should be respected by a protocol state machine and validate them against protocols written in our DSL. Protocol models failing to pass any of the verifications should be remodeled by developers. There have been research efforts in protocol model checking [24, 27, 28, 38, 44, 53], but the majority of them target proving cryptographical correctness for AKA (authentication and key agreement) protocols or components of specific protocols (e.g., TCP/IP), while the model checking in PROFACTORY focuses on generic functional correctness (e.g., correct state transitions). Our work is therefore orthogonal and complements existing work. This phase will be explained in Section 6.

4 Protocol Modeling

In protocol specifications, message format and FSM are the two building blocks. Hence, our DSL focuses on describing these two perspectives. To facilitate precise discussion, we simplify and summarize the syntax of our DSL in Figure 4 and semantics in Figure 5. As illustrated in Figure 4, a protocol can be represented by a set of abstract data types related to message format, socket, and state machine. These abstract types are further instantiated to concrete types when describing individual protocols. Low level implementation is automatically generated from the DSL specification. In the following, we first explain the syntax and then the semantics of the DSL, which are followed by an example.

4.1 DSL Syntax

Message Format. Most network messages have hierarchical structure, meaning that a network message m is often the raw data field of a message at a lower layer. It often has its own structure too, encapsulating some message(s) at a higher layer. Note that even the messages of a same protocol are organized in multiple layers. For example, L2CAP command messages are encapsulated in generic L2CAP messages. With such layered structures, the corresponding network message parsing code largely follows a fixed pattern, namely, the parser for a message m of a particular layer unfolds the structure at that

MESSAGE FORMAT RELATED ABSTRACT TYPES
Fix f ::= $(n_{size}, n_{low}, n_{high})$
Var v ::= (n_{low}, n_{high})
Hdr h ::= $f^+ \cdot f_{len}$
Para p ::= $(n_{key}, f_{key} \cdot f_{val})$
Plist ℓ ::= $(n_{size}, \mathcal{P}(p))$
Msg m ::= $v \mid p \mid f^+ \mid \ell \mid h \cdot m_{sub} \mid h \cdot (f_{type} = n_{type}\ ?:\ m_{sub})^+$
SOCKET RELATED ABSTRACT TYPES
Chan ch ::= (n^*_{key})
Conn cn ::= (n^*_{key})
STATE MACHINE RELATED ABSTRACT TYPES
State s ::= $n_{timeout}$
Recv r ::= $(m_{in}, (e, s_{from}, s_{to}, (m_{out}|\varepsilon), \{S\})^+)$
Send d ::= $n_{act} \cdot (m_{out}, (e, s_{from}, s_{to}, \{S\})^+)$
STATEMENT
S ::= $S_1; S_2 \mid x := e$
$\mid$ **if** (e) $\{S_t\}$ **else** $\{S_f\}$
$\mid$ **for** $(x$ **from** e **to** $e)$ $\{S\} \mid$ **iter** $(\ell, \{S\}) \mid \ldots$
EXPRESSION
e ::= $n \mid string \mid bool \mid x \mid \ominus e \mid e \oplus e \mid \ldots$

Figure 4: PROFACTORY DSL Syntax (n denotes an unsigned integer constant and operator $\cdot$ denotes field concatenation)

level, checks some field that determines the (inner) message type of the raw data field, and further invokes the corresponding parser(s) of the inner message(s). This regularity allows us to abstract network messages and message parsing to a set of general abstract types that have hierarchical relations. An example of abstracting L2CAP messages using our DSL can be found later in the section.

We introduce 6 abstract types to describe (hierarchical) message formats, two basic field types *Fix* and *Var* denoting a fixed-sized field and a variable-sized field, respectively, and four other types: Header *Hdr*, Parameter *Para*, Parameter List *Plist*, and Message Format *Msg* that are built on the two basic types. Besides, we provide two socket related abstract types *Conn* and *Chan* to model network connections. We explicitly model connection types to allow easy code generation for the interface with the kernel.

Fixed-sized Field (Fix) A fixed-sized field consists of three integer attributes, describing its size (n_{size}) and range (n_{low} and n_{high}). The size is measured in bytes. The range attributes (can be *nil*) specify the lower and upper bounds of the field. This could be further extended to support other value constraints. The attributes allow safe code generation (with bound checks and validity checks).

Variable-sized Field (Var) A variable-sized field represents a byte sequence, with its size range specified by n_{low} and n_{high},

which enable mandatory bound checks in code generation.

Header (Hdr) A message header is a sequence of fields followed by a length field, which is a dedicated fixed-sized field describing the length of the following message content. Note that we are defining an abstract type, which is further instantiated to concrete types for a specific protocol. In general, the fields in a header describe the meta information of a message and instruct the parser to correctly extract and process sub-messages. For example, a message header $f_{type} \cdot f_{id} \cdot f_{len}$ consists of three fields describing the type of the message (that determines how the message body is interpreted), the connection ID and the length of the body. Note that the operator $\cdot$ denotes field concatenation. The length field is for automatic generation of validity check.

Parameter (Para) A parameter (in message) consists of two fields, a key field f_{key} and a value field f_{val}. A parameter abstract type consists of a constant n_{key} that uniquely identifies the parameter kind and the specifications of the two fields. Parameters denote configurations that can be negotiated across the peers of a connection. Intuitively, if the value of the key field (at runtime) matches the constant n_{key}, the parameter is of the corresponding type. For instance, $(2,\ (4,2,2) \cdot (4,0,1024))$ is an MTU (Maximum Transmission Unit) parameter type, with a static value $n_{key} = 2$ denoting the type, the first key field 4 bytes long with a fixed value of 2, and the second value field 4 bytes long with a value in [0,1024]. At runtime, a concatenation of two fields (in a message) is considered an MTU parameter when the first field has the value of 2.

Parameter List (Plist) A parameter list denotes a variable set of parameters (whose f_{key} sizes must be the same) with n_{size} specifying the maximum number of parameters in it.

Message (Msg) A message could be a variable-sized field, parameter, sequence of fixed-sized fields, parameter list, header followed by a sub-message, or header followed by multiple possible sub-messages. The last two alternatives describe the hierarchical structure of network messages. Specifically, in the last alternative, field f_{type} is a field in the header h. When it has the value of n_{type}, the sub-message is of the m_{sub} type. Note that a concrete message type may have multiple sub-message branches. PROFACTORY automatically generates parsing code based on the specified hierarchy. An example can be found later in the section.

Socket Related Types. In a production kernel, a protocol has a socket-like interface that serves the applications. The interface includes a number of socket peripheral data types such as protocol connection *Conn* and protocol channel *Chan*. A protocol may have multiple channels sharing an end-to-end connection (for multiplexing). The abstract type of a connection/channel consists of a list of static values n_{key} that uniquely identify the parameters for the connection/channel. The key values are a subset of the key values in parameter type definitions. Sample parameters include device type for a connection and MTU for a channel in L2CAP. Different from other abstract types, PROFACTORY only allows one instantiation for *Conn* and *Chan*, meaning that all the connections and channels in a protocol have to be homogeneous.

State Machine. Protocol execution is largely driven by state machines. In particular, besides message parsing, the other focus of protocol implementation is to properly update state machines. Upon receiving a message, protocol implementation parses the message, updates some state variable(s), composes and sends a response message if needed. In some cases, it performs side-effect operations such as logging critical events and collecting statistics. Although most protocols follow the same execution model, their low level implementations have substantial diversity. For example, they may or may not have explicit state variable(s); some protocols update state variables before sending response whereas some others the opposite. Our DSL leverages the inherent regularity of the execution model to produce uniform implementation, enabling security and easy verification. Our DSL allows developers to specify protocol state machines, through three abstract types *State*, *Recv* and *Send*.

Protocol State (State) A state type is defined by a constant denoting the timeout of the state, which specifies the maximum time a state must be retained before a connection is terminated if no legitimate connection activity is observed, in case of idle/crashed applications and lost connections. To specify a concrete protocol, the developer often needs to define multiple states. PROFACTORY pre-defines a number of them including the start state s_{init}, and the end state s_t.

State Transition (Recv, Send) *Recv* specifies the state transition and the associated operations that are triggered by a received message and *Send* specifies transitions and operations triggered by a message-sending request. In *Recv*, m_{in} is the received message, followed by a group of transition options. Each option is guarded by an expression e. If the expression is evaluated to true, a (compound) statement S is executed, the state is updated from s_{from} to s_{to}. Meanwhile, a response message m_{out} may be sent. S typically includes invocation of the receive function of the sub-message of m_{in}, creating a channel/connection, setting/getting a parameter value, constructing m_{out}, and collecting statistics. The syntax of *Send* is similar with m_{out} the message to send, but it specifies an action type n_{act} to express one of the only three socket operations, i.e., connection establishment, message delivery and connection shutdown, that perform active message sending. Note that socket errors (e.g., a message was not successfully received/sent) get automatically handled by the state timeout (see "Timer and Counter" in Section 5).

Statement and Expression. The syntax of statements and expressions in our DSL largely follow the C language. Statements and expressions are mostly used in type definitions. Different from a program in mainstream programming languages that often has a *main()* function that specifies the main logic of a program, protocol code is event-driven, for instance, by connection, send, and receive events. As such, in PROFAC-

TORY the main logic of a protocol is directly derived from the type definitions, in the form of a list of event handlers and functions called by these handlers. Statements and expressions are merely part of the type definitions such as e and S in *Recv* and *Send*. PROFACTORY supports assignment, `if` $-$ `else`, `for` loop, constants, variables, common unary or binary operations. In addition, it provides a number of statements convenient for network protocols. For instance, *iter* executes statement S on each element in a parameter list ℓ.

4.2 DSL Semantics

Figure 5 presents the semantics of a subset of DSL specifications, with the data structures and auxiliary functions used, followed by the rules. Different from a regular programming language, in which each statement has concrete semantics, our DSL is mainly for type definitions. As such, we define semantic rules for *concrete* type definitions. In the semantic rules section of Figure 5, the first column shows concrete type definitions and the corresponding semantic rules in the second column show a list of functions and symbolic constraints derived from the definition. The functions define a list of operations for a concrete type. Some of the functions are event handlers that constitute the interface of protocol. The symbolic constraints are used for symbolic modeling checking that validates the correctness of protocol specifications.

Specifically, for a type definition, PROFACTORY updates $\mathbb{F}$, which denotes the list of functions defined, and $\mathbb{P}$, which denotes the list of symbolic constraints derived. Inside a function, the semantics is described using C-like statements, many of which update a store σ that is similar to a store in classic programming language semantics. Intuitively, one can consider it as a hash-map that projects a key or a number of keys to some value. Here, key can be a name, a value, or a variable.

Rule 1 specifies that for a definition of fixed-sized field, two functions are introduced, with $f.parse(M,V)$ parses the field and $f.compose(V,x)$ composes the field (as part of whole message composition). In the parser function, parameter M is a handler for the message (kind of id for the message). One can intuitively consider each incoming message (to parse) has its unique handler; V is a buffer passed in from the kernel, containing the message (or part of a message). The function copies n_s (i.e., the size of the field) bytes from V to the store. Note that the concrete field value is indexed by the handler M and the symbolic field name f. This is because f is a field *type* instead of a concrete field. In the composition function, variable x denotes the value used to compose the field and V is the buffer storing the message to compose. V will be passed on to the kernel to send a message after composing the whole message. In addition to the functions, three symbolic constraints are added to $\mathbb{P}$ dictating that the (symbolic) length of the field is equal to n_s, and the (symbolic) value of the field must be in between n_l and n_u. These constraints will be used for model checking. The semantics for a variable-sized field definition is similar.

In Rule 3 for a header consisting of two fields f_1 and f_{len} (for message body length), the parser function parses the two fields in order by invoking their parser functions. This implies that the two field types need to be defined. The expression $V[f_1.len,]$ means that a sub-buffer starting at offset $f_1.len$ of V. The composition functions copies the two variables x_1 and x_{len} to the result buffer. The symbolic constraint dictates that the length of header be the sum of the lengths of both fields.

In Rule 4 for a parameter, the parser function parses the two fields and asserts that the value of the key field must be equal to the specified key value n_{key}. A global function, i.e., a function not specific to a definition, $setPara(x_c,x_k,x_v)$ is also introduced to set a parameter, with x_c denoting the connection/channel, x_k the key, and x_v the value. It sets the parameter denoted by the value of x_k to the value of x_v. It will be invoked when connections/channels are created/configured.

In Rule 5 for a parameter list, the parser function traverses through the buffer V and parses individual parameters until it reaches the end of V or the number of parameters reaches the upper bound n_s. The last symbolic constraint requires that the parameters (in the list) have the same key field size. Note that $strlen(V)$ means the dynamic length of buffer V which is not null-terminated.

In Rule 6 for a message with two possible sub-message formats, the parser function first parses the header. It then checks the value of the type field f_t in the header and invokes the parser of the corresponding sub-message ("?:" is similar to "switch-case"). The composition function is symmetrically defined, with x_t the type of the sub-message, V_{sub} a buffer containing the sub-message composed before-hand, and x_l the length of the sub-message. The symbolic constraints ensure that (1) the value of the type field must be n_{t1} or n_{t2}; (2) if it is n_{t1}/n_{t2}, the message length is the sum of the header and the sub-message m_{t1}/m_{t2} and the value of the length field f_{len} in the header must match the length of m_{t1}/m_{t2}.

In Rule 7 for a channel definition, two global functions are introduced. The first one is to create a new channel, with x_c denoting the connection to which the channel belongs, x_{addr} the address of the channel, and x_1, x_2 the values for the channel parameters n_1 and n_2. Inside the function, a new local channel id is created to uniquely represent the new channel. The state of the channel is set to s_{init} and the list of channels for the connection is updated. Some protocols explicitly specify channel id in their messages so that they can be properly attributed. However, there are protocols that implicitly encode channel id in some parameter(s). For example, L2CAP may encode channel id in PSM (Protocol Service Multiplexer). As such, we provide a *findChanByPara()* function to help look up a channel in connection x_c, using the parameter key x_k and value x_v. It returns the reference to the found channel or `NULL`. In Rule 8, the creation function of connection is similarly defined. The list of channels is initialized to empty.

Rule 9 specifies the semantics for the definition of a *Recv* state transition, which leads to the definition of a *receive*()

Data Structures and Auxiliary Functions

σ : store that maps key(s) to a value; $\mathbb{F}$: functions defined; $\mathbb{P}$: symbolic constraints; $\mathbf{Ch_{cur}}$: the current channel; $\mathbf{s_{init}}$: the initial connection state

newId(): acquire a new connection or channel id; **newMsgHandler**(): acquire a message handler (like a file handler)

Semantic Rules

1. f:=**Fix** (n_s, n_l, n_h)

$$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } f.parse(M,V)\ \{\sigma[M][f]=V[0,n_s];\},\\ \textbf{def } f.compose(V,x)\ \{V[0,f.len-1]=\sigma[x];\}\end{array}\right\}\quad \mathbb{P}+=\left\{\begin{array}{l}f.len=n_s, f.val\le n_h,\\ f.val\ge n_l\end{array}\right\}$$

2. v:=**Var** (n_l, n_h)

$$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } v.parse(M,V,x_l)\ \{\sigma[M][v]=V[0,\sigma[x_l]-1];\},\\ \textbf{def } v.compose(V,x_l,x_v)\ \{V[0,\sigma[x_l]-1]=\sigma[x_v];\}\end{array}\right\}\quad \mathbb{P}+=\{v.len\le n_u,\ v.len\ge n_l\}$$

3. h:=**Hdr** $f_1\cdot f_{len}$

$$\mathbb{F}+=\left\{\begin{array}{ll}\textbf{def } h.parse(M,V)\ \{ & \textbf{def } h.compose(V,x_1,x_{len})\ \{\\ \quad f_1.parse(M,V); & \quad V[0,f_1.len-1]=\sigma[x_1];\\ \quad f_{len}.parse(M,V[f_1.len,\ ]);\} & \quad V[f_1.len,f_1.len+f_{len}.len-1]=\sigma[x_{len}];\}\end{array}\right\}$$
$$\mathbb{P}+=\{h.len=f_1.len+f_{len}.len\}$$

4. p:=**Para** $(n_{key}, f_{key}\cdot f_{val})$

$$\mathbb{F}+=\left\{\begin{array}{ll}\textbf{def } p.parse(M,V)\ \{ & \textbf{def } setPara(x_c,x_k,x_v)\ \{\\ \quad f_{key}.parse(M,V); & \quad \textbf{switch}(x_k)\ \{\\ \quad \textbf{assert}(\sigma[M][f_{key}]\equiv n_{key}); & \quad\quad \textbf{case } n_{key}:\ \sigma[\sigma[x_c]][n_{key}]=\sigma[x_v];\\ \quad f_{val}.parse(M,V[f_{key}.len,\ ]);\}, & \quad\quad ...\}\\ & \quad\}\end{array}\right\}$$
$$\mathbb{P}+=\{n_{key}=f_{key}.val,\ p.len=f_{key}.len+f_{val}.len\}$$

5. l := **Plist**$(n_s, \mathcal{P}(\{$
$p_1:=(n_{k1}, f_{k1}\cdot f_{v1}),$
$p_2:=(n_{k2}, f_{k2}\cdot f_{v2})\})$

$$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } l.parse(M,V)\ \{\\ \quad \textbf{while}(i<n_s\ \&\&\ j<strlen(V))\ \{\\ \quad\quad \textbf{switch}(V[j,\ j+f_{k1}.len-1])\ \{\\ \quad\quad\quad \textbf{case } n_{k1}:\ p_1.parse(M,V[j,]);\ j+=p_1.len\\ \quad\quad\quad ...\\ \quad\},\end{array}\right\}$$
$$\mathbb{P}+=\{l.len=p_1.len+p_2.len,\ f_{k1}.len=f_{k2}.len\}$$

6. m := **Msg**$(h:=f_t\cdot f_{len})\cdot$
$(f_t=n_{t1}\ ?:\ m_{t1}\ |$
$f_t=n_{t2}\ ?:\ m_{t2})$

$$\mathbb{F}+=\left\{\begin{array}{ll}\textbf{def } m.parse(M,V)\ \{ & \textbf{def } m.compose(V,x_t,V_{sub},x_l)\ \{\\ \quad h.parse(M,V); & \quad \textbf{if}(\sigma[x_t]\equiv n_{t1})\ \{\\ \quad \textbf{if}(\sigma[M][f_t]\equiv n_{t1}) & \quad\quad h.compose(V,n_{t1},x_l);\\ \quad\quad m_{t1}.parse(M,V[h.len,\ ]); & \quad\quad V[h.len,h.len+x_l-1]=V_{sub};\\ \quad \textbf{if}(\sigma[M][f_t]\equiv n_{t2}); & \quad\}\\ \quad\quad m_{t2}.parse(M,V[h.len,\ ]);\}, & \quad ...\end{array}\right\}$$
$$\mathbb{P}+=\{n_{t1}=f_t.val\vee n_{t2}=f_t.val,\ f_t.val=n_{t1}\rightarrow(m.len=h.len+m_{t1}.len\wedge f_{len}.val=m_{t1}.len),$$
$$f_t.val=n_{t2}\rightarrow(m.len=h.len+m_{t2}.len\wedge f_{len}.val=m_{t2}.len)\}$$

7. ch:=**Chan** (n_1, n_2)

$$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } createChan(x_c,x_{addr},x_1,x_2)\ \{\\ \quad id=\textbf{newId}();\ \sigma[id][addr]=\sigma[x_{addr}];\ \sigma[id][n_1]=\sigma[x_1];\ \sigma[id][n_2]=\sigma[x_2];\\ \quad \sigma[id][state]=\mathbf{s}_{init};\ \sigma[\sigma[x_c]][channels]+=\{id\};\ \textbf{ret } id;\}\\ \textbf{def } findChanByPara(x_c,x_k,x_v)\ \{\\ \quad \textbf{foreach}\ (id\in\sigma[\sigma[x_c]][channels])\\ \quad\quad \textbf{if}\ (\sigma[id][\sigma[x_k]]\equiv\sigma[x_v])\ \textbf{ret } id;\}\end{array}\right\}$$

8. cn:=**Conn** (n_1, n_2)

$$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } createConn(x_1,x_2)\ \{\\ \quad id=\textbf{newId}();\ \sigma[id][channels]=\{\};\ \sigma[id][n_1]=\sigma[x_1];\ \sigma[id][n_2]=\sigma[x_2];\\ \quad \textbf{ret } id;\}\end{array}\right\}$$

9. r := **Recv** $(m_{in},\ (e_1, s_{from},$
$s_{to},\ m_{out},\ \{S\}), (e_2, ...))$

$$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } r.receive(V)\ \{\\ \quad M=\textbf{newMsgHandler}();\\ \quad m_{in}.parse(M,V);\\ \quad \textbf{if}\ (e_1)\ \{\ S;\ \textbf{assert}(\sigma[\mathbf{Ch}_{cur}][state]\equiv s_{from});\ \sigma[\mathbf{Ch}_{cur}][state]=s_{to};\\ \quad\quad\quad m_{out}.compose(V,...);\ \textbf{send}(V);\}\\ \quad \textbf{if}\ (e_2)\ ...\ \}\end{array}\right\}$$
$$\mathbb{P}+=\{(e_1\wedge state=s_{from})\rightarrow state=s_{to},\ e_2...\}$$

Figure 5: ProFactory DSL Semantics

Figure 6: A running example of modeling a subset of Bluetooth L2CAP specifications

function. If the message m_{in} is a top level message, the function is invoked by another protocol at the lower layer. Otherwise, it is invoked by the receive functions of higher level messages. Inside the function, a handler is first allocated to denote the message. One can intuitively consider it as an id. Message m_{in} is then parsed. If the expression e_1 is satisfied, statement S is executed; the state is updated from s_{from} to s_{to}; a response message is composed and sent. Similarly, if the expression e_2 is satisfied, a different transition is performed. The symbolic constraint specifies the possible state transitions. The semantics for *Send* is similarly defined and elided.

4.3 A Real-world Example

In the Bluetooth protocol stack, L2CAP is one of the most critical protocols, responsible for protocol multiplexing and data delivery between applications and the protocol stack. It sits on top of the HCI (Host Controller Interface) layer (i.e., a link layer) and serves a large number of upper layers such as RFCOMM (Radio Frequency Communication), HIDP (Human Interface Device Profile), and BNEP (Bluetooth Network Encapsulation Protocol). Figure 6 (a) shows a few simplified code snippets from a Linux Bluetooth 5.0 implementation. They are to handle L2CAP command messages.

Function `l2cap_sig_channel` is invoked by a callback from the lower HCI layer to process a L2CAP command. Depending on the command type (Line 5335) in the command header (Line 5702), it invokes function `l2cap_connect_req` to process a connection request or `l2cap_config_req` to process a configuration request. Inside `l2cap_connect`, Line 3777–3820 leverage the PSM (protocol service multiplexer, like port for TCP) field to look up a parent channel listening to this kind of service request. If there is such a channel and no existing channel is using the requested source channel ID (Line 3771, 3798), it spawns and initializes a new L2CAP channel (the channel is initialized to the initial `BT_START` state, which is not explicitly shown in the snippets). It then sets the channel to `BT_CONFIG` state, and uses `l2cap_send_cmd` which is a wrapper API of the lower HCI layer to send a response message (Line 3841–3870). Inside `l2cap_parse_conf_req`, the loop in Line 3316–3325 traverses a list of configuration options. For example, it sets the MTU of the channel if an mtu option is included (Line 3323). Inside `l2cap_config_req`, a configuration request cannot be accepted/parsed unless the channel is at `BT_CONFIG` state (Line 4052). After parsing, it sends a response message and sets the channel to the ready state `BT_CONNECTED` (if the configuration is successful, Line 1251–1261).

Using our DSL, we can rewrite the complex implementation (hundreds of LOC) to 31 LOC in DSL as shown in Figure 6 (b). We use yellow, blue and green in (a) and (b) to mark artifacts related to L2CAP commands, connection requests, and configuration requests, respectively. We also use circled annotations in (a) to associate concrete variables and operations to abstract types in (b). For example, `u8*data` at Line 5700 is abstracted to Line T21, meaning that it is a L2CAP command message whose header directs the parsing to different commands. The corresponding message formats and state machine are shown in Figure 6 (b.i, b.ii).

5 Code Generation

ProFactory automatically generates C code from the DSL specification. The semantic rules in Figure 5 specify the functions we need to generate and the semantics of these functions. However, those functions are still abstract. In this subsection,

we discuss how the concrete C code is generated.

Specifically, each type definition in a protocol specification leads to a C data structure. For example, a header type definition $h :=$ **Hdr** $f_{type} \cdot f_{len}$ leads to a C data structure definition "*typedef struct* {... f_{type} ...; f_{len} ...;} h;". Functions like those described in the semantic rule of the type definition are generated. For the above header example, the two functions in Rule 3 in Figure 5 are generated. Hash-map operations through the store, e.g., $\sigma[M][f_{type}]$, are compiled to the corresponding data structure field accesses.

Sanity Checks An important advantage of PROFACTORY is that it ensures parser security by inserting bounds checks and input validity checks. Note that the length of each field, header, message is clearly specified in our DSL. If necessary, value ranges are also specified. Runtime checks like those in the symbolic constraints set $\mathbb{P}$ (derived while compiling the protocol specification) are automatically inserted.

Multiplexing PROFACTORY supports multiplexing which entails concurrent connections. To avoid races, PROFACTORY automatically adds mutexes to guard accesses to data structures involved in multiplexing, such as the channel list field in a connection data structure, the current connection/channel variable, and connection/channel parameter data structures.

Packet Fragmentation Developers are oblivious to the implementation details of packet fragmentation. They only need to specify how the MTU value is negotiated as part of the protocol specification. To support fragmentation, PROFACTORY automatically inserts an additional fragmentation header (including fragment ID, offset and continuation flag) into the header of each message and the fragmentation logic is injected in message send/receive functions.

Timer and Counter PROFACTORY does not customize timer or counter constructs. Instead, it leverages the generic kernel socket timer *sk_sndtimeo* (40s is a reasonable timeout value) for message sending, and the delayed callback registration *schedule_delayed_work*() (value is set by the timeout attribute of *State*) for message receiving, where the timeouts trigger the close of sockets. They are transparent to developers, and they are automatically generated for *Recv* and *Send*.

Cryptographic Operations Modeling cipher suites is out of the scope of our DSL, while cryptographic constructs are packed into prepared interfaces. Specifically, PROFACTORY offers two expressions *setSec*(*int*) and *getSec*() (omitted in Figure 4 for brevity) for security level setting and fetching. *setSec*(*int*) sets the security level to a predefined integer value, which indicates whether the protocol performs encryption/decryption. The lower-layer protocol checks the setting, establishes the corresponding lower-layer connection and delivers messages. The lower-layer of the other communication peer updates this security level after the lower-layer connection is established, but this is oblivious to the upper-layer. If the other peer wants to know what security level the communication operates on, it should explicitly check it by using *getSec*(). Developers are oblivious to the implementation details of those interfaces but only regard them as cryptographic delegating pipes of the lower-layer. For instance, L2CAP encloses a security level *sec_level* in the channel data structure, and the lower-layer (HCI) accesses this value for encryption jurisdiction when performing message delivery. NWK maintains *nwkSecurityLevel* in NIB (Network Information Base), and the lower-layer (MAC) accesses this value to apply corresponding security strategies.

Interfacing with Application, Kernel and Other Protocols While our DSL is platform-independent, allowing to specify the main logic of network protocols, the generated code has to be platform dependent, interfacing with four parties: *user space applications*, *the underlying kernel*, *lower layer protocol* and *upper layer protocol(s)*. PROFACTORY currently supports Linux. The generated implementation for a protocol is packaged as a Linux kernel module. In the following, we explain the four interfaces. Then we present an example.

All network protocols in Linux interface with user applications through the socket interface, which includes socket data structure and a number of API functions such as *bind*(), *listen*(), *accept*(), *connect*(), *send*() and *recv*(). To setup the user space interface, the kernel module initialization needs to register the protocol by providing the protocol name such as "`L2CAP`" and the socket data structure of the protocol (containing a reference to a channel data structure). It also registers a list of functions implementing the aforementioned APIs, e.g., *l2cap_sock_sendmsg*() is registered for *send*() and *sendmsg*(). According to the action type n_{act}, *connect*(), *send*()/*sendmsg*() and *shutdown*() are connected to the message sending functions emitted through *Send* instances. This is transparent to developers. The user space *send*() and *recv*() functions are merely sending and receiving raw data such that the underlying protocol is completely transparent to them.

The generated code interfaces with the upper layer protocols through a provided callback function. Theoretically, such functions can be registered. However, the current Linux protocol stack implementation hardcodes them. For example, the callback function provided by the Bluetooth family to L2CAP for raw data delivery has a fixed name *l2cap_data_recv*(), which is invoked inside the body of m_{data}.*recv*() that receives an L2CAP data message. Note that invoking the actual callback is transparent to developers but they only need to write a *deliver* expression (omitted in Figure 4 for brevity) to fulfill this. The interface with the lower layer protocol is similar. Upon receiving a data message in the lower layer (addressing our protocol), a fixed function is invoked that further invokes the various parser functions generated from DSL. Developers are also oblivious to the connection between the callback and the generated parser functions. The generated code also makes use of kernel functions such as socket allocation *sk_alloc*() when creating new channels, to which the developers are oblivious. An example can be found in Appendix A.

Code Generation Algorithm The algorithm of code generation is simplified and summarized in Algorithm 1. First,

data structures are emitted to compose a header file (Line 1–8). Note that the emission of some fields (e.g., timer, state, linked list, mutex and lock) in channel and connection structure is transparent to developers, and PROFACTORY leverages those fields to fulfill state transitions and multiplexing. Then, *findChanByPara* function is generated for each of the parameters defined in the channel structure (Line 9–11), and *createChan* and *createConn* are also emitted (Line 12–13), where concurrency control operations are included. For each message containing a header, a parser is generated to extract (*skb_pull*) the header (Line 16–17), perform sanity checking (Line 18) and invoke an inner parser (Line 19), composing a hierarchical parsing tree. Similarly, a hierarchical message constructor (Line 23–24) is generated to set header fields (e.g., length field). In contrast, a receiving transition defines the parser of a base message (leaf parsing node) without a header. Similarly, it extracts all the fields (Line 28–29) and performs sanity checking (Line 30). In addition, it allocates memory (Line 32), prepares (*skb_put*) local references (Line 33–34), conducts the concrete state transition (Line 37–38), and performs packet delivery (Line 39) for the outgoing message (if applicable). In particular, the optional header argument $[h]$ in Line 19 only applies to the parser of base message, which is reflected in Line 27. This design aims to handle message formats where the innermost content parsing involves the information of the adjacent header. Finally, a sending transition defines a message serializer which is almost the same as the construction of the outgoing message in a receiving transition, but the only difference is that it is allowed to carry a user-space data chunk passed through a socket sending function (Line 46, 56). Note that the recursive code generation of expressions/statements, and the channel/connection unlocking operations at the end of parsing are omitted for brevity.

6 Automated Verification

An important goal of PROFACTORY is to achieve correctness and security by construction. The goal is validated by automated verification, which includes the following three aspects: verifying *concurrency control correctness using VCC [2, 52]*, *memory safety using Frama-C [17]*, and *state-machine correctness using Z3 [42]*. Note that bugs in any of these aspects could lead to security exploits. We use different tools for the three aspects as they have different focuses. For example, VCC was designed to prove concurrency correctness and has limited support for type/pointer casting, whereas Frama-C was designed to prove memory safety and can hardly reason about concurrent program behaviors. Z3 is a general reasoning engine, and hence very suitable for reasoning about high-level state machine behaviors. The inputs to the first two are C-like functions that can be automatically emitted by PROFACTORY. However, as in most verification systems, the proof process may require developers' manual intervention, e.g., providing a small number of additional pre-conditions

Algorithm 1: Code Generation Algorithm

```
Def. : H,P,M,R,D - sets of defined headers, defined parameters,
       defined messages, defined receiving transitions and defined
       sending transitions
     : structGen - generate data structures
     : localGen - generate local pointers
     : extractGen - refer pointers to socket buffer data
     : checkGen - generate sanity check block
     : allocGen - generate socket buffer allocation block
     : baseMsg - get message base of a layered message
     : timerGen - generate timer updating block
     : sendGen - generate message delivery block
     : codeGen - generate type-specific block
1  foreach h ∈ H do
2  |  structGen(h)                                  ▷ packed data structure
3  end
4  foreach p ∈ P do
5  |  structGen(p)                                  ▷ packed data structure
6  end
7  structGen(ch)
8  structGen(cn)                                   ▷ end of header file generation
9  foreach p ∈ ch do
10 |  codeGen(p)                                    ▷ generate findChanByPara
11 end
12 codeGen(ch)                                     ▷ generate createChan
13 codeGen(cn)                                     ▷ generate createConn
14 foreach m = h(f = n1 ? : m1)... ∈ M do
15 |  codeGen(m) = define parse_m(ch,cn,skb_in){
16 |  localGen(h)
17 |  extractGen(h)
18 |  checkGen(h)                                  ▷ failure is directed to drop
19 |  if(f == n1) return parse_m1(ch,cn,[h],skb_in);
20 |  ...
21 |  else goto drop;
22 |  drop: kfree_skb(skb_in); return error;}
23 |  + define compose_m(ch,cn,skb_out){
24 |  compose_m1(ch,cn,skb_out);...}               ▷ details omitted
25 end
26 foreach r ∈ R do
27 |  codeGen(r) = define
   |    parse_baseMsg(m_in)(ch,cn,h_prev,skb_in){
28 |  localGen(baseMsg(m_in))
29 |  extractGen(baseMsg(m_in))
30 |  checkGen(baseMsg(m_in))                      ▷ failure is directed to drop
31 |  if(codeGen(e1) ∧ ch → state == s_f1){
32 |  allocGen(m_out)                              ▷ failure is directed to drop
33 |  localGen(m_out)
34 |  extractGen(m_out)
35 |  codeGen(S)
36 |  compose_m_out(ch,cn,skb_out);
37 |  (ch → state) ← s_t1
38 |  timerGen(s_t1)
39 |  sendGen(m_out)}
40 |  ...
41 |  else goto drop;
42 |  return success;
43 |  drop: kfree_skb(skb_in); kfree_skb(skb_out); return error;}
44 end
45 foreach d ∈ D do
46 |  codeGen(d) = define send_m_out(ch,cn,data,len){
47 |  allocGen(m_out)                              ▷ failure is directed to drop
48 |  localGen(m_out)
49 |  extractGen(m_out)
50 |  if(codeGen(e1) ∧ ch → state == s_f1){
51 |  codeGen(S)
52 |  (ch → state) ← s_t1
53 |  timerGen(s_t1)}
54 |  ...
55 |  else goto drop;
56 |  if(data ∧ len)memcpy(skb_put(skb_out,len),data,len);
57 |  compose_m_out(ch,cn,skb_out);
58 |  sendGen(m_out); return success;
59 |  drop: kfree_skb(skb_out); return error;}
60 end
```

and/or loop invariants. The input to Z3 is a set of symbolic constraints derived by interpreting the DSL specification (i.e., $\mathbb{P}$ in Figure 5).

Concurrency Control Correctness. In generated code, concurrency control takes place in connection multiplexing operations, where multiple channels share common information. The verification aims to ensure accesses to such common information do not cause races or deadlocks. During code generation, PROFACTORY also emits code that is amenable for VCC. Particularly, it makes the code self-contained by providing mock data structures and API functions, and replaces mutex operations with VCC-specific lock acquisition and release. With the annotations of the shared data structures, VCC automatically determines concurrency correctness. Figure 7 illustrates an example of this procedure. The code snippet in (a) presents a function generated by PROFACTORY that updates the `mtu` field of a channel indexed by a channel id `cid`. Observe that a lock is acquired at Line 4 before accessing the list of channels and then released at Line 10. The channel is further locked at Line 12 and then released at Line 14 after updating the `mtu` field. The snippet in (b) shows the corresponding version for VCC verification with those in green being VCC-specific keywords declaring shared objects and auxiliary objects. The tags show the correspondences of the mutex operations in (a) and (b). VCC then proves that the code in (b) is race-free and deadlock-free. Details of VCC are not the focus of our work and hence elided. Interested readers are referred to [2].

Memory Safety. Frama-C verification requires per-function code annotations written in its own ACSL specification language [17]. Most such annotations (e.g., pointer validation, memory span validation, memory span separation, loop variant and loop invariant) can be generated by PROFACTORY, but due to the difficulty in automatically producing the complete set of annotations, developers may need to manually insert additional (very limited) preconditions and/or loop invariants to assist the verification process. All the kernel data structures (e.g., `sk_buff`) and their operations are manually pre-simplified (one-time effort) as they are not supported. The verification excludes memory access vulnerabilities, i.e., buffer overflow, invalid pointer dereference, memory leakage, use after free and double free. In particular, being free from buffer overflow and invalid pointer dereference are deductively verified, where intermediate targets such as being free of infinite loop, integer overflow/underflow and dividing by zero are also guaranteed, while Frama-C guarantees being free of memory leakage, use after free and double free by tracking memory allocation/deallocation operations. Figure 8 showcases a generated function with Frama-C annotations. The function iterates a parameter list stored in `skb_in`, where the lines highlighted in green are the annotations. It was successfully verified by Frama-C. Specifically, the annotations consists of two parts, *function annotations* (Line 21–28) and *loop annotations* (Line 34–44). The former de-

```
01 static int parse_config(int cid, int mtu, ...,
02 struct conn* pconn) {
03   struct chan* pchan = NULL;
04   mutex_lock(conn->list_lock);   L1
05   pchan = pconn->chan_list;
06   while(pchan){
07     if(pchan->cid == cid) break;
08     pchan = pchan->next;
09   }
10   mutex_unlock(pconn->list_lock);   R1
11   if(!pchan) return CHAN_NOT_EXIST;
12   mutex_lock(pchan->lock);
13   pchan->mtu = mtu;   L2
14   mutex_unlock(pchan->lock);   R2
15   return SUCCESS;
16 }
```

(a)

```
17 static int parse_config(int cid, int mtu, ... _(ghost \claim c))
18   _(always c, (&ChanDataListContainer)->\closed)// channel
19 list must not be claimed
20   _(requires cid > 0)
21 {
22   CHAN_DATA* pchan = NULL;
23   _(assume \thread_local(pchan))// thread-local assumption
24   Acquire(&ChanDataListLock _(ghost c));// claim list lock
25     _(unwrapping &ConnData.ChanDataList)// enable access
26     _(writes pchan)   L1
27     {
28       pchan = ConnData.ChanDataList;
29       while(pchan) {
30         if(pchan->cid == cid) break;
31         pchan = pchan->next;
32       }
33 } ...   R1
34   Release(&ChanDataListLock _(ghost c));// release list lock
35   _(ghost \claim d = \make_claim({&ChanDataContainer},
36 (&ChanDataContainer)->\closed);)
37   Acquire(&ChanDataLock _(ghost d));// claim channel lock
38   _(unwrapping &ChanData)// enable access
39   _(writes \span(pchan))   L2
40   {
41     pchan->mtu = mtu;
42   }   R2
43   Release(&ChanDataLock _(ghost d));// release channel lock
44 ...}
```

(b)

Figure 7: Concurrency correctness for updating a channel

```
01 #define H_CONF_SIZE 8
02 #define P_OPT_SIZE 4
03 #define PL_MAX_CONF_SIZE 8
04 struct sk_buff {
05   int len;
06   char* data;
07   ...
08 };
09 struct conf {
10   int conf_type;
11   unsigned int len;
12 }__attribute__((packed));
13 struct opt {
14   unsigned int optVal;
15 }__attribute__((packed));
16 struct chan {
17   int mtu;
18   int fcs;
19   ...
20 };
21 /*@ requires ArgReq: \valid(pchan) &&
22 \valid(skb_in);
23   @ requires SkbReq: skb_in->len >= 0 && \
24 valid(skb_in->data + (0..skb_in->len));
25   @ requires SeparationReq: \
26 separated((char*)pchan + (0..sizeof(struct chan)),
27 (char*)skb_in + (0..sizeof(struct sk_buff)), skb_in-
28 >data + (0..skb_in->len));*/
29 static int parse_config(struct chan* pchan, struct
30 sk_buff* skb_in) {
31   int iter_cnt = 0;
32   struct conf* conf_hdr = 0;
33   struct opt* opt_para = 0;
34   /*@ loop invariant \valid(skb_in);
35     @ loop invariant \valid(pchan);
36     @ loop invariant skb_in->len >= 0;
37     @ loop invariant \valid(skb_in->data + (0..skb_in-
38 >len));
39     @ loop invariant iter_cnt <= PL_MAX_CONF_SIZE;
40     @ loop invariant \separated((char*)pchan +
41 (0..sizeof(struct chan)), (char*)skb_in +
42 (0..sizeof(struct sk_buff)), skb_req->data + (0..skb_in-
43 >len));
44     @ loop variant skb_in->len;*/
45   while(skb_in->len > H_CONF_SIZE && iter_cnt <
46 PL_MAX_CONF_SIZE) {
47     conf_hdr = (void*)skb_in->data;
48     skb_in->data += H_CONF_SIZE;
49     skb_in->len -= H_CONF_SIZE;
50     if(conf_hdr->len > skb_in->len || conf_hdr-
51 >len != P_OPT_SIZE) goto drop;
52     opt_para = (void*)skb_in->data
53     if(con_hdr->conf_type == 1) {
54       chan->mtu = opt_para->optVal;}
55     else if(con_hdr->conf_type == 2) {
56       chan->fcs = opt_para->optVal;}
57     ...
58     skb_in->len -= P_OPT_SIZE;
59     skb_req->data += P_OPT_SIZE;
60     iter_cnt++;
61   }
62   return 1;
63 drop:
64   ...
65   return 0;
66 }
```

Figure 8: Memory safety for iterating a parameter list

notes the function preconditions (which need to be verified at each callsite of the function). For instance, the execution of `l2cap_conf_parse` requires valid pointers (Line 21–22), valid socket buffer size (Line 23–24) and *separated argument spans* (Line 25–28), meaning they do not overlap. In contrast, loop annotations assist proving loop termination and iterative memory access safety. It usually includes a loop variant that changes across iterations and hence is related to loop termination (e.g., Line 44, length of remaining data in the socket buffer), and a list of loop invariants that specify predicates that must hold across iterations and substantially facilitate the proof procedure (e.g., Line 34–43). Internally, the invariants are auxiliary lemmas which help the proof and must also be deductively verified from function annotations. More details about Frama-C verification can be found in [17].

State Machine Verification by Symbolic Model Checking. To verify state machine correctness, we translate the DSL specification to symbolic constraints and check if the model satisfies a number of general properties. PROFACTORY rewrites DSL specification to the SMT-LIB [19] representa-

tion of Z3, a well-known theorem prover. Z3 supports reasoning of constraints in a large number of theories such as integer, string, array, and bit-vector theories. Specifically, symbolic variables are introduced to describe attributes of abstract data types, such as value of a field f, denoted as $f.val$ in $\mathbb{P}$ of Rule 1 in Figure 5, and variables such as channel state. Operations in $\mathbb{P}$ such as additions and multiplications are translated to Z3 operations in the corresponding theories. Logical operations, such as conjunctions, disjunctions, and implications (e.g., those in $\mathbb{P}$ of Rule 6 in Figure 5 describing the different possible state transitions guarded by different conditions) are directly supported by Z3. General statements such as assignments (whose semantics are standard and elided from Figure 5) are also translated to symbolic constraints. The translation of these statements is standard [29] and elided. Loops are unrolled with the unroll bound of 10, which is practically sufficient for the protocols we model. Note that while at runtime there are multiple channels, we do not have to model these instances during symbolic model checking as we are interested in state transition properties. Variables that can be defined by the user-space, kernel, and remote requests are largely free variables. That is, they are only constrained by range specifications if there are any. We say a property is satisfied (SAT) if Z3 can find a value assignment to all these free variables that can satisfy the property. We validate a number of general properties of state machine behaviors.

State Reachability The first property asserts that a destination state s_1 is reachable from the initial state s_0, denoted as $reachable(s_0, s_1)$. Let the symbolic encoding of the protocol be M, which includes the symbolic constraint encodings of all the protocol specifications and statements, and the state variable be $state$. We use $reachableInOne(s_0, s_1)$ to denote that s_1 can be reached from s_0 by one step transition. It is hence defined as $M \wedge state = s_0 \rightarrow state_1 = s_1$. Note that we have to rename $state$ to $state_1$ to denote the new state. Intuitively, it is SAT if Z3 can find a valuation to free variables (e.g., a message) that induces the state to change from s_0 to s_1 in one step. Constraint $reachableInTwo(s_0, s_1)$ is defined as $reachableInOne(s_0, s_k) \wedge reachableInOne(s_k, s_1)$. Note that the M encodings in $reachableInOne(s_0, s_k)$ and $reachableInOne(s_k, s_1)$ need to be renamed as well since they need to be resolved independently (representing different messages). Therefore, $reachable(s_0, s_1)$ is defined as follows.

$$reachableInOne(s_0, s_1) \vee reachableInTwo(s_0, s_1) \vee ...$$

Currently, our reasoning is bounded at 15 steps, that is, the maximum transition path has 15 steps.

Transition Coverage This property dictates that any transition defined in the protocol is feasible, meaning that it can be triggered by some message sequence(s). Assume the condition guarding a transition from s_1 to s_2 is e, we assert $reachable(s_0, s_1) \wedge e$. Intuitively, we assert that s_1 is reachable from the initial state and e is satisfiable.

Absence of Transition Conflict This property states that if a message can trigger two or more transitions, there are not two of them satisfiable simultaneously. Assume s can lead to s_1, s_2, ..., s_k, guarded by e_1, e_2, ..., e_k, respectively. For any $i, j \in [1, k]$ and $i \neq j$, we assert $reachable(s_0, s) \wedge e_i \wedge e_j$. Any SAT result indicates the protocol is buggy. If all are UNSAT, the property holds.

Race-free Message Sends This is the property illustrated in Section 2. When two peers are both in some state that is expected to send out a message, the protocol may be trapped into an asynchronous sending race that may lead to message loss. Suppose we have a state s_1^a for device A and a state s_1^b for device B and they have transitions to states s_2^a and s_2^b respectively, which are both triggered by a message send event, with A sending m_a and B sending m_b. As both devices are in a sending race, A may stay at s_1^a or reach s_2^a when m_b arrives. Correspondingly, B may stay at s_1^b or reach s_2^b when m_a arrives. Therefore, message loss may happen when (1) s_1^a or s_2^a does not accept m_b, or (2) s_1^b or s_2^b does not accept m_a, since the message m_a or m_b can be dropped. Validating this property requires coupling the reasonings of both sides of a connection. In the following, we define a constraint $coReachInOne(s_1^a, s_1^b, s_2^a, s_2^b)$ that states that by exchanging a message, device A can reach s_2^a (from s_1^a) and device B can reach s_2^b (from s_1^b).

$$(M_a \wedge s_1^a \rightarrow s_2^a) \wedge (M_b \wedge s_1^b \rightarrow s_2^b) \wedge (\overleftarrow{m}_a = \overrightarrow{m}_b \vee \overleftarrow{m}_b = \overrightarrow{m}_a)$$

The first two clauses assert there are messages that induce the state transitions and the last asserts that the incoming message at A must be the outgoing message at B or vice-versa. Note that the messages (e.g., $\overleftarrow{m}_a$ and $\overrightarrow{m}_a$) are essentially a subset of symbolic variables in the model M_a. They are instantiated when Z3 resolves M_a. We can define $coReach(s_1^a, s_1^b, s_2^a, s_2^b)$ to dictate that starting from s_1^a and s_1^b, the two devices can reach s_2^a and s_2^b, respectively, after exchanging a sequence of messages, in a way similar to defining $reachable()$ from $reachableInOne()$. For all state pairs s_1^a and s_1^b that can both send messages, guarded by conditions e_a and e_b. We assert $coReach(s_0^a, s_0^b, s_1^a, s_1^b) \wedge e_a \wedge e_b$. If none is SAT, the property holds. Otherwise, it is vulnerable.

Deadlock-free Message Receives The property states that at any time when two peers are both expecting an incoming message at some state, the protocol may get stuck in a receiving deadlock. Therefore, given a pair of states s_1^a and s_1^b at the two peers, respectively, we assert the following.

$$coReach(s_0^a, s_0^b, s_1^a, s_1^b) \rightarrow ((M_a \wedge s_1^a \rightarrow s_2^a \wedge \overrightarrow{m}_a \neq nil) \vee (M_b \wedge s_1^b \rightarrow s_2^b \wedge \overrightarrow{m}_b \neq nil))$$

The antecedent is the reachability of the two states. The consequent is to say either one can move forward with a message send, meaning that the peer does not have to wait for an incoming message. Any pair that yields UNSAT indicates a deadlock problem.

Consistent Security Level In Linux each protocol maintains a security level variable that varies during the lifetime of a connection, depending on the states of authentication and encryption. PROFACTORY supports the mechanism although

we do not model it in the DSL syntax/semantics for brevity. Given a state s, we use $sec(s)$ to denote the security level at the state. The security consistency property dictates that if a state can be reached through different message sequences, it must have the same security level. We assert the following.

$$reachable(s_0, s) \rightarrow sec(s) \ \wedge \ reachable(s_0', s') \rightarrow sec(s') \\ \wedge \ sec(s) \neq sec(s')$$

Here, s_0' and s' denote a renamed version of s_0 and s, respectively, indicating that they are considered different symbolic variables internally although they have the same meaning. This is to allow Z3 to resolve them independently. Any SAT result suggests an inconsistency problem. In Section 7, we would exhibit a violation of the property.

For the above discussion, one can observe that the model checking is performed in two modes: *isolated* and *coupled*. In the former, we only consider one side of the connection and assume messages from the other side can be anything, even corrupted intentionally by the adversary. In the latter, we reason both sides together and trust the connection to deliver messages properly such that messages on the two sides can be coupled (e.g., in the race-free and deadlock-free properties).

7 Evaluation

We implement a prototype of PROFACTORY in Haskell. We use it to customize 8 IoT protocols, including various Bluetooth (v5.0) protocols for Linux 4.10 kernel and Zigbee (v1.0) NWK layer for ZBOSS simulator [3]. Note that PROFACTORY can also be ported to other protocol stacks or systems if corresponding platform-dependent interfaces are provided. Currently, PROFACTORY does not fully support code generation for the Android kernel. Hence, for the evaluation purposes, we generate core communication components of Bluetooth protocols and manually adapt and integrate them into Android to obtain a customized Bluedroid (or Fluoride) [4].

According to original Bluetooth specifications, we write SDP (Service Discovery Protocol), PAN (Personal Area Network), BNEP, HIDP, RFCOMM and L2CAP in our DSL, and generate code for Linux. Note that BlueZ operates SDP and PAN in the user space but we generate kernel versions for them. Those implementations all pass the verification and they work properly when communicating with real devices (or simulator in the case of Zigbee). We customize RFCOMM and L2CAP, only focusing on the functionality of connection-oriented data delivery, and separating L2CAP classic and L2CAP BLE (Bluetooth Low Energy). Table 1 shows the lines of code in DSL, in the original implementation, and in the generated implementation for each protocol. Compared to the original implementation, the DSL specifications are much more succinct. Even the generated code is of smaller size.

7.1 System Performance

With the generated protocol implementation, we deploy two Raspberry Pi 3 devices, two desktop computers and two Android phones (Google Pixel 2) to measure performance for paired communication. Specifically, Raspberry Pi 3 devices and desktop computers load our customized L2CAP and RFCOMM, while we manually replace communication functions with our generated codes (adapted for Android) for L2CAP and RFCOMM in Bluedroid. We perform file transfer (object exchange, using RFCOMM and L2CAP) of a 20MB file for both of the original Bluetooth implementations (BlueZ and Bluedroid) and our customized ones. We repeat the experiment 10 times and collect the geometric mean of time costs in Figure 9. As illustrated, the customized implementation is about 4% less efficient. The efficiency loss in customization is mainly caused by the sanity checks enforced on fields, and the lack of data structure layout optimization in type-based code generation. Meanwhile, the memory footprints of the original bluetooth module are 536KB, 536KB and 439KB for desktop, Raspberry Pi and Phone, while the ones of the customized module are 533KB, 533KB and 438KB. The difference is negligible, and the customized module consumes slightly less because we trimmed unused components in customization.

Table 1: LoC comparison between original BlueZ implementations and codes generated by PROFACTORY

Protocol	Lines of Codes		
	Model Definition	Original	Generated
SDP	971	5500	3478
PAN	183	1023	635
BNEP	590	1447	1162
HIDP	578	1966	1580
RFCOMM	738	4547	2465
L2CAP-CLA	1148	10328	3419
L2CAP-BLE	1247		3868
Zigbee-NWK	782	2373	1471

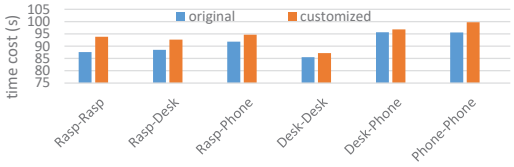

Figure 9: Comparison of time costs in paired file transfer

Zigbee Evaluation It is challenging to obtain devices with a programmable Zigbee stack. Therefore, we select the Zigbee simulator ZBOSS to conduct our evaluation. Note that this is a reasonable option because mainstream manufacturers are using ZBOSS to test Zigbee implementations before product shipment. In Zigbee NWK, we model the complete NLDE (Network Layer Data Entity) which is responsible for data delivery, consisting of request (for data sending), confirm (for confirming receipt) and indication (for updating link quality). In particular, we customize NLDE by removing the alias fields which are rarely used. For NLME (Network Layer Management Entity), we only model the message formats and generate the primary parsers, but all the payload processing is delegated to the original implementation in ZBOSS. NLME is hardware-specific, highly coupled with lots of hardware physical features and routing operations. Without introducing additional specifications to describe those semantics, PROFACTORY is not able to correctly express the whole procedure.

Message formats of Zigbee are flatter than that of Bluetooth and a header tailing with a variable-sized field is sufficient to depict all the messages. Because Zigbee establishes a connectionless ad-hoc network, it does not have multiplexing, while all the shared connection information is stored in the NIB data structure for concurrent access. Also, since it does not maintain connections, timer operations are excluded (router nodes require timers to repeatedly send out broadcast messages, but our evaluation only focuses on message delivery of end nodes). ZBOSS is not a kernel-oriented implementation and hence the generation of standard socket interfaces are excluded. ZBOSS maintains a buffer pool, where a message buffer is allocated by `ZB_BUFF_FROM_REF` and released by `zb_free_buf`. Correspondingly, kernel socket buffer operations are shifted to the two functions in code generation for Zigbee. Between layers, data are delegated through a ring buffer (kernel simply passes the socket buffer). Ring buffer operations are wrapped by ZBOSS, and we can directly generate code to invoke them. Note that different from kernel, the lower/upper layer in ZBOSS is not responsible to release the buffer, hence invoking `zb_free_buf` is always generated after writing to the ring buffer.

In evaluation, we create 5, 10, 15 and 20 nodes to measure the communication time costs. One node communicates with each of the other nodes to transfer 1MB application data. In addition to those nodes, we also create a forwarding node serving as a router to prevent noises caused by pairwise communication. Results are collected in Figure 10. As demonstrated, no significant overhead could be observed. Meanwhile, the memory footprints of the original ZBOSS are 16.9KB, 17.8KB, 19.5KB and 22.4KB, while the ones of the customized version are 16.9KB, 17.7KB, 19.7KB and 22.4KB. The difference is also negligible.

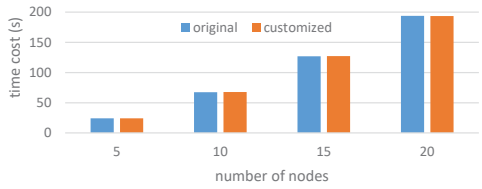

Figure 10: Comparison of time costs in Zigbee data transfer

7.2 Vulnerability Averting

Appendix B lists 81 (excluding the one in Section 2) recently-released CVEs that could have been averted if protocols had been defined and generated in ProFactory. Specifically, we searched for target CVEs by keyword "Bluetooth" (from 2017) and "Zigbee", and collected the ones related to field boundary checking and security downgrading. These represent the CVEs that we can find from public sources for the protocols we consider. We do not intend to claim the list is complete as there may be vulnerabilities that we are not aware of. The symbolic model checking does not disclose state-machine bugs in most of the protocols rewritten in our DSL. This is expected because these are widely-used protocols which are considered well specified and maintained. However, we do find a known state-machine vulnerability in PAN that could have been averted. Next, we showcase two of the 81 vulnerabilities (one for secure message parsing and one for state machine verification) in details.

```
/sdpd-request.c
722 sdp_buf_t *pCache = sdp_get_cached_rsp(cstate); ...
726 if (pCache) {
      if (pCache && cstate->cStateValue.maxBytesSent < pCache->data_size) // patch
727     short sent = MIN(max_rsp_size, pCache->data_size - cstate->cStateValue.maxBytesSent);
728     pResponse = pCache->data;
729     memcpy(buf->data, pResponse + cstate->cStateValue.maxBytesSent, sent);
```

Figure 11: Code and patch of CVE-2017-1000250

CVE-2017-1000250. This vulnerability [56] can cause information leaks. It resides in the SDP implementation of Linux BlueZ (version 5.46 or earlier). When BlueZ responds to an SDP request, the response message size may exceed the MTU of L2CAP and is dropped by L2CAP. Hence SDP must realize its own fragmentation mechanism. Specifically, a sender peer marks a field to notify a receiver peer that the current packet has continuous fragments. Accordingly, the receiver peer responds with an offset denoting the bytes that have been received so far. Then, the sender peer continues sending the next fragment from the exact offset. Figure 11 illustrates the buggy code and its original patch [7]. As demonstrated, before patching, the value of offset `maxBytesSent` is not checked and `sent` can be overflowed. Therefore, out-of-bound bytes can be copied to `buf` → `data` and sent to the remote peer. Note that automatically fixing the bug on the existing implementation (i.e., generating the illustrated patch) is extremely difficult without developers' intervention as it is very hard for an automatic analysis to infer the needed data-flow relations. In contrast, since we automatically generate secure packet fragmentation/assembly code all together and perform code verification with Frama-C, the vulnerability is avoided in the first place.

CVE-2017-0783. This vulnerability [5] allows a man-in-the-middle attack and it resides in Android/Windows's PAN implementation. PAN offers the service of network proxy via Bluetooth devices. In the protocol, a device can serve as any of the three roles, GN (Group Ad-hoc Network), NAP (Network Access Point) and PANU (PAN User). Among them, PANU acts as a network client user and GN/NAP represents a proxy/router/bridge. When a PANU device connects to a PANU/GN/NAP device, neither peer checks the security level. In contrast, when a GN/NAP device connects to a PANU device, the PANU device must perform the security check because an unauthorized GN/NAP device can redirect connections to malicious targets. In the vulnerable Bluedroid implementation [6], a PANU device is allowed to act as a GN/NAP device after it connects to a PANU device, bypassing the security check and performing reverse tethering. Figure 12 illustrates the problematic state machine, where a PAN device can send/receive wrapped network messages at `PAN_ready`

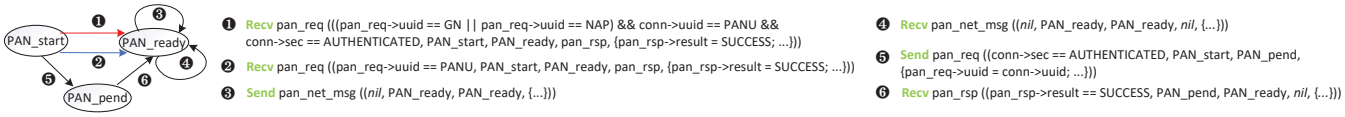

Figure 12: PAN state machine of CVE-2017-0783

state. The security issue lies in that a device has two transition paths (❶ and ❷) from `PAN_start` to `PAN_ready` but they have inconsistent security levels (❶ requires `AUTHENTICATED` but ❷ does not). The bug is detected when we model check the *consistent security level* property. Note that we integrate the `uuid` which is exchanged in ATT (Attribute Protocol) into PAN and omit the failure processing branches for PAN messages to simplify our modeling and demonstration. The official patch [6] roughly prohibits any connection to/from remote GN/NAP devices when an Android device performs as PANU, while splitting the `PAN_ready` state to deal with ❶ and ❷ separately may be a better solution.

8 Discussion

Flow Control. ProFactory currently does not support modeling flow control algorithms. Specifying flow-control is challenging due to the lack of regular design of the various flow-control algorithms. However in the context of IoT, due to resource constraints, existing protocols rarely have complex flow control. In fact, early BlueZ implementations did not have it at all. Regularity may be abstracted out of popular light-weight flow control algorithms and modeled in our DSL. We will leave it to our future work.

Specification Flaws. The working of ProFactory largely relies on the robustness of the specification (or DSL). If any part of the specification was further found flawed (e.g., a protocol model specified by ProFactory cannot be securely handled in the generated implementation), the claimed security guarantees should be degraded.

Security Properties. Cryptographic constructs are not covered in current specification set, but all the cipher operations are delegated to a pre-established lower layer. This leads to the lack of security guarantees for cryptographic properties (e.g., authentication and secrecy) in ProFactory. The future work of integrating existing cryptographic modeling/verification tools into ProFactory can bridge the gap.

Firmware Deployment. We foresee two modes of deployment in IoT firmware. The first one is in-production customization, in which the manufacturers make use of ProFactory to generate secure and correct implementation and customize their networking dialects before shipping the products. The second is post-deployment customization. Through the update interface of a firmware, hardened and customized networking code can be uploaded to deployed products.

Platform Dependence. As discussed in Section 5, the code generation is heavily dependent on the underlying platforms because they can have diverse underlying protocol implementation primitives. This degrades the portability of ProFactory. Nevertheless, if we target code generation for a particular kernel, this seems to be an inevitable issue. Adding virtualization layers may potentially mitigate the problem.

Semantic Preservation. Currently, semantic-preservation correctness of ProFactory code translation has not been comprehensively verified. This will be part of our future work for compiler verification. However, the post-modeling verification still offers security guarantees.

9 Related Work

Protocol Modeling Lots of tools towards secure parser generation have been proposed in recent years [12,22,43,45,55,57]. In EverParse [55] researchers devise a compiler transforming tag-length-value message formats to low-level F* code that calls a library of parser combinators which are formally verified in F*. In this way, the security of parsers is guaranteed and it proves to be effective in averting existing TLS vulnerabilities. In [45] a USB-specific message DSL is proposed to emit a hardening suite, which is then be integrated into a production kernel to avert USB parsing vulnerabilities. PADS [43], Spicy [57], Hammer [12] and Nail [22] are all message formalization tools that produce robust parsers from customized message specifications, covering common text or binary protocols across different languages. In particular, PADS is more data-oriented, which offers auxiliary tools to convert data in XML and XQuery formats to formalized specifications. However, as aforementioned, those tools largely focus on messages, and some of them are not able to describe non-context-free formats, we hence develop ProFactory to realize comprehensive customization for low-level protocols.

Protocol Verification Verifying implementation correctness is a persistent research effort in protocol security area [24–28,30,38,44,51,53,59]. In ProVerif [30] security protocols are represented by Horn clauses to prove (strong) secrecy, authentication and process equivalence. It is applied in [26] to verify the security of symbolic TLS1.3 models. Tamarin [51] specifies protocol actions taken by agents in different roles, using an expressive DSL based on multiset rewriting rules, to automatically construct proofs for security protocols. It is applied in [24, 38] to perform formal analysis of 5G AKA protocols. AGVI [59] applies iterative deepening to perform cost-constraint searching in order to generate a near-optimal security protocol, with the lowest cost, satisfying all the security requirements that are encoded in a DSL. The authors of [25] resolve the TLS composite state machine issue (unexpectedly accepting invalid handshakes due to state machine fusion) by writing a secure TLS implementation that is verified by Frama-C. In [44] researchers leverage adversarial

testing technique to disclose an authentication vulnerability in 4G LTE. Those techniques are targeting security protocols that are considered orthogonal and complementary to PROFACTORY. In [27, 28, 53] various components of TCP implementation are verified through different symbolic modeling techniques, however, PROFACTORY aims to resolve protocol security issues at the beginning, generating secure protocol implementations.

Protocol Reverse Engineering and Fuzzing Reverse-engineering protocol specifications from network traces and/or program execution has been well studied in the past decade [31, 33, 36, 39–41, 46–48, 50, 54, 62, 63, 65]. In Discoverer [39] and ScriptGen [48], protocol communication pattern is heuristically learned to infer message formats. Hence, in [31, 47], researchers improve the learning by performing message clustering based on protocol context and semantic information. In [33, 41, 50, 63], researchers extract message formats in a different direction, leveraging dynamic tainting to monitor how a message is processed on the receiver side. In Prospex [36], apart from message formats, an approximate but meaningful state machine can also be extracted based on an augmented prefix tree acceptor. These efforts are complementary to PROFACTORY as the reverse engineered protocol specification can be formalized with our DSL. Protocol fuzzing [23, 34, 58, 60] mutates network messages and network states to disclose vulnerabilities in protocols. They often require protocol specifications to operate. Our DSL provides a way to formulate such specifications.

10 Conclusion

We propose PROFACTORY, in which a protocol could be modeled, checked and securely generated, averting common vulnerabilities residing in protocol implementations. We leverage PROFACTORY to generate Bluetooth and Zigbee protocols and the evaluation shows that PROFACTORY can help to avert 82 known CVEs.

Acknowledgments

This research was supported, in part by NSF 1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947, DARPA HR001119S0089-AMP-FP-034, and ERC H2020 grant 850868. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] https://github.com/JacobFeiWang/USENIX22_ProFactory.

[2] VCC: A Verifier for Concurrent C. https://bit.ly/2m4fCHt, 2008.

[3] Zigbee. https://bit.ly/3lGtrp1, 2011.

[4] Bluedroid. https://bit.ly/2JUPydq, 2015.

[5] CVE-2017-0783. https://bit.ly/2UmQ3Rn, 2017.

[6] CVE-2017-0783 Patch. https://bit.ly/2YH8Shr, 2017.

[7] CVE-2017-1000250 Patch. https://bit.ly/2FR7WzN, 2017.

[8] CVE-2017-1000251. https://bit.ly/2xymD8a, 2017.

[9] CVE-2017-1000251 Patch. https://bit.ly/2WKLkqw, 2017.

[10] Forbes Roundup of IoT Market. https://goo.gl/UPpmkn, 2018.

[11] Statista IoT. https://goo.gl/mqt3T7, 2018.

[12] Hammer. https://bit.ly/3s3bMfI, 2019.

[13] SIG Bluetooth Specifications. https://bit.ly/2Vw9Y1Q, 2019.

[14] Zigbee Market. https://bit.ly/3kdNmvr, 2019.

[15] Bluetooth Core Specification. https://bit.ly/2YwR4VD, 2020.

[16] Bluetooth Market Report. https://bit.ly/3fRXkAv, 2020.

[17] Frama-C. https://frama-c.com, 2021.

[18] Protobuf. https://github.com/protocolbuffers, 2021.

[19] SMT-LIB. http://smtlib.cs.uiowa.edu/, 2021.

[20] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, 2007.

[21] Wahhab Albazrqaoe, Jun Huang, and Guoliang Xing. Practical bluetooth traffic sniffing: Systems and privacy implications. In *Proc. of MobiSys '16*, 2016.

[22] Julian Bangert and Nickolai Zeldovich. Nail: A practical tool for parsing and generating data formats. In *Proc. of OSDI '14*, 2014.

[23] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. Snooze: Toward a stateful network protocol fuzzer. In *Proc. of ISC '06*, 2006.

[24] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In *Proc. of CCS '18*, 2018.

[25] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. In *Proc. of IEEE S&P '15*, 2015.

[26] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *Proc. of IEEE S&P '17*, 2017.

[27] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. In *Proc. of SIGCOMM '05*, 2005.

[28] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: Hol specification and symbolic-evaluation testing for tcp implementations. In *Proc. of POPL '06*, 2006.

[29] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph Wintersteiger. Programming z3. https://stanford.io/2GS004D, 2017.

[30] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, October 2016.

[31] Georges Bossert, Frédéric Guihéry, and Guillaume Hiet. Towards automated protocol reverse engineering using semantic information. In *Proc. AsiaCCS '14*, 2014.

[32] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proc. of USENIX Security '07*, 2007.

[33] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proc. of CCS '07*, 2007.

[34] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Proc. of NDSS '18*, 2018.

[35] Jiongyi Chen, Chaoshun Zuo, Wenrui Diao, Shuaike Dong, Qingchuan Zhao, Menghan Sun, Zhiqiang Lin, Yinqian Zhang, and Kehuan Zhang. Your iots are (not) mine: On the remote binding between iot devices and users. In *Proc. of DSN '19*.

[36] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *Proc. of IEEE S&P '09*, 2009.

[37] J. A. Crain and S. Bratus. Bolt-on security extensions for industrial control system protocols: A case study of dnp3 sav5. *IEEE Security Privacy*, 13(3):74–79, 2015.

[38] Cas Cremers and Martin Dehnel-Wild. Component-based formal analysis of 5g-aka: Channel assumptions and session confusion. In *Proc. of NDSS '19*, 2019.

[39] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proc. of USENIX Security '07*, 2007.

[40] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy H. Katz. Protocol-independent adaptive replay of application dialog. In *Proc. of NDSS '06*, 2006.

[41] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proc. of CCS '08*, 2008.

[42] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[43] Kathleen Fisher and David Walker. The pads project: An overview. In *Proc. of ICDT '11*, 2011.

[44] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. Lteinspector: A systematic approach for adversarial testing of 4g LTE. In *Proc. of NDSS '18*, 2018.

[45] Peter C. Johnson, Sergey Bratus, and Sean W. Smith. Protecting against malicious bits on the wire: Automatically generating a usb protocol parser for a production kernel. In *Proc. of ACSAC '17*, 2017.

[46] Tammo Krueger, Hugo Gascon, Nicole Krämer, and Konrad Rieck. Learning stateful models for network honeypots. In *Proc. of AISec '12*, 2012.

[47] Tammo Krueger, Nicole Krämer, and Konrad Rieck. Asap: Automatic semantics-aware analysis of network payloads. In *Proc. of PSDML '10*, 2010.

[48] Corrado Leita, Ken Mermoud, and Marc Dacier. Scriptgen: An automated script generation tool for honeyd. In *Proc. of ACSAC '05*.

[49] Amit A. Levy, James Hong, Laurynas Riliskis, Philip Levis, and Keith Winstein. Beetle: Flexible communication for bluetooth low energy. In *Proc. of MobiSys '16*, 2016.

[50] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proc. of NDSS '08*, 2008.

[51] Simon Meier, Benedikt Schmidt, C. Cremers, and D. Basin. The tamarin prover for the symbolic analysis of security protocols. In *Proc. of CAV '13*, 2013.

[52] Michal Moskal, Wolfram Schulte, Ernie Cohen, and Stephan Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-2019, 2009.

[53] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proc. of NSDI '04*, 2004.

[54] James Newsome, David Brumley, Jason Franklin, and Dawn Song. Replayer: Automatic protocol replay by binary analysis. In *Proc. of CCS '06*, 2006.

[55] Tahina Ramananandro, Antoine Delignat-Lavaud, Cedric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. Everparse: Verified secure zero-copy parsers for authenticated message formats. In *Proc. of USENIX Security '19*, 2019.

[56] Ben Seri and Gregory Vishnepolsky. BlueBorne. https://armis.com/blueborne/, 2017.

[57] Robin Sommer, Johanna Amann, and Seth Hall. Spicy: A unified deep packet inspection framework for safely dissecting all your data. In *Proc. of ACSAC '16*, 2016.

[58] Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *Proc. of CCS '16*, 2016.

[59] Dawn Song, Adrian Perrig, and Doantam Phan. Agvi — automatic generation, verification, and implementation of security protocols. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proc. of CAV '01"*, 2001.

[60] Octavian Udrea and Cristian Lumezanu. Rule-based static analysis of network protocol implementations. In *Proc. of USENIX Security '06*, 2006.

[61] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *Proc. of CCS '17*, 2017.

[62] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proc. of ESORICS '09*, 2009.

[63] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Krügel, and Engin Kirda. Automatic network protocol analysis. In *Proc. of NDSS '08*, 2008.

[64] Fenghao Xu, Wenrui Diao, Zhou Li, Jiongyi Chen, and Kehuan Zhang. Badbluetooth: Breaking android security mechanisms via malicious bluetooth peripherals. In *Proc. of NDSS '19*, 2019.

[65] Yapeng Ye, Zhuo Zhang, Fei Wang, Xiangyu Zhang, and Dongyan Xu. Netplier: Probabilistic network protocol reverse engineering from message traces. In *Proc. of NDSS '21*, 2021.

A Platform-dependent Interface Example

Figure 13 illustrates the platform-dependent interfaces of L2CAP, with the user space on the left, the lower layer protocol (HCI) on the right, the box with green header in the center generated from the DSL and the boxes with headers of other colors the interfaces. Specifically, module initialization (the orange box in the middle of the second column) registers the protocol structure (the top orange box in the third column) and L2CAP socket operations that are invoked by user space (the bottom orange box). The registered socket operations need to invoke the functions generated from DSL (in the center) to fulfill message parsing and state transition. Inside those functions, common operations of Bluetooth family, kernel socket, kernel data structure, and kernel memory are invoked (through the interfaces in the red boxes). In particular, L2CAP needs to invoke these interfaces to accomplish data delivery and connection establishment. The lower layer protocol HCI (Host Controller Interface) invokes a callback *l2cap_recv_acldata* to inform L2CAP upon receiving a message. L2CAP invokes a callback *l2cap_data_recv* after unwrapping the raw data.

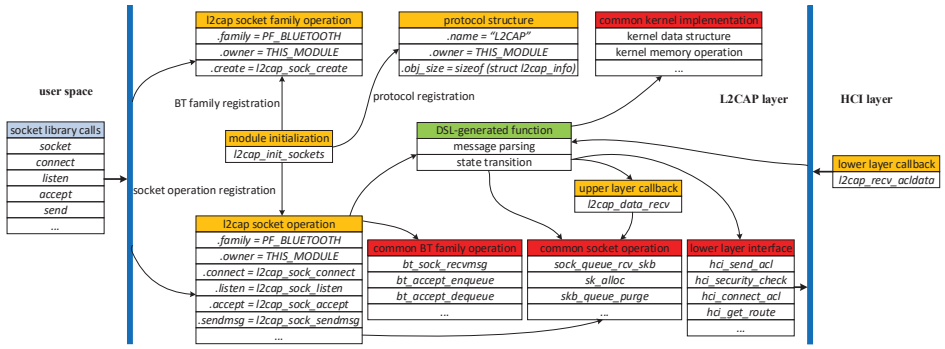

Figure 13: The platform-dependent interfaces of L2CAP

B CVEs Averted By PROFACTORY

Table 2 summarizes 82 recently-disclosed IoT vulnerabilities (including Bluetooth and Zigbee), 81 of which could have been averted if the corresponding protocols were modeled and generated from PROFACTORY. When PoCs are available, we checked if they can attack our generated code. Otherwise, we manually checked the CVE patches and ensure the triggering conditions are precluded in generated code. The avertable vulnerabilities are caused by either the lack of message boundary checks or the possible transition paths leading to security downgrading (i.e., inconsistent security levels). Further, we also find a representative vulnerability example that PROFACTORY cannot avert, cracking the protocol session key. Since current version of PROFACTORY cannot model cryptographical interactions, any vulnerability residing in the key negotiation procedure of a protocol cannot be averted.

Table 2: Averting IoT vulnerabilities

CVE No.	Protocol	Type	Avertable	CVE No.	Protocol	Type	Avertable
CVE-2020-0022	Bluetooth	missing boundary check	✓	CVE-2019-9474	Bluetooth	missing boundary check	✓
CVE-2019-9473	Bluetooth	missing boundary check	✓	CVE-2019-9462	Bluetooth	missing boundary check	✓
CVE-2019-9435	Bluetooth	missing boundary check	✓	CVE-2019-9434	Bluetooth	missing boundary check	✓
CVE-2019-9426	Bluetooth	missing boundary check	✓	CVE-2019-9425	Bluetooth	missing boundary check	✓
CVE-2019-9422	Bluetooth	missing boundary check	✓	CVE-2019-9419	Bluetooth	missing boundary check	✓
CVE-2019-9417	Bluetooth	missing boundary check	✓	CVE-2019-9413	Bluetooth	missing boundary check	✓
CVE-2019-9404	Bluetooth	missing boundary check	✓	CVE-2019-9402	Bluetooth	missing boundary check	✓
CVE-2019-9401	Bluetooth	missing boundary check	✓	CVE-2019-9398	Bluetooth	missing boundary check	✓
CVE-2019-9397	Bluetooth	missing boundary check	✓	CVE-2019-9396	Bluetooth	missing boundary check	✓
CVE-2019-9395	Bluetooth	missing boundary check	✓	CVE-2019-9394	Bluetooth	missing boundary check	✓
CVE-2019-9393	Bluetooth	missing boundary check	✓	CVE-2019-9390	Bluetooth	missing boundary check	✓
CVE-2019-9389	Bluetooth	missing boundary check	✓	CVE-2019-9388	Bluetooth	missing boundary check	✓
CVE-2019-9387	Bluetooth	missing boundary check	✓	CVE-2019-9368	Bluetooth	missing boundary check	✓
CVE-2019-9367	Bluetooth	missing boundary check	✓	CVE-2019-9363	Bluetooth	missing boundary check	✓
CVE-2019-9355	Bluetooth	missing boundary check	✓	CVE-2019-9353	Bluetooth	missing boundary check	✓
CVE-2019-9343	Bluetooth	missing boundary check	✓	CVE-2019-9342	Bluetooth	missing boundary check	✓
CVE-2019-9341	Bluetooth	missing boundary check	✓	CVE-2019-9333	Bluetooth	missing boundary check	✓
CVE-2019-9332	Bluetooth	missing boundary check	✓	CVE-2019-9331	Bluetooth	missing boundary check	✓
CVE-2019-9330	Bluetooth	missing boundary check	✓	CVE-2019-9328	Bluetooth	missing boundary check	✓
CVE-2019-9327	Bluetooth	missing boundary check	✓	CVE-2019-9326	Bluetooth	missing boundary check	✓
CVE-2019-9312	Bluetooth	missing boundary check	✓	CVE-2019-9289	Bluetooth	missing boundary check	✓
CVE-2019-9287	Bluetooth	missing boundary check	✓	CVE-2019-9286	Bluetooth	missing boundary check	✓
CVE-2019-9285	Bluetooth	missing boundary check	✓	CVE-2019-9284	Bluetooth	missing boundary check	✓
CVE-2019-9265	Bluetooth	missing boundary check	✓	CVE-2019-9260	Bluetooth	missing boundary check	✓
CVE-2019-9250	Bluetooth	missing boundary check	✓	CVE-2019-9249	Bluetooth	missing boundary check	✓
CVE-2019-9241	Bluetooth	missing boundary check	✓	CVE-2019-9237	Bluetooth	missing boundary check	✓
CVE-2019-2009	Bluetooth	missing boundary check	✓	CVE-2019-1996	Bluetooth	missing boundary check	✓
CVE-2018-9588	Bluetooth	missing boundary check	✓	CVE-2018-9583	Bluetooth	missing boundary check	✓
CVE-2018-9566	Bluetooth	missing boundary check	✓	CVE-2018-9560	Bluetooth	missing boundary check	✓
CVE-2018-9555	Bluetooth	missing boundary check	✓	CVE-2018-9544	Bluetooth	missing boundary check	✓
CVE-2018-9541	Bluetooth	missing boundary check	✓	CVE-2018-9540	Bluetooth	missing boundary check	✓
CVE-2018-9510	Bluetooth	missing boundary check	✓	CVE-2018-9509	Bluetooth	missing boundary check	✓
CVE-2018-9508	Bluetooth	missing boundary check	✓	CVE-2018-9507	Bluetooth	missing boundary check	✓
CVE-2018-9506	Bluetooth	missing boundary check	✓	CVE-2018-9505	Bluetooth	missing boundary check	✓
CVE-2018-9504	Bluetooth	missing boundary check	✓	CVE-2018-9502	Bluetooth	missing boundary check	✓
CVE-2018-9363	Bluetooth	missing boundary check	✓	CVE-2018-9358	Bluetooth	missing boundary check	✓
CVE-2017-0785	Bluetooth	missing boundary check	✓	CVE-2017-13283	Bluetooth	missing boundary check	✓
CVE-2017-1000250	Bluetooth	missing boundary check	✓	CVE-2020-0379	Bluetooth	downgrading security level	✓
CVE-2020-9770	Bluetooth	downgrading security level	✓	CVE-2019-2225	Bluetooth	downgrading security level	✓
CVE-2017-0783	Bluetooth	downgrading security level	✓	CVE-2015-8732	Zigbee	missing boundary check	✓
CVE-2015-6244	Zigbee	missing boundary check	✓	CVE-2020-15802	Bluetooth	cracking session key	✗