MCFICS: Model-based Coverage-guided Fuzzing for Industrial Control System Protocol Implementations

Uchenna Ezeobi*, Sena Hounsinou†, Habeeb Olufowobi‡, Yanyan Zhuang*, Gedare Bloom*
*University of Colorado Colorado Springs, Colorado Springs, CO, USA
†Metro State University, Saint Paul, MN, USA
‡University of Texas at Arlington, Arlington, TX, USA
{uezeobi, yzhuang, gbloom}@uccs.edu, habeeb.olufowobi@uta.edu, sena.houeto@metrostate.edu

***Abstract*—Industrial control system (ICS) protocols face the threat of adversaries launching cyber-physical attacks against protocol endpoints. Vulnerability discovery approaches such as fuzzing can be effective at reducing the risk of such threats. In this paper, we present MCFICS, a coverage-guided greybox fuzzing framework that uses (1) active automata learning for stochastic reactive systems to infer the state machine of a stateful ICS protocol server implementation, and (2) guided fuzzing to explore the state space using this learned state machine. During fuzzing, new input sequences that increase code coverage are used to improve the state space exploration of the ICS protocol implementations. We implemented and tested MCFICS with six example server implementations spanning three widely used ICS protocol implementations. Experimental results show that MCFICS achieves higher branch coverage than the AFLNwe, AFLNet and StateAFL fuzzers by an average (mean of means) of 15.82%, 1.99%, and 37.52%, respectively, with an overall average of 18.44% increased branch coverage. Furthermore, using MCFICS we discovered a new bug in a protocol implementation that we have reported to its upstream maintainer.**

1. Introduction

Industrial control systems (ICS) are a broad category of real-time embedded systems used in industrial processes to control, monitor, and connect physical processes, including those found in industrial facilities and critical infrastructure. ICSs comprise hardware and software components linked over a network to enable industrial automation and communication using a wide variety of protocols. ICS protocols differ from others in that they prioritize low latency and safety over throughput and security. In recent years, the ICS design space has undergone a sea change from being isolated air-gapped operational technology (OT) systems to being tightly-coupled and integrated with information technology (IT). The growing overlap between IT and OT enables the industrial Internet of Things (IIoT) [**?**]. This integration has enhanced system operations but opened up the traditionally closed ICS to cyber and physical exploits such as the attacks on the Ukrainian power grid [**?**], and more.

Addressing zero-day exploits in critical infrastructure ICS involves their proactive identification and remediation [**?**]. Traditional methods like manual testing, formal verification, and static analysis struggle to achieve good coverage and accuracy [**?**], [**?**]. Fuzz testing is an effective dynamic analysis technique to discover vulnerabilities [**?**], [**?**]. However, the inherently complex and stateful nature of ICS protocols, combined with the stochastic behavior of their operating environments, make thorough vulnerability detection challenging. The challenges often lead to the dilemma of state space explosion and make robust ICS security an ongoing effort [**?**].

In this paper, we introduce Model-based Coverage-guided Fuzzing for Industrial Control Systems (MCFICS) as a novel approach for greybox fuzzing tailored to the unique challenges of ICS protocol implementations. Our approach is inspired by prior work that combines fuzzing with automata learning [**?**], [**?**]. The performance of these fuzzers is based on the completeness of the generated state machine for the system-under-test (SUT). We improve upon the state-of-the-art by using active model learning of stochastic reactive systems [**?**] to learn the state machine model for an ICS protocol implementation without specific knowledge of the protocol implementation. We use coverage-guided fuzzing to dynamically discover new inputs and states with code coverage to guide the fuzzer through the protocol implementation's state space.

Our contributions are as follows.

- We introduce a coverage-guided greybox fuzzing framework called MCFICS that leverages active model learning of stochastic reactive systems.
- We evaluate a prototype of MCFICS on server implementations of three widely-used ICS protocols: IEC 60870-5-104 (IEC 104), Modbus, and EPICS PVAccess. We compare the performance of MCFICS against three state-of-the-art greybox fuzzer tools: AFLNwe, AFLNet, and StateAFL. The experimental results show that MCFICS outperforms these fuzzers in terms of branch coverage by 18.44% on average.
- We submitted a bug report for a crash discovered by MCFICS in the EPICS `pvxs` implementation. The `pvxs` maintainer fixed the bug based on our report.

- We provide an open-source[1] prototype of MCFICS.

2. Related Work

The most closely related work are in stateful protocol fuzzing. Fang et al. developed ICS3Fuzzer to detect ICS supervisory software flaws by deriving state machines from server execution logs and network interactions [**?**]. SGFuzz tracks state variables but may struggle with unclear state information [**?**]. In contrast to these approaches, MCFICS runs without detailed SUT software information. AFLNet relies on response codes for state identification. It faces challenges with protocols that do not embed state information, potentially leading to incorrect state machine generation [**?**]. MCFICS, however, does not require state information within response messages.

Previous work has used active learning for vulnerability detection, but required human inspection of the resulting state machine [**?**], [**?**]. Furthermore, active automata learning without an oracle may result in incomplete state machines [**?**], [**?**]. StateFuzzer employs the L* and TTT algorithms for learning and test case creation, but struggles to learn non-deterministic SUTs [**?**]. MACE combines active model learning and fuzzing, however its reliance on symbolic execution and relearning the state machine increases its complexity and time [**?**]. MCFICS uses L_{SMM} algorithm [**?**] to manage non-determinism in state machine learning and is the first to use active model learning and fuzzing to identify vulnerabilities in ICS protocol implementations.

3. System Design and Implementation

In this section, we present the design and implementation of MCFICS, which integrates automata learning and fuzzing as illustrated in Figure 1. MCFICS takes a packet capture (PCAP) file from the SUT as input and produces code coverage and crash reports as output. As shown, MCFICS has five main components, which we describe in more details below.

3.1. Input Corpus

MCFICS requires a behavior model of the SUT, i.e., the ICS protocol implementation used for learning and testing. To obtain the behavior model, we use the automata learning algorithm of stochastic reactive systems L^*_{SMM} [**?**]. The L^*_{SMM} algorithm takes as input a subset of the SUT input alphabet, which is a collection of symbols that are mapped from messages. Communications between the SUT and an example client are collected and used to infer the alphabet of the ICS protocol implementation. This collection uses a network sniffer (`tcpdump`) to listen on the communication port to collect live network data in a PCAP file. This file records communications sent from the example client to a server implementing the ICS protocol. The captured data is then provided to the alphabet extractor to obtain the initial alphabet for the SUT.

1. https://github.com/Embedded-Systems-Security-Lab/automata-learning-fuzzer

3.2. Alphabet Extractor

Cho et al. [**?**] pointed out that using all of the messages in an input corpus to learn the abstract model is unrealistic. Hence, MCFICS extracts the SUT's alphabet from an input PCAP file to reduce the problem space from messages to alphabet symbols. In line with Fang et al. [**?**], we utilize Netzob [**?**] for its superior performance in reverse engineering, extracting symbols for unique message types and formats, and facilitating message de-duplication and grouping based on format [**?**], [**?**].

3.3. Automata Learning

MCFICS uses the extracted alphabet provided by the Alphabet Extractor (Section 3.2) to generate sequences of messages that aim to learn the abstract model (state machine) of the SUT. The Automata Learning component, depicted in Figure 1, comprises the Mapper and Learning Algorithm.

3.3.1. Mapper. The mapper maps each symbol in the alphabet to a set of messages, out of which one message is sent to the SUT. For any symbol with multiple messages, we send each message to the SUT and store the response in a dictionary to keep track of unique response per symbol. If the SUT response is not already in the dictionary for that symbol, the mapper assigns a unique symbol to it and updates the alphabet accordingly. Any non-empty message from the SUT is treated as a response.

This approach has two drawbacks: some messages may change the state of the SUT without a response, and messages that require a specific sequence before triggering a response will not be learned. We address these drawbacks in the fuzzing phase as described in Section 3.5.1. If none of the messages from a symbol receive a response from the server, we choose the first message from the message set and assign the symbol to that message arbitrarily. The symbols with only one message are directly added to the alphabet of the SUT.

3.3.2. Learning Algorithm. Conventional L* algorithms require the SUT to be deterministic, which means that the same sequence of input messages will always yield the same output of messages. Non-deterministic behavior, e.g., as a result of timeouts, delay or loss of packet, asynchronous communication, or non-deterministic multiplex networks, requires adaptation of the L* algorithm. MCFICS uses a modified implementation of the AAlpy library [**?**] for L^*_{SMM}. The L^*_{SMM} is a client that can connect to the SUT to send tree and equivalence queries. In stochastic learning, to reduce uncertainties that arise from the same sequence given different responses, membership queries are replaced with tree queries to gather more information about

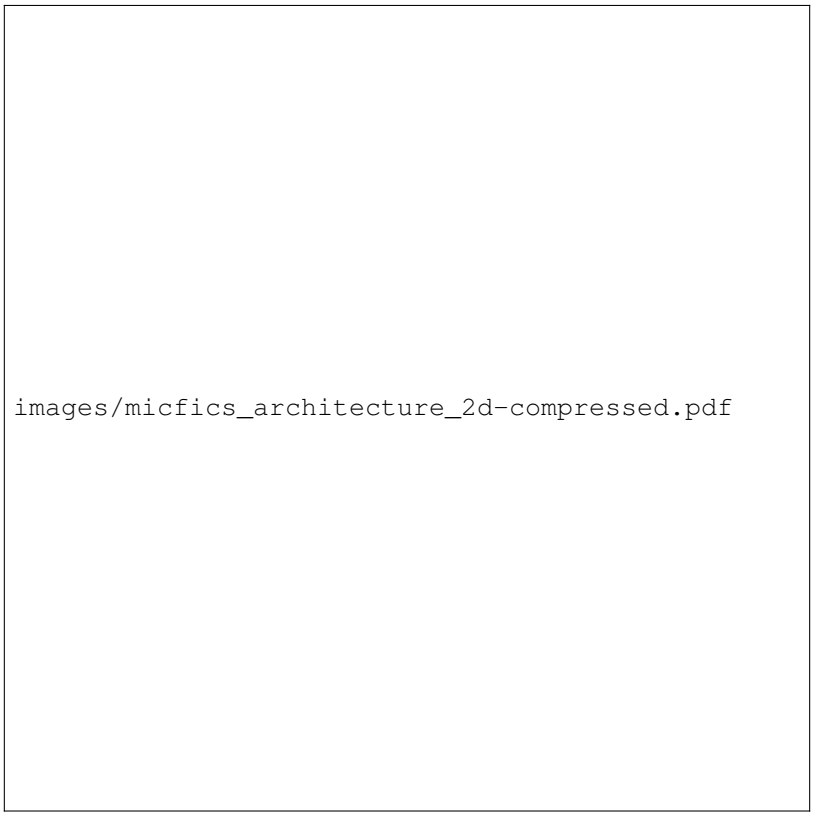

Figure 1. MCFICS Architecture.

sequences in the observation table. We use random sampling to check the conjectures (equivalence queries).

3.4. System Manager

The system manager further consists of four components: the instrumented SUT, the network manager, the monitor manager, and the crash log.

3.4.1. Instrumented SUT. In this work, the SUT is a standalone, reactive server for an ICS protocol, instrumented using AFL++ buildchain [**?**] to measure code coverage throughout automata learning and fuzzing. The setup requires an SUT reset after each membership query for L^*_{SMM} learning. AFL++'s compiler plugin assigns a unique identifier to each code block in shared memory while keeping track of number of times the branch is toggled. Similarly, MCFICS and the SUT use a common 64 KiB memory region optimized for L1 cache, assuring AFL compatibility and allowing for incremental updates of coverage data. The branch coverage, calculated by the number of non-zero bytes in this memory, is used to evaluate code coverage performance.

3.4.2. Managers and Log. As shown in Figure 1, MCFICS has two managers. The network manager sends requests from the fuzzer and learning algorithm to the SUT and receives responses back. The monitor manager examines the SUT to see whether it has become defective. A defective state occurs when the SUT terminates or enters an unexpected zombie state after the network manager sends a test case to the SUT. This determination is reinforced by cross-verifying the operating system's return code associated with the SUT process. The monitor manager records defective states and the test cases that caused the SUT to fail for further investigation by storing them in the crash log.

3.5. Stateful Fuzzer

The Stateful Fuzzer in our system, shown in Figure 1, uses learnt model and coverage data to effectively navigate the SUT's state space. To generate different test cases, it uses a structured method that includes per-state queues, a scheduler, and a mutation engine. The fuzzer uses the learned Mealy machine (see in Section 3.3) to explore states in depth and improve code coverage. However, the SUT's non-deterministic nature presents complications, since identical inputs might produce different results. In the following we describe how MCFICS handles these complications in state space exploration.

3.5.1. Per-state Queues. The L^*_{SMM} algorithm has an observation table that stores all the transitions to every state from the initial state. The transfer sequence for the initial

state is an empty sequence. To get to any state, MCFICS obtains the shortest sequence from the initial state to the target state from the observation table. The state queues are seeded with the unique messages from the PCAP file, because the same message sent from different states can generate different transitions. These state queues are used by the scheduler and mutation engine.

3.5.2. Scheduler. The scheduler allocates time to fuzz each state and selects queues to use that may increase code coverage or discover new paths. MCFICS uses a round-robin scheduling algorithm with a fixed time quantum that treats each queue as a FIFO.

3.5.3. Mutation Engine. The mutation engine selects and mutates messages from the state queue to create test cases, similar to AFL++'s [**?**] havoc and splice strategy. The message sequence is sent to the SUT as the shortest sequence from the initial state to reach the test case. If the branch coverage increases, then the test case is added to the state's queue. If the SUT does not crash, the test case is further put in a global queue that is used during the state space exploration.

We discovered that for specific states, the sequence of messages leading to the test case—the prefix sequence—can cause the SUT to time out, crash, or disconnect from the fuzzer. As a result the test case is not transmitted to the SUT and fuzzing time is wasted. To eliminate such prefix sequences from consideration by the scheduler, MCFICS tracks the number of times a test case is run for each state and removes it from the state queues and mutation engine if the rate of failures in the prefix sequence exceeds a threshold. We manually tuned and discovered that a threshold of 0.001 worked well; determining the best threshold is beyond the scope of this paper.

3.5.4. State Space Exploration. The first abstract model learned depends on the initial alphabet of the input corpus, i.e., messages in the PCAP file. This model might not fully capture the behavior of the SUT. Hence, MCFICS incorporates a state space exploration strategy to discover new states. The premise of this strategy is that any input that increases code coverage during fuzzing either discovered a new transition or a new state. The full message sequence of that input is used to seed a new state whose queue is initialized with the input corpus. In addition, that input is added to the queue of each state and to the input corpus.

4. Evaluation

We conducted experiments to evaluate the effectiveness of MCFICS in exploring the code space of stateful ICS protocol implementations and discovering bugs. The study specifically aimed to assess MCFICS's code coverage capabilities, its ability to identify unique crashes, and its overall fuzzing throughput. In Section 4.1, we detail the experimental setup and design, and then we present the experimental results in Section 4.2.

Table 1. ICS PROTOCOL IMPLEMENTATIONS ANALYZED.

Protocol	Subject	LOC	SUT
IEC 60870-5-104 (IEC104)	lib60870	32K	cs104_server_no_threads, cs104_redundancy_server
MODBUS	libmodbus	57K	unit_test_server, random_test_server
EPICS PVAccess	pvxs	40K	rpc_server, mailbox

4.1. Experimental Setup

We implemented and evaluated MCFICS using Python 3.8.10 on a machine with a 64-bit Intel Core i7-9700K @ 3.60 GHz CPU (16 core) and 32 GB RAM. The host operating system is Ubuntu 20.04.6 LTS. We compare MCFICS with three state-of-the-art fuzzers: AFLNwe [**?**], AFLNet [**?**], and StateAFL [**?**], using the recommendations for fuzzer assessment by Kless et al. [**?**]. In the following we describe in more details the fuzzers, fuzzer targets (SUTs), and the performance metrics for the experiments.

4.1.1. Fuzzers. AFLNwe [**?**] extends AFL's functionality to fuzz network connections by utilizing TCP/IP sockets instead of file-based inputs, but retaining its basic mutation and coverage tracking methods. AFLNet [**?**] enhances AFL with message-level input structuring, mutations, and state machine models from response codes, but requires manual parser creation for each protocol. StateAFL [**?**] enhances AFLNet by including state-aware instrumentation, enabling direct state tracking without relying on response codes. It uses locality-sensitive hashing to identify program states during protocol fuzzing. AFLNwe is a stateless fuzzer, while AFLNet and StateAFL are stateful. AFLNwe and MCFICS are protocol-agnostic, hence they eliminate the need for protocol-specific parsers and simplify fuzzing across protocols. In contrast, AFLNet lacks ICS protocol support.

4.1.2. Fuzzing Targets. Table 1 provides a comprehensive overview of the ICS protocol implementations analyzed including the specific protocol name, the library source of its implementation, the size of the protocol implementation in lines of code (LOC), and the example servers used as the SUT interface for fuzzing. The protocols chosen are widely used in real-world industrial control systems and span a wide range of operational settings. Our study is thus representative of the common scenarios seen in industrial settings [**?**].

4.1.3. Performance Metrics. To evaluate MCFICS's fuzzing effectiveness, we measure branch coverage by counting non-zero bytes in AFL++'s shared memory region, using a 64 KiB space for all fuzzers as specified in Section 3.4.1.

To improve bug-finding efficiency, we count unique crashes based on distinct stack traces compared to previously reported crashes, as explained in Section 3.4.2. Crashes with identical termination functions and backtraces are treated as duplicates and are thus removed. We use AddressSanitizer (ASan) [**?**], UndefinedBehaviorSanitizer

(a) cs104_server_no_threads (b) cs104_redundancy_server (c) unit_test_server

(d) random_test_server (e) rpc_server (f) mailbox

Figure 2. Branch coverage over 24-hour fuzzing experiment with 95% confidence interval, targeting example server implementations. Missing data is explained in Section 4.2.1.

(UBSan) [**?**], and gdb to look for vulnerabilities in these crashes. Each crash is replayed using gdb to detail faults and retrieve stack traces, allowing for manual analysis of reported vulnerabilities or crashes.

4.2. Experimental Results

We conducted a fuzzing experiment with each of the four fuzzers (MCFICS, AFLNwe, AFLNet, StateAFL) targeting each of the 6 SUTs. Each fuzzer was seeded with the same PCAP file containing the network traffic that we captured between example clients and servers provided by the targeted protocol libraries. We ran each fuzzer against each SUT 16 times in parallel with distinct random seeds. Each of these 16 instances was run for 24 hours. Performance metrics were then averaged over the 16 instances to yield descriptive statistical performance measurements (mean, variance, median) for each fuzzer and SUT combination. In the following we report the results of these measurements.

4.2.1. Code Coverage. Figure 2 demonstrates the branch coverage achieved by AFLNwe, AFLNet, StateAFL, and MCFICS during the fuzzing experiment. Figure 2 depicts the mean coverage over the 16 runs of each fuzzer and SUT with a 95% confidence interval.

A key observation is that MCFICS achieves consistently better branch coverage across two of the three tested protocols implementations; the only exception is the `pvxs` protocol, which includes `mailbox` and `rpc_server`. Quantitatively, MCFICS outperforms AFLNwe, AFLNet and StateAFL by an average (mean of means) of 15.82%, 1.99%, and 37.52%, respectively, with an overall average of 18.44% increased branch coverage. These results show that MCFICS is efficient at exploring the code space of these protocol implementations.

We also encountered problems that led to missing data for some fuzzers. AFLNet failed to identify any state in `cs104_redundancy_server` (Figure 2(b)) with the input corpus, thus AFLNet is not present in the figure. For the `pvxs` protocol, both `mailbox` and `rpc_server` (Figure 2(h) and (i)) were not able to compile with `afl-clang-fast` for StateAFL. As a result, StateAFL was not able to fuzz these targets.

Table 2 shows the mean ($\bar{x}$), median ($\tilde{x}$), standard deviation (σ), min, and max of branch coverage over each fuzzing experiment (corresponding with the results in Figures 2). This data shows that MCFICS outperforms the other fuzzers in all respects except on the `pvxs` protocol implementations. For individual comparison, Table 3 shows a pair-wise comparison between MCFICS and each of the other fuzzers.

Table 2. COMPARISON OF BRANCH COVERAGE STATISTICS OVER 24-HOUR OF FUZZING. MISSING DATA IS EXPLAINED IN SECTION 4.2.1.

Example Server	AFLNwe					AFLNet					StateAFL					MCFICS				
	$\overline{x}$	$\tilde{x}$	σ	max	min	$\overline{x}$	$\tilde{x}$	σ	max	min	$\overline{x}$	$\tilde{x}$	σ	max	min	$\overline{x}$	$\tilde{x}$	σ	max	min
cs104_server_no_threads	2608	2606	4	2614	2601	2596	2610	70	2715	2477	2562	2609	67	2626	2483	2813	2810	32	2884	2739
cs104_redundancy_server	1777	1569	352	2517	1441	-	-	-	-	-	1592	1599	31	1648	1520	2928	2913	53	3113	2913
unit_test_server	1231	1218	30	1275	1194	1275	1275	1	1276	1274	998	998	0	998	998	1312	1310	4	1322	1310
random_test_server	1035	1035	0	1041	1041	1041	1041	0	1041	1041	850	847	6	861	847	1062	1062	1	1062	1060
rpc_server	13873	17006	4227	17387	8602	15755	15724	910	16975	14206	-	-	-	-	-	15910	16111	434	16385	15182
mailbox	16775	17379	2156	17498	8744	17246	17231	111	17474	17091	-	-	-	-	-	16515	16494	119	16718	16282

Table 3. PERCENT CHANGE IN MEAN BRANCH COVERAGE AND STATISTICAL SIGNIFICANCE (MANN-WHITNEY U TEST) OF STATE-OF-THE-ART FUZZERS WITH RESPECT TO MCFICS. MISSING DATA IS EXPLAINED IN SECTION 4.2.1.

Example Server	AFLNwe		AFLNet		StateAFL	
	Change in $\bar{x}$	p-value	Change in $\bar{x}$	p-value	Change in $\bar{x}$	p-value
cs104_server_no_threads	7.89%	< 0.001	8.37%	< 0.001	9.83%	< 0.001
cs104_redundancy_server	64.78%	< 0.001	-	-	83.93%	< 0.001
unit_test_server	6.58%	< 0.001	2.86%	< 0.001	31.43%	< 0.001
random_test_server	2.60%	< 0.001	2.00%	< 0.001	24.88%	< 0.001
rpc_server	14.68%	0.78	0.98%	0.37	-	-
mailbox	-1.54%	1.00	-4.24%	1.00	-	-
Average Mean Branch Coverage	**15.82%**		**1.99%**		**37.52%**	

It shows the percent change of the mean coverage attained over all fuzzing experiments, and the significance of the difference in the mean based on the Mann-Whitney U test.[2] The test confirms that MCFICS significantly outperforms (p < 0.001) AFLNwe, AFLNet, and StateAFL across 4 SUTs. Overall, MCFICS has on average between ≈2%–38% more branch coverage than the other fuzzers.

4.2.2. Bug Discovery. MCFICS (and AFLNet) found one new, previously unknown bug in `pvxs` that we have submitted to the `pvxs` maintainer who has fixed it [**?**]. This bug is caused by a `Segmentation fault` at `pvxs::impl::from_wire()`.

5. Conclusion

In this paper, we introduce MCFICS as a novel greybox fuzzing approach for stateful ICS protocol implementations. MCFICS uses automata learning to discover the underlying state machine of an ICS protocol implementation, and explores that state machine during fuzzing by expanding from inputs that increase code coverage. We evaluated MCFICS in comparison with three state-of-the-art fuzzers (AFLNwe, AFLNet, StateAFL) targeting six server implementations of three ICS protocols. MCFICS achieves on average 18.44% (mean) better performance in terms of branch coverage and discovered new bugs in `pvxs`. Future work can evaluate the runtime performance of MCFICS and triage crashes for bug and vulnerability discovery.

2. Mann–Whitney (or Wilcoxon rank-sum) is a nonparametric alternative to the two-sample t-test useful when data are not normally distributed.

Acknowledgements

This work is supported in part by NSF grants OAC-2001789, OAC-2325369, and Colorado State Bill 18-086.