Automata Learning for Symbolic Execution

Bernhard K. Aichernig, Roderick Bloem, Masoud Ebrahimi, Martin Tappler, Johannes Winter
Graz University of Technology, Austria

***Abstract*—Black-box components conceal parts of software execution paths, which makes systematic testing, e. g., via symbolic execution, difficult. In this paper, we use automata learning to facilitate symbolic execution in the presence of black-box components. We substitute black-boxes in a software system with learned automata that model them, enabling us to symbolically execute program paths that run through black-boxes. We show that applying the approach on real-world software systems incorporating black-boxes increases code coverage when compared to standard techniques.**

I. Introduction

Symbolic execution is a method to analyze software systems. It has gained attention since its introduction in the 1970s [1, 2] and is used in testing, invariant detection, model checking, and proving software correctness [3, 4, 5, 6]. Symbolic execution achieves high test coverage in a setting where the source code is completely available.

In practice, many software systems incorporate black-box components for which the source code is not available (e. g., third-party software units, hardware peripherals). A thorough behavioral analysis in the presence of black-box components is challenging using methods like symbolic execution, because black-box components conceal parts of software execution. To symbolically execute such software systems, Cadar et al. [7] proposed to replace the calls to a black-box component with calls to manually written stubs that model the component's behavior. This is a very challenging and error-prone task because either one does not have access to documentations of black-box components or this labor-intensive effort is not worth it since the resulting model will only be used once. Consequently, we often use methods that are applicable in the presence of black-boxes; e. g., random testing, or model-based testing, which requires behavioral models. Alternatively, symbolic execution may be combined with concrete execution of black-box paths, such an approach is for instance used by concolic execution [4, 8, 9].

In this paper, we use automata learning to enable symbolic execution in the presence of black-box components. Figure 1 depicts the overall execution flow for our proposed setting. Given a System Under Test (SUT), divided into a white-box and a black-box component, we learn a finite-state machine (FSM) model of the black-box component, and compose it with the white-box to generate system-level test cases via symbolic execution; i. e., we replace the black-box with its model for test-case generation. Finally, we execute the generated test cases on the original system incorporating the black-box component itself to obtain high coverage on the white-box. Testing software units in isolation often results in a broad set of test cases that are not worth the effort of manual inspection. On the other hand, systematic testing of interactive software units results in a reduced number of system-level test cases. An advantage of the proposed approach is that it enables systematic testing in the presence of black-boxes.

This work was supported by the Graz University of Technology's LEAD project "Dependable Internet of Things in Adverse Environments". This work was partially supported by the ECSEL Joint Undertaking (ENABLE-S3, grant no. 692455), by the European Union (IMMORTAL, grant no. 644905).

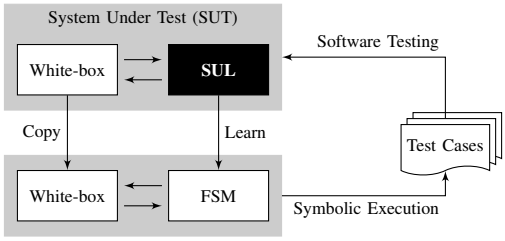

Fig. 1: System-level testing in the presence of a black-box. The system under test (SUT) comprises a white-box and a black-box component that is the system under learning (SUL).

Our approach is currently applicable to black-boxes that we can model as FSMs, and not applicable to other types of black-boxes like arithmetic functions. Moreover, there are concerns about the practicality of automata learning mostly due to the abstraction layer to counter state space explosion. Meanwhile, automata learning is successfully applied for systems with thousands of states [10] and there are techniques to support up to a million states [11]. Finally, learned FSMs of small abstract state space are shown to be sufficient for many interesting scenarios [12, 13] and this paper elaborates on one.

We built our method on top of KLEE [4], and LearnLib [14]. Applying our approach to a variety of real-world scenarios showed not only the test coverage increases, but the testing time also decreases, both compared to concolic execution. Results show coverage increase for units of interest in three real-world software systems that are dependent on an SPI controller (52.94%), an MQTT Broker (5.9% & 8.36%), and an SD-Card controller (75.36%).

Outline. This paper has the following structure. Section II summarizes automata learning and symbolic execution. Section III explains how to learn an automaton from a black-box and use it to execute the software system incorporating that black-box symbolically. Sections IV to VI provide case studies demonstrating the applicability of our approach in real-world scenarios. Section VII covers related work. Section VIII concludes and discusses future research directions.

II. Preliminaries

A. Symbolic Execution

To infer what inputs cause which parts of a program to execute, symbolic execution assigns symbolic values to input variables and then explores the control-flow of the program [1, 2, 3, 4]. By keeping track of the program counter and constraints on symbolic input values, an execution engine discovers how inputs influence the execution path. Along each execution path, symbolic execution collects constraints from branch conditions and forms a conjunction of these constraints, called path condition. An execution path is feasible if its path condition is satisfiable; thus, a constraint solver can reveal with which input values an execution path is feasible and with which input values it is not. A feasible execution path represents multiple program runs whose concrete values satisfy the path condition; i. e., solutions to the path condition are concrete test cases.

Definition 1 (Execution State): An execution state is a triple $S = \langle \text{PC}, \sigma, \pi \rangle$, where PC is the program counter, σ is a function from program variables to terms over concrete and symbolic values, and π is the path condition; i. e., a formula that imposes a set of constraints on the symbolic values.

To symbolically execute a program, symbolic execution evolves the execution state as soon as (1) an assignment is evaluated, (2) a conditional branch is evaluated, and (3) the program counter changes. Executing an assignment statement will update σ. Executing a conditional branch with the condition c duplicates the current execution state S into $S_{\textbf{true}}$ and $S_{\textbf{false}}$ and forks execution. Subsequently, the execution engine computes a symbolic formula ϑ_c from c by replacing program variables with the corresponding terms as determined by σ; next, it duplicates the path condition π for different branches and sets $\pi_{\textbf{true}} = \pi \wedge \vartheta_c$ and $\pi_{\textbf{false}} = \pi \wedge \neg\vartheta_c$. Finally, sometimes the program execution evolves through unconditional branches like `goto` statements, which affects the program counter PC of the execution state.

To tackle the problem of symbolic execution in the presence of black-box components, one can combine symbolic and concrete execution such that whenever the program counter is leaving the program's scope symbolic values that flow through the black-box component are concretized and upon returning to the program's scope symbolic execution continues with concrete values. The approach is often called concolic execution, dynamic symbolic execution, or directed automated random testing [4, 8, 9]. In this paper, we propose an alternative approach via automata learning. For a thorough survey on symbolic execution please refer to [15].

B. Automata

Definition 2 (Finite-State Transducer): A finite-state transducer over input alphabet I and output alphabet O is a tuple $\mathcal{M} = \langle I, O, Q, q_0, \delta, \lambda \rangle$, where Q is a nonempty set of states, q_0 is the initial state, $\delta \subseteq Q \times I \times Q$ is the transition relation, and $\lambda \subseteq Q \times I \times O$ is the output relation.

Definition 3 (Mealy Machine): A Mealy machine is a finite-state transducer $\mathcal{M} = \langle I, O, Q, q_0, \delta, \lambda \rangle$ where its δ and λ are functions $\delta : Q \times I \to Q$ and $\lambda : Q \times I \to O$.

From this point forward, we write $q \xrightarrow{i/o} q'$ if $q' = \delta(q, i)$ and $o = \lambda(q, i)$ for Mealy machines, and if $(q, i, q') \in \delta$ and $(q, i, o) \in \lambda$ for finite-state transducers.

Definition 4 (Observation): An observation over input/output alphabet I and O is a pair $\langle \bar{\imath}, \bar{o} \rangle \in I^* \times O^*$ such that $|\bar{\imath}| = |\bar{o}|$. Given a Mealy machine $\mathcal{M}$, the set of observations of $\mathcal{M}$ from state q denoted by $obs_{\mathcal{M}}(q)$ are:

$$obs_{\mathcal{M}}(q) = \{\langle \bar{\imath}, \bar{o} \rangle \in I^* \times O^* \mid \exists q' : q \xrightarrow{\bar{\imath}/\bar{o}}{}^* q'\},$$

where $\xrightarrow{\bar{\imath}/\bar{o}}{}^*$ is the transitive and reflexive closure of the combined transition-and-output function to sequences which implies $|\bar{\imath}| = |\bar{o}|$. From this point forward, $obs_{\mathcal{M}} = obs_{\mathcal{M}}(q_0)$.

Definition 5 (Observation Equivalence): Given states $q, q' \in Q$, we define $q \approx q'$, that is q and q' are observation equivalent, only if $obs_{\mathcal{M}}(q) = obs_{\mathcal{M}}(q')$. Given Mealy machines $\mathcal{M}_1$ and $\mathcal{M}_2$ over the same alphabet, $\mathcal{M}_1 \approx \mathcal{M}_2$ if $obs_{\mathcal{M}_1} = obs_{\mathcal{M}_2}$.

C. Learning and Abstraction

Angluin [16] proposed an active automata learning algorithm called L*. This algorithm learns a deterministic finite automaton accepting an unknown regular language L. It requires a *minimally adequate* teacher that needs to be able to answer two types of queries, *membership* and *equivalence* queries. First, the learner asks membership queries, checking inclusion of words in the language L. Once the learner has gained enough information to build a hypothesis automaton, it asks an equivalence query, checking whether the hypothesis accepts L. The teacher either responds with *yes*, signaling that learning was successful, or with a counterexample to equivalence. If provided with a counterexample, the learner integrates it into its knowledge and starts a new round of learning by issuing membership queries, which is concluded by an equivalence query. L* was adapted to learn various forms of automata, including Mealy machines [17]. The basic principle remains the same, but *output queries* replace membership queries, which ask for outputs produced in response to input sequences.

To learn models of software systems, teachers are usually implemented via testing, as shown in Fig. 2 [18]. Output queries typically reset the System Under Learning (SUL), execute a sequence of inputs and collect the produced outputs. Equivalence queries can be approximated with model-based testing [19]. For that, a Conformance Testing (CT) component derives test queries from the hypothesis, which are executed to find discrepancies between SUL and hypothesis, i. e., counterexamples to observation equivalence (see Def. 4 and 5).

L* is only affordable for small alphabets $I \cup O$; hence, Aarts et al. [20] suggested that we abstract away the concrete domain of the data, by forming equivalence classes in $I \cup O$. This is usually done by a mapper placed in between the learner and the SUL. For abstraction, the mapper maps concrete inputs I and outputs O to abstract inputs X and outputs Y.

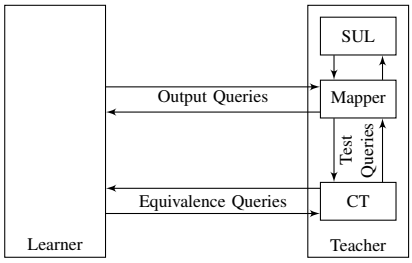

Fig. 2: Abstract automata learning through a mapper using L* (adapted from a figure in [18]).

Definition 6 (Mapper): A mapper for concrete inputs I, and concrete outputs O is a tuple $\mathcal{A} = \langle I, O, R, r_0, \Delta, X, Y, \nabla \rangle$, where R is the set of mapper states, r_0 is the initial state, $\Delta : R \times (I \cup O) \to R$ is a transition function, X is a set of abstract inputs and Y is a set of abstract outputs, and $\nabla : (R \times I \to X) \cup (R \times O \to Y)$ is an abstraction function. From this point forward, we write $r \xrightarrow{a} r'$ if $\Delta(r, a) = r'$.

The mapper communicates with the SUL via the concrete alphabet, and with the teacher and learner via the abstract alphabet. In the setting shown in Fig. 2, the learner behaves the same as the original L* algorithm, but the teacher answers to the queries by indirectly interacting with the SUL through the mapper. Consequently, whenever the teacher receives a reset signal from the learner it resets the mapper along with the SUL to their initial states. Moreover, an individual step executing a single input and observing the output is performed as follows:

1) Given mapper's current state r, upon receiving abstract input $x \in X$, the mapper non-deterministically picks a concrete input symbol $i \in I$ such that $\nabla(r, i) = x$. If such $i \in I$ exists, then the mapper jumps to state $r' = \Delta(r, i)$ and forwards i to the SUL, otherwise it returns the output $\perp$ to the learner.
2) If the mapper has selected and forwarded an $i \in I$, then upon receiving a concrete output $o \in O$ from the SUL, the mapper forwards an abstract version $y = \nabla(r', o)$ to the learner and jumps to state $r'' = \Delta(r', o)$.

Learning an abstract Mealy machine is a slight generalization of L* [20]. From the learner's point of view nothing has changed; it learns a hypothesis $\mathcal{H}$ from observations; but it actually queries an abstraction $\alpha_{\mathcal{A}}(\mathcal{M})$ of a Mealy machine $\mathcal{M}$ induced by a mapper $\mathcal{A}$ as described by Def. 7. Meanwhile, the concretization of $\alpha_{\mathcal{A}}(\mathcal{M})$ induced by a mapper $\mathcal{A}$ is a finite-state transducer $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(\mathcal{M}))$ defined by Def. 8.

Definition 7 (Abstraction): Let $\mathcal{M} = \langle I, O, Q, q_0, \delta, \lambda \rangle$ be a Mealy machine, and let $\mathcal{A} = \langle I, O, R, r_0, \Delta, X, Y, \nabla \rangle$ be a mapper. The abstraction of $\mathcal{M}$ via $\mathcal{A}$ is a finite-state transducer denoted as $\alpha_{\mathcal{A}}(\mathcal{M}) = \langle X, Y \cup \{\perp\}, Q \times R, \langle q_0, r_0 \rangle, \delta', \lambda' \rangle$, where δ' and λ' are given by the following rules:

$$\frac{q \xrightarrow{i/o} q',\ r \xrightarrow{i} r' \xrightarrow{o} r'',\ \nabla(r, i) = x,\ \nabla(r', o) = y}{(\langle q, r \rangle, x, \langle q', r'' \rangle) \in \delta' \wedge (\langle q, r \rangle, x, y) \in \lambda'}$$

$$\frac{\nexists i \in I : \nabla(r, i) = x}{(\langle q, r \rangle, x, \langle q, r \rangle) \in \delta' \wedge (\langle q, r \rangle, x, \perp) \in \lambda'}$$

Note that two issues may arise from abstraction. The abstraction function ∇ may be undefined for some inputs (second rule of Def. 7) and non-deterministic behavior may be introduced by the mapper. This non-determinism might occur if we have two pairs of concrete input/outputs pairs (i_1, o_1) and (i_2, o_2), observable in the same state, such that $\nabla(r, i_1) = \nabla(r, i_2)$ but $\nabla(r', o_1) \neq \nabla(r', o_2)$; i. e., the inputs map to the same abstract symbol, but the outputs map to different ones. While Aarts et al. [20] described a method to automatically refine the mapper, we manually refine it if we encounter such issues to ensure the learned model is an abstract Mealy machine.

Definition 8 (Concretization): Let $\alpha_{\mathcal{A}}(\mathcal{M}) = \langle X, Y \cup \{\perp\}, Q, q_0, \delta, \lambda \rangle$ be an abstract Mealy machine, and let $\mathcal{A} = \langle I, O, R, r_0, \Delta, X, Y, \nabla \rangle$ be the mapper. The concretization of $\alpha_{\mathcal{A}}(\mathcal{M})$ via $\mathcal{A}$ is a finite-state transducer denoted as $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(\mathcal{M})) = \langle I, O \cup \{\perp\}, Q \times R, \langle q_0, r_0 \rangle, \delta'', \lambda'' \rangle$ where δ'' and λ'' are given by the following rules:

$$\frac{q \xrightarrow{x/y} q',\ r \xrightarrow{i} r' \xrightarrow{o} r'',\ \nabla(r, i) = x,\ \nabla(r', o) = y}{(\langle q, r \rangle, i, \langle q', r'' \rangle) \in \delta'' \wedge (\langle q, r \rangle, i, o) \in \lambda''}$$

$$\frac{q \xrightarrow{x/y} q',\ r \xrightarrow{i} r',\ \nabla(r, i) = x,\ \nexists o \in O : \nabla(r', o) = y}{(\langle q, r \rangle, i, \langle q, r \rangle) \in \delta'' \wedge (\langle q, r \rangle, i, \perp) \in \lambda''}$$

III. Method

In this section we describe our method as it is depicted in Fig. 1. First, we give an overview of the proposed configuration and then discuss the involved steps in detail. We start by learning an FSM of the black-box component with a manually defined mapper. Then, we compose this with the white-box. Finally, we execute the SUT symbolically, to generate test cases exercising as many execution paths as possible.

A. Model Learning

As described in Sect. II-C, we learn models by interacting with the SUL via a mapper performing abstraction. The concrete alphabet $I \cup O$ of the SUL generally contains input/output actions of the form $e(p_1, \ldots, p_n)$, i. e., we have input/output events $e \in E$ with n parameters. Mappers create equivalence classes of $I \cup O$ by defining constraints on parameters.

The state of the mapper comprises a fixed number of m variables recording the occurrence of events and storing action parameters. We can therefore view the mapper state as a tuple $r \in R \subseteq (E \cup \mathcal{X})^m$, where E is the set of events and $\mathcal{X}$ is a set of values relevant to the application domain, i. e., it includes the domains of the action parameters, as well as terms formed from parameter values. For the update Δ of the mapper state, we have l guarded update rules for each event e: $\Delta(\langle r_1, \ldots, r_m \rangle, e(p_1, \ldots, p_n)) = \langle r'_1, \ldots, r'_m \rangle$ *if* g_j, where the guard g_j is a quantifier-free formula over R and the parameters of e such that $\bigvee_{j=1}^{l} g_j = \top$ and $i \neq j \to g_i \wedge g_j = \perp$. Similarly, we have k guarded abstraction rules $\nabla(r, e(p_1, \ldots, p_n)) = z$ *if* g_z for each e, where z is a unique abstract symbol in $X \cup Y$.

Input: 1. $\mathcal{M} = \langle X, Y, Q, q_0, \delta, \lambda \rangle$,
2. $\mathcal{A} = \langle I, O, R, r_0, \Delta, X, Y, \nabla \rangle$

1: **function** $i(p_1, \ldots, p_n)$
2: **switch** $\nabla(r, i(p_1, \ldots, p_n))$ **do**
3: **case** x_1
4: $y \leftarrow \lambda(q, x_1)$
5: $q \leftarrow \delta(q, x_1)$
6: $r \leftarrow \Delta(r, i(p_1, \ldots, p_n))$
7: $o(p'_1, \ldots, p'_j) \leftarrow \nabla^{-1}(r, y)$
8: $r \leftarrow \Delta(r, o(p'_1, \ldots, p'_j))$
9: **return** $o(p'_1, \ldots, p'_j)$
10: **case** x_2
11: $\vdots$
12: **function** $\nabla^{-1}(r, y)$
13: $o(p'_1, \ldots, p'_j) \leftarrow o_c$ s.t. $\nabla(r, o_c) = y$ *if* g_y
14: **for all** $p' \in \{p'_1, \ldots, p'_j\}$ **do**
15: MAKESYMBOLIC(p')
16: ASSUME(g_y)

Fig. 3: Composition of learned model and mapper.

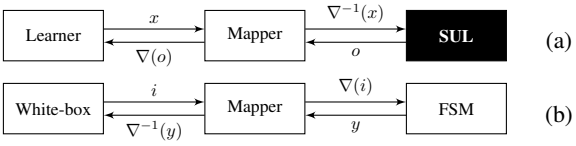

Fig. 4: Mapper in (a) learning vs. (b) symbolic execution

B. Symbolic Execution

Once an abstract model of the black-box component is learned, we compile it alongside the mapper to symbolically execute it. For that, we reverse the role of mapper as compared to learning; see Fig. 4. Therefore, we implement abstraction and concretization as described in Def. 7 and 8 via translation to source code. The interface to the translated composition of mapper and learned model consists of functions $i(p_1, \ldots, p_n)$, for each input event i, called by the white-box component. Figure 3 shows abstractly how these functions are implemented. First, we perform abstraction of inputs (Line 2). Consequently, if such a function is symbolically executed, execution initially forks to each **case**-branch and the execution engine adds the abstraction-rule guard g_x of each abstract input x to the respective path condition, thereby constraining symbolic parameters of i. After that, we update the model state (Line 5) and the mapper state (Lines 6 and 8). Finally, we return concretized outputs $o(p'_1, \ldots, p'_j)$ (Line 9). To update the mapper state, we actually need to check the guards of the update rules defining Δ. This detail is left implicit in Fig. 3.

Abstraction and updates of the state work as described by the ∇, Δ, and δ. For the concretization of an abstract output y, we need ∇^{-1}, but since ∇ performs abstraction, there is no immediate definition of ∇^{-1}. Instead, we retrieve the output event $o(p'_1, \ldots, p'_j)$ and the abstraction-rule guard g_y for y from the ∇ definition (Line 13). We declare the parameters of the output event to be symbolic values via MAKESYMBOLIC (Line 15) and through ASSUME(g_y) we instruct the symbolic execution engine to add g_y to the path condition (Line 16). Hence, we let the execution engine find an instantiation of the output-event parameters satisfying g_y; i. e., it picks a value in O that is in the equivalence class corresponding to y.

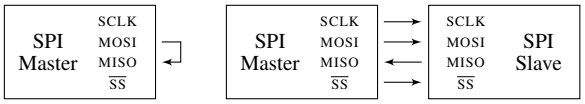

Fig. 5: Master-mode SPI peripheral in loopback along with SPI master-slave setup. SCLK is the serial clock, which is an output line of master, MOSI is the data output line from master to slave, MISO is the data output line from slave to master, and $\overline{\text{SS}}$ is the slave select and an output line of master.

```
1  byte tx_data = 0xdd, rx_data = 0;
2  LPC_SPI->CFG = SPI_CFG_MASTER | SPI_CFG_ENABLE;
3  while(~LPC_SPI->STAT & SPI_STAT_TXRDY);
4  LPC_SPI->TXDATCTL = SPI_TXDATCTL_SSEL_N(0xe)|
       SPI_TXDATCTL_FLEN(7)|SPI_TXDATCTL_EOT|tx_data;
5  while(~LPC_SPI->STAT & SPI_STAT_RXRDY);
6  rx_data = LPC_SPI->RXDAT;
7  if(rx_data != tx_data)
8      abort();
9  while(~LPC_SPI->STAT & SPI_STAT_MSTIDLE);
```

Listing 1: Transmit and receive to/from slave [21, p. 349]

C. Testing

After translation, we generate system-level test cases via symbolic execution of the composition of the white-box, the learned model, and the mapper. We then run these test cases on the actual SUT, i. e., the white-box interacting with the black-box component, while profiling the observed behaviour, outputs, and executed code paths in the white-box. This step is necessary, because the learned model may not be equivalent to the black-box under abstraction. This is due to the fact that learning relies on conformance testing which is incomplete in general. Hence, running the generated test cases serves as a spuriousness check, i. e., we ensure that we will not report spurious errors, or spuriously covered code paths. Our method is therefore sound, but incomplete as it involves black-box conformance testing in the learning phase.

IV. Serial Peripheral Interface

In this section, we demonstrate how we can symbolically execute code that depends on a Serial Peripheral Interface (SPI). First, we study how to learn an SPI controller in its master-mode with a loopback setup to execute Listing 1 symbolically. Then, we show how we can extend our experiment to the whole master-slave setup of SPI.

A. Learning Master-Mode Controller of SPI

In this subsection, we symbolically execute Listing 1 that depends on the SPI bus of the NXP LPC810 micro-controller (MCU). Listing 1 drives an SPI controller in its master-mode with the purpose of sending a single byte to a slave-mode SPI controller and receiving a byte from it. The execution aborts when the received byte does not conform to the sent byte.

TABLE I: Δ & ∇ functions of master-mode SPI mapper.

State (s)	Symbol (a)	$\Delta(s,a)$	$\nabla(s,a)$
r	`void`	r	ε
r	`read(STAT)`	r	ST
r	`read(RXDAT)`	r	RX
r	`write(TXDATCTL,`n`)`	n	TX
r	`STAT(`m`)`	r	
	if m = `0x01`		$\langle 0,0,1\rangle$
	if m = `0x03`		$\langle 0,1,1\rangle$
	if m = `0x102`		$\langle 1,1,0\rangle$
	if m = `0x103`		$\langle 1,1,1\rangle$
r	`RXDAT(n)`	r	
	if $n \neq r$		0
	if $n = r$		1

Learning. To simplify the learning, we learn the master-mode SPI controller in a loopback setup; that is, the same controller receives the transmitted byte; please see Fig. 5a. Primarily we need to know how to reset the SPI controller to its master-mode should the learner ask for a reset. In Line 2, we initialize the SPI controller to its master-mode by writing bit masks `SPI_CFG_MASTER` and `SPI_CFG_ENABLE` to `LPC_SPI->CFG` register.

Alphabet. To extract alphabets we ought to know a thing or two about the NXP LPC810 MCU. In Line 3, we read the `LPC_SPI->STAT` register and check if `SPI_STAT_TXRDY` bit is set to see if the transmission line is ready or not. The act of accessing and evaluating the value of `LPC_SPI->STAT` is an input symbol in I; accordingly, possible values for `SPI_STAT_TXRDY` (bit 0) represent outputs in O. The `STAT` register provides more SPI status flags whose possible values represent more outputs in O. Remaining SPI status flags are `SPI_STAT_RXRDY` (bit 1) and `SPI_STAT_MSTIDLE` (bit 8) [21, p. 239]. We extracted the following concrete alphabets from Listing 1:

$$I = \{\texttt{read(STAT)}, \texttt{read(RXDAT)}, \texttt{write(TXDATCTL,}n\texttt{)} \mid n \in \mathbb{N}\},$$
$$O = \{\texttt{void}, \texttt{STAT(0x01)}, \texttt{STAT(0x03)}, \texttt{STAT(0x102)}, \texttt{STAT(0x103)}, \texttt{RXDAT(}n\texttt{)} \mid n \in \mathbb{N}\}.$$

Mapper. We define a mapper over states $\mathbb{N} \cup \{\bot\}$ where $\bot$ is the initial state. We define the mapper's Δ and ∇ functions by Table I. The concrete values of the `STAT` register are mapped to triples $\langle$`SPI_STAT_MSTIDLE`, `SPI_STAT_RXRDY`, `SPI_STAT_TXRDY`$\rangle$.

Finally, the learning experiment results in the automaton that is depicted in Fig. 6, with which we were able to execute Listing 1 symbolically. An interesting observation that we made is according to the FSM depicted in Fig. 6, a data transmission in state s_0 triggers a state transition to state s_1 and an immediate data write to `TXDAT`, then a move to transmit holding register, and finally transmit to `RXDAT`. A subsequent data transmission results in a data write to `TXDAT`, then a move to transmit holding register. Since `RXDAT` register is occupied in s_1 the transmission to `RXDAT` never occurs, instead the Master Idle flag is cleared, indicating the transmit holding register is not empty, and current state changes to state s_2. If we do another data transmission in this state, current state changes to state s_3; where any data transmission rewrites `TXDAT` and clears Transmitter Ready flag.

On the other hand, according to [21], when the transmit holding register is empty and the transmitter is not sending data the Master Idle flag (i. e., `SPI_STAT_MSTIDLE`) is set otherwise it is cleared. The Transmitter Ready flag (i. e., `SPI_STAT_TXRDY`) indicates whether data may be written to the transmit buffer or not. It is unset when writing data to `TXDAT` and set when the data is moved from the transmit buffer to the transmit shift register. The Receiver Ready flag (i. e., `SPI_STAT_RXRDY`) indicates if data is available to be read from the receiver buffer, and it is cleared after reading `RXDAT` or `RXDATSTAT`.

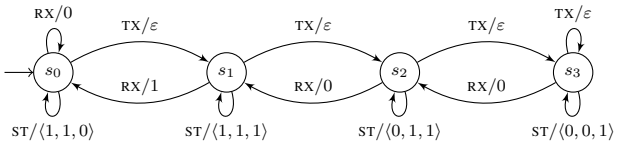

Fig. 6: Automaton of SPI controller as shown in Fig. 5a.

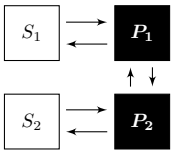

Fig. 7: Software units interacting via black-box SPI.

B. Learning Master-Slave Setup of SPI

In this subsection, we demonstrate how to generate system-level test cases for embedded software systems in the presence of a black-box communication channel.

Test Setup. Our embedded software system implements a padlock using two software units S_1 and S_2 that communicate through black-box peripherals P_1 and P_2; see Fig. 7. S_1 implements a user interface that unlocks a padlock with a 4-digit pin $p_0p_1p_2p_3$; see Listing 2. Meanwhile, S_2 implements a variant of a *combination lock automaton*; see Fig. 8. This automaton progresses on correct inputs $p_0p_1p_2p_3$ and resets otherwise. In each step, S_2 emits 1 in case of success and 0 otherwise. Finally, to check the pin, S_1 sends it to S_2.

We implemented our embedded software system on a pair of NXP LPC810 MCUs. Our port of S_1 to the primary MCU firmware (i. e., user interface) uses the SPI controller in its master-mode configuration to communicate with S_2 on the secondary MCU that uses the SPI controller in its slave-mode

```
1  int main(int argc, char* argv[]) {
2      do {
3          int pin = getchar();
4      } while(!S2.check(pin))
5      grant_user_access();
6      return 0;
7  }
```

Listing 2: S_1 runs user interface and access control.

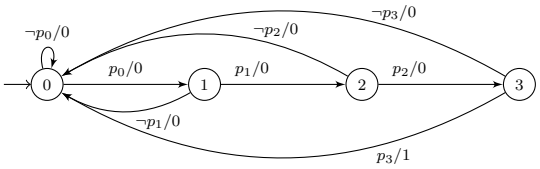

Fig. 8: S_2 runs a combination lock automaton for pin checks.

TABLE II: Code coverage in the presence of SPI controllers.

Coverage Metric	S_1		S_2	
	Concolic	Symbolic	Concolic	Symbolic
Line Coverage	84.78%	**86.96%**	38.24%	**91.18%**
Branch Coverage	57.14%	**71.43%**	25.00%	**85.00%**

configuration. Finally, due to the master/slave architecture of the SPI bus, communication is always initiated by S_1.

Alphabet. We extracted the learning alphabets as follows:

- Skimming the code from [21, p. 350] to work with an SPI in slave-mode, we extracted alphabets I_S and O_S.
- Ensuring $I_S \cap I_M = O_S \cap O_M = \emptyset$ by adding a distinguishing prefix to symbols, we fixed the alphabets I_M and O_M to work with an SPI in master-mode.
- Finally, we defined the concrete alphabets as $I = I_M \cup I_S$ and $O = O_M \cup O_S$ along with a mapper.

After roughly three hours, the experiment resulted in an automaton that models the interactive behaviour of the black-boxes shown in Fig. 7, with 348 states; i. e., $P_1 \times P_2$.

Symbolic Execution. The granting execution path in S_1 is unlikely to occur using concolic execution and random testing because it is very improbable to progress in S_2 not knowing the exact combination. On the other hand, unit-level symbolic execution of both S_1 and S_2 might reveal numerous execution paths; most of which, are not possible through interactive execution of S_1 and S_2; therefore, not worth the effort of manual inspection. Therefore, it is necessary to symbolically reason about how S_1 and S_2 restrict one another's behavior.

We symbolically executed S_1 along with S_2 interactively against the learned $P_1 \times P_2$ automaton. Symbolic execution resulted in five different execution paths almost immediately, while concolic execution through SPI communication channel only revealed one execution path after 22 hours. Table II summarizes the increase in test coverage gained by our proposed methodology against concolic execution.

V. Message Queuing Telemetry Transport

Message Queuing Telemetry Transport (MQTT) is a publish-subscribe connectivity protocol for the Internet of Things. Whenever publishers publish a message to a topic, that message gets posted to a broker server. Subscribers register with the broker on a topic to receive messages published on it. Testing and verifying MQTT clients is difficult because they communicate through a black-box message broker.

Test Setup. Library implementations of the MQTT protocol specifications exist. We implemented our padlock software system using two MQTT libraries (i. e., libemqtt [22] and MQTT-C [23]) in C language. The goal is to execute the padlock software system along with the MQTT libraries against an MQTT broker symbolically; please see Fig. 9.

Test Driver. In our implementation, S_1 and S_2 agree on the MQTT Quality of Service level of 1 for a predefined topic to interact with each other. S_1 is the publisher providing the pin while S_2 is the subscriber implementing the combination lock automaton. Initially, both clients connect to the MQTT broker. Next, they exchange the pin and S_2 performs the pin check

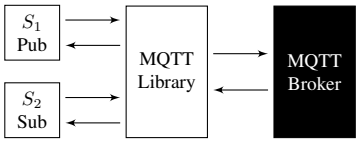

Fig. 9: Clients communicating via a broker using MQTT.

TABLE III: Δ & ∇ functions of MQTT mapper.

Source (s)	Symbol (a)	$\Delta(s,a)$	$\nabla(s,a)$
$\langle l,t,m\rangle$	`Publish`(S_1,t',m')	$\langle l,t',m'\rangle$	PUB
$\langle l,t,m\rangle$	`Subscribe`$(S_2,t',$`QoS1`$)$	$\langle l\cup\{t'\},t,m\rangle$	SUBQ1
$\langle l,t,m\rangle$	`UnSubscribe`(S_2,t')	$\langle l\setminus\{t'\},t,m\rangle$	UNSUB
$\langle l,t,m\rangle$	`Receive`(S_2,t',m')	$\langle l,t,m\rangle$	
	if $(t'\in l\wedge m=m'\wedge t=t')$		RECV
	if $(t'\notin l\vee m\neq m'\vee t\neq t')$		ε
$\langle l,t,m\rangle$	everything else	$\langle l,t,m\rangle$	a

granting access to the user should the pin be correct. Finally, both disconnect from the MQTT broker.

Learning. We used the learning setup configured by Tappler et al. [24] to learn the automaton of an MQTT broker. We extracted the concrete input alphabet for the learning experiment from the test driver as follows:

$$I = \{\texttt{Connect}(c), \texttt{Disconnect}(c), \texttt{Publish}(S_1,t,m), \texttt{Subscribe}(S_2,t,\texttt{QoS1}), \texttt{UnSubscribe}(S_2,t)\}\ .$$

where $c \in \{S_1, S_2\}$ is the client, $t \in \mathbb{S}$ is a topic, $m \in \mathbb{S}$ is a message and $\mathbb{S}$ is the set of character strings. Moreover, in response to above input events, we observe following concrete output events; set O_{S_1} in S_1, and set O_{S_2} in S_2.

$$O_{S_1} = \{\texttt{ConnClosed}(S_1), \texttt{ConnAck}(S_1), \texttt{PubAck}(S_1), \texttt{void}\}\ ,$$
$$O_{S_2} = \{\texttt{ConnClosed}(S_2), \texttt{ConnAck}(S_2), \texttt{SubAck}(S_2), \texttt{UnSubAck}(S_2), \texttt{Receive}(S_2,\texttt{Topic},\texttt{Msg}), \texttt{void}\}\ .$$

Finally, since the broker triggers outputs in both clients, we define the concrete output alphabet for this experiment as

$$O = O_{S_1} \times O_{S_2}\ .$$

Mapper. The state space R of the mapper is $(2^{\mathbb{S}} \times \mathbb{S} \times \mathbb{S} \cup \{\langle\emptyset,\bot,\bot\rangle\})$ where $\langle\emptyset,\bot,\bot\rangle$ is the initial state. Each state is a triple $\langle l,t,m\rangle$ where l is the set of topics to which S_2 is subscribed, and m is the last message published to the last topic t. We define the mapper according to Table III. We learned an automaton of 10 states and 100 transitions from the EMQ broker (v. 2.3.6).

Symbolic Execution. Since KLEE does not support software sockets, we compare the coverage obtained by symbolically executing S_1 and S_2 against the learned broker automaton with that of random testing. For random testing, we generated the test data for the pin randomly and executed n^3 times as many tests as generated by symbolic execution. Table IV summarises the increase in test coverage for MQTT libraries and dismisses the coverage of S_1 and S_2 since their coverage were similar to that of the previous case study. The gap between coverage of libemqtt and MQTT-C is due to the fact that MQTT-C implements more of MQTT protocol.

TABLE IV: Code coverage in the presence of a MQTT broker.

Coverage Metric	libemqtt		MQTT-C	
	Random Testing	Symbolic Execution	Random Testing	Symbolic Execution
Line Coverage	85.00%	**90.90%**	47.47%	**55.83%**
Branch Coverage	47.62%	**57.44%**	30.11%	**33.96%**

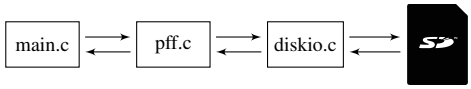

Fig. 10: Petit FAT File System.

VI. Petit FAT File System

The barrier in symbolic execution of software systems that are built on top of file systems is already addressed in KLEE [4]. KLEE models a basic file system that consists of a directory with n user-specified symbolic files. However, symbolic execution of file system implementations still remains an issue, because they are usually built using library level functionalities of disk controllers. In this section, we enable the symbolic execution of a file system that depends on a Secure Digital Card (SD-Card) controller. This helps to generate interesting test cases, which increase test coverage not only for the file system implementation, but also for the software system that is built on top of it.

Test Setup. Petit FAT File System (PFF) is an implementation of the FAT file system for 8-bit micro-controllers [25]. At the moment of writing this paper, the PFF consists of two main source files. The first source file, i. e., "diskio.c", contains SD-Card specific code that is to be implemented based on the target MCU's interface. The second, i. e., "pff.c", is built on top of the first source file and implements the file system.

Test Driver. For learning and testing PFF, we used the setup shown in Fig. 10, i. e., we have diskio.c and pff.c communicating with an SD card. On top of that, we implemented a driver, i. e., "main.c", that (1) mounts a partition, then (2) opens an arbitrary file, and eventually (3) reads 10 bytes of the file's content and tests them against a predetermined value. Once we learned the SD-Card controller, we are able to symbolically execute not only our simple software, but also the PFF itself.

Learning. PFF uses the SD Memory Card protocol in SPI mode. The Physical Layer Simplified Specification [26] contains functional description of SD-Cards and the SD Memory Card protocol in SPI mode. By inspecting the functional description of SD-Card and PFF source code, we extracted following concrete alphabets with which the PFF can run the SD-Card communication protocol in SPI mode.

$$I = \{\texttt{GO\_IDLE\_STATE(0x0)}, \texttt{SEND\_IF\_COND(0x1AA)}, \texttt{APP\_CMD(0x0)}, \texttt{SD\_SEND\_OP\_COND(0x0)}, \texttt{SD\_SEND\_OP\_COND(1<<30)}, \texttt{SEND\_STATUS(0x0)}, \texttt{READ\_SINGLE\_BLOCK}(n), \texttt{SD\_STATUS(0x0)}, \texttt{READ\_OCR(0x0)}\}$$

$$O = \{\texttt{R1}(n), \texttt{R2}(n), \texttt{R3}(n), \texttt{R7}(n), \langle \texttt{R1}(n), \texttt{DATA}(b)\rangle \mid n \in \mathbb{N}\}\ .$$

In case of successful execution, input `READ_SINGLE_BLOCK` returns two outputs `R1`(n), and `DATA`(b) where b is a data block of size 512 bytes. Therefore, we define a compound symbol $\langle \texttt{R1}(n), \texttt{DATA}(b)\rangle$ in our output alphabet.

TABLE V: Abstraction for PFF.

State (s)	Symbol (a)	$\nabla(s,a)$
r_0, r_1	`R1`(n)	`R1`$(n$ & `0x7F)`
r_0, r_1	`R2`(n)	`R2`$(n$ & `0x7FFF)`
r_0	`R3`(n)	`R3`$(n$ & `0x7FFFFFFFFF)`
r_1	`R3`(n)	`R3`$(n$ & `0x00F0000000)`
r_0, r_1	`R7`(n)	`R7`$(n$ & `0x7FF0000FFF)`
r_0, r_1	$\langle$`R1`(n), `DATA`$(b)\rangle$	
	if n & `0x7F` $\neq 0$	ε
	if n & `0x7F` $= 0$	DATABLOCK
r_0, r_1	`READ_SINGLE_BLOCK`(n)	READBLOCK
r_0, r_1	everything else	a

TABLE VI: Coverage results for PFF & Certgate SD-Card.

Coverage Metric	pff.c		main.c	
	Concolic	Symbolic	Concolic	Symbolic
Line Coverage	8.53%	**83.89%**	28.57%	**100.0%**
Branch Coverage	4.41%	**55.15%**	10.00%	**100.0%**

Mapper. We define the state space R as $\{r_0, r_1\}$, and Δ by:

$$\Delta(r,a) = \begin{cases} r_1 & \text{if } a = \texttt{READ\_OCR(0x0)} \\ r_0 & \text{otherwise} \end{cases}$$

and we define the abstraction method by Table V. We ran the learning on three SD-Card controllers namely Certgate SDC, Kingston SDC, and Kingston SDHC. Although the abstract alphabet is very large, in practice we only observed 23, 51, and 44 abstract outputs for Certgate SDC, Kingston SDC, and SDHC respectively. The learned Mealy machines are of size 39, 68, and 41 states and 351, 612, and 369 transitions for Certgate SDC, Kingston SDC, and SDHC respectively.

Symbolic Execution. We ran the experiment for 24 hours using concolic execution and discovered one execution path. Meanwhile, symbolic execution increases the code coverage for both "pff.c" and "main.c" drastically; please see Table VI. Since "diskio.c" implements the interface to the black-box component we considered its code coverage to be irrelevant.

VII. Related Work

Anand et al. [27] used type-dependence analysis to automatically pinpoint the variables to which the flow of symbolic values will cause a problem (e. g., parameters of black-box methods). They were able to automatically indicate problematic variables before performing symbolic execution along with contextual information that can help manual interventions. Although a first step towards coping with black-boxes in symbolic execution, a user had to implement models manually.

Cadar et al. [4] implemented 2500 lines of code to define simple models for roughly 40 system calls to model the execution environment. They also compiled and linked software systems that were built on top of the C standard library against a much more straightforward implementation (i. e., µClibc [28]) to facilitate symbolic execution of the whole software system. This manual effort is only worth for commonly used components. Moreover, since deployed software

systems often consist of more sophisticated implementations of components, this solution shifts system-level correctness to the correctness of handwritten models of the black-boxes.

Chipounov et al. [29] point out that manual modelling of black boxes is labor-intensive and that models are often inaccurate, especially when systems evolve. They present the S^2E platform, which avoids such problems by allowing symbolic execution of binaries, if source code is not available. In this paper, we proposed a method to symbolically execute codes that are dependent on black-boxes other than binaries.

Davidson et al. [30] encountered the same issue while extending symbolic execution to embedded platforms. They elaborated on scenarios in which the black-box is a hardware component. Not being aware of an architectural specification in the hardware component, the symbolic execution engine may follow an incorrect execution path. They manually modeled certain aspects of the hardware to facilitate symbolic execution. The problem is, architectural specifications are often abstruse, not well documented, or not published. Similarly, manual modeling of hardware components is often not practical, because it is both tedious and error-prone.

Jeon et al. [31] proposed to use program synthesis for modeling Java libraries to facilitate symbolic execution of software systems that are built on top of them. They instrumented the library source-code such that they can log simulations of tutorial programs exercising the library. Logs descriptively record either a call to or a return from a method that happened in a tutorial program discarding details of what happened inside the library after invocation. They successfully synthesized models that produced the same instantiations of design patterns as the library, should it run against the same tutorial program under the same inputs. This approach requests white-box access to the third-party components for instrumentation; moreover, it is based on instantiations of design patterns while we based our approach on finite-state machines; therefore, addressing a different and possibly broader set of components.

Godefroid et al. [8] showed that concolic execution might lead to divergence during system-level testing. Hence, the method with which concolic execution concretizes symbolic variables should be black-box specific. A program may induce exponentially many execution paths and concolic execution in a way prunes them unsystematically by replacing symbolic variables with random concrete values. This results in wandering through random execution paths pretty much like random testing; and like random testing, concolic execution also provides no sensible guarantees in terms of system-level coverage in presence of black-box components. Hence, concolic execution does not excel in presence of a black-box component whose behavior matters during path exploration.

Păsăreanu et al. [32] applied symbolic execution in unit-level testing while performing a system-level concrete execution to generate test cases that satisfy user-specified testing criteria. They outperformed random testing and manual test-case generation regarding both coverage and time. In a follow-up study, Davies et al. [33] used treatment learning to reduce number of system-level inputs that affect values of unit-level variables for a path condition of interest. Next, they applied function fitting to find a predictive relationship between the unit-level inputs and associated system-level inputs. Once they have calculated an approximation function for unit-level inputs with respect to system-level inputs, they form a higher-level path condition that also takes the approximation function (i. e., potentially interesting unit-level inputs) into account. They achieved higher coverage with fewer test cases compared to their previous study. The issue with this work is that approximated inputs of a software unit are not accurate enough to get a black-box, like a communication-protocol implementation, to run in practice.

VIII. Conclusion & Future Work

System-level test-case generation is complicated in the presence of black-box components; e. g., communication channels, communication protocols, locking mechanisms. This hardship arises from the fact that exact input values often trigger interesting behaviors of a software system, but the execution path affecting the system-level inputs is only partially visible. To cope with black-boxes, we propose to learn automata of them and instead execute software units against learned automata symbolically. Through this system-level symbolic execution, we can generate test cases for the actual software system under test. Using multiple case studies, we showed the applicability of our approach in generating test cases that cover corner cases and achieve higher coverage.

In this paper, we manually crafted mappers for our learning experiments using our own domain knowledge. Although labour intensive, mapper creation requires less effort compared to modeling systems manually, e. g., for model-based testing. Moreover, mappers are more easily reusable, e. g., [19] uses a single mapper for five different but similar systems. Additionally, we can avoid manual effort of crafting mappers for a certain class of systems through register automata learning [34], or through abstraction refinement [20, 35], which is our first direction for future research. For the second research direction, we speculate concolic execution might as well benefit from the additional information provided by the mappers; yet, we could not think of an easy way to enable that unless we assume the state space of black-box component is irrelevant. The third research direction would be to investigate how to embed the concept of time into our approach and a primary step can be extending our approach to the class of Mealy machines with timers [36]. Finally, we could extend the applicability of symbolic execution in system-level testing to a more comprehensive class of systems by investigating the possibility of approximating outputs of a black-box from its inputs using machine learning methods like treatment learning and function fitting as proposed in [33].

References

[1] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[2] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT—a formal system for testing and debugging programs by

symbolic execution," *ACM SigPlan Notices*, vol. 10, no. 6, pp. 234–245, 1975.

[3] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *TACAS'03*, 2003, pp. 553–568.

[4] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI'08*, 2008.

[5] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid, "Invariant discovery guided by symbolic execution," in *Java PathFinder Workshop*, 2013.

[6] N. Tillmann and J. de Halleux, "Pex-white box test generation for .NET," in *TAP'08*, 2008, pp. 134–153.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, 2008.

[8] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI'05*, 2005.

[9] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference, Lisbon, Portugal, September 5-9, 2005*, M. Wermelinger and H. C. Gall, Eds. ACM, 2005, pp. 263–272.

[10] W. Smeenk, J. Moerman, F. W. Vaandrager, and D. N. Jansen, "Applying automata learning to embedded control software," in *ICFEM'15*, 2015, pp. 67–83.

[11] M. Merten, F. Howar, B. Steffen, and T. Margaria, "Automata learning with on-the-fly direct hypothesis construction," in *Leveraging Applications of Formal Methods, Verification, and Validation - International Workshops, SARS 2011 and MLSC 2011, Held Under the Auspices of ISoLA 2011 in Vienna, Austria, October 17-18, 2011. Revised Selected Papers*, 2011, pp. 248–260.

[12] "Automata Learning benchmarks," accessed: August 24, 2018. [Online]. Available: http://automata.cs.ru.nl/

[13] B. K. Aichernig, W. Mostowski, M. R. Mousavi, M. Tappler, and M. Taromirad, "Model learning and model-based testing," in *Machine Learning for Dynamic Software Analysis: potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, 2018, pp. 74–100.

[14] M. Isberner, F. Howar, and B. Steffen, "The open-source LearnLib - a framework for active automata learning," in *CAV'15*, 2015, pp. 487–495.

[15] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 50:1–50:39, 2018. [Online]. Available: http://doi.acm.org/10.1145/3182657

[16] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, 1987.

[17] M. Shahbaz and R. Groz, "Inferring Mealy machines," in *FM 2009*, 2009.

[18] F. W. Vaandrager, "Model learning," *Commun. ACM*, vol. 60, no. 2, pp. 86–95, 2017.

[19] B. K. Aichernig and M. Tappler, "Learning from Faults: mutation testing in active automata learning," in *NASA Formal Methods - NFM 2017*, 2017, pp. 19–34.

[20] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. W. Vaandrager, "Automata learning through counterexample guided abstraction refinement," in *FM 2012*, 2012.

[21] *LPC81x User manual*, NXP Semiconductors, 4 2014, rev. 1.6.

[22] "Libemqtt," https://github.com/menudoproblema/libemqtt, Accessed: May 1, 2018.

[23] "MQTT-C," https://github.com/LiamBindle/MQTT-C, Accessed: May 1, 2018.

[24] M. Tappler, B. K. Aichernig, and R. Bloem, "Model-based testing IoT communication via active automata learning," in *ICST'17, Tokyo, Japan*, 2017, pp. 276–287.

[25] "Petit FAT File System Module," http://elm-chan.org/fsw/ff/00index_p.html, accessed: May 1, 2018.

[26] *SD Specifications (Part 1) - Physical Layer Simplified Specification*, SD Association, 4 2017, ver. 6.00.

[27] S. Anand, A. Orso, and M. J. Harrold, "Type-dependence analysis and program transformation for symbolic execution," in *TACAS, Braga, Portugal, Proceedings*, 2007.

[28] "µClibc: a C library for embedded linux," https://www.uclibc.org/, accessed: May 1, 2018.

[29] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: design, implementation, and applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, 2012.

[30] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: finding vulnerabilities in embedded systems using symbolic execution," in *Proceedings of the 22th USENIX Security Symposium*, 2013, pp. 463–478.

[31] J. Jeon, X. Qiu, J. Fetter-Degges, J. S. Foster, and A. Solar-Lezama, "Synthesizing framework models for symbolic execution," in *ICSE'16*, 2016, pp. 156–167.

[32] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. R. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software," in *ISSTA'08*, 2008, pp. 15–26.

[33] M. Davies, C. S. Păsăreanu, and V. Raman, "Symbolic execution enhanced system testing," in *VSTTE 2012, Philadelphia, PA, USA, Proceedings*, 2012, pp. 294–309.

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

[35] F. Howar, B. Steffen, and M. Merten, "Automata learning with automated alphabet abstraction refinement," in *Verification, Model Checking, and Abstract Interpretation*, R. Jhala and D. Schmidt, Eds., 2011, pp. 263–277.

[36] B. Jonsson and F. Vaandrager, "Learning Mealy machines with timers," January 2018, preprint on webpage at www.sws.cs.ru.nl/publications/papers/fvaan/MMT/.