ProXray: Protocol Model Learning and Guided Firmware Analysis

Farhaan Fowze, Dave Tian, Grant Hernandez, Kevin Butler, and Tuba Yavuz

Abstract—The number of Internet of Things (IoT) has reached 7 billion globally in early 2018 and are nearly ubiquitous in daily life. Knowing whether or not these devices are safe and secure to use is becoming critical. IoT devices usually implement communication protocols such as USB and Bluetooth within firmware to allow a wide range of functionality. Thus analyzing firmware using domain knowledge from these protocols is vital to understand device behavior, detect implementation bugs, and identify malicious components. Unfortunately, due to the complexity of these protocols, there is usually no formal specification available that can help automate the firmware analysis; as a result significant manual effort is currently required to study these protocols and to reverse engineer the device firmware. In this paper, we propose a new firmware analysis methodology using symbolic execution called ProXray, which can learn a protocol model from known firmware, and apply the model to recognize the protocol relevant fields and detect functionality within unknown firmware automatically. After the training phase, ProXray can fully automate the firmware analysis process while supporting user's queries in the form of protocol relevant constraints. We have applied ProXray to the USB and the Bluetooth protocols by learning protocol constraint models from firmware that implement these protocols. We are then able to map protocol fields and identify USB functionality automatically within all 6 unknown USB firmware while achieving more than an order of magnitude speedup in reaching protocol relevant targets in unknown Bluetooth firmware. Our model achieved high coverage of the USB and Bluetooth specifications for several important protocol fields. ProXray provides a new method to apply domain knowledge to firmware analysis automatically.

Index Terms—Protocol Learning, Model Extraction, Firmware, Symbolic Execution.

1 Introduction and Motivation

The number of Internet of Things (IoT) devices has reached 7 billion globally in early 2018 [1]. These devices have penetrated into almost every aspect of society, such as medicine and industrial control systems. Due to their interactions with the physical world, e.g., via sensors and actuators, and low-cost microcontroller architectures, IoT devices have a wide attack surface, which can be exploited to cause significant damage as in the case of the Mirai botnet [2]. Analyzing the firmware of IoT devices for safety and security is becoming critical.

IoT devices usually implement communication protocols such as USB and Bluetooth within firmware to allow a variety of functionality. Different vendors often have their own protocol stack implementations based on their interpretation of a protocol specification. Unfortunately, there is usually no formal specification to check whether the firmware implements the protocol(s) correctly. Transforming an informal protocol specification into a formal representation is often infeasible due to the complexity of these protocols. For instance, the core Bluetooth 5.0 specification [3] has almost 3,000 pages excluding different application protocols (a.k.a., profiles).

Protocol related bugs contribute to the IoT's wide attack surface. Linux kernel maintains a long list of "unusual" USB devices called "quirks" [4], which essentially violate some parts of the USB specification. Over 80 bugs were found within the Linux USB subsystem as well in the past year [5]. Even worse, attackers can exploit these vulnerabilities to steal sensitive information or achieve privilege escalation within target systems. BlueBorne [6] allows attackers to inject worms into Android devices by exploiting a Bluetooth protocol stack vulnerability within Android. BleedingBit [7] enables attackers to break into an enterprise environment undetected by exploiting a Bluetooth Low Energy (BLE) vulnerability within TI devices. In the worst case, attackers can reprogram the firmware to add malicious functionality, such as BadUSB attacks [8].

Accordingly, existing firmware analysis frameworks either ignore the domain knowledge from these protocols, e.g., treating firmware the same as typical binary executables [9], [10], [11], [12], or require manual effort to study these protocols and reverse engineer the firmware [13]. The complexity of protocol implementations, i.e., complex dataflow and implicit calls through function pointers, leaves pure NLP-based approaches and light-weight static analysis ineffective for reasoning about protocol-relevant behavior. Dynamic symbolic execution, on the other hand, serves as a feasible approach to solving this problem due to its precise memory model.

In this paper, we propose a new *firmware analysis methodology* called ProXray, which utilizes dynamic symbolic execution. As illustrated in Figure 1, ProXray can learn a protocol model from known firmware and apply the model to recognize the protocol and to identify functionality within unknown[1] firmware automatically.

ProXray has three stages. To learn a protocol model without a formal specification of the protocol, we use various path prioritization heuristics for symbolic execution to extract protocol field constraints by running some known firmware implementing the protocol. The protocol model is essentially the collection of those constraints. Once the protocol model is available, ProXray can apply the model

1. We use the term *unknown firmware* to denote firmware that is known to implement a specific protocol and for which the source code is not available.

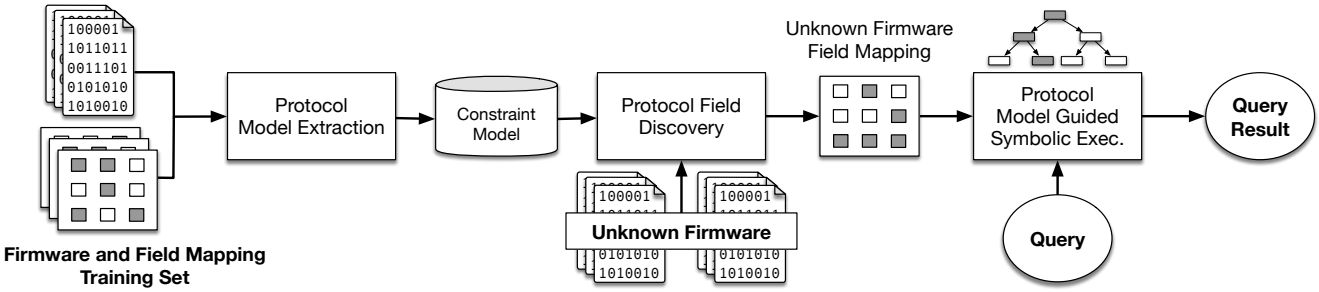


Fig. 1: ProXray: Protocol model extraction and its utilization in guided symbolic execution.

to unknown firmware to recognize the protocol usage by mapping binary execution into protocol field constraints using symbolic execution. After the firmware is tagged with protocol fields automatically, ProXray accepts queries in the form of protocol relevant data constraints, leverages the protocol model again to accelerate path explorations in symbolic execution, and generates answers.

The first stage of our approach, in which we extract protocol constraints from the firmware in the training set, can be viewed as an instantiation of the generic feature extraction approach of Eisenbarth et al. [14]. The novelty of our approach stems from the fact that we represent the extracted features formally and combine these formal models extracted from different source code to generate a protocol constraint model. The benefit of having a protocol model is that when we do not have the source code available for a system under analysis and, therefore, cannot apply the approach of Eisenbarth et al. [14], we can reverse engineer the protocol relevant features of the code via Protocol Field Discovery and leverage this information in Protocol Model Guided Symbolic Execution.

We have applied ProXray to the USB protocol by learning a USB protocol model from 23 known USB firmware images. We are then able to automatically map protocol fields and identify USB functionality by answering queries like "Does the firmware contain a USB keyboard functionality?" within all 6 unknown USB firmware images. We have also applied ProXray to the Bluetooth protocol and we achieved more than an order of magnitude speedup in analyzing unknown firmware using the learned HCI and L2CAP models. Specifically, ProXray answers four important research questions:

1) *Does constraint-based search prioritization perform better than the baseline symbolic execution (BSE)?*
2) *Can we use the extracted constraint model to discover the program variables that correspond to the protocol fields in unknown firmware?*
3) *How effective is protocol-guided symbolic execution in driving symbolic execution to the protocol relevant targets in unknown firmware?*
4) *How precise is constraint model based functional classification on unknown firmware?*

Our contributions can be summarized as follows:

- We introduce a learning algorithm that employs several symbolic execution prioritization and pruning heuristics to generate a protocol model from firmware directly. *Our experiments show that our constraint-based search prioritization heuristics perform better than the BSE and extract up to 1.54 and 1.76 times more unique constraints on average for the USB and the Bluetooth protocols*[2]*, respectively.*
- We present a discovery algorithm that uses the protocol model to recognize the protocol usage within unknown firmware automatically. *Our experiments show that our approach achieves on average 100% and 75% precision in mapping the variables that correspond to the two fields,* `wValue` *and* `bRequest`*, respectively, which are instrumental in determining the functionality of USB devices. For the Bluetooth protocol, we achieved 100% precision for the HCI* `event type` *and the L2CAP* `signaling command code` *fields.*
- We design a protocol guided symbolic execution algorithm that identifies functionality and answers queries about unknown firmware.
- We have applied our methodology to the USB protocol and the HCI and L2CAP layers of the Bluetooth protocol and evaluated ProXray using the firmware that runs on MSP430 and Intel 8051 microcontrollers. *Our experimental results show at least an order of magnitude speedup for the Bluetooth firmware and up to 73.8 times speedup in reaching USB protocol related targets for MSP430 firmware and 1.5 times speedup for 8051 firmware. Our approach can correctly match functionality of every USB firmware in our test set from both MSP430 and Intel 8051 architectures.*

The rest of the paper is organized as follows. Section 2 provides background information on the symbolic execution of firmware. Section 3 presents an overview of our methodology. Section 4 presents the individual steps of our approach. Section 5 applies our methodology to the USB and Bluetooth protocols. Section 6 presents experimental results. Section 7 discusses threats to validity and other challenges. Section 8 presents the related work, and Section 9 concludes and considers future work.

2 Background

2.1 Protocol Model Learning

Model learning refers to inferring a model of software components [15]. This model could be a Hidden Markov-Chain Model (HMM), for example, or relations between

2. The extracted protocol models can be found at https://firmware-analysis.org/

objects, class hierarchies, or implemented protocols. For the latter, protocol state transitions are often set as the learning target. Depending on the availability of the code in some analyzable form, either a black-box or a white-box method can be applied. White-box methods learn the model by analyzing the source code or the binary executable. When it is not feasible to analyze the implementation of a system under analysis, black-box methods are used to infer the state machines by observing the inputs and outputs of the program. Both passive and active learning are possible. In passive learning, the training data is labeled upfront. In active learning, however, labeling is performed on the specific instances when explicit queries are submitted. Unlike previous protocol learning techniques (see Section 8 for a detailed discussion), ProXray targets learning of protocol constraints rather than a state machine of the protocol. ProXray is essentially a white-box and a passive model learning method.

2.2 Firmware Analysis using Symbolic Execution

Classical symbolic execution uses symbolic values for inputs and executes the instructions symbolically to propagate symbolic data flow among the program variables. When a branch instruction that involves a symbolic condition gets executed, multiple successors may potentially be created to represent the feasible paths in the program. Each path is associated with a symbolic expression, the *path condition*, to represent all decisions made on the symbolic inputs along that path.

Dynamic symbolic execution extends classical symbolic execution by mixing concrete and symbolic values to deal with challenging cases such as library calls and non-linear expressions. Concolic Testing [16] and Execution-Generated Testing (EGT) [17] are two specializations of dynamic symbolic execution. In Concolic Testing the program is executed with concrete input values while computing both concrete states and symbolic states for variables that have symbolic values. It uses the symbolic path expressions to generate new concrete input values that can potentially execute new parts of the code. EGT, on the other hand, uses symbolic input values, keeps a symbolic state for the relevant variables, and performs concrete computation only when all the variables are concrete. So an EGT based symbolic execution engine, such as KLEE [18], can mix concrete inputs with symbolic inputs. Please see [19] for an overview on various symbolic execution approaches. In this paper, the algorithms we present in Section 4 assume that the underlying symbolic execution engine implements the EGT approach.

Symbolic execution of firmware poses challenges that do not exist for the symbolic execution of user space applications. These include specification of the architectural elements, e.g., special function registers of a specific microcontroller architecture, and the interrupt service routines (ISRs) . FIE [9] extends the KLEE symbolic execution engine to enable analysis of MSP430 firmware. It enables specification of the microcontroller specific memory layout and symbolic regions as well as ISRs. It implements an approximate interrupt scheduling policy. In previous work [13], we have extended FIE with binary execution capability and a support for Intel 8051 firmware by developing an LLVM lifter for Intel 8051 ISA and modeling the architectural elements. ProXray uses FIE with our extensions to extract USB protocol relevant constraints from a set of 23 MSP430 firmware. The combination of extracted constraints that are rewritten in terms of the protocol fields forms the learned protocol model. We have used ProXray and the learned model to a test set of firmware that consists of four MSP 430 firmware images and two Intel 8051 firmware images to show the effectiveness of our approach in reaching protocol relevant code locations, which we call *targets*, and identifying the specific functionality implemented by these firmware images.

3 System Overview

This section presents an overview of ProXray, which extracts a formal protocol model from known firmware and leverages it in the analysis of unknown firmware. Figure 1 shows the data flow and major processing phases.

Model Extraction Phase: The first phase involves protocol model extraction. We assume the availability of a representative set of firmware implementing the protocol of interest. This is a reasonable assumption as microcontroller vendors provide software developer packages (SDKs), e.g., the MSP430 SDK provided by Texas Instruments [20], for their boards and the associated toolchains. These developer packages demonstrate programming of both the microcontroller specific features and the communication protocols that they use. This type of program-based documentation of the protocol supplements the original protocol specification and helps developers understand how to program their firmware to use the protocol. SDKs often use identifier names that are similar to the protocol field names. Thus, in firmware that comes with an SDK, a simple text search often suffices to identify the variables that implement the protocol field names.

Our approach can leverage the domain knowledge encoded in such sample firmware to extract a constraint-based model of the protocol. As shown in Figure 1, each firmware that is used in the training set is accompanied with a mapping of the protocol fields to the memory locations (program variables) in the firmware[3]. In this paper, we assume that the mapping information needed for the model extraction phase is generated manually by scanning the source code of the sample firmware.

The protocol model extraction stage takes the sample firmware along with the associated mapping between the protocol fields and the program variables. It performs symbolic execution on a set of training firmware in a way that prioritizes exploration of protocol-relevant paths. The main information that guides the symbolic execution of this stage is the protocol field mapping and the protocol-related symbolic program constraints extracted from various branches in the firmware as paths are explored.

Let us assume that the variables $sdata$ and req of a firmware in the training set have been mapped to the USB protocol fields $bmRequestType$ and $bRequest$, respectively (see Section 5.1 for a brief background on the USB protocol).

3. In this paper, we use memory locations and program variables interchangeably. The memory location concept facilitates explanation of our algorithms that extend basic symbolic execution.

One way of guiding the exploration is to prioritize paths that have explored new symbolic regions representing some of the protocol fields. As an example, assume that a path has so far executed branch conditions involving the variable $sdata$. If a successor of this path has recently executed a branch condition involving req while another successor has not executed such a branch condition yet then the former path should be prioritized over the latter. Another way is to prioritize paths based on the number of different branch conditions that they have executed. As an example, a path that has recently checked a new condition, e.g., $sdata == 1$ in addition to $sdata == 0$ should be prioritized over a path that has only checked $sdata == 0$.

Protocol relevant program constraints are captured in a canonical form, $exp\ OP\ \kappa$ where OP and κ denote a relational operator and a constant, respectively. exp may consist of a protocol field, m, or an expression that involves one or more bitwise operators manipulating some protocol fields, e.g., $m\ BOP\ \kappa_1$ or $(m_1\ BOP_1\ \kappa_1)\ BOP_2\ m_2$, where m, m_1, and m_2 denote protocol fields, BOP, BOP_1 and BOP_2 denote bitwise operators, and κ_1 denotes a constant. Basically, our atomic constraints correspond to the atomic constraints of the Bitvector Logic [21]. Extracted protocol-relevant constraints are rewritten by replacing each variable with the corresponding protocol field. Thus, the constraint model shown in Figure 1 consists of a set of canonical constraints, where the identifiers in the constraints correspond to some protocol field. Protocol constraints that are extracted from each firmware in the training set are eventually combined to represent the protocol constraint model. Specifically, we compute the union of all sets of canonical constraints to combine them. The protocol model is leveraged by the second and third phases of our approach to uncover functionality of unknown firmware.

Protocol Field Discovery: In the second phase of our approach, a given unknown firmware is analyzed to discover the mapping between the variables of the firmware and the protocol fields. It should be noted that for arbitrary firmware, which may be in binary form only, it may not be feasible to do this manually. We propose an automated technique to discover this relationship between the firmware variables/memory regions and the protocol fields. We achieve this by leveraging the protocol constraint model. Our approach employs standard symbolic execution to explore some of the paths and compares the constraints in the path condition with those in the protocol constraint model for detecting semantic matching. Those constraints that match semantically provide potential mappings. We use statistical information, i.e., the number of times a memory region is matched to a protocol field, to refine the discovered mapping.

Protocol Model Guided Symbolic Execution: The third phase of our approach leverages the discovered mapping for the unknown firmware to perform protocol-guided symbolic execution. The inputs consist of an unknown firmware that has not been analyzed w.r.t. to protocol use, the discovered mapping between the firmware variables and the protocol fields, and a query in the form of a protocol constraint representing the part of the protocol of interest. Note that the query helps to focus the analysis to a specific functionality of the protocol. Our guided symbolic execution rewrites the input protocol constraint to reflect all possible mappings that have been discovered. As an example, in one of the USB firmware we examined, the $bmRequestType$ field was implemented by the variable $tSetupPacket.bmRequestType$. So, the constraint $bmRequestType == 161$ as a query is rewritten as $tSetupPacket.bmRequestType == 161$ if $bmRequestType$ has been mapped to $\{tSetupPacket.bmRequestType\}$ and is rewritten as $tSetupPacket.bmRequestType == 161 \vee sdata == 161$ if $bmRequestType$ has been mapped to $\{tSetupPacket.bmRequestType, sdata\}$ to reflect all possible candidate mappings. Then we apply a customized pruning algorithm to expand the paths that satisfy the transformed protocol constraint. In this paper, we focus on two applications of protocol-guided analysis of unknown firmware: 1) checking whether the unknown firmware handles a given protocol constraint and 2) discovering the functionality class(es) an unknown firmware implements.

4 Approach

The goal of ProXray is to automatically extract a constraint-based model of a protocol using a representative set of firmware and then use this model to perform guided symbolic execution in new firmware. We present our protocol model extraction in Section 4.1, discovery of protocol elements in a new firmware in Section 4.2, and protocol guided symbolic execution in Section 4.3.

4.1 Protocol Model Extraction

Algorithm 1 Protocol Constraint Model Extraction Algorithm

1: **ExtractProtocolConstraints**(F : $Firmware$, M : $MemLoc \rightarrow Identifier$, C_{scope} : $\{perPath, perGroup\}$,C_{cov} : $\{code, constraint, field\}$, C_{window} : $\mathcal{N}$, τ : $\mathcal{R}$): $\mathcal{P}(Constraint)$
2: $s_0 : SEState$
3: Let s_0 denote the initial symbolic execution state/path for F
4: **global** $ActivePaths \leftarrow \{s_0\}$
5: **global** $SC \leftarrow \emptyset$
6: **global** $stashStack \leftarrow$ empty stack of $SEState$
7: **global** $newPaths \leftarrow \emptyset$
8: **global** $toBeStashed \leftarrow ActivePaths$
9: start the new window of size C_{window} for executing paths in $ActivePaths$
10: **while** τ seconds not elapsed and $ActivePaths \neq \emptyset$ **do**
11: **while** exists some path in $ActivePaths$ for which end of window has not been reached **do**
12: $s \leftarrow$ **chooseNext**($ActivePaths$)
13: update s's coverage based on C_{cov} and the next instruction
14: $s.successors \leftarrow$ **ExecuteNextInstruction**(s)
15: **for** each $s' \in s.successors$ **do**
16: Let Mem denote the memory locations that appear in $s'.PC$
17: **ExtractAtomicConstraints**($Mem, s'.PC, M$)
18: **end for**
19: **if FilterStates**(C_{scope}, s) **then**
20: break
21: **end if**
22: **end while**
23: **NextFrontierSet**()
24: start the new window of size C_{window}
25: **end while**
26: **return** SC

Algorithm 1 presents our approach for extracting a protocol constraint model from a single firmware F. It takes

as input a mapping M from firmware memory locations to the protocol fields and configuration options that we will explain below. It uses symbolic execution to explore the paths in F and returns a set of atomic constraints, SC, on the protocol fields.

One challenge of symbolic execution is the *path explosion* problem. The exponential growth in the number of paths slows down the progress made for each path. Depending on the goal of the underlying analysis, the achieved coverage may be far from the ideal. Our goal is to explore the paths of the firmware that implement the protocol functionality while traversing as diverse set of protocol relevant paths as possible. Although symbolic execution engines come with path exploration heuristics such as those based on random selection and coverage, we need customized heuristics to maximize the unique number of protocol constraints extracted.

We thus designed three types of "knobs" to fine tune our model extraction process. The first knob, C_{cov}, configures the type of coverage we would like to measure as the exploration progresses. In addition to traditional code coverage, we also consider constraint coverage and protocol field coverage. The second knob, C_{scope}, configures the scope of coverage computation. One possible scope is per-path and the other possibility is per-group (considering all active paths). The third knob, C_{window}, configures the duration of coverage computation, denoting the number of blocks executed.

Algorithm 1 keeps track of a frontier set of symbolic execution states or paths, $ActivePaths$, that is initialized with the initial state (line 4). It keeps a stack of stashed paths, $stashStack$, representing all paths other than the current frontier set and to be considered later. The algorithm runs until a time bound τ is reached or the frontier set becomes empty. Each time $ActivePaths$ is initialized, a window of size C_{window} starts (lines 9 and 24). Until the window ends, paths in $ActivePaths$ execute as in standard symbolic execution by choosing a path from the frontier set (line 12), updating coverage based on the type of coverage configuration (line 13), and executing the next instruction to compute possible successors (line 14).

After executing an instruction for a path s, the path conditions of the successors are analyzed (lines 15-18) using Algorithm 2 to check the appearance of a new protocol relevant constraint. Algorithm 2 uses the protocol field mapping M to identify such constraints (lines 3-4), which are stored in the set SC (line 5).

Algorithm 2 An algorithm for extracting atomic constraints from a given expression using a mapping from the memory locations to the protocol fields.

1: **ExtractAtomicConstraints**(Mem : $\mathcal{P}(MemLoc)$, E : Expression, $M : MemLoc \rightarrow Identifier$)
2: $MMem \leftarrow \{m \mid m \in Mem \ \wedge \ M(m) \neq undef\}$
3: **for** each $\chi \in ATOMIC(E)$ and $m \in MMem$ **do**
4: **if** $m \in Var(\chi)$ **then**
5: $SC \leftarrow SC \cup \chi[M(m)/m]$
6: **end if**
7: **end for**

Figure 2 shows a sample code snippet from the MSP430 SDK for the USB protocol [20]. In this SDK, class specific descriptors are defined by class specific code (lines 5-18). The common functionality (lines 21-58) uses data structures defined by the class specific functionality, e.g., the `tUsbRequestList` array, to check for the matching request types and the requests. Figure 3 shows the extracted protocol relevant constraint on the protocol field `bRequest` that corresponds to the condition at line 32 when the request matches the fields of the array entry on lines 7-16.

Figure 4 shows a sample code snippet from an implementation of the Bluetooth protocol [22]. The transport layer function `hci_transport_h4_block_read` (line 22) reads the raw HCI packet and calls a callback function through the function pointer `packet_handler` (line 23), which gets resolved to the `packet_handler` function defined on line 28. The `packet_handler` function checks the HCI packet type and calls the `event_handler` function for HCI event type packets. Line 41 shows one of the cases handled by the `event_handler` function that involves checking the `HCI Command Complete` event for the sub-event `hci_read_local_name`. To check this particular case, it uses the `HCI_EVENT_IS_COMMAND_COMPLETE` macro and according to the definition of this macro (lines 16-17) it checks two conditions: 1) the event code corresponding to `HCI_EVENT_COMMAND_COMPLETE` (line 16) and 2) the sub-event code that consists of two bytes and when read in little endian mode corresponds to the `opcode` of the `hci_read_local_name` command (line 17). Figure 5 shows the extracted atomic constraint that corresponds to the condition at line 17. In this case the bitwise operators are the bitwise `Or` and the shift left (`Shl`) operators.

In addition to $ActivePaths$, Algorithm 1 keeps two sets of paths, $newPaths$ and $toBeStashed$, denoting the next set of frontier paths and the paths from the current frontier set that will be stashed away. Filtering of paths into the next frontier set and stashing of others are performed by Algorithm 3 and depends on the configured scope of the analysis. At the beginning of each window, $newPaths$ and $toBeStashed$ are initialized to an empty set and to $ActivePaths$, respectively (lines 7-8 of Algorithm 1 and 11-12 of Algorithm 4). If the scope is per group (lines 2-14) then filtering does not happen until all the paths in the frontier set reach the end of the current window. While inside the window, the algorithm updates the frontier set (line 4) and the paths to be stashed away (line 5).

When a path reaches the end of the current window, it is removed from the frontier set (line 7). If such a path achieves new coverage based on the type of coverage, its successors are added to the next frontier set (line 9) and it is not considered for stashing (line 10). Otherwise, its successors are stashed (line 12).

If the scope is per path (lines 15-19) then new coverage is checked after each instruction execution (line 15) and handled immediately even if the end of the window is not reached yet. As in the per group case, when a path is filtered the next set of frontier paths and the set of stashed paths get updated (line 16 and 17). Unlike in the per group case, when a path is filtered it signals to stop executing the paths in the current frontier set (line 18).

After executing the paths in the current frontier set, Algorithm 1 calls Algorithm 4 to compute the next frontier set (line 23). Algorithm 4 pushes the stashed paths, if any, onto the stack (line 3) and updates the frontier set using the

```
1  //  MSP430_USB_API/USB_API/USB_Common/usb.h
2  #define USB_MSC_GET_MAX_LUN            0xFE // 254 in decimal
3
4  // MSP430_USB_API/examples/MSC_massStorage/M2_SDCardReader/USB_config/descriptors.c
5   const tDEVICE_REQUEST_COMPARE tUsbRequestList[] =
6  {    ...
7       {
8               // Get Max Lun
9               USB_REQ_TYPE_INPUT | USB_REQ_TYPE_CLASS | USB_REQ_TYPE_INTERFACE,
10              USB_MSC_GET_MAX_LUN,
11              0x00,0x00,                      // always zero
12              MSC0_DATA_INTERFACE,0x00,       // MSC interface is 0
13              0x01,0x00,                      // Size of Structure (data length)
14              0xff,&Get_MaxLUN,
15              ...
16      },
17     ...
18 };
19
20 //  MSP430_USB_API/USB_API/USB_Common/usb.c
21 uint8_t usbDecodeAndProcessUsbRequest (void){
22          ...
23      const uint8_t* pbUsbRequestList;
24      ptDEVICE_REQUEST ptSetupPacket = &tSetupPacket;
25          ...
26      pbUsbRequestList = (uint8_t*)&tUsbRequestList[0];
27      while (1) {
28          bRequestType = *pbUsbRequestList++;
29          bRequest     = *pbUsbRequestList++;
30          ...
31          if ((bRequestType == tSetupPacket.bmRequestType) &&
32              (bRequest == tSetupPacket.bRequest)){
33              bResult = 0xc0;
34              bMask   = 0x20;
35              for (bTemp = 2; bTemp < 8; bTemp++) {
36                  if (*((uint8_t*)ptSetupPacket + bTemp) == *pbUsbRequestList){
37                      bResult |= bMask;
38                  }
39                  pbUsbRequestList++;
40                  bMask = bMask >> 1;
41              }
42              if ((*pbUsbRequestList & bResult) == *pbUsbRequestList){
43                  pbUsbRequestList -= 8;
44                  break;
45              } else {
46                  pbUsbRequestList += (sizeof(tDEVICE_REQUEST_COMPARE) - 8);
47              }
48          }
49          else {
50              pbUsbRequestList += (sizeof(tDEVICE_REQUEST_COMPARE) - 2);
51          }
52      }
53      ....
54      lAddrOfFunction =
55          ((tDEVICE_REQUEST_COMPARE*)pbUsbRequestList)->pUsbFunction;
56
57      bWakeUp = (*lAddrOfFunction)();
58 }
```

Fig. 2: Code snippets from the USB SDK for MSP430 MCUs [20].

```
(Eq 254 (Read w8 0 bRequest))
```

Fig. 3: The extracted USB protocol constraint (the `USB_MSC_GET_MAX_LUN` request) that corresponds to the condition on line 32 in Figure 2 when the request matches the fields of the array entry bw lines 7-16 in Figure 2.

paths that have been filtered, if any (line 6). On the other hand, if no paths could be filtered from the frontier set, the next set of frontier paths is received from the top of the stack of stashed paths (lines 8 and 9).

4.2 Protocol Field Discovery

The goal of protocol field discovery is to analyze an arbitrary firmware known or suspected to implement a protocol functionality and identify the set of potential memory locations corresponding to each data field of the protocol. The idea is to use symbolic execution to explore paths of the firmware under analysis and utilize the protocol constraint model that has been extracted as presented in Section 4.1 for this discovery process.

```
1  // bluetooth.h
2  #define OGF_CONTROLLER_BASEBAND   0x03
3  // bluetooth.h
4  #define HCI_EVENT_COMMAND_COMPLETE  0x0E
5
6  // src/hci_cmd.c
7  #define OPCODE(ogf, ocf) (ocf | ogf << 10)
8  //  ===> (0x0014 | 0x0003 << 10) ====> (0x0014 | 0x0C00) ====> 0x0C14 = 3092
9
10 // src/hci_cmd.c
11 const hci_cmd_t hci_read_local_name = {
12 OPCODE(OGF_CONTROLLER_BASEBAND, 0x14), ""
13 };
14
15 // src/hci.h
16 #define HCI_EVENT_IS_COMMAND_COMPLETE(event,cmd) (event[0] == HCI_EVENT_COMMAND_COMPLETE &&
17                                     little_endian_read_16(event,3) == cmd.opcode)
18
19 // src/hci_transport_h4.c
20 static uint8_t * hci_packet = &hci_packet_with_pre_buffer[HCI_INCOMING_PRE_BUFFER_SIZE];
21
22 static void hci_transport_h4_block_read(void){
23     packet_handler(hci_packet[0], &hci_packet[1], read_pos-1);
24 }
25
26
27 // src/hci.c
28 static void packet_handler(uint8_t packet_type, uint8_t *packet, uint16_t size){
29     hci_dump_packet(packet_type, 1, packet, size);
30     switch (packet_type) {
31         case HCI_EVENT_PACKET:
32             event_handler(packet, size);
33             break;
34     ...
35      }
36 }
37
38 // src/hci.c
39 static void event_handler(uint8_t *packet, int size) {
40 ...
41    if (HCI_EVENT_IS_COMMAND_COMPLETE(packet, hci_read_local_name)){
42         ...
43    }
44 }
```

Fig. 4: Code snippets from a Bluetooth stack implementation [22].

Algorithm 3 An algorithm for filtering the symbolic execution states based on relevance to extracting protocol relevant constraints.

FilterStates(C_{scope} : $\{perPath, perGroup\}$, s : $SEState$): boolean
if $C_{scope} = perGroup$ **then**
  if not end of current window for $s.successors$ **then**
    $ActivePaths \leftarrow ActivePaths \cup s.successors \setminus \{s\}$
    $toBeStashed \leftarrow toBeStashed \cup s.successors \setminus \{s\}$
  else ▷ end of current window for $s.successors$
    $ActivePaths \leftarrow ActivePaths \setminus \{s\}$
    if s covers new based on C_{cov} in the current window **then**
      $newPaths \leftarrow newPaths \cup s.successors$
      $toBeStashed \leftarrow toBeStashed \setminus s$
    else ▷ no new coverage by the end of the window
      $toBeStashed \leftarrow toBeStashed \cup s.successors \setminus \{s\}$
    end if
  end if
else if s covers new based on C_{cov} **then** // $C_{scope} = perPath$
  $newPaths \leftarrow newPaths \cup s.successors$
  $toBeStashed \leftarrow toBeStashed \setminus s$
  return true ▷ Terminate current window
end if
return false

```
(Eq 3092
     (Or w16
          (Shl w16
               (ZExt w16 (Read w8 5 hci_packet))
          8)
          (ZExt w16 (Read w8 4 hci_packet))
     )
)
```

Fig. 5: The extracted Bluetooth protocol constraint (`HCI_Read_Local_Name` command) that corresponds to the condition on line 17 in Figure 4.

Algorithm 5 presents the details of our protocol field discovery algorithm. The inputs include a set of atomic constraints over protocol fields PM, the set of protocol fields PF, and a time threshold τ. The algorithm returns mappings between the protocol fields and the set of memory locations. It collects candidate mappings as it explores program paths by running the standard symbolic execution algorithm. When a path condition is updated, it checks to see if each constraint in the path condition matches some

Algorithm 4 An algorithm for computing the next frontier set.

```
1: NextFrontierSet()
2: if toBeStashed ≠ ∅ then
3:     stashStack.push(toBeStashed)
4: end if
5: if newPaths ≠ ∅ then
6:     ActivePaths ← newPaths
7: else
8:     ActivePaths ← stashStack.top()
9:     stashStack.pop()
10: end if
11: newPaths ← ∅
12: toBeStashed ← ActivePaths
```

Algorithm 5 Protocol Field Discovery Algorithm

```
1: DiscoverProtocolFields(F : Firmware, PM : P(Constraint),
   PF : P(Identifier), τ : R) : Identifier → P(MemLoc)
2: Let M, M_best : Identifier → P(MemLoc)
3: M, M_best ← λx.∅
4: Let Freq : MemLoc → Identifier → N
5: Freq ← λx.λy.0
6: Let s_0 denote the initial symbolic execution state/path for F
7: ActivePaths ← {s_0}
8: while τ seconds not elapsed and ActivePaths ≠ ∅ do
9:     s ← chooseNext(ActivePaths)
10:    s.successors ← ExecuteNextInstruction(s)
11:    ActivePaths ← ActivePaths ∪ s.successors \ {s}
12:    for each s' ∈ s.successors do
13:        for each c_pc ∈ ATOMIC(s'.PC) do
14:            for each pf ∈ Identifier do
15:                if ∃c ∈ PM. (isValid(c ↔ c_pc[pf/m]) then
16:                    M ← M[pf ← M(pf) ∪ {m}]
17:                    Freq ← Freq[(m, pf) ← Freq(m, pf) + 1]
18:                end if
19:            end for
20:        end for
21:    end for
22: end while
23: M_best ← λpf.{m | Freq(m, pf) = MAX_{pf_i ∈ PF}(Freq(m, pf_i))}
24: return M_best
```

constraint in the protocol model. If so, it maps the memory location that appears in the matching constraint from the path condition to the protocol field in the matching protocol constraint (line 16). It also updates a frequency function that keeps track of the number of unique matches that have been observed between a memory location and a protocol field (line 17).

By the time the symbolic execution stage terminates, the algorithm has some mappings between protocol fields and sets of memory locations, M, and the number of times a match has been observed between two entities, $Freq$. We choose the candidate mapping with the highest frequency as the final mapping (line 23). The main idea behind this selection is that the more number of times a protocol field has been matched to a memory location, the more confidence we have that the match is correct.

Figure 6 shows a code snippet from the Phison BadUSB firmware [23]. Using the extracted constraints including the one shown in Figure 3, we can map the protocol field `bRequest` to the `SETUPDAT[1]` using Algorithm 5. This is because `SETUPDAT` array will be marked symbolic due to being one of the data ports of the firmware. Symbolic execution will identify the correspondence between the program variable `bRequest` and the memory region `SETUPDAT[1]` due to the assignment statement at line 18. The symbolic condition that gets captured at line 63 will semantically match the constraint in Figure 3 (line 15 of Algorithm 5) and the memory region `SETUPDAT[1]` will be added to the mapping for the protocol field `bRequest` (line 16 of Algorithm 5).

4.3 Protocol Model Guided Symbolic Execution

The goal of guided symbolic execution is to steer the execution into the specific parts of the program. ProXray utilizes the extracted protocol constraint model to guide symbolic execution to explore paths that implement protocol related functionality. We assume that the protocol field discovery has already been performed on the firmware of interest as explained in Section 4.2 and that we have a specific protocol relevant query specified in terms of the fields of the protocol representing a specific protocol functionality.

Algorithm 6 Protocol Guided Symbolic Execution Algorithm

```
1: ProtocolGuidedSymEx(F : Firmware, Q : Constraint, M :
   Identifier → P(MemLoc), τ : R): P(SEState)
2: Let Q ≡ ⋁_{i=1}^{N} ⋀_{j=1}^{k_i} q_ij
3: Let
        T(q) = { ⋁_{m∈M(pf)} q[m/pf]   M(pf) ≠ ∅ // case 1
               { true                  otherwise // case 2
4: Let Q' ≡ ⋁_{i=1}^{N} ⋀_{j=1}^{k_i} T(q_ij)
5: Let s_0 denote the initial symbolic execution state/path for F
6: ActivePaths ← {s_0}
7: while τ seconds not elapsed and ActivePaths ≠ do
8:     s ← chooseNext(ActivePaths)
9:     s.successors ← ExecuteNextInstruction(s)
10:    filtered, pruned ← false
11:    filteredPaths ← ∅
12:    for each s' in s.successors do
13:        if s'.PC ∧ Q' ≠ false then
14:            filtered ← true
15:            filteredPaths ← filteredPaths ∪ {s'}
16:        end if
17:        if s'.PC ∧ Q' = false then
18:            pruned ← true
19:        end if
20:    end for
21:    if filtered ∧ pruned then
22:        ActivePaths ← filteredPaths
23:    else
24:        ActivePaths ← ActivePaths ∪ filteredPaths \ {s}
25:    end if
26: end while
27: return ActivePaths
```

Algorithm 6 presents our protocol model guided symbolic execution algorithm. The input includes a reachability query, Q, in the form of a protocol relevant constraint (the query in Figure 1), discovered mappings from protocol fields to a set of memory locations, M (generated by Algorithm 5), and a time threshold τ. The algorithm first transforms the protocol relevant input query into a program specific constraint by utilizing the mapping M (line 4). If a protocol field is mapped to a nonempty set of memory locations (line 3, case 1) then each memory location is considered to be legitimate separately and the query is rewritten by replacing the protocol field with that memory location. Constraints that are obtained through rewriting are combined using the disjunction operator. However, if a protocol field could not be mapped to a memory location (line 3, case 2) then the constraint is replaced with $true$.

```
1  // firmware/defs.h
2  __xdata __at 0xF0B8 volatile BYTE SETUPDAT[8];
3
4  // firmware/usb.c
5  BYTE        bmRequestType, bRequest;
6  WORD        wValue;
7
8  //firmware/main.c
9  void main() { ...
10   while (1) { HandleUSBEvents(); } ...
11 }
12
13 void usb_isr(void) __interrupt USB_VECT {
14  if (UsbIntStsF080 & 1) {
15     XVAL(0xF080) = 1;
16     if (EP0CS & bmSUDAV) {
17        bmRequestType = SETUPDAT[0];
18        bRequest = SETUPDAT[1];
19        wValue = SETUPDAT[2] | (SETUPDAT[3] << 8);
20        wIndex = SETUPDAT[4] | (SETUPDAT[5] << 8);
21        wLength = SETUPDAT[6] | (SETUPDAT[7] << 8);
22     }}}
23
24 void HandleUSBEvents(void) {
25   ... HandleControlRequest(); ... }
26
27 static void HandleControlRequest(void) {
28  BYTE res;
29  switch(bmRequestType & 0x60) {
30    case 0:
31      res = HandleStandardRequest(); break;
32    case 0x20:
33      res = HandleClassRequest(); break;
34    case 0x40:
35      res = HandleVendorRequest(); break;
36    default:
37      res = FALSE;
38  } ...
39 }
40
41 // firmware/control.c
42 BYTE HandleStandardRequest() {
43  switch(bRequest) {
44    case 0x06: { ... return GetDescriptor(); }
45    case 0x05: case 0x09: default: ...
46  }
47 }
48
49 static BYTE GetDescriptor() {
50  BYTE type = (wValue >> 8) & 0xFF;
51  switch (type) {
52    case 0x22: ...
53    case 0x01:...case 0x02:...case 0x06:...
54    default: ...
55  }
56  return ret;
57 }
58
59 // firmware/control.c
60 BYTE HandleClassRequest() {
61  switch(bRequest) {
62   case 0x09: ... case 0x0A: ...
63   case 0xFE: ... {
64     return GetMaxLUN(); }
65   default:
66   { return FALSE;}
67  }
68 }
```

Fig. 6: Code snippets from the Phison BadUSB firmware [23].

The algorithm keeps a set of frontier paths, $ActivePaths$, and starts the symbolic execution from the initial symbolic execution state for F. As in standard symbolic execution, it chooses the next path to execute (line 8) and executes the instruction to produce the successors (line 9). Some of the successors may be filtered (lines 13-16) and some of them may be pruned (line 17-19). It keeps a set to record which successors of the current path gets filtered in $filteredPaths$, which has been initialized to an empty set at line 11. Path condition of each successor will be checked to decide if it satisfies part of the transformed protocol constraint. If so, the successor will be added to the set of filtered paths (line 15). If there are filtered paths as well as pruned paths, the frontier set is updated with the filtered successors of the current path. Otherwise, the frontier set is expanded with the filtered successors (line 24). The intuition behind this is to detect branches where decisions related to the query are made and to aggressively replace the active paths with the successors of the current path that make decisions consistent with the query. At branches without query related decisions, the algorithm preserves the current active set by expanding it with all the successors.

In one extreme case, each protocol field might be mapped to an empty set and the transformed query would evaluate to $true$. In the other extreme case, each protocol field might be mapped to a large set of memory locations yielding a large transformed query. In both cases, steering the execution to the relevant part of the program will not be effective as effective pruning will not be achieved. Moreover, the latter will have an additional overhead in terms of constraint solving due to the size of the transformed query. So the effectiveness of Algorithm 6 in steering the execution to the desirable part of the program depends on the precision of the mapping between the protocol fields and the memory locations provided as an input.

As an example, consider the BadUSB code given in Figure 6 and a query that describes the USB standard request for reporting of the HID functionality: $\texttt{bRequest} = 0x06 \wedge \texttt{wValue\_high} = 0x22$. Assume that Algorithm 5 has mapped the `bRequest` field and the `wValue_high`[4] field to $\{\texttt{SETUPDAT[1]}, \texttt{SETUPDAT[3]}\}$ and $\{\texttt{SETUPDAT[3]}\}$, respectively, in the context of the BadUSB firmware. This mapping gets passed to Algorithm 6 along with the query. The query would be rewritten as $(\texttt{SETUPDAT[1]} = 0x06 \vee \texttt{SETUPDAT[3]} = 0x06) \wedge \texttt{SETUPDAT[3]} = 0x22$ to generate a constraint, as shown on line 4 of Algorithm 6, that refers to the memory regions of the BadUSB firmware. Using this constraint, the guided symbolic execution stage, as shown on lines 7-26 of Algorithm 6, prunes the paths that satisfy the conditions of the lines 45, 53, 54, 62, and 63 and reaches the target location of reporting the HID functionality by filtering paths that satisfy the conditions of the lines 44, 52, and 65 of the BadUSB firmware.

5 Applying ProXray to the USB and Bluetooth Protocols

5.1 Universal Serial Bus

We choose the USB protocol as one of our case studies for ProXray because of its ubiquity in embedded systems and IoT devices. On one hand, the core USB specification, e.g., USB 2.0 [24] or USB 3.0 [25] is still approachable with

4. Refers to the high byte of the `wValue` field.

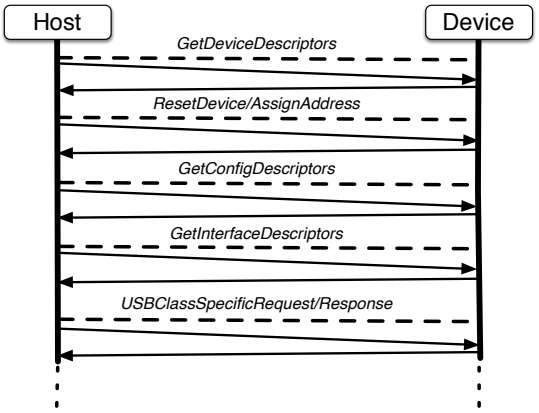


Fig. 7: USB Enumeration Procedure.

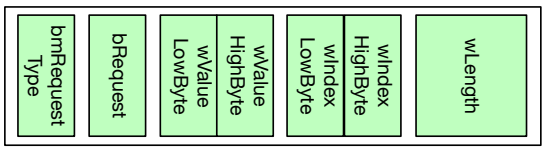


Fig. 8: Protocol fields used in the USB Setup packet in the Enumeration Procedure.

few hundreds of pages. On the other, its ability to support versatile functionality via different USB classes [26] also reflects the challenges in firmware analysis.

Each USB class defines one kind of functionality. The most common classes are Communication Device Class (CDC), Human Interface Device (HID), and Mass Storage Class (MSC). Different USB classes introduce their own request/response messages, which follow the standard request/response structure defined by the USB spec. As shown in Figure 7, all USB devices follow the same procedure called *enumeration* once plugged into the host machine. Initiated by the host, this procedure is used to provide the device configuration information, including *GetDeviceDescriptors*, *GetConfigDescriptors*, and *GetInterfaceDescriptors*. Once the enumeration phase is complete, the corresponding device driver loaded by the OS starts to serve the device using USB class-specific requests.

All USB requests start with a *Setup* packet, as shown in Figure 8, This is an 8-byte structure, containing 1-byte *bmRequestType* and *bRequest* fields, and 2-byte *wValue*, *wIndex*, and *wLength* fields. *bmRequestType* is a bitmap determining data transfer direction, type, and recipient. *bRequest* is the request code defined by the USB and class-specific specs. Both *wValue* and *wIndex* are separated into low and high bytes, which act as parameters passed by a given request type. *wLength* shows the number of bytes to be transferred during the data stage if one exists.

5.2 Bluetooth

While USB dominates wireline connections for embedded systems and IoT devices, Bluetooth [27] provides a most common way to connect with these devices wirelessly, especially after the introduction of Bluetooth Low Energy (BLE) and Bluetooth Mesh [28]. Comparing to the USB protocol,

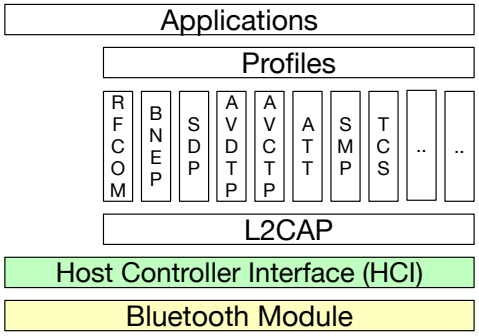


Fig. 9: Bluetooth protocol stack.

Bluetooth [29] is a more complex protocol suite, including different layers within the protocol stack. As shown in Figure 9, from the bottom up we have Bluetooth Module, Host Controller Interface (HCI), Logical Link Control and Access Protocol (L2CAP), a variety of sub protocols used to support different Bluetooth Profiles, and finally the Applications.

The HCI layer is the bottom part of the Bluetooth software stack passing HCI packets to the hardware Bluetooth modules. Packets at this layer follow typical TLV format. There are four types of HCI packets in total, including Command, Event, ACL, and SCO. As its name implies, the L2CAP layer maintains logical connections between different Bluetooth devices, and can be treated as the transport layer within the stack (e.g., TCP and IP). Similar to TCP, L2CAP provides the foundation of different application protocols defined by the Bluetooth specification. Given their importance, we focus on the protocols fields within the HCI and the L2CAP layers.

5.3 Model Generation and Analysis

FIE [9] is a firmware analysis tool that leverages KLEE to perform symbolic execution on MSP430 firmware. We extend FIE to record all conditions evaluated on any protocol field while executing the training firmware set. The core execution engine of FIE considers every read from a symbolic memory region independently and as a new version of that location on the executing path. As a result, every read from a specific memory creates a new node in the underlying abstract syntax tree (AST) used for representing symbolic expressions. This procedure helps in capturing the dynamic interactions between the firmware and its environment. However, this complicates our model extraction as we have to deal with multiple versions of the same memory location. To get the unique constraints imposed on each protocol field, we first find the memory regions used in each branch condition. We developed a custom expression AST traversal to emit the atomic constraints from the branch conditions. If the memory region in the atomic constraint turns out to be a protocol field, we rewrite the conditional expression in terms of that protocol field. For every such expression we evaluate uniqueness of that constraint using validity checking interface of the Simple Theorem Prover (STP), which is the SMT solver that FIE uses. If the new conditional expression turns out to be different from all the constraints in our model, we accept it as a newly found

unique constraint. We store all the unique constraints in the syntax of the KQuery [30] language.

We also use the KQuery language in specifying protocol relevant queries for guided execution. Based on the protocol field mapping found for the firmware under test, we rewrite these queries in terms of the mapped memory regions of the firmware. We utilize the caching solver provided by KLEE in all phases of our approach to minimize the runtime overhead.

6 Evaluation

To evaluate the effectiveness of ProXray in the context of the four research questions we mentioned in Section 1, we applied it to the USB and Bluetooth protocols. For the USB protocol we used the USB development package available from MSP's USB developer site [20] and two Intel 8051 firmware, Phison BadUSB firmware [23] and EzHID firmware [31]. The MSP430 package is provided as an example for USB firmware developers working on the MSP430 architecture. It contains a rich set of example firmware images with each demonstrating a different use-case of the USB protocol. In general, a single firmware image focuses on a specific device class, except for the composite firmwares, which combine functionalities from multiple classes. For the Bluetooth protocol, we used the BTStack framework [22] that implements the Bluetooth Core Specification version 4.0 and provides a variety of embedded firmware examples, eight of which were ported to an MSP 430 architecture. We chose BTStack due to being a fully open source implementation and having MSP 430 examples that we could analyze with FIE[5].

We divide all the firmware into the training set and the testing set for each protocol type. The USB training set contains 23 firmware images from the MSP430 package, implementing 3 different USB classes, including Communication Device Class (CDC), Human Interface Device (HID), and Mass Storage Class (MSC). The USB testing set contains 6 firmware, including another 4 firmware from the MSP430 package and 2 Intel 8051 firmware. Table 1 shows the individual USB firmware images and their size while Table 2 shows the number of firmware images by the USB class type along with the quantitative details on the unique constraints extracted for each. Table 1 also shows the individual firmware images used for Bluetooth experiments. The sizes include the size of the BTStack core. We evaluate our Protocol Model Extraction (6.1) using the training set, Field Discovery (6.2), and finally Guided Execution (6.3) using the testing set.

Firmware Name	Lines of Code
Training Set	
USB CDC	
C0_SimpleSend	18,764
C1_LedOnOff	18,855
C2_ReceiveData	18,764
C3_EchoToHost	18,493
C4_PacketProtocol	18,429
C5_SendDataWaitTillDone	18,323
C6_SendDataBackground	18,269
USB MSC	
M2_SDCardReader	19,128
M3_MultipleLUN	20,403
M4_DoubleBuffering	19,053
M5_CDROM	23,055
USB HID	
H0_SimpleSend	18,777
H1_LedOnOff	18,850
H2_ReceiveData	18,540
H3_EchoToHost	18,404
H4_PacketProtocol	18,476
H5_SendDataWaitTillDone	18,491
H6_SendDataBackground	18,432
H8_Keyboard	18,876
H7_Mouse	18,649
H9_Remote_Wakeup	18,865
H10_ReceiveData_EncryptDecrypt	18,911
H11_LedOnOff_EncryptDecrypt	18,724
Bluetooth	
ble_server	32,387
gap_inquiry	47,428
led_counter	47,315
spp_and_le_counter	58,121
Test Set	
USB	
CC1_term2term	18,946
CH1_term2hidDemo	18,640
CHM1_term2HidDemo_2LUN	24,287
HH1_hidDemo2hidDemo	18,647
BadUSB_Firmware	1,696
EzHID_Firmware	8,683
Bluetooth	
spp_counter	47,397
spp_flowcontrol	47,389
sdp_general_query	47,378
sdp_rfcomm_query	47,368

TABLE 1: The list of firmware examples analyzed and their lines of code generated from David A. Wheeler's `SLOCCount` tool.

6.1 Protocol Model Extraction

In this subsection, we answer the question: *Does constraint-based search prioritization perform better than the baseline symbolic execution (BSE)?* We have found out that all of our heuristics provided in Algorithm 1 perform better than the baseline symbolic execution (BSE), i.e., they extract more protocol information from the training firmware set in a given amount of time compared to BSE as shown in Figure 10, which shows the average number of unique constraints extracted over a period of 15 minutes (900s). Figure 11, 12, and 13 shows model extraction data for USB, Bluetooth HCI, and Bluetooth L2CAP protocols, respectively. The graphs compare the performance of the heuristics on firmware where BSE showed its best performance, i.e, BSE's number of extracted constraints were highest in the given time among all the firmware using that protocol. Our heuristics achieved much better performance than BSE in all cases.

The USB host request in the protocol is dependent on these five fields: *bmRequestType*, *bRequest*, *wValue*, *wIndex*, and *wLength*. For Bluetooth, the requests are based on

5. The MSP430 Bluetooth SDK contains 3rd party libraries that are in binary only form, which prevents us from generating the LLVM bitcode that was needed for symbolic execution.

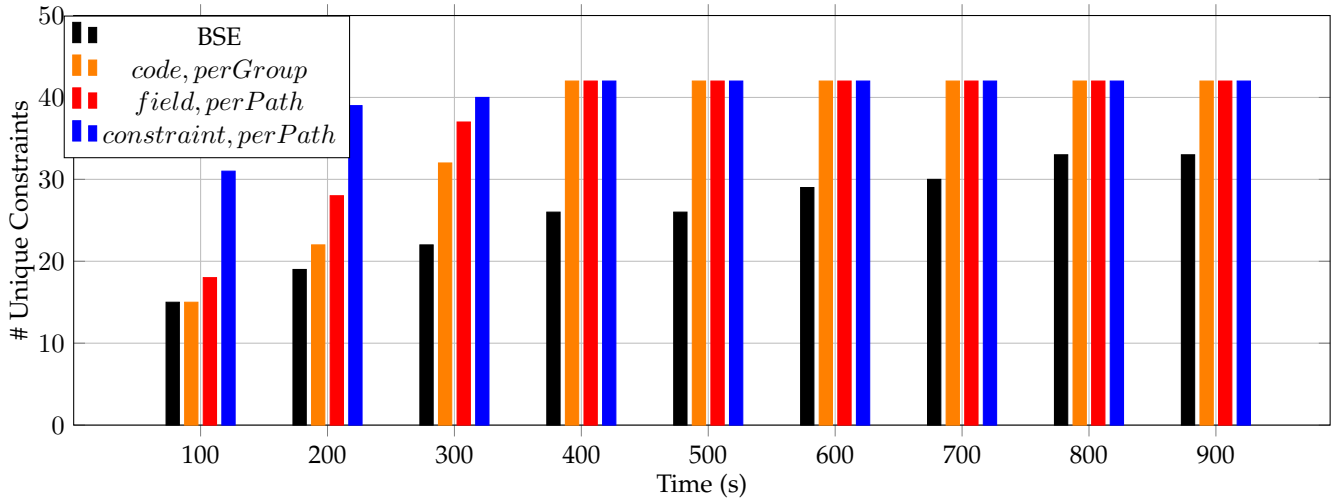


Fig. 10: Performance of heuristics compared to BSE. The figure shows average number of constraints recovered at different time intervals for MSP430 USB firmware.

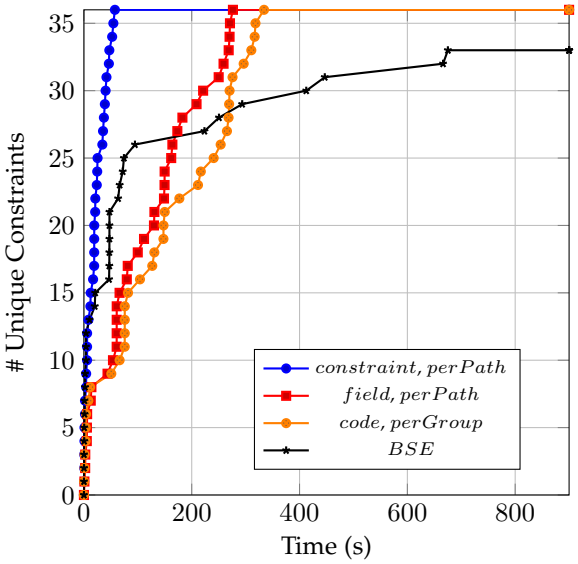


Fig. 11: Performance of different heuristics compared to baseline symbolic execution (BSE) for USB. C_{window} is 5 for the heuristics.

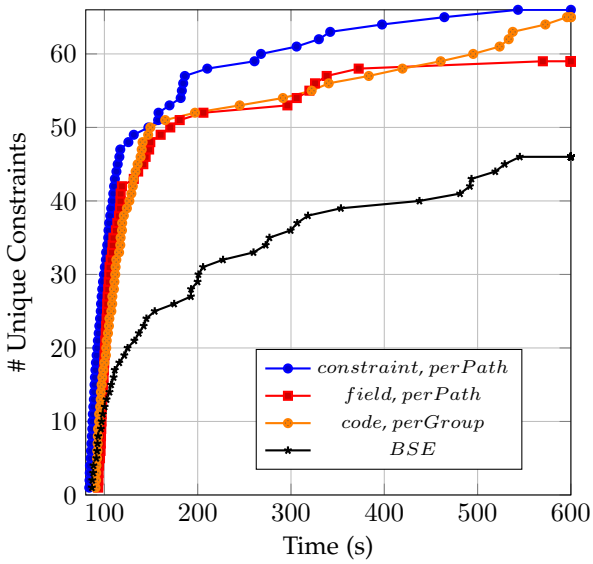


Fig. 12: Performance of different heuristics compared to baseline symbolic execution (BSE) for Bluetooth HCI. C_{window} is 5 for the heuristics.

Type	FW#	CBE			BSE		
		Min.	Max.	Avg.	Min.	Max.	Avg.
CDC	7	37	42	40	18	24	22
HID	12	34	41	38	19	33	27
MSC	4	34	39	37	27	30	29

TABLE 2: USB protocol constraints extracted from the training firmware set by Constraint Based Extraction (CBE) and Baseline Symbolic Execution (BSE). CBE corresponds to $C_{cov} = constraint, C_{scope} = perPath$. Since CBE achieved the best extraction results over time amongst our heuristics, we compare the results against BSE here. The diversity of the firmware set in terms of functionality can be seen from this table.

packet_type, *event_type*, HCI *command*, and the L2CAP signaling command code (*sig_cmd_code*). To extract the protocol model from a given firmware, first we manually find the addresses of these protocol relevant fields within that firmware and pass this information as an input to Algorithm 1 so that the relevant memory locations can be tracked throughout the symbolic execution. For USB, the protocol fields had the same name in the firmware as in the specification (lines 24, 31, and 32 in Figure 2). In Bluetooth firmware, the protocol fields corresponded to the various elements of an array named `hci_packet_with_pre_buffer` (line 20, Figure 4). So, the protocol fields could be easily mapped since the `hci_packet_with_pre_buffer` array was always used to pass data from the HCI layer to the L2CAP layer and the conditional statements in the firmware mostly used different elements of this array.

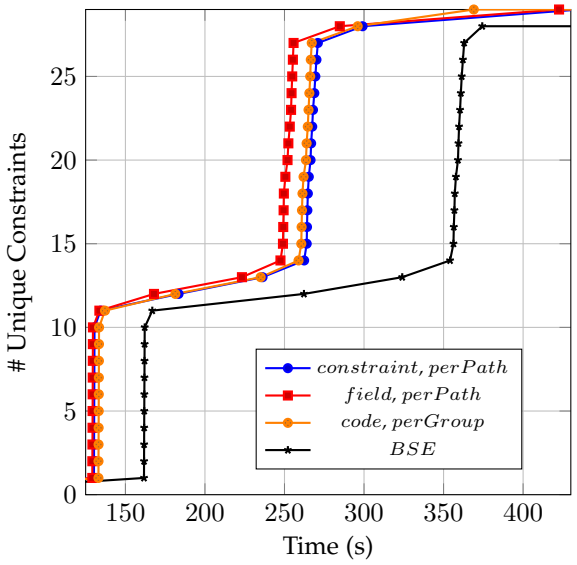


Fig. 13: Performance of different heuristics compared to baseline symbolic execution (BSE) for Bluetooth L2CAP. C_{window} is 5 for the heuristics.

We have used three configuration parameters, C_{cov}, C_{scope}, and C_{window}, to control the exploration during constraint extraction. C_{cov} denotes the coverage criteria, which can be $code$, $constraint$ or $field$ coverage. Code coverage is used to prioritize paths that cover new instructions, constraint coverage prioritizes paths that lead to more unique condition extraction, and field coverage prioritizes paths based on the visibility of multiple fields in a certain path.

Depending on the firmware's implementation, each of these criteria has unique advantages. For a small firmware with few branches just doing $code$ based coverage may be enough to extract information. For a large firmware with many branches, on the other hand, constraint coverage is more suitable, since it produced better yield in a short amount of time by prioritizing paths that provide new constraints. Field coverage showed faster extraction of constraints deeper in the code which involved multiple protocol fields.

C_{scope} denotes the scope for which the coverage is evaluated and can be $perPath$ or $perGroup$. C_{window} specifies the granularity of the window for executing the paths in the frontier set. It is given in terms of the number of basic blocks. C_{scope} combined with C_{window} determines the frequency of action taken based on C_{cov}. In a $perPath$ scoping, the frontier set gets updated every time new coverage is achieved. However, for $perGroup$ scoping, updating the frontier set is delayed until all paths have executed C_{window} number of blocks. There are six possible configurations based on the values of C_{cov} and C_{scope}. We have combined these with window sizes from one through ten. We observed that too small a window size does not let paths cover much and too large a window size lets all paths cover something new. We found window sizes of 4-7 performs better in all cases than others. The graph in Figure 11 shows results for $C_{window} = 5$. Due to space restrictions we will discuss the best performing combinations for each C_{cov} here. We should note that all configurations discussed below could extract on average 33 unique constraints for USB by the end of the analysis window, which was set to 900 seconds. For Bluetooth HCI and L2CAP layers the average number of unique constraints extracted are 59 and 29, respectively, within an analysis window of 900 seconds.

1) $C_{cov} = code, C_{scope} = perGroup$: This configuration is the closest to BSE. Focusing on paths with better code coverage reduced the number of paths to execute and led to more constraint extraction. As Figure 10 shows, this configuration extracts the least number of unique constraints compared to the other two configurations in the early phases of the analysis. Using $perPath$ scope with $code$ coverage achieves worse performance as only one path is chosen until it stops achieving new coverage. So, it may cause divergence from the protocol relevant parts of the code.
2) $C_{cov} = field, C_{scope} = perPath$: This configuration performs slightly better than configuration (1) as it extracts more constraints in the early phases as shown in Figure 10. Protocol field based extraction performs better than instruction coverage because it better relates to our goal. We choose paths based on finding new protocol fields. If a path has seen at least one new protocol field we expand on that path and stop executing it when it does not reach any new protocol field in the given window. Since the scope is per path, the window size only comes into play when there is no field coverage. This configuration performed the best for the Bluetooth L2CAP layer, which has a higher variation of constraints on different fields. So focusing on paths with different fields gets the constraints faster. For the Bluetooth HCI layer and the USB protocol it is beneficial to focus on the unique constraints found on the same field than variety of fields, that is why this configuration does not have the best results there.
3) $C_{cov} = constraint, C_{scope} = perPath$: In general, this configuration produces the best result as it extracts more constraints than the other configurations in a given amount of time as shown in Figures 10 - 12 for USB and Bluetooth HCI. This approach is most related to the goal of extraction. Since we want to extract as many unique constraints as possible from a firmware, we choose paths that provide us with constraints that have not been seen yet. Only the constraints on protocol fields are considered while evaluating paths. The $perGroup$ scope combined with constraint coverage performs slightly worse than (3) as it executes more paths in the same time which delays extraction compared to $perPath$ scope and performs much better than all other combinations.

6.1.1 Ground Truth Evaluation

We extracted 58 unique constraints in total across three different USB classes; see Table 2 for the number of constraints for each class of firmware. It is important to note that the extracted constraints come in different formats. For example, two constraints on *bmRequestType* are $((bmRequestType$ &

USB	*bmRequestType*	*bRequest*	*wValue*	*wIndex*	*wLength*
CDC	10	12	6	4	5
HID	9	10	5	3	10
MSC	10	11	5	3	4

BT	packet_Type	event_Type	sig_cmd_code	commar
HCI	3	27	-	12
L2CAP	-	7	6	-

TABLE 3: Unique values contained in the equality constraints for each protocol field in the extracted model from the training set for both USB and Bluetooth (BT). Equality constraints directly compare the data read from a memory location to a specific value.

Type	USB 2.0	CDC 1.2	HID 1.11	MSC 1.3
***bmRequestType* Coverage**				
CDC	6/6	2/2	–	–
HID	5/6	–	3/4	–
MSC	5/6	–	–	2/2
***bRequest* Coverage**				
CDC	9/11	8/39	–	–
HID	9/11	–	7/8	–
MSC	9/11	–	–	3/5

TABLE 4: *bmRequestType* and *bRequest* value coverage comparing with different USB specifications based on the 23 firmware in the training set.

Field	BT 4.0	BTStack	Extracted
packet_type	4	4	3
command	137	15	12
event_type	66	40	27
sig_cmd_code	23	16	5

TABLE 5: Value coverage for Bluetooth fields in the training set. BT 4.0 and BTStack lists the number of possible values for the field in Bluetooth 4.0 specification and the number of values listed in BTStack core. Extracted shows the number of values extracted by ProXray.

$128) \neq 0)$ and $(bmRequestType == 161)$. Both constraints appear on the same path and the latter satisfies the former. The first is used to identify the direction of the request i.e., from host to device or device to host. The second is used to identify the specific request. We examine all extracted constraints to find all possible values for each protocol field contained, and list the number of unique values for each field in Table 3. Both *bmRequestType* and *wIndex* share the same possible values among all these classes. For *wIndex*, the variation is limited. In most cases, the default value is zero. For *bmRequestTypes*, which tells the data transfer direction, recipient, etc, each value can also be reused by different USB requests. This means that given enough USB requests, it is possible to enumerate every *bmRequestType* permitted for this USB class. *bRequest* shows minor differences among different classes due to the class-specific USB requests. Other fields vary since they heavily rely on the semantics of the USB request.

We then look into all the values found in the constraints for *bmRequestType* and *bRequest*. We first manually extract

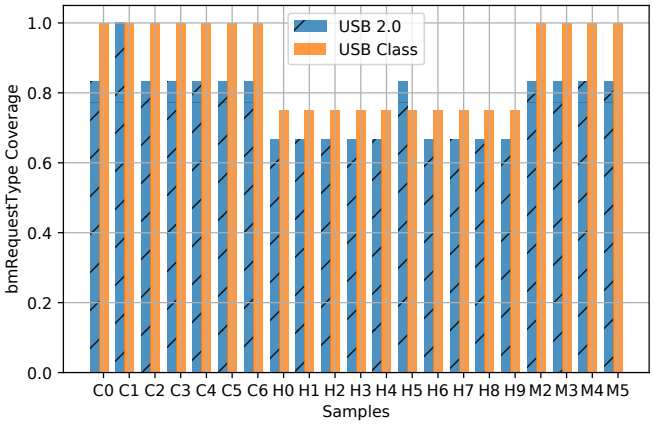


Fig. 14: bmRequestType value coverage comparing with different USB specifications for each firmware within the training set.

all possible values explicitly listed in different USB specifications for these fields. We compare the values found in the constraints with the ones included in the specifications. The coverage for these fields are shown in Table 4. For *bmRequestType*, all three classes of firmware show a minimum 83% and even 100% coverage on the standard USB 2.0. This is expected since different USB classes still follow the enumeration procedure using the standard USB requests. We also find 75% and 100% coverages on CDC 1.2, HID 1.11, and MSC 1.3 class-specific protocols accordingly based on different classes of the firmware.

For *bRequest*, all these three classes of firmware show an 82% coverage on the standard USB requests defined by USB 2.0. The only missing ones are "SET_DESCRIPTOR" and "SYNCH_FRAME", which are optional or only used by audio streaming devices. CDC firmware shows the lowest coverage comparing to the CDC 1.2 specification due to a large number of requests defined by its four different sub specifications, including Public Switched Telephone Network (PSTN), Integrated Services Digital Network (ISDN), Ethernet Control Model (ECM), and Abstract Control Model (ACM). HID firmware demonstrates 87.5% coverage comparing to the HID 1.11 specification. Both the 2 requests defined by the MSC 1.3 specification but not covered by our MSC firmware, are for Lockable Mass Storage device, a different kind of MSC devices.

We further look into each firmware sample used during the model extraction, and investigate its *bmRequestType* and *bRequest* coverage. As shown in Figure 14, every firmware in our training set has over 60% coverage of the USB 2.0 specification and over 70% coverage of other USB class specifications respectively on *bmRequestType*. For *bRequest* shown in Figure 15, most firmware except the CDC class have over 50% coverage of different USB specifications. Again, the coverage limitation of the CDC class is due to its sub protocol variations defined by the spec. Note that we did not include *wValue* and *wIndex* in our ground truth study, because although important, these fields depend on *bRequest* rather than being self-contained.

Due to the complexity of the Bluetooth HCI and L2CAP layers, we focus on protocols fields *packet_type*, *command*,

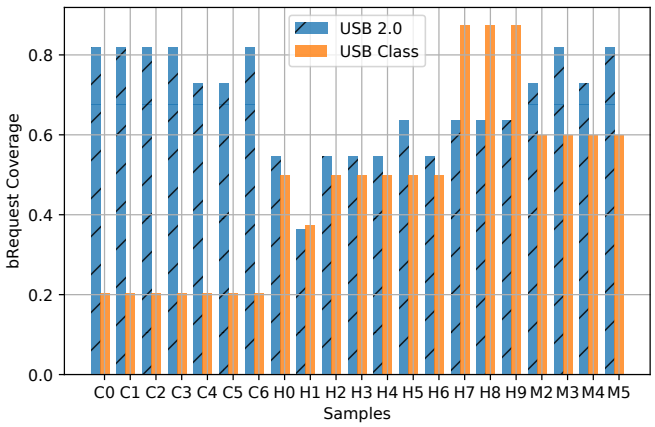


Fig. 15: bRequest value coverage comparing with different USB specifications for each firmware within the training set.

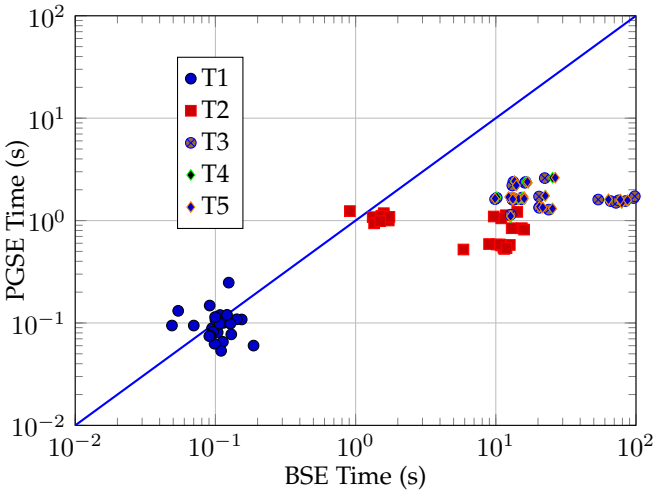


Fig. 16: Comparison of PGSE and BSE w.r.t. time to target for MSP430.

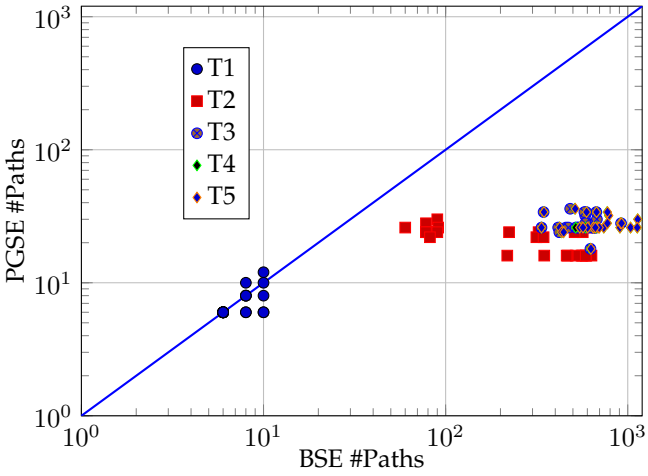


Fig. 17: Comparison of PGSE and BSE w.r.t. number of paths generated for MSP430.

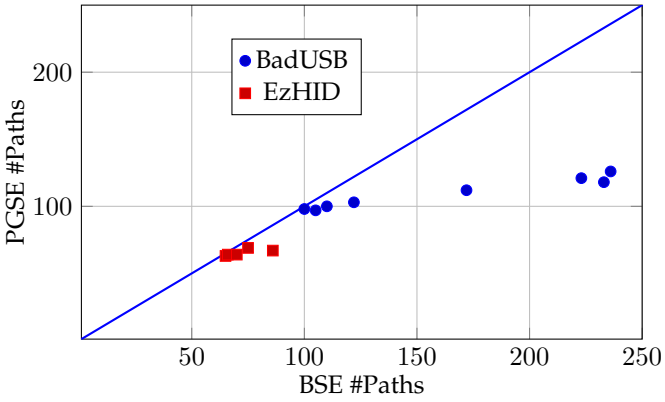


Fig. 18: Comparison of PGSE and BSE w.r.t. number of paths generated for BadUSB and ezHID

and *event_type* within HCI and *sig_cmd_code* within L2CAP. As shown in Table 5, for *packet_type* within HCI, we have successfully recovered 3 types out of 4. The only missing one is the "Command" type. We realize all the firmware examples we used do not build HCI Command packets directly. Instead, the BTStack lower level takes care of sending these packets automatically during the initialization. This is further proved by the recovery of *command* protocol fields within HCI Command packet. While the Bluetooth spec defines over 100 different commands, the BTStack only includes 15 of them and we are able to recover 12 with 80% coverage. For *event_type* within HCI Event packets, the BTStack again only covers 40 of 60 defined by the Bluetooth spec. The coverage of our extraction is 67.6% against the BTStack implementation. For the *sig_cmd_code* within L2CAP signaling packets, we have recovered 5 different values out of 16 defined by the BTStack implementation. A further investigation reveals a classic path explosion issue that prevents us from reaching another 5 different values.

In summary, although not 100% coverage for all possible values of each field defined by the specs, our constraints did a good job to cover the most common values of the most important fields (e.g., *bRequestType* and *command*), which help pinpoint the usage of the USB and Bluetooth protocols and potential functionality with a high confidence. Our study of the training set proves that each firmware does provide a lot of information about different specifications, which bases our model extraction methodology using firmware. Since we extracted the constraints from the most commonly available firmware images, other firmware images that contain less common functionality, e.g., PSTN, can help us to improve our constraint set. But those are relatively rare and less used. We also notice the limitations of firmware built upon certain protocol stack implementations, e.g., the BTStack implementation. In general, with the help of more diverse firmware collection, the coverage of our constraints is expected to get better.

6.2 Protocol Field Discovery

In this subsection, we answer the following research question: *Can we use the extracted constraint model to discover the program variables that correspond to the protocol fields in unknown firmware?*. One of our main contributions in this paper is automatic discovery of the protocol fields within a

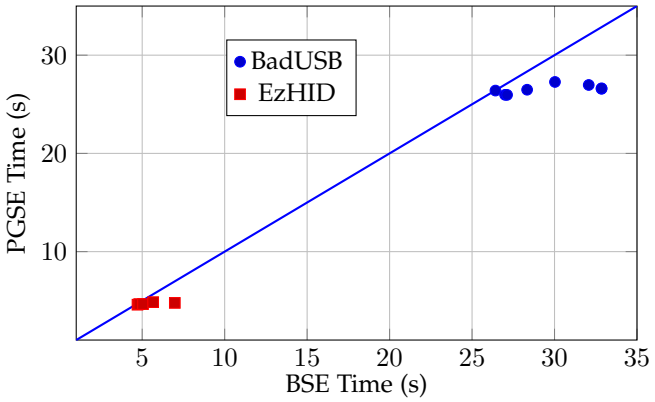


Fig. 19: Comparison of PGSE and BSE w.r.t. time to target for BadUSB and ezHID

Protocol Fields	Mapped Firmware Fields	Precision		
		Min.	Max.	Avg.
bmRequestType	**bmRequestType**	100%	100%	100%
bRequest	**bRequest**, wLength	0%	100%	75%
wValue	**wValue**	100%	100%	100%
wIndex	**wIndex**	100%	100%	100%
wLength	**wLength**	100%	100%	100%

TABLE 6: USB Field Mapping precision results for MSP430 (4) Composite firmware.

Protocol Fields	Mapped Firmware Fields	Precision		
		Min.	Max.	Avg.
bmRequestType	**bmRequestType**	100%	100%	100%
bRequest	**bRequest**, wLength	50%	100%	95.45%
wValue	**wValue**	100%	100%	100%
wIndex	**wIndex**	100%	100%	100%
wLength	**wLength**, wValue, bRequest	0%	100%	79.54%

TABLE 7: USB Field Mapping precision results for MSP430 (23) Test firmware.

Firmware	Protocol Field	Firmware Field	Precision
BadUSB	bmRequestType	wValue	0%
	bRequest	**bRequest**, wValue	50%
	wValue	**wValue**	100%
	wIndex	wLength, wValue	0%
	wLength	wValue	0%
EzHID	bmRequestType	**bmRequestType**	100%
	bRequest	**bRequest**	100%
	wValue	**wValue**	100%
	wIndex	SM0	0%
	wLength	bRequest, wValue	0%

TABLE 8: USB Field Mapping precision results for Intel 8051 firmware BadUSB and EzHID.

firmware address space. For an arbitrary firmware it may be difficult to identify the memory locations or variables that implement the protocol fields. The source code may not be available at all. Using Algorithm 5 explained in Section 4.2,

Protocol Fields	Mapped Firmware Fields	Precision		
		Min.	Max.	Avg.
packet_type	**packet_buffer[6]**, packet_buffer[7]	50%	100%	62.5%
event_type	**packet_buffer[7]**	100%	100%	100%
command	**packet_buffer[9]**, packet_buffer[12]	50%	100%	62.5%
code	**packet_buffer[15]**	100%	100%	100%

TABLE 9: Bluetooth Field Mapping precision results for MSP430 firmware. `packet_buffer` is an abbreviation for `hci_packet_with_pre_buffer` shown in Figure 4.

we could discover the protocol fields in our benchmarks with high precision. We compute the precision of matching a protocol field pf in a given firmware as

$$100 \times \frac{\begin{cases} 1 & rc \in M_{best}(pf) \\ 0 & \text{otherwise} \end{cases}}{|M_{best}(pf)|}$$

, where rc denotes the actual variable that implements pf and M_{best} is the output of Algorithm 5.

Tables 6 shows the field discovery data for the four composite firmware from the MSP430 SDK for USB. For USB, we could discover all fields except the *bRequest* field as sometimes another field, *wLength*, yielded a higher matching frequency. However, we achieve 100% precision in the discovery of the remaining protocol fields.

Table 8 shows the field discovery data for the Intel 8051 firmware. Compared to MSP430 benchmarks, we achieve less precision for protocol field discovery. However, we are able to discover one protocol field, wValue, for each Intel 8051 firmware with 100% precision and an additional protocol field, *bRequest*, for BadUSB firmware with a precision of 50%. In case of EzHID firmware the first 3 fields were mapped with 100% accuracy. We found that in all of the 0% precision cases, such as *wLength* for both, the actual field was matched as a candidate. However, the correct mapping was not selected as another variable that corresponded to a different field (*wValue* for BadUSB, *wValue, bRequest* for EzHID) had a higher score and prevented the correct candidate from being included in the final set.

We also evaluated field discovery for the firmware in the training set by excluding the firmware under analysis from the model extraction phase and by using the constraint model extracted from the remaining 22 firmware. Table 7 shows that we achieve similar precision values compared to those for the composite MSP430 firmware.

Table 9 shows the field discovery data for the four testing firmware for Bluetooth. In the case of Bluetooth, we could discover both the HCI and the L2CAP fields properly. The reason for lower precision for *packet_type* is due to the lack of variety in the values of those fields. For example, the set of values for *packet_type* has a lot common with the set of values for *event_type*. So, when a firmware uses the common values the mapping precision is lower. However, due to the high variety of values for *event_type* the mapping has 100% precision every time.

In terms of recall, we achieve 100% recall for the Bluetooth protocol. For the USB protocol, we achieved an aver-

Targets	Query (Q)	Time (secs)		#Paths	
		PGSE	BSE	PGSE	BSE
T1	(packet_type0 == $0x04$) ∧ (event_type0 == $0x04$)	173.315	174.571	61	61
T2	C(T1) ∧(packet_type1 == $0x04$) ∧ (event_type1 == $0x03$)	254.991	-	232	-
T3	C(T2) ∧(packet_type2 == $0x04$) ∧ (event_type2 == $0x0E$)	361.009	-	316	-
T4	C(T3) ∧(packet_type3 == $0x03$)	399.706	-	365	-
T5	C(T4) ∧(code == $0x02$)	412.259	-	398	-
T6	C(T4) ∧(code == $0x0A$)	414.205	-	404	-

TABLE 10: Comparison of PGSE and BSE for Bluetooth firmware w.r.t. time to target and the number of paths at the time of reaching the target. - denotes not reaching the target within a timeout of 2 hours.

age recall of 95% for the MSP430 USB firmware, a recall of 40% for BadUSB, and a recall of 60% for EzHID. The missed fields were not selected as the best matches although they appeared in the candidate set of mappings.

6.3 Protocol Guided Execution

Based on our extracted model, we can increase the efficiency of a symbolic execution engine in terms of exploring the required paths/code locations in a goal based execution, where the goal, or the *query* as shown in Figure 1, is described in terms of a generic protocol constraint. This is different from preconditioned symbolic execution, where the constraint is expressed in terms of the variables/memory locations of the system under test. Our field mapping phase automatically discovers potential mappings and rewrites the constraints of the query based on these potential mappings. So this phase involves the field mapping phase of Section 6.2 and uses the same experimental setup for USB that uses the 23 MSP430 USB firmware as the training set and six firmware from both MSP430 and Intel 8051 architecture as the testing set. Similarly for Bluetooth it has the same training and testing set as earlier.

6.3.1 Target Finding

In this subsection, we answer the following research question: *How effective is protocol-guided symbolic execution in driving symbolic execution to the protocol relevant targets in unknown firmware?*. To achieve this, we identified several protocol relevant targets in our benchmarks. For MSP430 USB firmware, we have identified protocol relevant targets T1-T5 shown in Table 11. As seen from Figure 17, we have achieved great reduction in the number of paths that are relevant to the protocol functionality of interest. Our approach achieves up to 73.8x speedup (T5 in C1) as shown in Figure 16. Our guided execution could reach T5 in a minimum of 1.106s (9.883s minimum for BSE) to a maximum of 2.624s (121.866s maximum for BSE). The number of paths in this case ranged from a minimum of 18 (336 for BSE) to maximum 36 (1134 for BSE). In case of timing for T1 there are a few cases where PGSE times are marginally higher than BSE. It is because of the depth of the code point corresponding to T1. Since the code location is not that deep within execution BSE and PGSE have similar results. The deeper the code location gets with respect to execution the better PGSE performs by eliminating irrelevant code. Table 12 and Table 13 shows the constraint based targets for Intel 8051 test firmware BadUSB and EzHID respectively. It took 126 paths in 26.63s to reach T8 from Table 12 compared to 236 paths in 32.852s for BSE. From Table 1 it is evident that BadUSB_Firmware has much lower number of lines of code compared to the MSP430 firmware. Smaller code is easier for BSE to explore compared to exploring code with a high line count and more conditional paths. In case of EzHID firmware we do not see much difference between PGSE and BSE. It took 67 paths in 4.78s to reach T5 from Table 13 compared to 86 paths in 6.98s for BSE. Figure 18 and Figure 19 shows the performance of PGSE compared to BSE for the Intel 8051 test firmware. In almost all cases, PGSE has shown improvement over BSE both in terms of number of paths and time to reach targets.

Targets	Query (Q)
T1	(bmRequestType & $0x80$) ≠ 0
T2	C(T1) ∧(bRequest == $0x06$)
T3	C(T2) ∧(wValue == $0x01$)
T4	C(T3) ∧(wIndex == 0)
T5	C(T4) ∧(wLength == 0)

TABLE 11: Constraints used for Protocol Guided Symbolic Execution of MSP430 firmware.

For the Bluetooth HCI and L2CAP layers the performance of PGSE is far superior from BSE. Table 10 shows the targets for Bluetooth. T1 - T4 are in the HCI layer and T5 - T6 are in the L2CAP layer. *event_type0* and *event_type1* represent the value of event type on 1st and 2nd iteration of the main event loop[6]. To reach the targets in the L2CAP layer, a specific sequence of events needs to occur in the HCI layer. The Bluetooth firmware code contains too many branches for BSE to achieve this sequence. For example, there are 27 cases for only *event_type*. Also, two types of interrupts get fired on every iteration of the main event loop. As a result, BSE could not reach any targets beyond T1. Since T1 is not that deep in the code and does not require a sequence of events, the performance of both PGSE and BSE is similar. As the execution went deeper in the code BSE could not keep up with PGSE. At the end of the two hour timeout BSE had spawned 52,289 paths but not the required one. PGSE was able to reach the all the targets with only 404 paths in 414.205 seconds.

6.3.2 Functionality Matching

In this subsection, we answer the following research question: *How precise is constraint model based functional classification on unknown firmware?*. Thus, as an additional capability of our protocol model extraction approach, we tried to

6. Note that FIE assigns version numbers to differentiate different versions of symbolic regions throughout the execution. So, we leverage this version numbering to refer to different iterations during executions of these event-based systems.

Targets	Query (Q)
T1	(bmRequestType & $0x40$) == 0
T2	(bmRequestType & $0x20$) == 0
T3	((bmRequestType & $0x60$) == 0)∧ (bRequest == $0x05$)
T4	((bmRequestType & $0x60$) == 0)∧ (bRequest == $0x09$)
T5	(bRequest == $0x06$) ∧ (wValue == $0x01$)
T6	(bRequest == $0x06$) ∧ (wValue == $0x02$)
T7	(bRequest == $0x06$) ∧ (wValue == $0x06$)
T8	(bRequest == $0x06$) ∧ (wValue == $0x22$)

TABLE 12: Constraints used for Protocol Guided Symbolic Execution of BadUSB firmware.

Targets	Query (Q)
T1	(bRequest == $0x06$) ∧ (wValue == $0x01$)
T2	(bRequest == $0x06$) ∧ (wValue == $0x02$)
T3	(bRequest == $0x06$) ∧ (wValue == $0x03$)
T4	(bRequest == $0x06$) ∧ (wValue == $0x21$)
T5	(bRequest == $0x06$) ∧ (wValue == $0x22$)

TABLE 13: Constraints used for Protocol Guided Symbolic Execution of EzHID firmware.

Type	Type Specific Constraints
CDC	bRequest ∈ $\{32, 33, 34\}$
HID	bRequest == 2 ∨ wValue ∈ $\{33, 34\}$
MSC	bRequest ∈ $\{254, 255\}$

TABLE 14: USB type specific constraints used in functionality matching.

identify the USB subclasses implemented by a given USB firmware. The intention here is to be able report the types of functionalities a firmware can support. For this purpose, we have first identified the class specific constraints in the model by automatically removing those that appear in more than one subclass and recorded the associated subclass type for each class specific constraint. In our benchmarks, we have come across three USB subclasses: CDC, HID, and MSC.

Table 14 shows CDC, HID, and MSC specific constraints that we extracted from the 23 MSP430 firmware.

Next the protocol field discovery of the test firmware is done in the same process as explained in Algorithm 5. Guided symbolic execution is then used to find functionality by matching the type of class specific constraints shown in Table 14. All paths in the firmware are evaluated against these constraints. If a firmware is found to conform to a specific constraint of any functionality type, we report that the firmware implements that functionality. We have correctly identified the functionality type for every MSP430 and Intel 8051 firmware in our benchmarks using this process including matching all functionalities of the 4 composite firmware. The importance of this can be seen in case of BadUSB firmware. We found that it implements HID functionalities in addition to its reported MSC functionality which indicates that the firmware has a malicious aspect.

6.4 Usability

The applicability of our approach is not limited to the USB and the Bluetooth protocols. It can be applied to any protocol that processes requests that are structured as a set

Firmware	Matched Constraints	Class
EzHID	wValue == 33	HID
	wValue == 34	HID
BadUSB	bRequest == 254	MSC
	wValue == 34	HID

TABLE 15: Functionality matching for BadUSB firmware and EzHID firmware.

of attribute and value pairs. The protocol model extraction phase of our approach requires some domain knowledge. Specifically, the analyst should understand whether the protocol involves multiple layers and the format of the packet types for each layer. In our study, we consulted the core specifications of both protocols to achieve this knowledge.

This level of understanding of the protocol will guide the analyst in getting a better understanding of the corresponding SDKs. The complexity of the implementation is proportional to the complexity of the protocol. In our case, understanding the BTStack framework that implements the Bluetooth protocol took more time than understanding the MSP430 USB SDK. We should acknowledge that the documentation plays a significant role and BTStack's documentation was very helpful in this regard.

The most important stage in understanding a protocol implementation is to identify the data structures used for the packet types of the protocol. In our experience with these two implementations, we found out that the developers try to use the terminology from the protocol as identifiers in their code. For instance, MSP430 SDK uses the identifier `tSetupPacket` to represent the Setup packet for the USB protocol as shown in Figure 2 (line 24). Similarly, BTStack framework uses the identifier `hci_packet` to represent the HCI packet for the Bluetooth protocol as shown in Figure 4 (line 20). So, we needed to identify these two data structures for all the training firmware we used in our experiments as they all used these specific data structures defined by the relevant libraries.

Despite the ease of finding the data structures that correspond to the protocol specific packet types, the data-flow in these protocol implementations is typically complicated as can be seen in the sample snippets of code provided in Figures 2 and 4. This is due to pointer arithmetic and using function pointers to implement callbacks for various event types in the protocol. These programming constructs require a precise analysis to avoid false positives. Dynamic symbolic execution provides a precise memory model and proved to be effective for protocol constraint model extraction.

We think that as more microcontoller (MCU) architectures get fully supported by the state-of-the-art symbolic execution engines such as FIE, more code bases can be explored for model extraction and model guided analysis. In this study, we have been restricted to MSP430 and 8051 architectures as these were the only MCUs that were supported by FIE with our extensions from [13]. Another challenge is the proprietary code that comes with the SDKs. As an example, we were not able to use TI's MSP430 Bluetooth SDK due to having some 3rd party libraries in binary only

form. This prevented us from generating the LLVM bitcode for that SDK, which was needed for symbolic execution.

Finally, any mistakes related to the setup of the model extraction phase, e.g., identifying a wrong data structure for the packet type, could lead to the extraction of wrong protocol constraints. A low-quality constraint model would lead to a low-quality field discovery and, hence, to a misleading analysis result during protocol model guided analysis. Another issue could be the implementation mistakes in the SDKs. One way to deal with these issues is to use a variety of SDKs and a diverse set of firmware examples. We think that with better support for diverse set of MCU types, it will be practical to analyze a variety of SDKs and extract high quality protocol models.

7 Discussion

In this paper, we have focused on the USB protocol, and, specifically, the USB (control) requests issued by the host, and the HCI and L2CAP layers of the Bluetooth protocol. It should be sufficient to analyze different device firmware to capture these parts of the USB and the Bluetooth protocols. Our training set was restricted to MSP430 firmware as FIE has been specialized for this microcontroller architecture. Although our evaluation is based on sample firmware implementations from the vendor rather than the USB and Bluetooth firmware in the wild, these non-toy implementations provide a strong base of how the real-world firmware would be implemented.

ProXray also needs the source files of the firmware to map program variables into protocol fields during the training phase. However, access to the source file might be infeasible due to various reasons. It is not uncommon to have only the binary format of firmware available. Fortunately, it is still possible to recover the protocol fields from the binary using static analysis. One way is to look for certain binary patterns from within the binary, e.g., the binary pattern of a USB setup packet. We will include binary parsing/mapping support in our future work.

Threats to Validity: Threats to internal validity include not having a training set that is comprehensive in terms of the USB subclass functionality. However, despite this limitation, our approach could precisely discover the functionality of all USB firmware in the test set. We think that the quality of the extracted model can be improved by enriching the training set. Threats to external validity include application of ProXray to the USB and Bluetooth protocols, which do not involve relational constraints, e.g., $p_1 \ ROP \ p_2$, where p_1 and p_2 are protocol fields. Although our implementation focuses on constraints that relate some protocol field to some constant value, the implementation of our approach can be easily adapted to protocols that involve relational constraints by incorporating 1) prioritization heuristics that can compare two paths w.r.t. the potential for covering richer relational constraints in the model extraction phase and 2) matching heuristics that consider pairwise associations in the potential mappings for the protocol field discovery phase.

8 Related Work

Protocol Model Learning: Black-box model learning has been used for extracting models of the Session Initiation Protocol [32], the Europay-MasterCard-Visa protocol (EMV) [33], the Transmission Control Protocol (TCP) [34], [35], the Transport Layer Security (TLS) protocol [36], a botnet command and control protocol [37], a smart-card reader [38], and the Secure Shell Protocol (SSH) [39]. White-box model learning has been combined with predicate abstraction and symbolic execution in [40] to systematically consider the input space for the learned protocol model. MACE [41] extracts finite state machines (FSMs) from reference implementations in an incremental way by using the extracted FSM to guide concolic execution and by using guided concolic execution to enrich the model. Our approach focuses on extraction of the protocol constraints rather than the underlying FSM of the protocol, which may not be relevant to protocols such as the USB protocol as the specification explicitly advises the firmware developers to handle any type of request at any time during the communication with the host. Also, we use the extracted models to reverse engineer unknown protocol implementations.

USB Protocol Modeling: In the hardware/software co-design, USB has been modeled as a 8-bit wide channel, where each transfer takes 8 cycles [42]. This bandwidth- and timing-based modeling was designed to ease the partitioning during the co-design, but ignored the semantics of the USB protocol. In the USBFirewall [43] work a DSL is designed to describe the syntax of the USB protocol. This DSL is used to compile into C code using Haskell, generating a "formal" USB packet parser for FreeBSD. The USB Type-C Authentication protocol [44] is formally verified in USBSoK [45] using ProVerif [46] . Unlike previous work, we build a constraint-based model based on standard USB request/response message structure. This model can be readily used by a SAT solver.

Model Extraction: There have been some interest in (semi-)automatically extracting models from software, which include extraction of relationships between features and computational units [14], object models [47], [48], finite-state models [49], [50], [51], dependence models [52], test scenarios conforming to some temporal logic formula defined over a data-flow graph [53], state predicates to guide state exploration of a cooperating model checker [54], a statistical language model [55], framework models for symbolic execution [56], symbolic data models for web applications [57], and behavioral models for USB firmware [13]. Our approach uses symbolic execution to extract protocol relevant constraints from firmware and it uses these constraints to learn a constraint-based model for the protocol of interest. Our approach differs from related work in that the extracted model is not used to improve the analysis of firmware from which the models get extracted, which we refer to as the training set, and is used to analyze unknown firmware that is known to implement the same protocol as the firmware in the training set.

Firmware Analysis: Symbolic execution has been used for firmware analysis in FIE [9], AVATAR [10], Firmalice [11], and FirmUSB [13]. FIE models the reactive nature of firmware via scheduling interrupt functions at various gran-

ularities. AVATAR uses the S2E symbolic execution engine to run firmware binaries in an emulator while forwarding I/O requests to the physical device and processing the responses from the device through state migration. S2E is further used [58] to generate test-cases for System Management Mode interrupt handlers in BIOS implementations for Intel-based platforms. Firmalice combines symbolic execution and program slicing to discover backdoors and their triggers in firmware images. FirmUSB uses static analysis and incremental symbolic execution to discover symbolic memory regions and uses domain-specific constraints for guided symbolic execution. Our approach can support these and any symbolic execution based firmware analysis with a priori extracted protocol constraint information.

Symbolic Execution with Pruning/Prioritization: Redundant paths are pruned in [59] based on the program point, the concrete write set, the path condition, and the variables that would be read after that program point. The generational search approach in [60] executes a set of child paths and uses code coverage information for prioritizing their exploration. Structural path coverage is used in [61] to prioritize the paths. In [62], structural path information is enriched with data-flow facts to include only the relevant program locations in path coverage. Dependence analysis is used in [63] to perform pruning based on symbolic path equivalence. The goal of our pruning heuristics is to achieve high coverage of the protocol of interest and, hence, our heuristics consider canonical constraints that involve the protocol fields. Our approach is similar to the generational search presented in [60] in that it also executes a set of child paths and uses coverage information for evaluating their usefulness. However, our approach also considers constraint coverage and prioritizes symbolic paths rather than feasible inputs.

Guided Symbolic Execution: Program dependence information and user provided abstraction strategies are used in [64] to compute equivalence classes of paths that can reach a specific program location. [65] uses dynamically computed relevant slice conditions to explore paths that are relevant to a slicing criteria. Control flow analysis and weakest-precondition computation are combined in [66] to generate tests that exercise code changes due to patches. Interpolation and subsumption checking are used in [67] to prune paths that do not lead to buggy program locations. Approximate weakest preconditions are used in [68] to prevent scheduling of paths that are guaranteed not to violate assertions in multithreaded programs. Abstract reachability graphs computed by a predicate abstraction based model checker are used in [69] to avoid exploration of infeasible paths. The program analysis phase of our approach discovers associations between the protocol fields and the memory locations. This information is used in guiding symbolic execution by prioritizing paths that produce constraints of the protocol model.

In [70] constraint normalization is used to efficiently support satisfiability and model counting queries for string manipulating programs. In our approach, constraint normalization enables efficient computation of constraint coverage and simplifies the formal representation of the protocol.

9 Conclusion

We presented a methodology for protocol model guided analysis using symbolic execution. We applied our approach to MSP430 and Intel 8051 firmware in the context of the USB and Bluetooth protocols. We extract a constrained-based model of the protocol using symbolic execution. Our path pruning and prioritization heuristics perform 1.54 and 1.76 times better than the baseline symbolic execution in terms of the number of unique constraints extracted in the context of the USB and Bluetooth protocols, respectively. Our constraint-based protocol model enables analysis of new firmware without prior knowledge of how it uses the protocol. Our protocol field discovery can find a mapping for each of the protocol fields and achieves high precision for the most significant fields of both protocols. Our protocol guided symbolic execution achieves up to 73.8 times speedup for unknown USB firmware and more than an order of magnitude speedup for unknown Bluetooth firmware in reaching the parts of the code that are relevant to a given protocol constraint. In future work we will explore other protocols as well as firmware for other microcontroller architectures.

Acknowledgments

This work is supported by the US National Science Foundation under grant CNS-1815883 and by the Semiconductor Research Corporation.

References

[1] Knud Lasse Lueth, "State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating," https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/, 2018.
[2] "Mirai (malware)," https://en.wikipedia.org/wiki/Mirai_(malware).
[3] S. Bluetooth, "Bluetooth core specification version 5.0," *Specification of the Bluetooth System*, 2016.
[4] LXR, "Unusual Devices," https://lxr.missinglinkelectronics.com/linux/drivers/usb/storage/unusual_devs.h, 2018.
[5] Syzkaller, "Found Linux kernel USB bugs," https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs_usb.md, 2018.
[6] armis lab, "BlueBorne," https://www.armis.com/blueborne/, 2017.
[7] ——, "BleedingBit," https://armis.com/bleedingbit/, 2018.
[8] K. Nohl and J. Lell, "BadUSB–On accessories that turn evil," *Black Hat USA*, 2014.
[9] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 463–478. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson
[10] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.
[11] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *NDSS*, 2015.
[12] I. Pustogarov, T. Ristenpart, and V. Shmatikov, "Using program analysis to synthesize sensor spoofing attacks," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, 2017, pp. 757–770.

[13] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. R. B. Butler, "Firmusb: Vetting USB device firmware using domain informed symbolic execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 2245–2262.

[14] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 210–224, March 2003.

[15] F. Vaandrager, "Model learning," *Commun. ACM*, vol. 60, no. 2, pp. 86–95, Jan. 2017. [Online]. Available: http://doi.acm.org/10.1145/2967606

[16] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13, 2005.

[17] C. Cadar and D. Engler, "Execution generated test cases: How to make systems code crash itself," in *Proceedings of the 12th International Conference on Model Checking Software*, ser. SPIN'05, 2005.

[18] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, 2008, pp. 209–224.

[19] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[20] T. Instruments, "Msp430 usb developers package," http://www.ti.com/tool/MSP430USBDEVPACK, 2017, online; accessed March 7th 2018.

[21] D. Kroening and O. Strichman, *Decision Procedures - An Algorithmic Point of View, Second Edition*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.

[22] M. Ringwald and M. Ringwald, "BTStack," https://github.com/bluekitchen/btstack, last accessed 20 March 2019.

[23] A. Caudill and B. Wilson, "Phison 2251-03 (2303) Custom Firmware & Existing Firmware Patches (BadUSB)," *GitHub*, vol. 26, Sep. 2014.

[24] Compaq, Hewlett-Packard, Intel, Microsoft, NEC, and Phillips, "Universal Serial Bus Specification, Revision 2.0," April 2000.

[25] Hewlett-Packard and Intel and Microsoft and NEC and ST-NXP Wireless and Texas Instruments, "Universal Serial Bus 3.0 Specification, Revision 2.0," November 2008.

[26] The USB Device Working Group, "USB Class Codes," http://www.usb.org/developers/defined_class, 2015.

[27] Bluetooth SIG, "Bluetooth core specification version 4.2," *Specification of the Bluetooth System*, 2014.

[28] S. Bluetooth, "Bluetooth mesh networking specifications," *Specification of the Bluetooth System*, 2017.

[29] ——, "Bluetooth core specification version 5.0," *Specification of the Bluetooth System*, 2016.

[30] "The reference manual for the kquery language," https://klee.github.io/docs/kquery/.

[31] Arnim Laeuger, "The EzHID Firmware Project," http://ezhid.sourceforge.net/, 2015.

[32] F. Aarts, B. Jonsson, and J. Uijen, "Generating models of infinite-state communication protocols using regular inference with abstraction," in *Proceedings of the 22Nd IFIP WG 6.1 International Conference on Testing Software and Systems*, ser. ICTSS'10, 2010, pp. 188–204.

[33] F. Aarts, J. De Ruiter, and E. Poll, "Formal models of bank cards for free," in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW '13, 2013, pp. 461–468.

[34] S. Cassel, F. Howar, B. Jonsson, and B. Steffen, "Active learning for extended finite state machines," *Form. Asp. Comput.*, vol. 28, no. 2, pp. 233–263, Apr. 2016.

[35] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager, *Combining Model Learning and Model Checking to Analyze TCP Implementations*. Cham: Springer International Publishing, 2016, pp. 454–471.

[36] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C., 2015, pp. 193–206.

[37] C. Y. Cho, D. Babi ć, E. C. R. Shin, and D. Song, "Inference and analysis of formal models of botnet command and control protocols," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10, 2010, pp. 426–439.

[38] G. Chalupar, S. Peherstorfer, E. Poll, and J. de Ruiter, "Automated reverse engineering using lego®," in *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. San Diego, CA: USENIX Association, 2014. [Online]. Available: https://www.usenix.org/conference/woot14/workshop-program/presentation/chalupar

[39] P. Fiterau-Brostean, T. Lenaerts, E. Poll, J. de Ruiter, F. W. Vaandrager, and P. Verleg, "Model learning and model checking of SSH implementations," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017*, 2017, pp. 142–151.

[40] N. Kothari, T. Millstein, and R. Govindan, "Deriving state machines from TinyOS programs using symbolic execution," in *Proceedings of the 7th International Conference on Information Processing in Sensor Networks*, ser. IPSN '08, 2008, pp. 271–282.

[41] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, "MACE: model-inference-assisted concolic exploration for protocol and vulnerability discovery," in *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*, 2011.

[42] P. V. Knudsen and J. Madsen, "Integrating communication protocol selection with partitioning in hardware/software codesign," in *Proceedings of the 11th international symposium on System synthesis*. IEEE Computer Society, 1998, pp. 111–116.

[43] P. Johnson, S. Bratus, and S. Smith, "Protecting Against Malicious Bits On the Wire: Automatically Generating a USB Protocol Parser for a Production Kernel," in *Proceedings of the 33th Annual Computer Security Applications Conference*, ser. ACSAC '17, 2017.

[44] USB 3.0 Promoter Group, "Universal Serial Bus Type-C Authentication Specification, Revision 1.0," March 2016.

[45] D. J. Tian, N. Scaife, D. Kumar, M. Bailey, A. Bates, and K. R. Butler, "Sok: "plug & pray" today – understanding usb insecurity in versions 1 through c," in *Security and Privacy (SP), 2018 IEEE Symposium on*. IEEE, 2018.

[46] B. Blanchet, V. Cheval, X. Allamigeon, and B. Smyth, "ProVerif: Cryptographic protocol verifier in the formal model," *URL http://prosecco. gforge. inria. fr/personal/bblanche/proverif*, 2010.

[47] D. Jackson and A. Waingold, "Lightweight extraction of object models from bytecode," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99, 1999, pp. 194–202.

[48] G. Ramalingam, R. Komondoor, J. Field, and S. Sinha, "Semantics-based reverse engineering of object-oriented data models," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, 2006, pp. 192–201.

[49] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng, "Bandera: Extracting finite-state models from java source code," in *Proceedings of the 22Nd International Conference on Software Engineering*, ser. ICSE '00, 2000, pp. 439–448.

[50] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, Robby, H. Zheng, and W. Visser, "Tool-supported program abstraction for finite-state verification," in *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada*, 2001, pp. 177–187.

[51] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, 2001, pp. 203–213.

[52] D. Jackson and E. J. Rollins, "A new model of program dependences for reverse engineering," in *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '94, 1994, pp. 2–10.

[53] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural, "Data flow testing as model checking," in *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, 2003, pp. 232–243.

[54] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler, "Conditional model checking: A technique to pass information between verifiers," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, 2012, pp. 57:1–57:11.

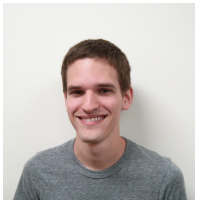
[55] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2013, pp. 532–542.

[56] J. Jeon, X. Qiu, J. Fetter-Degges, J. S. Foster, and A. Solar-Lezama, "Synthesizing framework models for symbolic execution," in *Pro-*

ceedings of the 38th International Conference on Software Engineering, ser. ICSE '16, 2016, pp. 156–167.
[57] I. Bocic and T. Bultan, "Symbolic model extraction for web application verification," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 724–734.
[58] O. Bazhaniuk, J. Loucaides, L. Rosenbaum, M. R. Tuttle, and V. Zimmer, "Symbolic execution for bios security," in *Proceedings of the 9th USENIX Conference on Offensive Technologies*, ser. WOOT'15, 2015, pp. 8–8.
[59] P. Boonstoppel, C. Cadar, and D. Engler, "Rwset: Attacking path explosion in constraint-based test generation," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08, 2008, pp. 351–366.
[60] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008.
[61] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13, 2013, pp. 19–32.
[62] H. Seo and S. Kim, "How we get there: A context-guided search strategy in concolic testing," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 413–424.
[63] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence guided symbolic execution," *IEEE Trans. Softw. Eng.*, vol. 43, no. 3, Mar. 2017.
[64] R. Santelices and M. J. Harrold, "Exploiting program dependencies for scalable multiple-path symbolic execution," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10, 2010, pp. 195–206.
[65] D. Qi, H. D. T. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, Oct. 2013.
[66] P. D. Marinescu and C. Cadar, "KATCH: high-coverage testing of software patches," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, 2013, pp. 235–245.
[67] J. Jaffar, V. Murali, and J. A. Navas, "Boosting concolic testing via interpolation," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2013, pp. 48–58.
[68] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta, "Assertion guided symbolic execution of multithreaded programs," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 854–865.
[69] P. Daca, A. Gupta, and T. A. Henzinger, "Abstraction-driven concolic testing," in *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, 2016, pp. 328–347.
[70] T. Brennan, N. Tsiskaridze, N. Rosner, A. Aydin, and T. Bultan, "Constraint normalization and parameterized caching for quantitative program analysis," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, pp. 535–546.

Farhaan Fowze received his B.Sc. from Bangladesh University of Engineering and Technology(BUET) in 2012. He is a Ph.D. candidate in the Department of Electrical and Computer Engineering in University of Florida. His research interests include model extraction, binary analysis, and program analysis. Contact at farhaan104@ufl.edu .

Dave (Jing) Tian is an Assistant Professor in the Department of Computer Science at Purdue University. His research involves operating system security, embedded system security, and trusted computing. He received his Ph.D. in computer science from the University of Florida in 2019. For more information, please access https://davejingtian.org.



Grant Hernandez is a Ph.D. Research Assistant in the Computer and Information Science and Engineering department at the University of Florida. His research focuses on automated embedded binary firmware analysis to discover vulnerabilities at scale. Contact him via email grant.hernandez@ufl.edu.

Kevin Butler is an Associate Professor of Computer and Information Science and Engineering at the University of Florida, and Associate Director of the Florida Institute for Cybersecurity Research. His research focuses on establishing the trustworthiness of computer systems and embedded devices. Butler received at Ph.D. in computer science and engineering from the Pennsylvania State University in 2010. He is a Senior Member of IEEE and ACM.

Tuba Yavuz Tuba Yavuz is an Assistant Professor at the Electrical and Computer Engineering Department of University of Florida. Her research interests include model extraction, model checking, and program analysis with applications to cyber-security. Yavuz received a Ph.D. in computer science from the University of California of Santa Barbara. She has been on the program committee of ICSE'19 and Usenix Security'19. She is a member of the IEEE and ACM. Contact her at tuba@ece.ufl.edu address.