Finding Causally Different Tests for an Industrial Control System

Christopher M. Poskitt*, Yuqi Chen†‡, Jun Sun*, and Yu Jiang§
*Singapore Management University, Singapore
†ShanghaiTech University, China
‡Shanghai Engineering Research Center of Energy Efficient and Custom AI IC, China
§Tsinghua University, China

Abstract—**Industrial control systems (ICSs) are types of cyber-physical systems in which programs, written in languages such as ladder logic or structured text, control industrial processes through sensing and actuating. Given the use of ICSs in critical infrastructure, it is important to test their resilience against manipulations of sensor/actuator inputs. Unfortunately, existing methods fail to test them comprehensively, as they typically focus on finding the simplest-to-craft manipulations for a testing goal, and are also unable to determine when a test is simply a minor permutation of another, i.e. based on the same *causal events*. In this work, we propose a guided fuzzing approach for finding 'meaningfully different' tests for an ICS via a general formalisation of sensor/actuator-manipulation strategies. Our algorithm identifies the causal events in a test, generalises them to an equivalence class, and then updates the fuzzing strategy so as to find new tests that are *causally different* from those already identified. An evaluation of our approach on a real-world water treatment system shows that it is able to find 106% more causally different tests than the most comparable fuzzer. While we focus on diversifying the test suite of an ICS, our formalisation may be useful for other fuzzers that intercept communication channels.**

Index Terms—**Cyber-physical systems, fuzzing, test diversity, equivalence classes, causality**

I. Introduction

Industrial control systems (ICSs) are types of cyber-physical systems (CPSs) consisting of programs, written in languages such as ladder logic or structured text, that control industrial processes through sensing and actuating. While these programs (the cyber part) are often simple when viewed in isolation, the system as a whole is more complex than the sum of its parts due to the integration of multiple physical processes. Given this high level of complexity, as well their ubiquity in domains such as critical infrastructure, it is important that ICSs are systematically tested and analysed against a diversity of sensor/actuator manipulations, so as to ensure that their various defences (e.g. anomaly detectors [1]–[6], digital fingerprinters [7]–[9], invariant checkers [10]–[14]) react appropriately to potential attacks and failures. Unfortunately, this is challenging when benchmarks do not exist for the system under test, as it takes significant time and expertise to construct quality test suites for ICSs from scratch.

An alternative approach, in the absence of benchmarks, is to assess ICS defences against tools such as 'CPS fuzzers' [15]–[18], which automatically search for potential tests, i.e. manipulations of different network inputs that drive the system towards a targeted unsafe physical state (e.g. a dangerously high tank level in a water treatment system). The fuzzers proposed in our previous work [15], [16], for example, find tests based on the manipulation of high-level actuator commands or low-level bit flipping in network packets. In both cases, the search is guided by machine learning (ML) models that predict the effects of potential manipulations. Experimental evaluations of these fuzzers show they can generate test suites that cover a similar range of testing goals to those of an established benchmark [19] for a water treatment testbed.

These fuzzers, unfortunately, share a critical common limitation in that they focus on the *destination rather than the journey*: their underlying search algorithms are designed to find *some* test as quickly as possible, but give little attention to the details of its steps, so long as it achieves the testing goal. First, the fuzzers do not consider whether all the individual manipulations in a test were actually necessary for it to have succeeded, potentially leading to tests that are easier for countermeasures to detect (since they are not as 'subtle' as they could have been). Second, once a test is identified, the fuzzers have no systematic way of determining whether there are different strategies that bring about the *same* unsafe state (i.e. same testing goal). This potentially leads to test suites that lack diversity, in that they are dominated by tests that are 'easier' for the search algorithms to find. This is exactly what we observe on an actual benchmark for a real-world water treatment system [20], which was accumulated over a period of several years with the help of industrial domain experts as well as researchers, multiple hackathons (with teams from academia and industry), and fuzzers. In particular, it only contains at most one or two tests per stated goal.

Addressing these two problems is harder for ICSs and CPSs in general than it might at first seem. Identifying the necessity of individual manipulations boils down to an analysis of *causality* (or cause and effect), and this is made more complicated by factors such as sequencing of steps. For example, the manipulation of a particular actuator might only be causal for a test if it takes place after the manipulation of another. Furthermore, the problem of finding *different* tests with the same goal (i.e. reaching the same unsafe physical state) is challenging due to the ambiguity of what it means for two tests to be 'different'. Suppose, for example, that manipulation x followed by y brings about a targeted unsafe state. If y
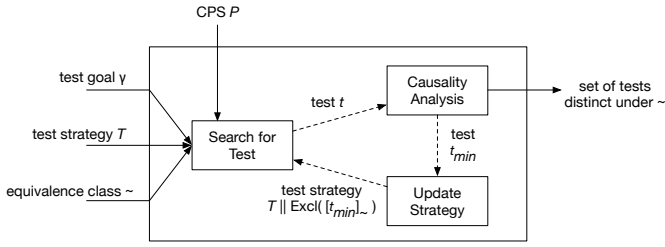
Fig. 1. Overview of our approach for finding causally different tests

followed by x brings about the same unsafe state, is this to be considered 'different', given that the same manipulations are used? What about a sustained sequence of x followed by y? Or x followed by x and y together? While all these tests are different in a strict sense, depending on the context, they may not be different in a *meaningful* way. Current fuzzers, unfortunately, lack any support for this kind of reasoning.

In this work, we present a comprehensive and practical guided fuzzing approach for finding *meaningfully different* tests, based on a general formalisation of sensor/actuator-manipulation strategies. Our algorithm identifies the *causal* manipulations in a test, i.e. without which the test would not have achieved its goal. It then generalises the test to a user-defined *equivalence class*, which formalises what it means for tests to be the 'same' based on common factors such as the set of (causal) manipulations used, or the sequence they were utilised in. Finally, it updates its fuzzing strategy so as to find new tests that are causally different based on the equivalence classes previously identified.

Figure 1 gives an overview of this general approach, showing how a test strategy $\mathcal{T}$ is updated to exclude tests equivalent to a previous one t under equivalence class $\sim$. Our underlying algorithm is based on the idea of modelling test strategies as transition systems labelled with the sensor/actuator values that can be manipulated. After deriving a test from a strategy, the strategy is updated to guide the search away from tests that simply belong to the same equivalence class, ensuring the next test succeeds for different reasons. This ability to exclude causally equivalent tests from the search is the key contribution of our formalism that differs it from other test case generation approaches on labelled transition systems (LTSs) (e.g. [21]).

We demonstrate the practical viability of our formalisation and approach by implementing a *causal fuzzer* for the Secure Water Treatment (SWaT) testbed [22], a complex multi-stage water purification system based on a real-world plant. First, our experiments show that our fuzzer is able to derive tests for 16 different goals, matching the coverage of previous fuzzers that were applied to this ICS. Second, but most importantly, we find 33 *causally different* tests for achieving those 16 goals, i.e. 106% more causally different tests than before. Furthermore, we found that using our causal equivalence classes allowed our fuzzer to reduce the test suite by 4 orders of magnitude while still covering all relevant causal events.

Overall, this paper makes the following contributions:

- A general formalisation of test strategies based on LTSs and traces of sensor/actuator manipulations;
- Three equivalence classes for characterising what it means for two tests to be the 'same';
- A linear-time over-approximation algorithm for identifying the causal events in a test;
- A goal-driven test generation strategy that combines fuzzing with our LTSs;
- An implementation of the approach for a water treatment system that found 33 causally different tests.

While our focus is on diversifying the test suite of an ICS, it is possible that our formalisation could be applied to any kind of software/system in which the execution trace can be influenced by intercepting some communication channel.

II. CASE STUDY: SWAT TESTBED

We use a complex real-world ICS as our running example: the Secure Water Treatment (SWaT) testbed [22], [23]. SWaT is a scaled-down version of a working water purification plant, intended to support research on securing and defending critical infrastructure. SWaT consists of a modern six-stage process that produces up to five gallons of safe drinking water per minute. Each stage focuses on a particular chemical process (e.g. dosing, ultrafiltration, or dechlorination) and is controlled by a dedicated Allen-Bradley ControlLogix Programmable Logic Controller (PLC). These PLCs repeatedly cycle through their programs, computing the appropriate *commands* to send to actuators based on the latest sensor readings received. The testbed consists of 36 sensors in total, including water flow indicator transmitters (FITs), tank level indicator transmitters (LITs), and chemical analyser indicator transmitters (AITs). Among the 30 actuators are motorised valves (MVs) for controlling the inflow of water to tanks, and pumps (Ps) for pumping the water out. Figure 2 provides an overview of the six stages, as well as the main sensors and actuators involved.

The network of the SWaT testbed is organised into a layered hierarchy compliant with the ISA99 standard, providing different levels of segmentation and traffic control. The 'upper' layers of the hierarchy, Levels 3 and 2, respectively handle operation management (e.g. the historian server that logs data) and supervisory control (e.g. touch panel, engineering workstation). Level 1 is a star network connecting the PLCs, and implements the Common Industrial Protocol (CIP) over EtherNet/IP. Finally, the 'lowest' layer of the hierarchy is Level 0, which consists of ring networks that connect individual PLCs to their relevant sensors and actuators.

Each sensor in SWaT is associated with a manufacturer-defined range of *safe* values, which they are expected to remain within at all times during normal operation. If a sensor reports a (true) reading outside of this range, we say the physical state of the system has become *unsafe*. If the tank level indicator transmitter LIT101 in stage one, for example, reports a reading below 250mm, then the physical state has become unsafe due to the risk of an underflow scenario (which can damage the pumps). Similarly, a pressure reading outside of the safe range indicates that a pipe is at risk of bursting.
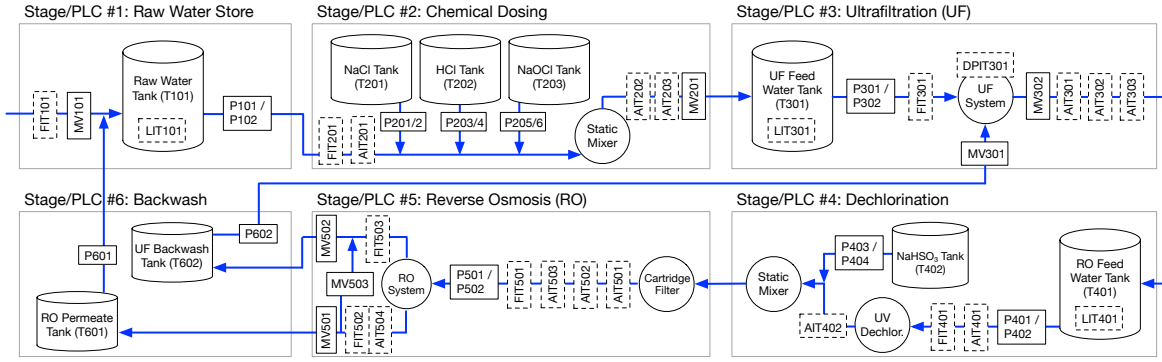
Fig. 2. Overview of the six sub-stages of SWaT (blue arrows indicate water flow; dashed/solid rectangles indicate sensors/actuators)

SWaT is associated with a number of additional resources beyond the physical testbed itself. First, it has an extensive dataset [19], [20], consisting of the sensor readings and actuator states observed over seven days of continuous normal operation, and four days of various benchmarked attack scenarios. Furthermore, SWaT has a simulator [24] that can be used for offline analyses. Built in Python, it faithfully simulates the control logic of the PLCs, and executes models of some key physical processes (e.g. water flow). As it was cross-validated against real data from the dataset, the simulator can be used as a way to initially evaluate various over- and underflow attacks without wasting the resources of the physical testbed.

Across the six sub-processes of SWaT, there are several different unsafe physical states as measured by the sensors. In Stage 3, for example, if the differential pressure transmitter DPIT301 detects a pressure level less than 0.1 bar, the ultra-filtration process is in an unsafe state because the subsequent backwash process will become stuck. A CPS fuzzer that aims to get DPIT301 below this threshold as soon as possible will typically find the simplest way to do so, i.e. by manipulating at least the motorised valve MV302. However, this is not the only way that this unsafe state can be brought about: one can also switch off two pumps (P301 and P302) or open MV301, MV303, and switch on P602. Finding these alternative tests automatically is difficult without a causal understanding of why the tests succeed, or a way to determine that a test is not simply a permutation of an 'equivalent' one.

III. SYSTEMS, STRATEGIES, AND TESTS

In order to solve the problem of finding causally different tests for ICSs such as SWaT, we need a formalisation that allows fuzzers to reason precisely about tests, manipulation strategies, and causal events. We thus present an intuitive formal definition of systems, define strategies as LTSs, and formalise tests as traces of sensor/actuator manipulations.

Our formalism aims for generality, modelling the key characteristics of CPSs (e.g. control states, physical states, and sensing/actuating components) in a way that abstracts away from specific ICS implementation details (e.g. ladder logic). We remark that while generality is our intention, our current evaluation and case study focus on an ICS representative of

the water domain. Assessing the utility of this formalisation for other kinds of CPSs is important future work.

A. Systems

Intuitively, a CPS consists of software interacting with one or more physical processes via components connected over a network. In this work, we assume there to be two types of components: sensors, which read continuous data from the physical state (e.g. temperature, pressure, flow), and actuators, which are mechanised devices for controlling the process (e.g. motorised pumps or valves).

Definition 1 (Component; sensor; actuator). A *component* c is a device for interacting with a physical process; it has an internal state derived from an associated *domain* of values D_c. A *sensor* s is a component with domain $D_s \subseteq \mathbb{R}$, i.e. modelling real-valued readings from a process. An *actuator* a is a component with a discrete finite domain D_a. □

In this paper, if an actuator is a pump, we assume the domain $\{\texttt{on}, \texttt{off}\}$, and if it is a motorised valve, we assume the domain $\{\texttt{open}, \texttt{closed}\}$. Actuators with 'partial' positions (e.g. 25% open) would require an appropriate discretisation.

For a given set of sensors or actuators, we refer to their current states as their readings or configurations respectively.

Definition 2 (Readings; configurations). Given a set of sensors (resp. actuators) C, the *readings* (resp. *configurations*) of C are denoted as a set of pairs $\overline{C} \subseteq \{\langle c, v \rangle \mid c \in C, v \in D_c\}$ containing exactly one pair per component, i.e. if $\langle c, v_1 \rangle \in \overline{C}$ and $\langle c, v_2 \rangle \in \overline{C}$ then $v_1 = v_2$. We let $\mathbb{C}$ denote the set of *all possible* readings/configurations $\overline{C}$. □

Our definition of a CPS elaborates on how control, physical, and component states all relate to each other. Note that the transition functions can be left implicit if the targeted system is available to be executed (as is the case for SWaT; Section V).

Definition 3 (Cyber-physical system). A *cyber-physical system (CPS)* $\mathcal{P}$ is a tuple of the form $(Q_\mathcal{P}, X_\mathcal{P}, S_\mathcal{P}, A_\mathcal{P}, \delta_\mathcal{P}, d_\mathcal{P}, \theta_\mathcal{P}, \tau_\mathcal{P})$, where $Q_\mathcal{P}$ is a set of control states, $X_\mathcal{P}$ is a set of states of the physical process, $S_\mathcal{P}$ is a set of sensors, $A_\mathcal{P}$ is a set of actuators, $\delta_\mathcal{P} : Q_\mathcal{P} \times \mathbb{S}_\mathcal{P} \to Q_\mathcal{P} \times \mathbb{A}_\mathcal{P}$ is the logic of the controllers,

$d_{\mathcal{P}} : X_{\mathcal{P}} \times \mathbb{A}_{\mathcal{P}} \rightarrow X_{\mathcal{P}}$ is a function characterising how the physical process evolves after a fixed time interval $\tau_{\mathcal{P}}$, and $\theta_{\mathcal{P}} : X_{\mathcal{P}} \rightarrow \mathbb{S}_{\mathcal{P}}$ is an observation function describing how sensor readings are extracted from the state of the physical process. $\square$

Given a CPS $\mathcal{P}$, a *run* of the system is a sequence of the form $(q_0, x_0) \rightarrow (q_1, x_1) \rightarrow \cdots$ such that each $q_i \in Q_{\mathcal{P}}$, $x_i \in X_{\mathcal{P}}$, and for every step $(q_i, x_i) \rightarrow (q_{i+1}, x_{i+1})$, $\delta_{\mathcal{P}}(q_i, \theta_{\mathcal{P}}(x_i)) = (q_{i+1}, \overline{A}_{\mathcal{P}})$ with $\overline{A}_{\mathcal{P}} \in \mathbb{A}_{\mathcal{P}}$ and $d_{\mathcal{P}}(x_i, \overline{A}_{\mathcal{P}}) = x_{i+1}$.

B. Test Strategies

CPS testing involves manipulating sensor readings and actuator states according to a defined strategy. Our definition consists of two parts. First, we define *capabilities*, which are the particular manipulations that are possible. Second, we define *strategies*, which are transition systems expressing how the capabilities will be utilised, and the conditions of the system that must be true for them to be able to do so. We can specify a range of test strategies, including optimistic ones that simply 'try out' various capabilities, or more sophisticated ones that conduct manipulations based on the system state.

Capabilities are denoted as pairs of components and values (we use square brackets to distinguish them from readings). A pair $[s, v]$ for sensor s and value $v \in D_s$ indicates that the tester is capable of spoofing the reading reported by s as v, regardless of what the actual reading is. Analogously, a pair $[a, v]$ for actuator a and value $v \in D_a$ indicates that the tester is capable of forcing actuator a into configuration v, regardless of the commands that should be being issued to the actuator at a given moment of the system's execution.

Definition 4 (Capability). Let $\mathcal{P}$ be a CPS. A *capability over* $\mathcal{P}$ is a pair $[c, v]$, where c is a component in $S_{\mathcal{P}} \cup A_{\mathcal{P}}$ and $v \in D_c$. $\square$

A set of capabilities may be infinite if the tester can manipulate sensor values over a continuous domain. In such cases, we use the notation $[s, R]$ for sensor s and $R \subseteq D_s$ to represent all capabilities $[s, v]$ for $v \in R$. For example, $[s, \mathbb{N}_0]$ would represent $[s, 0], [s, 1], [s, 2]$, and so on.

As an example, the set of capabilities $\{[\mathtt{P}_{101}, \mathtt{on}], [\mathtt{MV}_{101}, \mathtt{close}]\}$ over SWaT expresses the ability to override the configurations of two actuators in stage one. In particular, the capabilities respectively express manipulations that cause pump $\mathtt{P}_{101}$ to switch on and motorised valve $\mathtt{MV}_{101}$ to close. Note that our model abstracts away from how these capabilities are realised, specifying only that they *can* be.

Test *strategies* are defined as LTSs with labels of the form $\gamma \vdash \varphi$. Here, γ is a *sensor condition* that must be true for the transition to fire, whereas φ is a *capability condition* that constrains the capabilities that it is allowed to utilise.

Definition 5 (Sensor condition). Let $\mathcal{P}$ denote a CPS. A *sensor condition* over $\mathcal{P}$ is a Boolean formula of simple linear inequalities over the sensors $S_{\mathcal{P}}$. The set of all possible sensor conditions over $\mathcal{P}$ is denoted by $\mathrm{Cond}_{\mathcal{P}}$. Given a condition

$\gamma \in \mathrm{Cond}_{\mathcal{P}}$ and physical state $x \in X_{\mathcal{P}}$, the *valuation* of the condition, $\gamma^x \in \mathbb{B}$, is obtained by evaluating the Boolean expression in the standard way, but with each sensor symbol s interpreted as v such that $\langle s, v \rangle \in \theta_{\mathcal{P}}(x)$. $\square$

As an example, the sensor condition $\gamma = \mathtt{LIT}_{101} \geq 250 \wedge \mathtt{LIT}_{101} \leq 1100$ over SWaT expresses that the level of tank one is between 250mm and 1100mm, i.e. within its safe range. Given a physical state x, the valuation of γ^x is true if $250 \leq v \leq 1100$ for $\langle \mathtt{LIT}_{101}, v \rangle \in \theta_{\mathtt{SWaT}}(x)$, and false otherwise.

Capability conditions are simple Boolean expressions over capability sets. We use the symbol '_' to represent the capabilities that are utilised when the transition is fired. Conditions across multiple labels can be related using globally scoped variables from a set Var, e.g. $\mathtt{X} \in \mathrm{Var}$.

Definition 6 (Capability condition). Let Cap denote a set of capabilities. A *capability condition* over Cap is a Boolean formula of the form true, $\mathrm{Exp}_1 \subseteq \mathrm{Exp}_2$, $\mathrm{Exp}_1 == \mathrm{Exp}_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2$, or $\neg \varphi_1$. Here, $\mathrm{Exp}_1, \mathrm{Exp}_2$ can be expressions over variables in Var, subsets of Cap, or the symbol _, whereas φ_1, φ_2 are capability conditions. We let $\mathrm{Cond}_{\mathrm{Cap}}$ denote the set of all possible capability conditions over Cap. An *assignment* is a function $\alpha \colon \mathrm{Var} \rightarrow 2^{\mathrm{Cap}}$ mapping variables to subsets of Cap. Given a capability condition φ, a set of capabilities $Y \in 2^{\mathrm{Cap}}$, and an assignment α, the condition's *valuation* $\varphi^{Y,\alpha} \in \mathbb{B}$ is obtained by evaluating the set and Boolean expressions in the usual way, but with Y substituted for _, and variables $\mathtt{X} \in \mathrm{Var}$ interpreted as $\alpha(\mathtt{X})$. $\square$

For example, the capability condition $\varphi = [\mathtt{P}_{101}, \mathtt{on}] \notin$ _ specifies that any combination of capabilities can be used, so long as $[\mathtt{P}_{101}, \mathtt{on}]$ is not included. Note that we will use $y \in \mathrm{Exp}$, $\mathrm{Exp}_1 \not\subseteq \mathrm{Exp}_2$, and $\mathrm{Exp}_1 \neq \mathrm{Exp}_2$ to respectively abbreviate $\{y\} \subseteq \mathrm{Exp}$, $\neg \mathrm{Exp}_1 \subseteq \mathrm{Exp}_2$, and $\neg (\mathrm{Exp}_1 == \mathrm{Exp}_2)$. Furthermore, we let the capability set Y on its own abbreviate the capability condition _ $== Y$.

Finally, we define strategies as LTSs, expressing the possible sequences of capabilities that can be utilised by the tester.

Definition 7 (Test strategy). Let $\mathcal{P}$ denote a CPS and Cap a set of capabilities over $\mathcal{P}$. A *test strategy* for $\mathcal{P}$ and Cap is a tuple $\mathcal{T} = (Q_{\mathcal{T}}, \longrightarrow_{\mathcal{T}}, i_{\mathcal{T}})$ where $Q_{\mathcal{T}}$ is a set of states, $\longrightarrow_{\mathcal{T}} \subseteq Q_{\mathcal{T}} \times (\mathrm{Cond}_{\mathcal{P}} \times \mathrm{Cond}_{\mathrm{Cap}}) \times Q_{\mathcal{T}}$ is a transition relation, and $i_{\mathcal{T}} \in Q_{\mathcal{T}}$ is the initial state. A transition $(r_i, (\gamma, \varphi), r_j) \in \longrightarrow_{\mathcal{T}}$ is visualised as an arrow from r_i to r_j with the label $\gamma \vdash \varphi$ (we may omit γ or φ when they are true). $\square$

For simplicity, we make some assumptions on the form of strategies. First, we assume that it is *always* possible to fire at least one transition, regardless of the state of the strategy or the physical process. (One can always add a 'do nothing' transition that uses no capabilities.) Second, we assume that between any pair of states r_1, r_2, there is at most one transition from r_1 to r_2. As a consequence, the sequence of transitions fired can be uniquely determined from a sequence of states. Finally, if a capability set is specified in a transition, then it contains at most one capability per sensor/actuator.
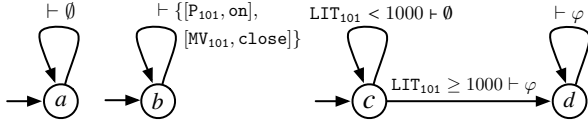
Fig. 3. Three examples of test strategies for SWaT

This definition allows us to capture a variety of test strategies, as shown by the examples in Figure 3. The first transition system $(\{a\}, \{(a, (\text{true}, \_ == \emptyset), a)\}, a)$ implements a *null* strategy. The sensor condition of the sole transition is vacuously satisfied, but no capabilities are ever utilised. The second transition system $(\{b\}, \{(b, (\text{true}, \_ == \{[\text{P}_{101}, \text{on}], [\text{MV}_{101}, \text{close}]\}), b)\}, b)$ implements a *sustained* test strategy. There is only ever one transition to fire (with a condition that is always satisfied), and it utilises the capability set $\{[\text{P}_{101}, \text{on}], [\text{MV}_{101}, \text{close}]\}$ for SWaT, i.e. overriding the configurations of pump P_{101} (to switch it on) and motorised valve MV_{101} (to close it), sustaining this for the whole test.

Finally, let $\varphi = [\text{MV}_{101}, \text{open}] \in \_ \wedge \text{X} == \_$. The rightmost transition system in Figure 3 then implements a *conditional* strategy. For as long as the actual sensor reading of LIT_{101} remains under 1000mm, no capabilities are used. As soon as that threshold is crossed during SWaT's operation, the transition from c to d is fired, and a set of capabilities is used that satisfies φ, i.e. includes at least the opening of motorised valve MV_{101}. Due to the global variable X in φ, that same set of capabilities is used for the rest of the test, i.e. the looping transition incident to d.

C. Tests

Next, we define *tests*, the specific sequence of manipulations derived from a strategy. A test can be thought of as a run of the system in which the transitions of a strategy are fired and tracked in parallel. The key difference is that the capabilities specified in strategies result in *modified observation functions* and *modified actuator commands*, and thus the resulting sequence can contain control and physical states that would not normally be observed in a run of the system.

We begin by defining how the utilisation of capabilities modifies the observation functions and actuator commands of systems. Given a CPS $\mathcal{P}$ and a capability set Y, the *usage of sensor capabilities in Y* is reflected by a modified observation function, $\theta_{\mathcal{P}}^Y$, defined the same as $\theta_{\mathcal{P}}(x)$ for all $x \in X_{\mathcal{P}}$, but with elements $\langle s, v \rangle$ replaced by $\langle s, v' \rangle$ if $[s, v'] \in Y$. Analogously, given a set of actuator configurations $\overline{A}_{\mathcal{P}} \in \mathbb{A}_{\mathcal{P}}$, the *usage of actuator capabilities in Y* is reflected by a modified set of configurations, $\overline{A}_{\mathcal{P}}^Y$, obtained from $\overline{A}_{\mathcal{P}}$ by replacing elements $\langle a, v \rangle$ with $\langle a, v' \rangle$ if $[a, v'] \in Y$.

Intuitively, a test is a sequence of synchronised steps between the control/physical states of the CPS and the states of a strategy. If a transition in the strategy is labelled with capabilities, they are utilised in the step directly. If instead it is labelled with $\vdash \varphi$, then *any* combination of capabilities that satisfies φ is utilised.

Definition 8 (Test). Given a CPS $\mathcal{P}$, a set of capabilities Cap, and a strategy $\mathcal{T}$, a *test* on $\mathcal{P}$ is a sequence of the form $(q_0, x_0, r_0) \rightarrow_{Y_1} (q_1, x_1, r_1) \rightarrow_{Y_2} \cdots$ where $r_0 = i_{\mathcal{T}}$ and each $q_i \in Q_{\mathcal{P}}$, $x_i \in X_{\mathcal{P}}$, $r_i \in Q_{\mathcal{T}}$, and $Y_i \in 2^{\text{Cap}}$. Furthermore, there is an assignment α such that for every $(q_{i-1}, x_{i-1}, r_{i-1}) \rightarrow_{Y_i} (q_i, x_i, r_i)$, there exists a transition $(r_{i-1}, (\gamma_i, \varphi_i), r_i) \in \longrightarrow_{\mathcal{T}}$ with $\gamma_i^{x_{i-1}} = \varphi_i^{Y_i, \alpha} = $ true, $\delta_{\mathcal{P}}(q_{i-1}, \theta_{\mathcal{P}}^{Y_i}(x_{i-1})) = (q_i, \overline{A}_{\mathcal{P}})$, and $d_{\mathcal{P}}(x_{i-1}, \overline{A}_{\mathcal{P}}^{Y_i}) = x_i$. □

Given a CPS $\mathcal{P}$, a strategy $\mathcal{T}$, and a *testing goal* $\gamma_g \in \text{Cond}_{\mathcal{P}}$, a *successful test* for γ_g is a finite test $(q_0, x_0, i_{\mathcal{T}}) \rightarrow_{Y_1} \cdots \rightarrow_{Y_n} (q_n, x_n, r_n)$ such that $\gamma_g^{x_n}$ is true. When analysing a successful test, it is helpful to be able to reason about the *capability history*, i.e. the trace of capability sets utilised. Given a test $t = (q_0, x_0, r_0) \rightarrow_{Y_1} (q_1, x_1, r_1) \rightarrow_{Y_2} \cdots$, the *capability history* of t, denoted π_t, is the corresponding sequence of capability sets $Y_1 Y_2 \cdots$.

For a capability history $\pi_t = Y_1 Y_2 \cdots$, we use the notation $\pi_t[k..l]$ with $1 \leq k \leq l$ to denote the *slice* of π_t from position k to l inclusive, i.e. $Y_k Y_{k+1} \cdots Y_{l-1} Y_l$. Furthermore, we define $\text{CSet}(\pi_t) = \bigcup_{i \in \{1, 2, \cdots\}} Y_i$ as the cumulative set of all capabilities used in π_t.

IV. IDENTIFYING EQUIVALENCE CLASSES OF TESTS

In this section, we address the problem of identifying classes of equivalent tests and performing online updates of strategies.

First, we propose a number of equivalence classes for expressing different ways that two tests can be related. In particular, we define classes based on the particular capabilities used and the order in which they are applied. These specific classes are defined to support our case study (SWaT), but in general, can be defined according to the user's needs.

Second, we show how a strategy can be updated to exclude the possibility of deriving further tests from a given class. Implementing this step provides testers (e.g. fuzzers) a mechanism by which subsequent tests can be diversified once a given equivalence class is suitably covered.

A. Equivalence Classes

We propose three initial equivalence classes that characterise some informal common understandings of what makes two tests the 'same'. In particular, whether two tests both use a given (causal) subset of capabilities, whether they use exactly the same ones, or whether they use them in the same order.

The first of our classes considers two successful tests equivalent if the capabilities they used include a given subset, regardless of the order or context of their application. This equivalence class is particularly useful for our implemented fuzzer (Section V-B), which is able to approximate the *causal* capabilities of a test, and thus can treat any other test that also uses them as equivalent.

Note that these initial equivalence classes were proposed after examining the existing benchmark of manually constructed SWaT attacks [19] and forming an intuition as to when attacks with the same goal (e.g. overflow a tank) were considered

distinct by the benchmark's designers. For ICSs and CPSs outside of the water domain, the user may need to define different equivalence classes if the specific context requires different criteria to distinguish two tests.

Definition 9 (Capability-set equivalence). Let t_1, t_2 denote two successful tests for goal γ_g, and Y a set of capabilities. We say that t_1 and t_2 are Y-*set equivalent*, denoted $t_1 \simeq_Y t_2$, if $t_1 = t_2$, or $Y \subseteq \mathrm{CSet}(\pi_{t_1})$ and $Y \subseteq \mathrm{CSet}(\pi_{t_2})$. □

A stronger version of this class considers two successful tests to be equivalent if the tests utilised *exactly the same* capabilities, regardless of when or how they were used.

Definition 10 (Strong capability-set equivalence). Let t_1, t_2 denote two successful tests for goal γ_g. We say that t_1 and t_2 are *strong capability-set equivalent*, denoted $t_1 \simeq t_2$, if $\mathrm{CSet}(\pi_{t_1}) = \mathrm{CSet}(\pi_{t_2})$. □

Our third equivalence class considers two successful tests to be equivalent if their capability histories are equal once duplicate and trailing steps are removed. For example, successful tests with histories $PQQRPP$ and $PPQRRP$ are strong capability-order equivalent, as both contain the same *order* of distinct sets—$PQRP$—when duplicates are removed. Furthermore, if a successful test derives a sequence of distinct sets with $PQRP$ as its prefix (e.g. $PQRPS$), then that test would be considered equivalent too.

Definition 11 (Strong capability-order equivalence). Let t_1, t_2 denote two successful tests for goal γ_g. We say that t_1 and t_2 are *capability-order equivalent*, denoted $t_1 \rightvec t_2$, if $\mathrm{COrd}(\pi_{t_1})$ is a prefix of $\mathrm{COrd}(\pi_{t_2})$ (or vice versa). Here, we inductively define $\mathrm{COrd}(Y_1 Y_2 Y_3 \cdots) = Y_1 \cdot \mathrm{COrd}(Y_k Y_{k+1} \cdots)$ where $\cdot$ denotes concatenation, k is chosen such that $Y_1 \neq Y_k$, and for every k' in $1 < k' < k$, $Y_1 = Y_{k'}$. For finite capability histories, COrd terminates on the empty sequence. □

As a notational convenience, for a given equivalence class $\sim$, we let $[t]_\sim$ denote the (potentially infinite) set $\{t' \mid t' \sim t\}$.

B. Excluding Classes of Tests

Given a strategy $\mathcal{T}$, a successful test t, and an equivalence class $\sim$, our goal is to transform $\mathcal{T}$ into a new strategy in which the same tests can be derived *except* for those equivalent to t, i.e. in $[t]_\sim$. This mechanism allows fuzzers to diversify by 'nudging' the search towards tests that are not simply minor permutations of t. Our algorithm involves two steps. First, we construct a strategy $\mathrm{Excl}([t]_\sim)$ that can derive every test excluding those that are equivalent to any in $[t]_\sim$. Second, we construct a *parallel composition* $\mathcal{T} \parallel \mathrm{Excl}([t]_\sim)$ of the strategies to return a new one that characterises all of the tests possible in $\mathcal{T}$ except for those belonging to $[t]_\sim$.

We present the construction of $\mathrm{Excl}([t]_\sim)$ for capability-set equivalence (the others are available in the Appendix). In the following, we use the notation $L(\mathcal{T})$ to denote the language of all finite capability histories that can be derived from $\mathcal{T}$.

Proposition 1 (Capability-set equivalence). Let t denote a successful test on CPS $\mathcal{P}$, and $Y \subseteq \mathrm{CSet}(\pi_t)$ a set of
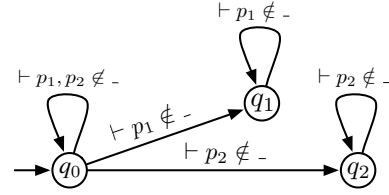

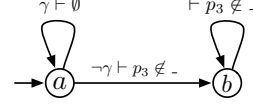Fig. 4. Constructing $\mathrm{Excl}([t]_{\simeq_Y})$ for capability set $Y = \{p_1, p_2\}$

Fig. 5. Example test strategy $\mathcal{T}$

capabilities. A strategy $\mathrm{Excl}([t]_{\simeq_Y})$ can be constructed such that for every successful test t' on $\mathcal{P}$:

$$\pi_{t'} \in L(\mathrm{Excl}([t]_{\simeq_Y})) \quad \text{if and only if} \quad t' \notin [t]_{\simeq_Y}$$

Construction. Suppose that $Y = \{y_1, \ldots, y_n\}$. Then:

$$
\begin{aligned}
\mathrm{Excl}([t]_{\simeq_Y}) = (&\{q_0, q_1, \ldots, q_n\}, \\
&\{(q_0, (\mathrm{true}, (y_1, \ldots, y_n \notin \_)), q_0)\} \\
&\bigcup_{i=1}^{n} \{(q_0, (\mathrm{true}, y_i \notin \_), q_i), (q_i, (\mathrm{true}, y_i \notin \_), q_i)\}, \\
&q_0)
\end{aligned}
$$

where $q_0, q_1, \ldots, q_n$ are fresh strategy states. □

For example, suppose that $\pi_t = PPPQQQPPP$ is the history of a successful test t, where $P = \{p_1\}$ and $Q = \{p_1, p_2\}$. Let $Y = \{p_1, p_2\}$. Then $\mathrm{Excl}([t]_{\simeq_Y})$ is as depicted in Figure 4. Intuitively, the strategy characterises three types of tests: those in which p_1 and p_2 are never used; those in which p_1 is used at least once but p_2 never is; and those in which p_2 is used at least once but p_1 never is. In particular, the strategy excludes any tests in which *both* p_1 and p_2 are used at some point (whether together or separately), i.e. the class of all tests that are Y-set equivalent to t.

Next, we define a parallel composition operator $\parallel$ for strategies. In particular, it allows us to construct $\mathcal{T} \parallel \mathrm{Excl}([t]_\sim)$, i.e. a strategy allowing all of the tests of $\mathcal{T}$ except those equivalent to t under $\sim$.

Definition 12 (Strategy composition). Let $\mathcal{P}$ denote a CPS. Suppose that $\mathcal{T}_1, \mathcal{T}_2$ are strategies for $\mathcal{P}$ with disjoint variables and $Q_{\mathcal{T}_1} \cap Q_{\mathcal{T}_2} = \emptyset$. Then the *composition* of the strategies, denoted $\mathcal{T}_1 \parallel \mathcal{T}_2$, is the tuple $(Q_{\mathcal{T}_1} \times Q_{\mathcal{T}_2}, \longrightarrow_{\mathcal{T}_1 \parallel \mathcal{T}_2}, (i_{\mathcal{T}_1}, i_{\mathcal{T}_2}))$ where:

$$
\begin{aligned}
\longrightarrow_{\mathcal{T}_1 \parallel \mathcal{T}_2} = \{&((q_i, r_k), (\gamma_1 \wedge \gamma_2, \varphi_1 \wedge \varphi_2), (q_j, r_l)) \\
&\mid (q_i, (\gamma_1, \varphi_1), q_j) \in \longrightarrow_{\mathcal{T}_1} \\
&\wedge (r_k, (\gamma_2, \varphi_2), r_l) \in \longrightarrow_{\mathcal{T}_2}\} \quad \square
\end{aligned}
$$

To illustrate, consider the strategy $\mathcal{T}$ of Figure 5, in which any capability (except p_3) may be used as soon as some sensor condition γ no longer holds. Suppose that $\emptyset\emptyset\emptyset PPQQQQ$ is the history of a successful test $\pi_t \in L(\mathcal{T})$, where $P = \{p_1\}$
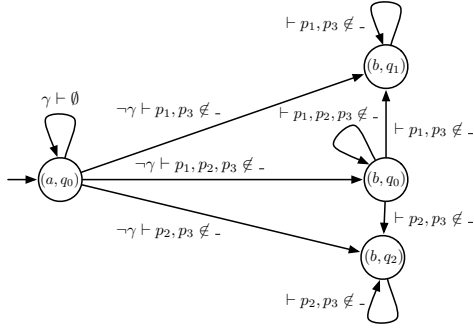
Fig. 6. Composition $\mathcal{T} \parallel \mathrm{Excl}([t]_{\simeq_Y})$ for capability set $Y = \{p_1, p_2\}$

and $Q = \{p_1, p_2\}$. Let $Y = \{p_1, p_2\}$. Suppose that we want to find tests that are *not* Y-set equivalent to t (perhaps, for example, because p_1, p_2 are causal; see Section V-B).

First, we construct $\mathrm{Excl}([t]_{\simeq_Y})$ in Figure 4, which characterises all tests that are not Y-set equivalent. Then, we compose the strategies together, i.e. $\mathcal{T} \parallel \mathrm{Excl}([t]_{\simeq_Y})$. Figure 6 depicts the result (with some simplifications), a strategy characterising all the same tests as $\mathcal{T}$ *except* for those that are Y-set equivalent to a. Note that $\emptyset\emptyset\emptyset PPQQQQ$ can no longer be derived as there are no paths allowing both p_1 and p_2; however, a test such as $\emptyset\emptyset PPPPP$ can still be derived, as this is not Y-set equivalent to t (and thus is 'different' under this equivalence class). Note that the capability p_3 was forbidden in $\mathcal{T}$, and remains forbidden in the composition.

Theorem 1 (Composition). Let $\mathcal{P}$ denote a CPS, $\mathcal{T}$ a strategy, and t a successful test on $\mathcal{P}$ such that $t \in L(\mathcal{T})$. For every equivalence class $\sim$ and successful test t' on $\mathcal{P}$:

$$\pi_{t'} \in L(\mathcal{T} \parallel \mathrm{Excl}([t]_{\sim})) \;\; \mathrm{iff} \;\; \pi_{t'} \in L(\mathcal{T}) \wedge t' \notin [t]_{\sim}$$

A proof is given in the Appendix.

V. FINDING DIFFERENT TESTS BY GUIDED FUZZING

Next, we show how our formalism can be combined with guided fuzzing approaches to automatically derive a set of mutually *non-equivalent* tests from an initial strategy.

Our overall fuzzing approach is summarised in Figure 1 and Algorithm 1: given some CPS $\mathcal{P}$ (e.g. the SWaT testbed), a strategy $\mathcal{T}$, a goal γ_g (e.g. cause an overflow in tank T101), and an equivalence class $\sim$ (e.g. strong capability-set equivalence), our fuzzing algorithm will find and return a set of tests that are all mutually non-equivalent under $\sim$.

The first step of the algorithm is to *plan* a finite sequence of transitions (a 'walk') from strategy $\mathcal{T}$ to fire on CPS $\mathcal{P}$. This plan can be derived in a number of ways (e.g. randomly), but we use a more intelligent approach which we describe separately in Section V-A. We aim to derive a sequence of transitions from $\mathcal{T}$ *predicted* to achieve the goal γ_g on $\mathcal{P}$.

Once a plan is obtained, the fuzzer proceeds to *fire* the transitions one-by-one, utilising the capabilities of each transition for the given time interval $\tau_{\mathcal{P}}$. The test t consists of the sequence of control and physical states of $\mathcal{P}$ that result from firing those transitions, with the sequence terminating when

---

Algorithm 1: Overall Guided Fuzzing Approach

Input: CPS $\mathcal{P}$; strategy $\mathcal{T}$; goal γ_g, equivalence class $\sim$
Output: Set of distinct $tests$ for γ_g

1 Let $tests := \{\}$;
2 **repeat**
3   Derive a sequence of transitions $w = tr_1 tr_2 \cdots tr_n$ from $\mathcal{T}$ where
        $tr_i = (r_{i-1}, (\gamma_i, \_ == Y_i), r_i)$; [Alg. 2]
4   Let q_0, x_0 denote the current control/physical states of $\mathcal{P}$;
5   Let $t := (q_0, x_0, r_0)$;
6   Let $i := 1$;
7   **while** $i \le |w|$ **do**
8   Utilise capabilities Y_i on $\mathcal{P}$ for $\tau_{\mathcal{P}}$ seconds;
9   Let q_i, x_i denote the current control/physical states of $\mathcal{P}$;
10   **if** $\gamma_i^{x_i}$ **then**
11             $t := t \rightarrow_{Y_i} (q_i, x_i, r_i)$;
12             $i := i + 1$;
13   **else**
14             $i := |w| + 1$;
15   **if** t is successful for γ_g **then**
16   **if** causal fuzzing enabled **then**
17             $t := t_{\min}$; [Alg. 3]
18         $tests := tests \cup t$;
19         $\mathcal{T} := \mathcal{T} \parallel \mathrm{Excl}([t]_{\sim})$;
20 **until** timeout;
21 **return** $tests$;

---

either the final transition of the plan is fired, or when the next transition has a sensor condition that the physical state does not satisfy. The test t is *successful* if γ_g is true of the final physical state, in which case t is added to the set of $tests$ returned by the fuzzing algorithm. (If causal fuzzing is enabled, non-causal capabilities are pruned first—see Section V-B.)

Before the main loop of the algorithm is repeated, the test strategy $\mathcal{T}$ is updated according to the algorithms of Section IV so as to ensure that the next test identified is not equivalent.

A. Searching for a Potential Test

A key step of our overall fuzzing algorithm (Alg. 1, Line 3) is deriving a walk from a strategy, i.e. planning a finite sequence of transitions from a strategy $\mathcal{T}$ to fire on the real system. In general, strategies are nondeterministic, so there are many potential sequences: some states may have multiple outgoing transitions, and there may be $\_$-labelled transitions in which different combinations of capabilities can be used. As the search space is large, we propose an intelligent approach that searches for a walk *predicted* to achieve the test goal γ_g.

Our approach for generating a plan is given in Algorithm 2. Given a CPS $\mathcal{P}$, strategy $\mathcal{T}$, objective function f_{γ_g}, and prediction model $M_{\mathcal{P}}$, the algorithm returns a sequence of transitions (a 'walk') in $\mathcal{T}$ that is predicted to maximise f_{γ_g}. Here, $f_{\gamma_g} : \mathbb{S}_{\mathcal{P}} \to \mathbb{R}$ is some function that maps the sensor readings of $\mathcal{P}$ to a number, such that the larger that number is, the 'closer' the physical state is to satisfying γ_g. While these objective functions must be defined, for our purposes, they are typically quite simple (see the Appendix). Furthermore, the algorithm requires a model $M_{\mathcal{P}} : \mathbb{S}_{\mathcal{P}} \times \mathbb{A}_{\mathcal{P}} \to \mathbb{S}_{\mathcal{P}}$ that can predict how the current sensor readings will evolve over a given time period $\tau_{\mathcal{P}}$ relative to some intended actuator commands. This model could be in the form of a simulator for $\mathcal{P}$, or it could be a machine learning model (e.g. a neural network) that was trained on some data logs [15], [16], [25].

Algorithm 2: Find a Potential Test from a Strategy

Input: CPS $\mathcal{P}$; strategy $\mathcal{T}$; objective function f_{γ_g}; prediction model $M_{\mathcal{P}}$
Output: Sequence of transitions w

1 Let $walks := \langle\rangle$; [empty sequence]
2 Let $scores := \langle\rangle$;
3 **repeat**
4   From $i_{\mathcal{T}}$, generate finite random walk $tr_1 tr_2 \cdots tr_n$ over transitions tr_i in $\mathcal{T}$;
5   Replace φ_i in each tr_i with Y_i for some Y_i satisfying φ_i;
6      $walks := walks.\text{append}(tr_1 tr_2 \cdots tr_n)$;
7 **until** timeout;
8 Let $\overline{S}_0, \overline{A}_0$ denote $\mathcal{P}$'s current sensor/actuator values;
9 **foreach** $w \in walks$ **do**
10   Let $\overline{S} := \overline{S}_0$;
11   Let $i := 1$;
12   **while** $i < |w|$ **do**
13   Let $\overline{A}_i$ denote the actuator commands resulting from Y_i;
14          $\overline{S} := M_{\mathcal{P}}(\overline{S}, \overline{A}_i)$;
15          $i := i + 1$;
16      $scores.\text{append}(f_{\gamma_g}(M_{\mathcal{P}}(\overline{S}, \overline{A}_{|w|})))$;
17 Select a walk w from $walks$ using *roulette wheel selection* with corresponding scores in $scores$;
18 **return** walk w;

Fig. 7. Example of a replay strategy

Algorithm 3: Approximating the Causal Capabilities

Input: Successful test t for γ_g
Output: Pruned successful test $t_{\min}$ for γ_g

1 Let $t_{\min} := t$;
2 Let $causal := \{\}$;
3 **repeat**
4   Pick y, k, l where $\pi_{t_{\min}}[k..l]$ is a maximal slice of equal capability sets, y is a capability in the set, and $\langle y, k, l\rangle \notin causal$;
5   **if** y is causal in $\pi_{t_{\min}}$ from k to l **then**
6          $causal := causal \cup \{\langle y, k, l\rangle\}$;
7   **else**
8   Let $t_{\min}$ denote the *counterexample*; [Def. 13]
9 **until** Line 4 fails;
10 **return** $t_{\min}$;

Intuitively, the algorithm begins by generating a large number of completely random finite walks through the strategy $\mathcal{T}$. Following this, the algorithm processes each random walk in turn, *predicting* the sensor readings that would result from actually firing the transitions relative to the current state of the CPS. This final prediction is achieved by applying the model $M_{\mathcal{P}}$ to each transition in turn, and then applying the objective function f_{γ_g} to the final state. The higher the number, the 'closer' the predicted effects of the walk are to achieving the testing goal. Rather than simply returning the walk with the highest score, we choose one using roulette wheel selection [26] in order to ensure diversity in the tests.

B. Causally Different Tests

Thus far, our equivalence classes have not taken into account the *causality* of capability usages. For example, a test that uses capabilities Y and a test that uses capabilities $Y \cup \{p\}$ (with $p \notin Y$) would be considered different under strong capability-set equivalence, even if p is totally irrelevant to the test's success (e.g. it modifies an actuator in an unrelated stage).

Our solution here is for equivalence classes to consider *only* those events that were causal for the test's success. As computing the exact causal set is NP-complete in general [27], [28], we propose an algorithm to efficiently approximate it. In particular, given a successful test, our algorithm prunes capability usages from it that are non-causal. To assess causality, we analyse counterfactual conditionals of the form "if p did not occur, then γ_g would not be achieved", by systematically replaying the test with individual capabilities removed. Depending on the CPS $\mathcal{P}$ involved, this replay can be performed on the system itself, or by using the prediction model $M_{\mathcal{P}}$ from the previous steps (i.e. either a learnt model or a high-fidelity simulator).

We can formalise this notion of 'replaying' a test as a strategy. Given a test t with capability history $\pi_t = Y_1 Y_2 \cdots Y_n$, the *replay strategy* for π_t is $\mathcal{T}_{\pi_t} = (\{s_0, \ldots, s_n\}, \{tr_1, \ldots, tr_n, tr_\em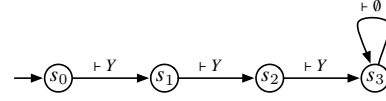ptyset\}, s_0)$, where each $tr_i =$ $(s_{i-1}, (\text{true}, \_ == Y_i), s_i)$ and $tr_\emptyset = (s_n, (\text{true}, \_ == \emptyset), s_n)$. Figure 7, for example, depicts the replay strategy constructed from a history $\pi_t = YYY$ for $Y = \{[\text{P}_{101}, \text{on}], [\text{MV}_{101}, \text{close}]\}$.

The following definition formalises our notion of causality in a successful test for γ_g. In particular, the usage of a capability p across a given range of positions is *causal* if removing p from those positions would mean the resulting test no longer satisfies γ_g. This is assessed using replay strategies in which p has been removed.

Definition 13 (Causal capability). Let $t = (q_0, x_0, r_0) \rightarrow_{Y_1} \cdots$ denote a successful test for γ_g with capability history $\pi_t = Y_1 Y_2 \cdots Y_n$. Given a capability y and indices k, l such that $1 \le k \le l \le n$, we say that y is *causal in π_t from k to l* if there does not exist a successful test for γ_g from q_0, x_0 using replay strategy $\mathcal{T}_{\pi_t \setminus \langle y, k, l\rangle}$. If such a test from q_0, x_0 exists, we call it the *counterexample*. Note that the sequence $\pi_t \setminus \langle y, k, l\rangle$ is obtained from π_t by substituting $Y_i \setminus \{y\}$ for Y_i across $k \le i \le l$. □

With these concepts defined, Algorithm 3 describes a linear-time approximation procedure that takes a successful test t for γ_g and returns a successful test $t_{\min}$ for γ_g in which non-causal capability usages have been removed. The algorithm operates by repeatedly identifying a maximal subsequence of equal, repeated capability set usages, and replaying the test with a specific capability removed from each of the sets in that subsequence. For example, for a sequence $PPPQ$ with $p \in P$, the algorithm may replay the sequence $P'P'P'Q$ where $P' = P \setminus \{p\}$. If the test goal is still met, the capability usage was not causal and can be pruned; if it is not, then it is recorded as causal and other subsequences of capabilities are tested. When no remaining capability can be pruned, the test is returned.

Proposition 2 (Over-approximation of causal capabilities). Given a successful test t for γ_g, Algorithm 3 returns a test containing an over-approximation of its causal capabilities. □

This is evident given that capabilities are only removed if there exist counterexamples to them being causal. The algorithm's complexity is linear in the number of events in the trace.

Finally, to ensure that Algorithm 1 finds *causally different* tests, we replace t with $t_{\min}$ (Line 17), and then use $\text{CSet}(\pi_{t_{\min}})$-equivalence as our equivalence class when updating $\mathcal{T}$ (Line 19). This ensures that any other tests identified by the algorithm are not based on the same causal manipulations.

VI. Evaluation on SWaT

In this section, we evaluate the effectiveness of our formalisation and guided fuzzing approach with respect to SWaT, a real-world testbed representative of ICSs in the water domain. In particular, we ask:

RQ1: How many *different tests* can our approach automatically find for each testing goal?

RQ2: Does causal fuzzing effectively prune minor permutations of otherwise equivalent tests?

RQ3: How long does it take to find these tests?

RQ4: Does causal fuzzing generate a more diverse set of tests than the SWaT benchmark?

Through this evaluation, we aim to judge whether our formal approach can really scale to a complex and real-world critical infrastructure system from the water domain.

Our programs are available online [24], and details of the implementation of our algorithms for SWaT are available in the Appendix.

Experiment #1: RQ1. Our first experiment assessed how many *different* tests for SWaT our algorithms can find according to two of our equivalence classes: strong capability-set equivalence, and strong capability-order equivalence. These classes only take into account patterns of capability usages: they do not consider the causality of individual manipulations, the importance of which is explored in our second experiment.

To achieve this, we initialised our implementation of the algorithm on a 'universal' test strategy, i.e. a single node incident to a transition permitting any capability usages. We ran our program for one minute per goal (i.e. per unsafe state) and equivalence class, using the SWaT simulator as our prediction model for Algorithm 2, randomly generating an initial configuration each time within normal operational ranges. For test goals concerning all sensors except the LITs, we set the time interval $\tau_{\mathcal{P}}$ as 15s, as the changes take effect very quickly (no longer than 15s in preliminary experimentation). Given the size of tanks, it takes somewhat longer to effect changes on the LIT readings. Thus, we chose a separate time interval of 10 minutes for test goals concerning the LITs. Note that we do not mix different time intervals in a single test: the value of $\tau_{\mathcal{P}}$ is selected according to the test goal, and applied for each capability usage in the test. (Future work could generalise this in the formal model, perhaps facilitating equivalence classes defined with respect to time.)

This value of $\tau_{\mathcal{P}}$ was determined in a simple pre-study: we randomly generated an initial configuration within the SWaT simulator's normal operational ranges, searched for a combination of actuators predicted to drive the LIT sensor to an unsafe state, then set that combination in the simulator and ran it 100 times, recording the median time taken to reach the unsafe state. We found that no sensor required more than 30 minutes, and with three transitions per walk (see the Appendix), this suggested a time interval of 10 minutes. Note that as part of this pre-study, we recreated one run per test goal on the physical testbed to validate the time intervals identified by experimenting on the simulator.

Our results are given in the first two rows of Table I. We found that our implementation was able to quickly generate large numbers of effective tests (as predicted by the simulator) for every test goal, with each test distinct according to the equivalence class definition used. As mentioned before, these equivalence classes are *non-causal*, defined only over patterns of the capabilities present, meaning that minor permutations of the same test are not filtered out. Our next experiment considers whether our causal-aware fuzzing solves this problem.

Experiment #2: RQ2–3. Our second experiment assesses how many *causally different* tests for SWaT our algorithm can find, and how much additional time our causality approximation adds in comparison to Experiment #1. In particular, we searched for new tests on the basis of our Y-set equivalence class, where Y is taken to be the *set of causal events* in the test (as approximated by Algorithm 3). As for our previous experiment, we started with a 'universal' test strategy that could derive any possible combination of capabilities.

First, we used our program to search for a single test t predicted to achieve the test goal. We then applied Algorithm 3 to approximate the causal capabilities, pruning out all others to get $t_{\min}$. For all non-LIT goals, we performed this approximation directly on the testbed. Due to the time requirements for LIT goals, we implemented those causality tests using the simulator (on a random initial state within normal operational ranges), *except* for the final minimised sequence which was tested and verified on the real system. Next, we parallel composed the strategy with $\text{Excl}([t]_{\simeq_Y})$, where Y is taken as $\text{CSet}(\pi_{t_{\min}})$. We repeated this process for the next test found, which was guaranteed by our formalisation not to contain this causal set of capabilities.

Our results are presented in the third row of Table I, and reflect the total number of tests that could be found based on different causality sets of capabilities. For each non-LIT goal, the causality approximation algorithm took up to an hour per test to complete on the testbed. For each LIT goal, in which the final step was run on the testbed but others on the simulator, the approximation took 10 minutes per test.

The number of different tests represents a significant reduction from those of the non-causal experiment, and illustrates the power of causality-based equivalence classes in reducing the search space. In fact, we found that 100% of the tests from the previous experiment utilised at least one of the causality sets identified in this one, meaning that they were minor permutations of the key manipulations that made the test succeed. The only exception was DPIT301-Low ($*$), for which 97% of the tests utilised one of the three causality sets. We believe that this was due to a poorer prediction model

| | | Tanks (High) | | | Tanks (Low) | | | Flow (High) | | | Flow (Low) | | | | | | Pr. (L) |
		LIT101	LIT301	LIT401	LIT101	LIT301	LIT401	FIT101	FIT201	FIT601	FIT101	FIT201	FIT301	FIT401	FIT501	FIT601	DPIT301
Non-Causal	Strong capability-set equivalence	34,019	30,034	38,890	23,659	29,853	37,013	42,355	51,902	7,713	41,091	54,109	41,023	41,001	16,265	41,045	56,834*
	Strong capability-order equivalence	40,199	28,100	34,191	19,908	27,118	44,213	54,239	56,122	10,902	46,124	54,213	49,008	42,123	23,455	44,214	59,901*
Causal	Causal-set equivalence	**3**	**6**	**2**	**2**	**2**	**4**	**1**	**1**	**1**	**1**	**2**	**2**	**1**	**1**	**1**	**3**
Other	SWaT dataset/benchmark	2	1	—	1	—	1	—	—	—	—	—	—	—	—	—	—

for the sensor in our simulator, although the 3% of tests still pushed the sensor reading towards the testing goal.

The approximated causality sets are given in the Appendix. Upon manual inspection, we found these approximations to be exact in all cases. We note that for some test goals, some causality sets are subsets of others (e.g. for LIT301-High). The reason for this is due to the initial configuration of the system that the test is launched in: some tests based on those causal sets will only succeed when the water tanks are at certain levels (e.g. close to unsafe ranges).

> *Our approach avoids generating causally equivalent tests for SWaT, pruning the test suite by 4 orders of magnitude, and is practically implemented with a linear-time approximation.*

Note that for other systems, one may wish to generate multiple tests for each equivalence class. This could be achieved by triggering the strategy update only after certain thresholds (e.g. after generating n number of tests for a given class). Due to resource limitations, we do not evaluate this alternative approach for SWaT.

Experiment #3: RQ4. Finally, we assessed whether causal fuzzing led to tests that achieve better coverage/diversity than existing CPS fuzzers [15] and the SWaT benchmark [19], [20] (Section II) by counting the number of tests that had the goal of driving a sensor into an unsafe range. We chose the latter as our benchmark as it is the most extensive publicly available dataset for the SWaT testbed, and has become established as a baseline in the critical infrastructure community (cited over 300 times since 2016, and downloaded by researchers in 82 countries [20].)

As shown in Table I, none of the tests in the benchmark target the unsafe states of the FITs or DPIT, covering only the unsafe states of LITs. Even limited to LITs, our causal fuzzer still has more coverage/diversity: the benchmark does not contain tests for the LIT401 (High) and LIT301 (Low) ranges, and contains only 1–2 tests for the others. We note, however, that some of the tests in the benchmark are incomparable to ours as they do not target the manipulation of a physical state, but rather just spoof a sensor (without physical effects); causal fuzzing focuses only on tests that caused physical changes.

With the CPS fuzzer of [15], we are only able to find one causally distinct test per each of our 16 test goals. This is explained by its search approach, which is always guided towards the 'simplest' causal capabilities. Thus, across these goals, causal fuzzing finds 106% more causally different tests.

> *Causal fuzzing found 106% more causally different tests for SWaT than the most comparable CPS fuzzer.*

Threats to Validity. While SWaT is a fully operational testbed, it is not as large as the plants it is based on, meaning our results may not scale-up (this is difficult to assess due to strict confidentiality). Furthermore, due to practical constraints, our causality analysis (Algorithm 3) is partially evaluated on the simulator (although the final results are all validated on the actual testbed). As a result, the success of a test may depend on the starting state/time: occasionally, the system may not reach the unsafe state even for causal events, thus causal events might be removed. This could be addressed with a weaker requirement for being causal, e.g. an event could be considered causal as long as there is a physical state change towards the test goal.

We remark that our evaluation has focused on a real-world ICS in the water domain. While our formalisms and algorithms were designed with generality in mind, the results of these experiments may not generalise to different kinds of CPSs (e.g. drones). This should be explored in future work.

VII. RELATED WORK

In this section, we highlight how our approach relates to other works involving fuzzing and test generation for CPSs and ICSs in particular. While fuzzing has been applied to these kinds of systems before, the aims are often quite different (e.g. to find bugs in network protocols) in comparison to ours, which is to provide a formal causality-guided way of diversifying the test suite of an ICS.

Our approach was motivated by problems observed in the original 'CPS fuzzers' developed in our previous work [15], [16]. For example, in [15], a genetic algorithm is used to find actuator configurations of SWaT that were predicted to drive the system into unsafe physical states. The search attempts to find *some* test of this kind as quickly as possible, which in practice means that it will usually return the same test for a given goal (typically the 'simplest' test, but perhaps with some random non-causal manipulations as a result of the optimisation). Our new fuzzer avoids this problem first by determining the causal manipulations and excluding the entire class of causally equivalent classes from the search space, ensuring that multiple causally different tests can be found for a given goal (see e.g. Table I). We also found more tests than the active learning-based fuzzer we proposed in [16], although a direct comparison here is less fair as it focused

on manipulating low-level network packets (i.e. bit-vectors) rather than high-level actuator commands.

Other CPS fuzzers are more domain-specific, focusing on finding bugs in code or protocols, rather than sensor/actuator manipulations. CyFuzz [29], for example, tests CPS tool chain components that have been implemented in Simulink by randomly generating input models. DeepFuzzSL [30] goes further by guiding the generation with a neural network. Fuzzing has also been applied to specific protocols in CPSs, e.g. network protocols in order to test their intrusion detection systems (e.g. [31]). Our work differs in that subsequent phases of test generation adapt to ensure causally equivalent classes of tests are excluded from the search.

Several works have applied the concept of causality for *security analyses* of CPSs. In [32], Moradi et al. employed a STRIDE model as a reference for classifying attacks. While not explicitly built upon causality, their STRIDE model has a certain form of in-built causality. In [33], Zhang et al. applies causal models (defined based on maximum information coefficients and transfer entropy) to trace and detect ICS anomalies. Outside the domain of CPSs, causality has been increasingly applied to analyse or explain complex systems including in AI [34]–[36]. In contrast to these works, we use causality as a way to ensure diversity from our fuzzer.

Several authors have investigated the *falsification of CPS specifications*. Given a CPS and a formal specification (e.g. in some temporal logic), the idea is to search for counterexamples that violate the specification. Tools such as S-TaLiRo [37] use stochastic search methods to find simulation traces that minimise a global robustness metric. Yamagata et al. [38] also attempt to minimise how robustly specifications are satisfied but drive the search using deep reinforcement learning. The tests found by our work could be considered as similar to the counterexamples of these works, although otherwise our approach is quite different. Our approach may combine to help find causally different ways of violating the same specification.

Outside of ICSs and CPSs, many fuzzers are available for testing software, e.g., [39]–[43]. A common approach to improve test generation is to specify the class of valid inputs as a context-free grammar (CFG) [44], [45]. While it would be feasible to specify our 'strategies' as CFGs, and use CFG-guided fuzzers to generate tests, it may be difficult to exclude equivalence classes of tests from a CFG in general (especially as this requires handling language complements). It is also worth noting that while CFGs typically express (single-step) software inputs, our strategies express (multi-step) sequences of system manipulations that are evaluated by observing their effects on execution traces. Though our approach was motivated by a specific application in diversifying ICS test suites, it may general enough to adapt for fuzzing other types of software in which communication can be manipulated, e.g. distributed software systems.

Johnson et al. [46] proposed 'Causal Testing', an approach that helps Java developers to identify causal information associated with a failed test case by fuzzing with minimally different inputs that do not exhibit the faulty behaviour. Our

approach differs in that they use causality to *explain* an existing failure, while we use it to diversify the test suite.

VIII. Conclusion

Inspired by the limited diversity of existing test benchmarks for ICSs such as SWaT, we developed a formal guided fuzzing approach that allows us to systematically generate causally different tests with respect to an equivalence class and a given goal. By focusing on traces of sensor/actuator manipulations, our approach can be applied without having to construct any mathematical models of the targeted system's programs or physics. We assessed the utility of our approach by implementing it in a fuzzer for a real-world water treatment testbed, finding that it was able to identify multiple tests that were successful for *causally different* reasons.

In future work, we would like to explore practical ways of reducing the overhead of our online strategy update algorithms. For example, as an alternative to excluding classes of tests by construction, a guided fuzzer may be able to incorporate test diversity (i.e. being outside known equivalence classes) as a part of the 'reward' function when searching for new tests. We will also determine whether our approach can be applied to fuzzing different kinds of CPSs (e.g. drones), or other software/systems in which the execution trace can be influenced by intercepting some communication channel. Finally, we will explore its applicability to model checking (e.g. searching for different counterexamples) and compositional verification (i.e. verifying the system under the presence of different test strategies).

References

[1] J. Inoue, Y. Yamagata, Y. Chen, C. M. Poskitt, and J. Sun, "Anomaly detection for a water treatment system using unsupervised machine learning," in *Proc. IEEE International Conference on Data Mining Workshops (ICDMW 2017)*. IEEE, 2017, pp. 1058–1065.

[2] W. Aoudi, M. Iturbe, and M. Almgren, "Truth will out: Departure-based process-level detection of stealthy attacks on control systems," in *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*. ACM, 2018, pp. 817–831.

[3] M. Kravchik and A. Shabtai, "Detecting cyber attacks in industrial control systems using convolutional neural networks," in *Proc. Workshop on Cyber-Physical Systems Security and PrivaCy (CPS-SPC 2018)*. ACM, 2018, pp. 72–83.

[4] Q. Lin, S. Adepu, S. Verwer, and A. Mathur, "TABOR: A graphical model-based approach for anomaly detection in industrial control systems," in *Proc. Asia Conference on Computer and Communications Security (AsiaCCS 2018)*. ACM, 2018, pp. 525–536.

[5] M. A. M. Carrasco and C. Wu, "An unsupervised framework for anomaly detection in a water treatment system," in *Proc. IEEE International Conference On Machine Learning And Applications (ICMLA 2019)*. IEEE, 2019, pp. 1298–1305.

[6] S. Adepu, F. Brasser, L. Garcia, M. Rodler, L. Davi, A. Sadeghi, and S. A. Zonouz, "Control behavior integrity for distributed cyber-physical systems," in *Proc. ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS 2020)*. IEEE, 2020, pp. 30–40.

[7] C. M. Ahmed, A. P. Mathur, and M. Ochoa, "NoiSense Print: Detecting data integrity attacks on sensor measurements using hardware-based fingerprints," *ACM Transactions on Privacy and Security*, vol. 24, no. 1, 2020.

[8] Q. Gu, D. Formby, S. Ji, H. Cam, and R. A. Beyah, "Fingerprinting for cyber-physical system security: Device physics matters too," *IEEE Security & Privacy*, vol. 16, no. 5, pp. 49–59, 2018.

[9] M. Kneib and C. Huth, "Scission: Signal characteristic-based sender identification and intrusion detection in automotive networks," in *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*. ACM, 2018, pp. 787–800.

[10] S. Adepu and A. Mathur, "Distributed attack detection in a water treatment plant: Method and case study," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 1, pp. 86–99, 2021.

[11] Y. Chen, C. M. Poskitt, and J. Sun, "Learning from mutants: Using code mutation to learn and monitor invariants of a cyber-physical system," in *Proc. IEEE Symposium on Security and Privacy (S&P 2018)*. IEEE Computer Society, 2018, pp. 648–660.

[12] H. Choi, W. Lee, Y. Aafer, F. Fei, Z. Tu, X. Zhang, D. Xu, and X. Xinyan, "Detecting attacks against robotic vehicles: A control invariant approach," in *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*. ACM, 2018, pp. 801–816.

[13] J. Giraldo, D. I. Urbina, A. Cardenas, J. Valente, M. A. Faisal, J. Ruths, N. O. Tippenhauer, H. Sandberg, and R. Candell, "A survey of physics-based attack detection in cyber-physical systems," *ACM Computing Surveys*, vol. 51, no. 4, pp. 76:1–76:36, 2018.

[14] C. H. Yoong, V. R. Palleti, R. R. Maiti, A. Silva, and C. M. Poskitt, "Deriving invariant checkers for critical infrastructure using axiomatic design principles," *Cybersecurity*, vol. 4, no. 1, p. 6, 2021.

[15] Y. Chen, C. M. Poskitt, J. Sun, S. Adepu, and F. Zhang, "Learning-guided network fuzzing for testing cyber-physical system defences," in *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*. IEEE Computer Society, 2019, pp. 962–973.

[16] Y. Chen, B. Xuan, C. M. Poskitt, J. Sun, and F. Zhang, "Active fuzzing for testing and securing cyber-physical systems," in *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*. ACM, 2020, pp. 14–26.

[17] H. Wijaya, M. Aniche, and A. Mathur, "Domain-based fuzzing for supervised learning of anomaly detection in cyber-physical systems," in *Proc. International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS 2020)*. ACM, 2020, pp. 237–244.

[18] H. Kim, M. O. Ozmen, A. Bianchi, Z. B. Celik, and D. Xu, "PGFUZZ: Policy-guided fuzzing for robotic vehicles," in *Proc. Annual Network and Distributed System Security Symposium (NDSS 2021)*. The Internet Society, 2021.

[19] J. Goh, S. Adepu, K. N. Junejo, and A. Mathur, "A dataset to support research in the design of secure water treatment systems," in *Proc. International Conference on Critical Information Infrastructures Security (CRITIS 2016)*, ser. LNCS, vol. 10242. Springer, 2016, pp. 88–99.

[20] "iTrust Labs: Datasets," https://itrust.sutd.edu.sg/itrust-labs_datasets/, 2023, accessed: February 2023.

[21] J. Tretmans, "Test generation with inputs, outputs, and quiescence," in *Proc. International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1996)*, ser. LNCS, vol. 1055. Springer, 1996, pp. 127–146.

[22] A. P. Mathur and N. O. Tippenhauer, "SWaT: a water treatment testbed for research and training on ICS security," in *Proc. International Workshop on Cyber-physical Systems for Smart Water Networks (CySWater@CPSWeek 2016)*. IEEE Computer Society, 2016, pp. 31–36.

[23] "Secure Water Treatment (SWaT)," https://itrust.sutd.edu.sg/testbeds/secure-water-treatment-swat/, 2023, accessed: February 2023.

[24] "Code for SWaT experiments," https://github.com/yuqiChen94/Causally-Different-Attacks/, 2023.

[25] J. Goh, S. Adepu, M. Tan, and Z. S. Lee, "Anomaly detection in cyber physical systems using recurrent neural networks," in *Proc. International Symposium on High Assurance Systems Engineering (HASE 2017)*. IEEE, 2017, pp. 140–145.

[26] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[27] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. J. Trefler, "Explaining counterexamples using causality," in *Proc. International Conference on Computer Aided Verification (CAV 2009)*, ser. LNCS, vol. 5643. Springer, 2009, pp. 94–108.

[28] ——, "Explaining counterexamples using causality," *Formal Methods in System Design*, vol. 40, no. 1, pp. 20–40, 2012.

[29] S. A. Chowdhury, T. T. Johnson, and C. Csallner, "CyFuzz: A differential testing framework for cyber-physical systems development environments," in *Proc. Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy 2016)*, ser. LNCS, vol. 10107. Springer, 2017, pp. 46–60.

[30] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "DeepFuzzSL: Generating models with deep learning to find bugs in the Simulink toolchain," in *Proc. Workshop on Testing for Deep Learning and Deep Learning for Testing (DeepTest 2020)*. ACM, 2020.

[31] G. Vigna, W. K. Robertson, and D. Balzarotti, "Testing network-based intrusion detection signatures using mutant exploits," in *Proc. ACM Conference on Computer and Communications Security (CCS 2004)*. ACM, 2004, pp. 21–30.

[32] F. Moradi, S. A. Asadollah, A. Sedaghatbaf, A. Causevic, M. Sirjani, and C. L. Talcott, "An actor-based approach for security analysis of cyber-physical systems," in *Proc. International Conference on Formal Methods for Industrial Critical Systems (FMCIS 2020)*, ser. LNCS, vol. 12327. Springer, 2020, pp. 130–147.

[33] R. Zhang, Z. Cao, and K. Wu, "Tracing and detection of ICS anomalies based on causality mutations," in *2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC)*, 2020, pp. 511–517.

[34] A. Ibrahim, T. Klesel, E. Zibaei, S. Kacianka, and A. Pretschner, "Actual causality canvas: A general framework for explanation-based socio-technical constructs," in *Proc. European Conference on Artificial Intelligence (ECAI 2020)*, ser. Frontiers in Artificial Intelligence and Applications, vol. 325. IOS Press, 2020, pp. 2978–2985.

[35] J. Pearl, "The seven tools of causal inference, with reflections on machine learning," *Communications of the ACM*, vol. 62, no. 3, pp. 54–60, 2019.

[36] A. Forney, J. Pearl, and E. Bareinboim, "Counterfactual data-fusion for online reinforcement learners," in *Proc. International Conference on Machine Learning (ICML 2017)*, ser. Proceedings of Machine Learning Research, vol. 70. PMLR, 2017, pp. 1156–1164.

[37] Y. Annpureddy, C. Liu, G. E. Fainekos, and S. Sankaranarayanan, "S-TaLiRo: A tool for temporal logic falsification for hybrid systems," in *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011)*, ser. LNCS, vol. 6605. Springer, 2011, pp. 254–257.

[38] Y. Yamagata, S. Liu, T. Akazaki, Y. Duan, and J. Hao, "Falsification of cyber-physical systems using deep reinforcement learning," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2823–2840, 2021.

[39] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2017, accessed: February 2023.

[40] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proc. IEEE Symposium on Security and Privacy (S&P 2015)*. IEEE Computer Society, 2015, pp. 725–741.

[41] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. SIGSAC Conference on Computer and Communications Security (CCS 2017)*. ACM, 2017, pp. 2329–2344.

[42] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *Proc. USENIX Security Symposium (USENIX Security 2019)*. USENIX Association, 2019, pp. 1967–1983.

[43] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sFuzz: an efficient adaptive fuzzer for solidity smart contracts," in *Proc. International Conference on Software Engineering (ICSE 2020)*. ACM, 2020, pp. 778–788.

[44] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proc. ACM SIGPLAN Conference on Programming Lan-*

guage Design and Implementation (PLDI 2008). ACM, 2008, pp. 206–215.

[45] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The fuzzing book." CISPA Helmholtz Center for Information Security, 2023, accessed: February 2023. [Online]. Available: https://www.fuzzingbook.org/

[46] B. Johnson, Y. Brun, and A. Meliou, "Causal testing: understanding defects' root causes," in Proc. International Conference on Software Engineering (ICSE 2020). ACM, 2020, pp. 87–99.

[47] A. Ruscito, "pycomm," https://github.com/ruscito/pycomm, 2019, accessed: February 2023.

APPENDIX

EQUIVALENCE CLASS CONSTRUCTIONS

Proposition 3 (Strong capability-set equivalence). Let t denote a successful test on CPS $\mathcal{P}$. A strategy $\mathrm{Excl}([t]_\simeq)$ can be constructed such that for every successful test t' on $\mathcal{P}$:

$$\pi_{t'} \in L(\mathrm{Excl}([t]_\simeq)) \text{ if and only if } t' \notin [t]_\simeq$$

Construction. Suppose that $\mathrm{CSet}(\pi_t) = \{y_1, \ldots, y_n\}$. Let us define $Y^* = 2^{\mathrm{CSet}(\pi_t)} \setminus \mathrm{CSet}(\pi_t)$. Then,

$$
\begin{aligned}
\mathrm{Excl}([t]_\simeq) = \\
(\{q_*\} \cup \{q_Y \mid Y \in Y^*\}, \\
\{(q_*, (\mathrm{true}, \_, \mathrm{true}), q_*)\} \\
\bigcup_{Y \in Y^*} \{(q_Y, (\mathrm{true}, \_ \setminus \mathrm{CSet}(\pi_t) \neq \emptyset), q_*)\} \\
\bigcup_{\substack{Y, Y' \in Y^* \\ . Y \subseteq Y'}} \{(q_Y, (\mathrm{true}, Y' \setminus Y \subseteq \_ \wedge \_ \subseteq Y'), q_{Y'})\}, \\
q_\emptyset)
\end{aligned}
$$

where q_* and each q_Y are fresh strategy states. □

For example, suppose that $\pi_t = PPPQQQPPP$ is the history of a successful test, where $P = \{p_1\}$, $Q = \{p_1, p_2\}$, and thus $\mathrm{CSet}(\pi_t) = \{p_1, p_2\}$. Then $\mathrm{Excl}([t]_\simeq)$ is as depicted in Figure 8. Note that the constraints of some transitions have been simplified. Intuitively, the strategy characterises tests that do not involve *exactly* the same set of capabilities as t. This can be achieved by using strictly more capabilities than t, or by using strictly fewer capabilities than t. In particular, the strategy excludes any tests achieved by using *exactly* the capabilities p_1 and p_2 (whether together or in different transitions) and no others.

Proposition 4 (Strong capability-order equivalence). Let t be a successful test on CPS $\mathcal{P}$. A strategy $\mathrm{Excl}([t]_{\overset{\rightarrow}{\sim}})$ can be constructed such that for every successful test t' on $\mathcal{P}$:

$$\pi_{t'} \in L(\mathrm{Excl}([t]_{\overset{\rightarrow}{\sim}})) \text{ if and only if } t' \notin [t]_{\overset{\rightarrow}{\sim}}$$

Construction. Suppose that $\mathrm{COrd}(\pi_t) = Y_1 Y_2 \ldots Y_k$. Define:

$$
\begin{aligned}
\mathrm{Excl}([t]_{\overset{\rightarrow}{\sim}}) = \\
(\{q_0, q_1, \ldots, q_k\}, \\
\{(q_0, (\mathrm{true}, \_ \neq Y_1), q_k), (q_k, (\mathrm{true}, \mathrm{true}), q_k)\} \\
\bigcup_{i=1}^{k-1} \{(q_{i-1}, (\mathrm{true}, \_ == Y_i), q_i), (q_i, (\mathrm{true}, \_ == Y_i), q_i), \\
(q_i, (\mathrm{true}, \_ \neq Y_i \wedge \_ \neq Y_{i+1}), q_k)\}, \\
q_0)
\end{aligned}
$$

where $q_0, q_1, \ldots, q_k$ are fresh strategy states. □

For example, suppose that $\pi_t = PPPQQQPPP$ is the history of a successful test, where $P = \{p_1\}$, $Q = \{p_1, p_2\}$, and $\mathrm{COrd}(\pi_a) = PQP$. Then $\mathrm{Excl}([t]_{\overset{\rightarrow}{\sim}})$ is as depicted in Figure 8. The strategy characterises tests that do *not* consist of usages of P, followed by usages of Q, followed by usages of P again. Intuitively, it achieves this by using states q_1 and q_2 to 'track' the usages of Ps followed by Qs, with transitions to q_3 whenever a capability is used that makes it impossible to achieve the $P..Q..P..$ pattern. Note that from q_2 there are no transitions supporting the usage of P.

PROOF OF THEOREM 1

Proof Sketch. First, it is required to show that a capability history $\pi_{t'} = Y_1 Y_2 Y_3 \cdots$ is in $L(\mathcal{T} \parallel \mathrm{Excl}([t]_\sim))$ if and only if $\pi_{t'} \in L(\mathcal{T})$ and $\pi_{t'} \in L(\mathrm{Excl}([t]_\sim))$. This can be obtained by showing that any test trace in $\mathcal{T} \parallel \mathrm{Excl}([t]_\sim)$ that derives $\pi_{t'}$ can be mapped to a corresponding trace of states and transitions in $\mathcal{T}$ and $\mathrm{Excl}([t]_\sim)$ (and vice versa). This is evident from the definition of $\parallel$ as the composed strategy explicitly tracks (disjoint) state identifiers, combines all possible pairs of transitions, and fully synchronises on the steps of both strategies. The result is then obtained by appealing to the relevant Proposition of the equivalence class defined by the user (e.g. Propositions 1, 3, and 4), i.e. that $\pi_{t'} \in L(\mathrm{Excl}([t]_\sim))$ if and only if $t' \notin [t]_\sim$. □

IMPLEMENTATION FOR SWaT

The formalisation, equivalence classes, and algorithms we have presented are completely general: they can be applied to any test scenario characterised as performing sequences of sensor/actuator manipulations, and they do not require any modelling of the targeted CPS (which would be difficult to do in general, owing to the presence of physical processes). Our theory can be applied to a real system by identifying the sensors/actuators a tester can manipulate (along with their possible readings/configurations), an equivalence class of interest, and a test strategy of interest. Furthermore, in order to guide the search for effective tests from a strategy, it requires some black box predictive model of the system's behaviour (e.g. a simulator, or a neural network trained on sensor/actuator data of the real system).

To demonstrate the effectiveness of our theory in practice, we implemented causal fuzzing for SWaT as a suite of Python scripts, which are available online [24]. Our implementation
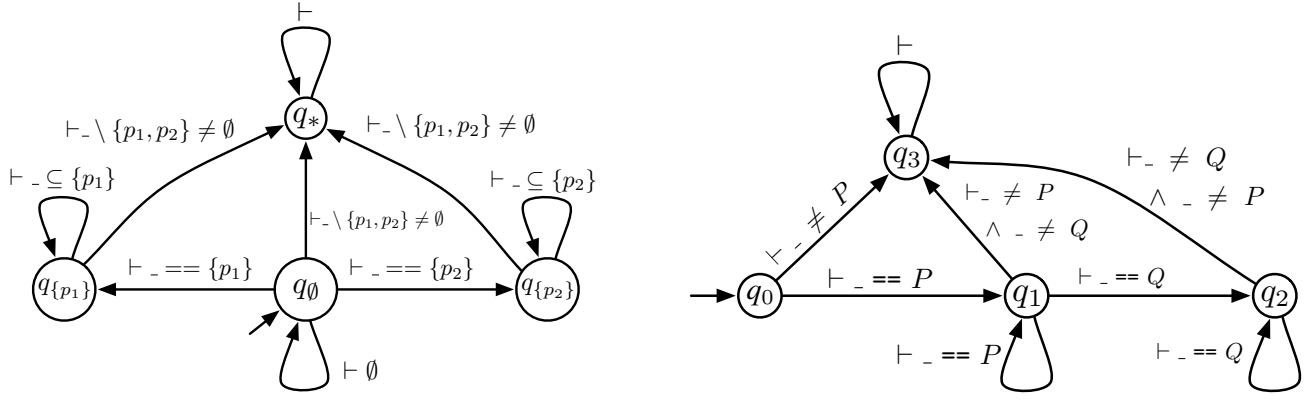
Fig. 8. *Left:* constructing $\mathrm{Excl}([t]_{\simeq})$ for $\mathrm{CSet}(\pi_t) = \{p_1, p_2\}$; *Right:* $\mathrm{Excl}([t]_{\rightsquigarrow})$ for $\mathrm{COrd}(\pi_t) = PQP$

is able to handle strong capability-set equivalence, strong capability-order equivalence, most importantly, Y-set equivalence, where Y is taken to be the *causal* capabilities of an identified test.

In Algorithm 2, to constrain the search space, we limit walks to three transitions, but use a longer fixed time interval of 10 minutes (determined experimentally; Section VI). As our prediction model, we make use of a high-fidelity simulator for SWaT [24], but note that other kinds of prediction models based on machine learning are available too [15], [16], [25]. We define simple objective functions in terms of the known safe operational ranges of sensors (similar to the fuzzer of [15]). To test whether a manipulation is causal (Algorithm 3), we make use of the actual testbed for all tests except those that target tank over- or underflows. For those tests, we use the simulator to avoid resource wastage, testing only the final output $t_{\min}$ on the real system.

Once a potential test has been identified and pruned, the rest of Algorithm 1 (i.e. identifying whether the potential test is *actually* successful) is executed on the SWaT testbed itself. To assess the effects of utilising capabilities, we fuzz the sensor readings and actuator commands accordingly over the network using the pycomm [47] package.

CAUSALLY DIFFERENT TESTS FOUND

Table II lists the causality sets identified in our evaluation (Section VI; Experiment #2).

TABLE II

Causal capability sets in tests identified by our fuzzer

Test Goal	Causal Capability Sets
FIT101 (High)	{[MV101, open]}
FIT201 (High)	{[MV201, open], [P101, on], [P102, on]}
FIT601 (High)	{[MV301, open], [MV303, open], [MV502, open], [P602, on]}
FIT101 (Low)	{[MV101, close]}
FIT201 (Low)	{[MV201, close]} {[P101, off], [P102, off]}
FIT301 (Low)	{[MV302, close]} {[P301, off], [P302, off]}
FIT401 (Low)	{[MV501, close]}
FIT501 (Low)	{[MV501, close], [MV503, close], [MV504, close]}
FIT601 (Low)	{[P602, off]}
DPIT301 (Low)	{[MV301, open], [MV303, open], [P602, on]} {[MV302, close]} {[P301, off], [P302, off]}
LIT101 (High)	{[MV101, open]} {[MV101, open], [P101, off], [P102, off]} {[MV101, open], [MV201, close]}
LIT301 (High)	{[MV201, open], [P101, on]} {[MV201, open], [P102, on]} {[MV201, open], [P101, on], [P301, off], [P302, off]} {[MV201, open], [P102, on], [P301, off], [P302, off]} {[MV201, open], [P101, on], [MV302, close]} {[MV201, open], [P102, on], [MV302, close]}
LIT401 (High)	{[MV302, open], [P301, on]} {[MV302, open], [P302, on]}
LIT101 (Low)	{[MV101, close], [MV201, open], [P101, on]} {[MV101, close], [MV201, open], [P102, on]}
LIT301 (Low)	{[MV201, close], [MV302, open], [P301, on]} {[MV201, close], [MV302, open], [P302, on]}
LIT401 (Low)	{[MV302, close], [P401, on]} {[P301, off], [P302, off], [P401, on]} {[MV302, close], [P402, on]} {[P301, off], [P302, off], [P402, open]}