Causality-Driven Test Case Minimisation for Cyber-Physical Systems

MICHAEL FOSTER, The University of Sheffield, United Kingdom
CHRISTOPHER M. POSKITT, Singapore Management University, Singapore
NICHOLAS R. LATIMER, The University of Sheffield, United Kingdom
NEIL WALKINSHAW, The University of Sheffield, United Kingdom
RICHARD SOMERS, The University of Sheffield, United Kingdom
ROBERT M. HIERONS, The University of Sheffield, United Kingdom

Cyber-physical systems allow digital control systems to interact with the physical world using sensors and actuators. They are increasingly being used to automate critical infrastructure, where software faults can have dire consequences. Due to the complex nature and unpredictability of these systems, their resilience is often tested using a technique called fuzzing, which generates quasi-random sequences of sensor and actuator manipulations with the goal of forcing a system into unsafe states. However, there is currently no way of determining which manipulations of a test case cause a failure without systematically removing each one and re-running the test, which can be extremely time-consuming and expensive. In this work, we present CausalCut, a technique that uses causal inference to estimate the causal contribution of each intervention from pre-existing runtime data, thereby reducing the number of times tests must be re-run. We evaluated CausalCut by applying it to two very different systems: an artificial pancreas and a water treatment plant. CausalCut typically managed to remove more than half of the spurious manipulations using fewer executions than the current state of the art, which represents a saving of up to 18 hours and $6300 per test case.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Causal reasoning and diagnostics.*

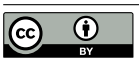
Additional Key Words and Phrases: Cyber-physical systems, fuzzing, test diversity, equivalence classes, causality, causal inference, test minimisation

ACM Reference Format:
Michael Foster, Christopher M. Poskitt, Nicholas R. Latimer, Neil Walkinshaw, Richard Somers, and Robert M. Hierons. 2026. Causality-Driven Test Case Minimisation for Cyber-Physical Systems. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (January 2026), 48 pages. https://doi.org/10.1145/3816435

1 Introduction

Cyber-physical systems (CPSs) give software the capability to respond to and interact with the physical world. Unlike traditional software, they are typically *hybrid systems* that combine continuous aspects, such as variation in the temperature or volume of liquid in a vessel, with discrete aspects, such as whether a pump or heater is switched on in the given state. These systems play a pivotal role in many areas of society and are increasingly being used to automate critical infrastructure systems,

Authors' Contact Information: Michael Foster, m.foster@sheffield.ac.uk, The University of Sheffield, Sheffield, United Kingdom; Christopher M. Poskitt, cposkitt@smu.edu.sg, Singapore Management University, Singapore, Singapore; Nicholas R. Latimer, n.latimer@sheffield.ac.uk, The University of Sheffield, Sheffield, United Kingdom; Neil Walkinshaw, n.walkinshaw@sheffield.ac.uk, The University of Sheffield, Sheffield, United Kingdom; Richard Somers, The University of Sheffield, Sheffield, United Kingdom; Robert M. Hierons, r.hierons@sheffield.ac.uk, The University of Sheffield, Sheffield, United Kingdom.



ACM 1557-7392/2026/1-ART
https://doi.org/10.1145/3816435

where failures or outages can have dire consequences. However, their combination of discrete and continuous dynamics is known to make a range of activities, including testing, particularly difficult [20].

The main practical challenge here is that CPSs respond to a continuous stream of sensor data and adjust the states of their actuators based on these data. Tests are therefore *sequences* of events rather than a single initial configuration. This leads to a particularly large (arguably infinite) input space, making it difficult to generate tests that reveal faults while still representing feasible usage patterns. Additionally, test runs can take a long time and be extremely expensive. For example, the water treatment plant that we use as part of our evaluation costs up to $350 (USD) per hour [34] to run. This is compounded if the system under test is nondeterministic, since it is then necessary to run each test case multiple times. Finally, because CPSs interact with the physical world, testing them can be dangerous or may require environmental conditions that are impractical to achieve. While simulated environments can be used to mitigate this, the simulations can still be time-consuming and computationally intensive to run [30].

A popular test-generation method for CPS resilience is guided fuzzing [10, 11]. This involves using a machine learning model learnt from data to guide the search for new failure-inducing input sequences by quasi-randomly mutating existing sequences and predicting the probability of failure. However, the random nature of these techniques means they tend to produce tests that include events that are not required to bring about the failure [50]. This can increase the time and cost of running tests, and make it difficult to trace the causes of a failure. It can also lead to test suites that lack diversity in that they are dominated by tests that manifest the same fault and only differ in irrelevant events

The potential for tests to include irrelevant events motivated Poskitt et al. [50] to add a minimisation step to their test generation technique. Given a failing test (sequence of events) produced by a fuzzer, the goal is to determine which of the events were necessary to cause the failure. As Poskitt et al. note, this is an NP-hard combinatorial problem [50], so there is no scalable optimal algorithm. Instead, they use a greedy heuristic that takes a failing test case and iteratively removes events from the test, rerunning the system to check that the failure remains, until the removal of any single event leads to the failure no longer occurring. While Poskitt et al. show that their approach led to a test suite that was four orders of magnitude smaller and found over twice as many structurally different ways to reach the same failure conditions than a conventional approach, repeatedly running tests can be expensive. In addition, it can be difficult to reliably repeat test runs for nondeterministic systems.

In this paper we present CausalCut, a novel test case minimisation approach that uses ideas from causal inference to reduce the number of reruns needed to minimise a test sequence. CausalCut has two phases. In the first phase, we use a set of pre-existing execution data and a directed graph, representing the expected causal relationships between variables, to estimate the causal contribution of each event in the test. We then remove events that do not have a significant relationship to the failure. An important feature of CausalCut is that this first phase does not require additional test runs. However, depending on the amount and suitability of the available data, the causal effect estimates may not be accurate or we may not be able to calculate them at all. If phase 1 results in a test that no longer leads to a failure, phase 2 iteratively reinstates events in order of their estimated causal contribution, until the failure manifests once again.

We evaluated CausalCut against Poskitt et al.'s greedy heuristic [50] using two real CPSs that are difficult to test and differ greatly in nature and application domain: an artificial pancreas system used by Somers et al. [54] and the water treatment system used in [50]. For the artificial pancreas system, CausalCut was 17% more cost efficient than the greedy heuristic on average, and up to 96% in the best case. For the water treatment system, even though the available data did not allow

a causal effect to be estimated for more than half of the interventions, CausalCut was still 2.8% more cost efficient on average, and up to 83.3% in the best case. For the longest test cases, this represents a saving of up 18 hours and $6,300.

This paper makes the following main contributions.

- A technique based on causal inference to discover the events in a test that are necessary to bring about a failure. To the best of our knowledge, this is the first work to apply statistical causal inference to this problem.
- An openly available implementation of this technique as an extension to the Causal Testing Framework [25].
- An experimental evaluation of CausalCut in the context of two real cyber-physical systems.

The remainder of this paper is structured as follows. Section 2 presents two real CPSs that serve as motivating examples for our work and formalises the test case minimisation problem. Section 3 introduces the aspects of causal inference that are directly relevant to our work. For a broader introduction, we refer the reader to [31, 48, 49]. Section 4 presents CausalCut, our test-minimisation technique. Section 5 presents a theoretical analysis of the cost and minimisation capabilities of CausalCut in terms of best and worst case scenarios. Section 6 lays out the methodology of our empirical evaluation, before Section 7 presents the results and Section 8 provides a more detailed discussion. Section 9 summarises related work. Finally, Section 10 concludes the paper.

2 Test Case Minimisation for Cyber-Physical Systems

We start this section by introducing two CPSs that motivate out work. Following this, we formally define CPSs and CPS tests. This then allows us to precisely characterise the test case minimisation problem.

2.1 Motivating Examples

Unlike traditional software domains, where large numbers of benchmark systems and datasets are readily available, empirically evaluating cyber-physical systems is constrained by the limited availability of realistic systems with accessible execution data. The two systems below were selected to balance realism, diversity, and data availability.

Our first motivating system is the Artificial Pancreas System (APS), which will be used as a running example for the rest of this section. APSs provide a modern approach to managing type 1 diabetes mellitus by using a continuous glucose monitor sensor, an insulin pump, and a control algorithm [58] to mimic the behaviour of a healthy pancreas, as illustrated in Figure 1. OpenAPS[1] is one such system with over 3,000 real users across the world and has amassed an estimated 100M operational hours[2]. It uses the oref0 control algorithm to determine how much insulin to inject at any given time to keep the body's blood glucose within a safe range, while accounting for day-to-day activities such as eating and exercise. There are two main test goals here: hypoglycaemia (low blood glucose) and hyperglycaemia (high blood glucose), both of which can have life-threatening consequences.

Our second motivating system is the Secure Water Treatment (SWaT) testbed [35, 44]: a fully operational, scaled-down industrial control system that replicates a modern water purification plant and can produce up to 19 litres of clean drinking water per minute. Figure 2, reproduced from [50, Figure 2], provides an overview of the six stages of SWaT, and the main sensors and actuators involved. Each stage has its own programmable logic controller that continuously reads from sensors and actuates valves and pumps to maintain safe operation. The system features 36

[1]https://openaps.org/

[2]https://openaps.org/outcomes/

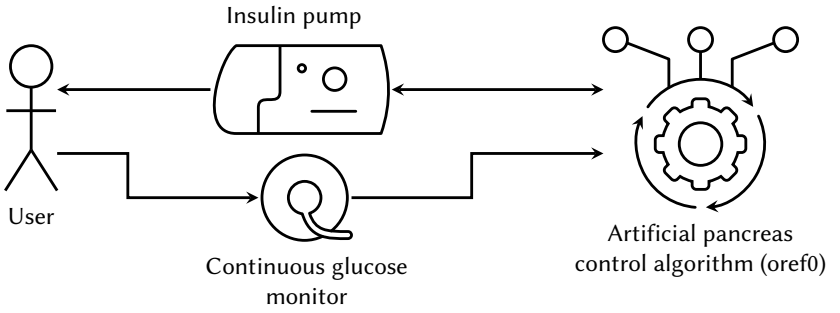


Fig. 1. The interaction between an APS and its user. Blood glucose data is measured by the monitor and sent to the APS control algorithm. The algorithm then, along with historical insulin data, suggests insulin prescription. The insulin pump prescribes the insulin based on the algorithm's output.

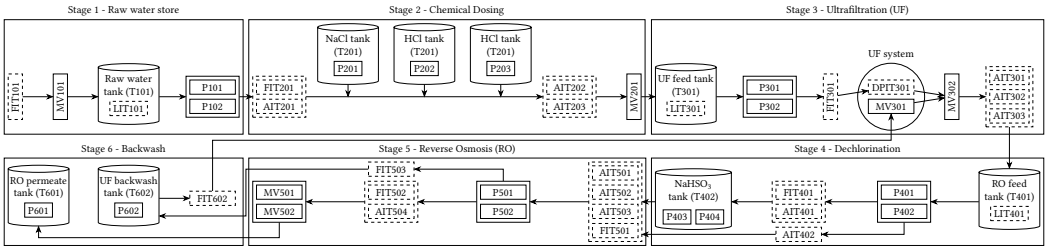


Fig. 2. Overview of the six sub-stages of SWaT. Arrows indicate water flow; dashed/solid rectangles indicate sensors/actuators, which include water flow indicator transmitters (FITs), tank level indicator transmitters (LITs), chemical analyser indicator transmitters (AITs), motorised valves (MVs), and pumps (Ps).

sensors and 30 actuators in total, each with well-defined safe operating ranges (e.g. tank underflow or pipe overpressure) that must be maintained to avoid damage or failure. The test goals here are then to exceed these ranges.

SWaT is a widely adopted reference system for cyber-physical systems research, particularly in industrial control system security, testing, and failure analysis. Since its introduction by Mathur and Tippenhauer [44], the SWaT testbed has accrued over 1,000 citations, and its publicly released datasets [35] have received more than 7,800 access requests, reflecting its sustained relevance and broad community uptake. Furthermore, it was targeted by the causal fuzzing approach [50] that this paper builds upon.

While OpenAPS and SWaT are very different systems in nature, scale, and domain, they share several common characteristics fundamental to CPSs that make them difficult to test.

Risk: System-level testing in real environments can be dangerous. In OpenAPS, incorrect insulin doses may cause hypo/hyperglycaemia, which can be life-threatening, so testing with real users must be limited to safe configurations. In SWaT, tanks can overflow and pipes can burst if put under too much pressure, causing costly damage to equipment and downtime while the system is repaired.

Test Duration and Costs: CPSs combine software with physical components that react in real time, often over extended durations. For instance, in OpenAPS, insulin may take 10–30 minutes to act, with effects lasting hours. SWaT operates over similar time scales, and can cost up to $350 per hour to run. This makes large-scale testing impractical and prohibitively expensive.

Nondeterminism: Sensor readings are inherently noisy and can be affected by uncontrollable environmental factors. As such, the response to any given configuration is nondeterministic and must be assessed statistically over multiple runs. For example, when OpenAPS adjusts insulin delivery to manage blood-glucose levels, the user's physiological response will depend on factors such as sleep, exercise, and sensitivity to insulin [17, 54]. When SWaT opens a valve to allow water into or out of the system, the flow rate will depend on the external water pressure outside the system.

Both systems support research by providing extensive log data. OpenAPS provides traces from real users of the system during normal operation [42] that frequently oscillate between hypo- and hyperglycaemia. SWaT provides a publicly available dataset [28, 35] containing logs from 11 days of continuous operation under both normal and attack scenarios. However, these datasets are far from complete. Somers et al. [54] note that much of the available OpenAPS data is not well-formed: logs contain time-skips, bad sensor readings, and human errors such as forgetting to record meals, while the SWaT data does not achieve every test goal or even implement every (discrete) intervention. That is, some sensors remain within their safe range, and some valves remain open throughout the entire run.

Somers et al. [54] facilitate the collection of synthetic OpenAPS data by providing a configurable equational model that can be used to simulate the dynamics of OpenAPS users. This prevents the need to put real patients at risk, but is still rather slow and computationally intensive. Collecting the runs for our evaluation in Section 6 took approximately a month and used $5,000 worth of HPC time. Chen et al. [10] do provide a Python-based SWaT simulator that replicates the control logic and selected physical dynamics. However, this simulator does not expose values for all sensors and actuators present in the real SWaT system, and does not support arbitrary or fine-grained intervention sequences over time. As a result, it is not possible to obtain complete execution traces or to run the kinds of counterfactual and intervention-heavy simulations required by our causal analysis.

2.2 Cyber-Physical Systems

To provide a precise characterisation of CPSs, we adopt the formal definitions proposed by Poskitt et al. [50]. Fundamentally, a CPS consists of software that interacts with physical processes via two types of interface components: sensors and actuators. Sensors capture continuous data from the physical state (e.g. glucose monitors), whereas actuators are mechanised devices (e.g. insulin pumps) that regulate or influence the process.

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. □

For instance, if an actuator is an APS insulin pump, its domain could be defined as {`on`, `off`}. In real APS systems, insulin pumps tend to deliver insulin at a particular *rate* defined over a continuous range. However, our temporal causal analysis approach (Section 3) requires a finite set of values. In such cases, an appropriate discretisation of that range should be selected, for example {`none`, `small`, `large`}.

We denote the current states of the components in a CPS as a set of component-value pairs. Note that for sensor and actuator components, we refer to their current states as their *readings* and *statuses* respectively.

Definition 2 (Component states)**.** Given a set of components C, the *component states* 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* component states $\overline{C}$. □

At a high level, a CPS is a system consisting of three layers: the *control* layer, containing traditional deterministic programming logic; the *physical* layer, in which continuous processes (e.g. water flow) evolve over time; and the *component* layer, containing sensors and actuators to allow the control and physical layers to interface with each other.

Formally, a CPS is a set of control states, physical states, components, and a number of transition functions that characterise how sensor and actuator states influence the control and physical states respectively. Note that in real-world systems like APS or SWaT, these transitions are often implicit and observed through execution rather than explicitly defined.

Definition 3 (Cyber-Physical System)**.** We define a *Cyber-Physical System (CPS)* $\mathcal{P}$ as an eight-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, $d_{\mathcal{P}} : X_{\mathcal{P}} \times \mathbb{A}_{\mathcal{P}} \to X_{\mathcal{P}}$ is a function describing how the physical process evolves after a fixed time interval $\tau_{\mathcal{P}}$, and $\theta_{\mathcal{P}} : X_{\mathcal{P}} \to \mathbb{S}_{\mathcal{P}}$ is an observation function describing how sensor readings are extracted from the state of the physical process. □

Given a CPS $\mathcal{P}$, its behaviour unfolds as a sequence of control and physical states over fixed time steps, $\tau_{\mathcal{P}}$. At each step, the system observes the current physical state through sensors, uses this information to determine control actions via its control logic, and then applies these actions through actuators, causing the physical process to evolve. This interaction results in a new control state and a new physical state, and the cycle repeats. A sequence of such steps forms an *execution* of the system.

Definition 4 (System execution)**.** An *execution* of a CPS $\mathcal{P}$ over fixed time steps $\tau_{\mathcal{P}}$ is a sequence of the form $(q_0, x_0) \to (q_1, x_1) \to \cdots$ such that each $q_i \in Q_{\mathcal{P}}$, $x_i \in X_{\mathcal{P}}$, and for every step $(q_i, x_i) \to (q_{i+1}, x_{i+1})$, $\delta_{\mathcal{P}}(q_i, \theta_{\mathcal{P}}(x_i)) = (q_{i+1}, \overline{A}_{\mathcal{P}})$ with $\overline{A}_{\mathcal{P}} \in \mathbb{A}_{\mathcal{P}}$ and $d_{\mathcal{P}}(x_i, \overline{A}_{\mathcal{P}}) = x_{i+1}$. □

2.3 Interventions and Tests

In our setting, a failure-inducing test is a sequence of interventions — manipulations of sensor readings or actuator commands [48] — that results in a system failure. We formalise this by adapting the definitions proposed by Poskitt et al. [50]. First, we define the *domain of interventions* for a CPS, i.e. the set of possible interventions that are available to the tester. An intervention is a manipulation of a sensor reading or actuator command, and is denoted as a component-value mapping. In particular, an intervention $s \mapsto v$ 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 mapping $a \mapsto v$ for actuator a and value $v \in D_a$ indicates that the tester is capable of forcing actuator a into status v, regardless of any actual command being issued to it at the given moment. Note that our model allows multiple interventions to occur at a given time point, and abstracts away from how these are realised by a tester, specifying only that they can be.

Definition 5 (Domain of interventions; intervention; intervention set)**.** Let $\mathcal{P}$ be a CPS. A *domain of interventions* $\mathbb{I}$ for $\mathcal{P}$ is a set of *interventions* $c \mapsto v$ such that c is a component in $S_{\mathcal{P}} \cup A_{\mathcal{P}}$ and $v \in D_c$. An *intervention set* is any finite subset $I \subseteq_{\text{fin}} \mathbb{I}$ containing at most one intervention per component, i.e. if $c \mapsto v_1 \in I$ and $c \mapsto v_2 \in I$ then $v_1 = v_2$. □

For example, a domain of interventions $\{\texttt{pump} \mapsto i \mid i \in \{\texttt{none}, \texttt{small}, \texttt{large}\}\}$ for the APS expresses that the tester can override the status of the insulin pump actuator to force it to give the user a `small` or `large` dose of insulin, or `none` at all. This notion of intervening to override a system's default behaviour parallels the causal inference techniques we discuss in Section 3.

Next, we define *tests*, which are sequences of (possibly empty) intervention sets. A *test execution* is similar to a system execution (Definition 4) but with these interventions applied at the corresponding time steps. Applying interventions results in *modified observation functions* and *modified actuator commands*, leading to a sequence of control and physical states that might not otherwise be observed among the normal (unmodified) runs of a system. Formally, given a CPS $\mathcal{P}$ and domain of interventions $\mathbb{I}$, the application of an intervention set I results in a modified observation function, $\theta^I_{\mathcal{P}}$, 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 $s \mapsto v' \in I$. Analogously, given a set of actuator statuses $\overline{A}_{\mathcal{P}} \in \mathbb{A}_{\mathcal{P}}$, the application of an intervention set I results in a modified set of statuses $\overline{A}^I_{\mathcal{P}}$, obtained from $\overline{A}_{\mathcal{P}}$ by replacing elements $\langle a, v\rangle$ with $\langle a, v'\rangle$ if $a \mapsto v' \in I$. Note that an intervention set may be empty ($I = \emptyset$), i.e. representing a 'do nothing' event. In this case $\theta^I_{\mathcal{P}} = \theta_{\mathcal{P}}$ and $\overline{A}^I_{\mathcal{P}} = \overline{A}_{\mathcal{P}}$. We remark that a system execution is essentially equivalent to a test execution in which every intervention is the empty set, $\emptyset$.

Definition 6 (Test; test execution)**.** Given a CPS $\mathcal{P}$ and a domain of interventions $\mathbb{I}$, a *test* t for $\mathcal{P}$ is a sequence of (possibly empty) intervention sets $I_0 I_1 \cdots$. An *execution* of t on some initial states q_0, x_0 is a sequence of the form $(q_0, x_0) \rightarrow_{I_0} (q_1, x_1) \rightarrow_{I_1} \cdots$ where each $q_i \in Q_{\mathcal{P}}$ and $x_i \in X_{\mathcal{P}}$. Furthermore, for every step $(q_{i-1}, x_{i-1}) \rightarrow_{I_{i-1}} (q_i, x_i)$ in the sequence, it is the case that $\delta_{\mathcal{P}}(q_{i-1}, \theta^{I_{i-1}}_{\mathcal{P}}(x_{i-1})) = (q_i, \overline{A}_{\mathcal{P}})$ and $d_{\mathcal{P}}(x_{i-1}, \overline{A}^{I_{i-1}}_{\mathcal{P}}) = x_i$. □

Notation 1 (Time-indexed interventions)**.** In practice, many tests only apply interventions at specific timesteps, leaving the system to evolve naturally at others. To simplify notation, we treat the omission of an intervention set at a given timestep as implicitly applying the empty intervention set. For example, writing $(\{\texttt{pump} \mapsto \texttt{on}\}, 3)(\{\texttt{pump} \mapsto \texttt{off}\}, 5)$ is shorthand for the test $\emptyset\emptyset\emptyset\{\texttt{pump} \mapsto \texttt{on}\}\emptyset\{\texttt{pump} \mapsto \texttt{off}\}$, where interventions are applied only at timesteps 3 and 5. □

A test execution is *failure-inducing* if, in a finite number of steps, it brings the system into a state that satisfies a *test goal* expressed over the (actual) sensor readings. For instance, a test goal for the APS could be blood glucose < 4mmol/L. This represents a state of hypoglycaemia (low blood sugar), which can be life threatening if left untreated. Thus, test goals represent undesirable system states that the testing process attempts to lead the system into.

Definition 7 (Test goal; failure)**.** A *test goal* over a CPS $\mathcal{P}$ is a Boolean formula of simple linear (in)equalities over the sensors $S_{\mathcal{P}}$. Given a test goal γ and physical state $x \in X_{\mathcal{P}}$, the *valuation* of the goal, $\gamma^x \in \mathbb{B}$, is obtained by evaluating the Boolean expression in the standard way, but with each sensor symbol s interpreted as v such that $\langle s, v\rangle \in \theta_{\mathcal{P}}(x)$. If γ^x holds true, we say that a *failure* has occurred. □

Definition 8 (Failure-inducing test)**.** Given a CPS $\mathcal{P}$, a finite test t, and test goal γ, we say that t is *failure-inducing* on states q_0, x_0 if executing t leads to an execution $(q_0, x_0) \rightarrow_{I_0} \cdots \rightarrow_{I_{n-1}} (q_n, x_n)$ for which there exists some $m \leq n$ such that γ^{x_m} is true. We denote this $t \vdash \gamma$. □

Note that Definition 8 is suffix-closed. That is, any failure-inducing test t for test goal γ remains failure-inducing for γ if an additional sequence of interventions t_x is appended to the end to form a new test $t' = t \cdot t_x$, even if the system returns to a state such that γ no longer holds while executing the t_x suffix. Conversely, every failure-inducing test $t = I_0, \ldots, I_n$ has a failure-inducing prefix $I_0, \ldots, I_m$ with $m \leq n$ for which there is no shorter failure-inducing prefix.

2.4 Identifying Failure-Inducing Tests

A common approach for identifying failure-inducing CPS tests is fuzzing [10, 11, 37, 61]. CPS fuzzers execute the system with large numbers of (pseudo-)random interventions in search of sequences that lead to failures. However, the challenges discussed in Section 2.1, namely the cost, risk, and nondeterminism associated with CPS test cases, make CPS fuzzing especially challenging compared to fuzzing traditional software. To mitigate these challenges, CPS fuzzers employ machine learning models that predict the effects of potential manipulations [9, 50]. This helps to reduce the number of times that the system under test is physically run by only executing it on inputs that are predicted to lead to a violation.

One limitation of fuzzers is their focus on the outcome (CPS failures) without considering what the *causes* of these failures might be. This leads to two consequences [50]. Firstly, generated test cases can contain large numbers of spurious interventions, which can make them unnecessarily expensive to run and difficult to debug. Secondly, it can lead to test suites that lack diversity and are dominated by minor variants of the same failure-inducing test case that only differ in terms of spurious events. In both cases, it is desirable to be able to 'minimise' the test to the interventions that are essential to bringing about the failure, thereby minimising the cost of a test execution, and making it easier to avoid repeatedly generating minor variants of tests that have already been identified.

2.5 Test Case Minimisation

In the following, we formally characterise the aforementioned test case minimisation problem for CPSs. Given a failure-inducing test (e.g. found by a fuzzer), the goal is to minimise it by finding the smallest 'subset' of interventions that result in the same failure (i.e. the same test goal). In determining the necessary interventions for a given failure, we not only provide the test engineer with more specific information about the failure, but also reduce the cost of testing and help to create more diverse test suites by guiding fuzzing away from tests with the same necessary interventions.

Definition 9 (Test reduction)**.** Let t denote a failure-inducing test for test goal γ. A test t' is a *reduction* of t, denoted $t' \sqsubseteq_\gamma t$, if $t' \vdash \gamma$ and t' can be obtained from t by pruning intervention sets I_i from the sequence or removing individual interventions $c_i \mapsto v_i$ from intervention sets. Note that for notational simplicity, we occasionally write $\sqsubseteq$ instead of $\sqsubseteq_\gamma$ if the test goal γ is clear from the context. □

The context of the test goal γ is critical here since it may still be possible to remove interventions from a test and have the result be failure-inducing for a *different test goal* γ'. For example, we may start with a test t for OpenAPS that brings the patient into a state of hypoglycaemia and, through removing interventions, end up with a test t' that brings the patient into a state of hyperglycaemia. While both tests are failure-inducing — both put the patient's blood glucose level outside the safe range — the resulting failures achieve different test goals, so t' is not a reduction of t.

Informally, t' is *minimal* with respect to t and γ if no reduction of t with respect to γ has fewer interventions. Note that there may be multiple minimisations for t if it contains a high level of redundancy. There may also be reductions of t from which no further interventions can be removed while still achieving γ but which contain more interventions than t'. We call such reductions *locally minimal*. For example, a test t for the OpenAPS system may contain four consecutive doses of insulin (one large, then two small, then another large) that lead to the patient experiencing hypoglycaemia. If it is case that either a single large dose or two small doses are sufficient to induce hypoglycaemia, then there are two minimal tests here: either the first or last dose of insulin. The two small doses represent a test which is locally minimal.

Algorithm 1 The greedy heuristic used by Poskitt et al. [50]

```
function GREEDY(testCase)
    for interventionSet ∈ testCase do
        for i ∈ interventionSet do
            interventionSet ← interventionSet \ {i}  ▷ Remove i from the test and see if it is still failure-inducing
            if testCase is no longer failure-inducing then  ▷ If not, put back i
                interventionSet ← interventionSet ∪ {i}
    return testCase
```

Definition 10 ((Locally) minimal test; Optimal test)**.** Let $\mathcal{P}$ denote a CPS, γ a test goal, and t, t' be failure-inducing tests for γ on states q_0, x_0. We say that t' is *locally minimal* with respect to t if (1) $t' \sqsubseteq_\gamma t$; and (2) $\forall t''.\, t'' \sqsubseteq_\gamma t' \implies t' \sqsubseteq_\gamma t''$. We say that t' is *minimal* with respect to t and γ if the above conditions are met and $\nexists t''.\, t'' \sqsubseteq t \wedge \sum_{I \in t''} .|I| < \sum_{I \in t'} .|I|$. Additionally, t' is *optimal* if $\nexists t''.\, t'' \vdash \gamma \wedge \sum_{I \in t''} .|I| < \sum_{I \in t'} .|I|$. □

Note that, even if test t' is minimal, this is still qualified with respect to some "original" test case t, as may be produced by some test generation technique. For a test to be *optimal*, there must not exist any test with fewer interventions that achieves γ. For example, if t achieved hypoglycaemia by giving the patient two small doses of insulin and a snack, then the two small doses of insulin would represent a minimal test, but would not be optimal since we can achieve hypoglycaemia using just a single large dose of insulin. In this paper, we focus on the test minimisation problem rather than the task of finding *optimal* tests, which is a test-generation problem.

Definition 11 (Test minimisation problem)**.** Given a failure-inducing test t for test goal γ, the *test minimisation problem* is to find a minimal test with respect to t and γ.

As noted by Poskitt et al. [50], test minimisation is a combinatorial optimisation problem in this context, which is known to be NP-hard, so there is no efficient method of precisely minimising a given test case. To mitigate this, they proposed the *greedy heuristic* outlined in Algorithm 1, which we subsequently refer to as Greedy. Intuitively, Greedy iteratively removes interventions from a failure-inducing test, re-executes the test on the CPS, and observes whether the failure still occurs. To the best of our knowledge, Greedy represents the current state of the art for CPS test minimisation, and forms the baseline of our evaluation in Section 6.

For more traditional programs, delta debugging [67] (which we subsequently refer to as DDmin) is a popular test case minimisation technique that follows an approach similar to binary search. DDmin searches for the interventions in the test case that are necessary to achieve a given test goal by iteratively dividing the test case into smaller components, and increasing the level of granularity when the test goal is no longer achieved. We refer the reader to [67] for the full algorithm.

For a test with n interventions, both Greedy and DDmin are guaranteed to produce a minimal result using n executions of the system as long as there is no interaction between interventions. However, the approaches can become problematic in the face of nondeterministic and expensive CPS test cases. Rerunning tests is time-consuming and expensive, and a system's nondeterministic nature can make violations difficult to repeat, making it impossible to precisely quantify the contribution of individual intervention to the violation from a single rerun of the system.

Furthermore, if interventions do interact with each other to bring about a failure, there may be groups of interventions that can be removed together but not individually. A somewhat byzantine example of this is a system that trivially fails for tests containing an odd number of interventions and succeeds for tests with an even number of interventions. Any failure-inducing test can thus be

reduced to a single intervention, but Greedy will fail to reduce the test at all since the removal of any individual intervention stops the test from being failure-inducing.

In fact, both Greedy and DDmin are only guaranteed to produce a 1-minimal result [67]. That is, no intervention can be *individually* removed from the result while still achieving the original test goal, even though there may still be sets of interactions which can be removed all together. We reproduce Zeller and Hildebrandt's formal definition in Definition 12.

Definition 12 (n-minimality; 1-minimality)**.** A test t is n-minimal for test goal γ if $\forall t' \sqsubseteq t.\, |t| - |t'| \leq n \implies t \nvdash \gamma$. Thus, a test is 1-minimal if $\forall I_n \in t.\, t - I_n \nvdash \gamma$.

While DDmin somewhat mitigates the effect of interaction by evaluating more combinations than Greedy, and will be unaffected by the above example, its performance can still be degraded by interaction. Furthermore, as we will examine in Section 5, it can require significantly more executions than Greedy, even without interaction, and is not guaranteed to produce a better result. While attempts have been made to reduce this [60], the technique relies on assumptions that do not necessarily apply for cyberphysical systems. This perhaps explains why, to the best of our knowledge, Greedy represents the current state of the art in cyberphysical test case minimisation.

2.6 Summary and Next Steps

In this section, we introduced the motivating challenges of testing CPSs, formalised the notions of interventions, (failure-inducing) tests, and the problem of test case minimisation. We also highlighted the NP-hard nature of minimisation, together with the limitations of Greedy in the face of interaction, nondeterminism, and high execution costs. To address these challenges, we require a principled way of reasoning about the causal contribution of interventions to failures without relying solely on repeated executions. In the next section, we introduce causal inference, a body of techniques that provides the statistical and conceptual foundation for such reasoning, before applying it to the test minimisation problem in Section 4.

3 Causal Inference

Causal Inference (CI) [48] is a family of statistical techniques that enables us to estimate *causal* effects, whereas traditional statistical methods are limited to *associational* reasoning. Informally, the causal effect of a treatment X on an outcome Y is the expected change in Y that arises from an intervention on X. For example, the expected change in blood glucose level Y brought about by a dose of insulin X. By employing domain knowledge supplied by the user, CI techniques can identify and remove sources of bias in data, allowing causal conclusions to be drawn from pre-existing, uncontrolled data. These techniques are increasingly being applied in the context of software testing, where they excel at testing properties of nondeterministic systems for which it is difficult to obtain large numbers of carefully controlled executions, such as computational models [15] and autonomous driving systems [26, 27].

Estimating the causal effect of X on Y using CI has four main steps: (1) Specify the causal graph, (2) Collect test data, (3) Perform causal identification, (4) Estimate the causal effect. These are elaborated in the following sub-sections. In principle, everything except the initial formation of the causal model can be (semi-)automated.

3.1 Specify the Causal Graph

Domain knowledge is commonly supplied in the form of a causal graph [31, 48]. This sets out the relevant variables within the system, and shows which variables could feasibly affect each other. In our case, these variables will be the components (sensors and actuators) of the CPS under test.

Definition 13. A causal graph G is a directed graph $G = (N, E)$ comprising a set of nodes representing random variables, N, and a set of edges, E, representing causality between these variables, where:

(1) The presence of an edge $N_i \rightarrow N_j$ represents the possible presence of a direct causal effect of N_i on N_j.
(2) The absence of an edge $N_i \rightarrow N_j$ represents a known causal independence between N_i and N_j.
(3) All common causes of any pair of variables on the graph are themselves present on the graph. □

Causal graphs form an intuitive representation of the system under test, and are widely used in fields such as epidemiology and sociology [31], where they are often hand drawn by researchers based on the literature and discussions with domain experts. In CI, the absence of an edge in a causal graph represents a much stronger assumption than the presence of one [48]. Where it is uncertain whether or not N_i has a direct causal effect on N_j, it is safer to include the edge in the graph.

As with any model-based technique, drawing a causal graph requires careful application of domain knowledge. However, an advantage over traditional software models, such as finite-state machines [12], is that causal graphs are very lightweight: they do not specify the precise form of the relationships between variables, merely their existence. For instance, we can construct a graph for SWaT directly from Figure 2, which simply depicts how sensors and actuators are connected together and does not require knowledge in the dynamics of water treatment.

While it is now fairly common [41] for cyber-physical systems to be developed from or alongside models like Figure 2, if there is no existing model of the system on which to base the causal graph, the approach described in Clark's thesis [14, Section 5.4.1] can be followed. We assume that there is a known set of variables N and start from a graph where every pair of nodes is connected (any variable may have a causal effect on any other variable). We then remove all of the edges where there is a temporal ordering that would prevent a causal effect (i.e. if variable A physically has to change its value before variable B, then it is not possible that a change in the value of B could affect A). Then it is a matter of applying domain knowledge to remove further edges where an underlying direct causal effect is known to be impossible.

The above approach is conservative, in the sense that a pair of variables are connected by default, unless there is firm knowledge to indicate that there should not be a causal link. This does allow for the possibility that spurious causal edges remain in the graph (i.e. edges that do not correspond to a genuine causal effect, but for which there is a lack of domain knowledge to rule them out). Empirical results on the use of causal DAGs for metamorphic testing (of traditional software systems) [16] indicate that causal analyses tend to be relatively robust to DAG misspecification, especially when edges are superfluous (missing edges have a more pronounced impact).

3.2 Collect Test Data

The second step is to collect data from which to estimate the causal effect. A major benefit of CI is that this data can be "observational", i.e., collected without needing to tightly control the inputs [16, 26]. The advantage of this from a software engineering standpoint is that we can reuse the same data to investigate multiple causal relationships, and can even use pre-existing log data recorded during normal use, which we demonstrate as part of our evaluation in Section 6.

One important caveat is that the test data must satisfy the *positivity* assumption, which is fundamental to many statistical methods used for CI [31]. Formally, this means that the probability of any treatment we wish to investigate (typically an input configuration when working in the

software testing context) must be non-zero. Intuitively, this means we need a good coverage of the input space. In practice, this can limit what we can effectively analyse when using pre-existing log data, especially when exploring causal effects over time. We will discuss this further in Section 3.5, and present our own results using observational data in Section 7.

3.3 Perform Causal Identification

Causal graphs present a useful basis for examining causal relationships between variables in data because they show which variables can directly or indirectly affect each other. In other words, if we wish to establish the causal effect of variable X on Y, the analysis of paths in the graph [48] enables the identification of variables that must be adjusted (controlled for) to remove bias and isolate the causal effect. In CI this activity is referred to as 'identification' [48].

A common source of bias is *confounding*, where a third variable Z directly affects both the treatment X and the outcome Y, as illustrated in Figure 3a. This can introduce a spurious correlation between X and Y, even if there is no direct causal link between the two. Another potential source of bias is *mediation*. This arises when the causal effect of treatment X on outcome Y "flows through" a third variable Z, as illustrated in Figure 3b. From the data alone it may not be clear to what extent X is responsible for a change in Y because of Z.

(a) Causal graph representing confounding. (b) Causal graph representing mediation.

Fig. 3. Causal structures of confounding and mediation.

Identification aims to identify problematic variables so that their influence on the relationship between the treatment and outcome of interest can be 'blocked'. For example, in Figure 3, if we want to estimate the direct causal effect between treatment X and outcome Y, then we must adjust for Z. This is achieved by restricting the analysis to situations where the value of Z is held constant. Conversely, in Figure 3b, if we want to estimate the *total* effect of X on Y, then we must not adjust for Z as this will block the causal path from X to Y through Z. Details on the criteria used to determine which variables to adjust for (d-separation and Back-door criteria) are described by Pearl [48] or Hernán and Robins [31].

3.4 Estimate the Causal Effect

The adjustment set computed from the causal graph provides us with the set of variables we need to control for in order to establish the causal effect of X on Y. This makes it possible to adjust our estimates without needing to carefully control the data-collection process. For example, if our adjustment set is $\{Z\}$, this can be achieved by using a regression equation of the form $Y \sim \beta_0 + \beta_1 X + \beta_2 Z$ [48]. For larger adjustment sets, the regression function is expanded with an additional coefficient for each variable. In cases where Y is a continuous value, approaches such as Ordinary Least Squares linear regression can be used. In our case, when the outcome tends to be binary (e.g. a failure event occurring), one can use logistic regression instead. Where variables have complex or unknown relationships, black-box Machine Learning estimation methods can also be applied.

In a conventional CI setting, such regression functions can form the basis for estimating the effect of X on Y whilst explicitly accommodating confounding variables $\mathbf{Z}$. This is achieved by using the regression function to simulate a controlled experiment. X can be varied, whilst holding

$\mathbf{Z}$ constant (even if this was not the case in the data from which the function was inferred). A typical measure is to compute the Average Treatment Effect (ATE) [31]:

$$\widehat{ATE} = \frac{1}{n}\sum_{i=1}^{n} \widehat{\mathbb{E}}[Y|X = x_t, \mathbf{Z_i}] - \widehat{\mathbb{E}}[Y|X = x_c, \mathbf{Z_i}]$$

Here, for each of the n instances for which we have data, we calculate the difference between the predicted value of Y whilst fixing the value of X to the *treatment value* x_t (and substituting the values of the variables Z in the adjustment set for their original values in the sampled data), and for when X is fixed to the *control value* x_c. The ATE is then computed as the sum of these differences, averaged out over the number of data-points n. For reference, we have included a fully worked through example of how the ATE is computed in Appendix A (we include this as an appendix since we ultimately use the IPCW technique instead of the ATE, as described below).

Point estimates such as the ATE are typically accompanied by confidence intervals, which specify a range of values in which the point estimate could fall for a given statistical significance level [46]. For example, the 90% confidence intervals define the region in which the ATE would fall 90% of the time if we repeatedly calculated the estimate using data sampled from the same distribution as the data actually used. For ATE, the causal relationship is said to be significant if and only if confidence intervals do not contain zero.

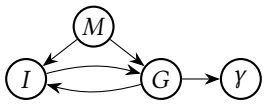
3.5 Handling Time Dependence and Cycles

One limitation of traditional CI as described above is that the causal graph must be *acyclic*. This prevents us from investigating causal effects involving feedback loops between variables over time. However, this sort of feedback loop is very common when working with CPSs, and occurs in both OpenAPS and SWaT, as shown in Figures 1 and 2.

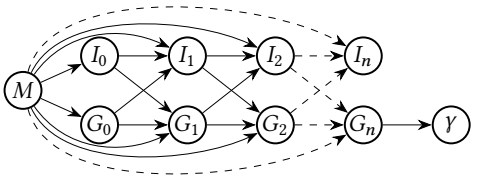
To introduce how CI handles such feedback loops, let us consider a simplified version of the feedback cycle in the OpenAPS system between the amount of glucose G in a patient's blood and the dose of insulin I delivered by the insulin pump, and the effect that this has on the test goal Y of hypoglycaemia. If the pump has just delivered a dose of insulin, G will be fairly stable and the pump may not need to deliver any more insulin for some time. As time passes, the pump may need to deliver extra insulin to manage the patient's blood glucose level. The amount required will be governed by their physiological characteristics, captured by a set of medical constants M [54], which includes various indicators such as the patient's sensitivity to insulin, and the amount of time it takes various parts of their digestive system to process carbohydrates. If the pump delivers too much insulin, the patient may enter a state of hypoglycaemia.

The causal relationships between M, G, I and Y are shown in Figure 4a. The standard approach to handling temporal feedback between variables is to "unroll" the graph for a fixed number of time steps [31], as shown in Figure 4b. As in [54], we here assume that the medical constants M remain constant during the period of observation, although the process would be the same if these values changed.

The passage of time is a key consideration here. It means that we do not simply examine the effect of a single treatment on a subsequent outcome, but rather we consider the causal effects of sequences of interventions (often referred to in CI as *treatment strategies*) on a subsequent binary outcome. This aligns well with our definitions of CPS tests and test goals from Definitions 6 and 7. For example, two simple treatment strategies (i.e. tests) for OpenAPS would be "give insulin at every time step" and "do not give insulin at any time step". We can then compare these treatment strategies to investigate which is more effective at achieving the test goal of hypoglycaemia.



(a) Cyclic causal graph of the feedback between G and I.



(b) Unrolled causal graph of Figure 4a, where subscripts represent time steps.

Fig. 4. Causal graphs of the relationships between the dose of insulin I delivered by a pump, the amount of glucose G in the patient's blood, their medical characteristics M, and the test goal Y of hypoglycaemia.

The main challenge when comparing treatment strategies is the causal identification step, discussed in Section 3.3. On the causal pathway from I_0 to Y in Figure 4b, G_1 is a *mediator*, since causality "flows" $I_0 \to G_1 \to I_2 \to \cdots \to Y$, so we should not adjust for it when estimating the final outcome, otherwise we cut off the indirect causal effect of I_0 on Y. However, G_1 is also a *confounder*, because it is a common cause of both I_2 and Y, so we simultaneously must adjust for it to remove its confounding bias. This situation is referred to as *time-varying confounding* [31], and cannot be handled by traditional identification methods. Confounders such as M, whose values remain constant during the study, are referred to as *baseline* confounders.

To tackle the problem of causal estimation in the presence of time-varying confounding, a family of specialised techniques called G-methods [31] has been introduced. Inverse probability of censoring weighting (IPCW) [51] is one such method, which we will apply in Section 4 during the first phase of CausalCut. IPCW estimates the expected risk of some event (for example an instance of hypoglycaemia) occurring within a certain time if the entire dataset had followed one particular treatment strategy (for example having the pump deliver a dose of insulin at every time step).

Algorithm 2 outlines the IPCW estimation process. The first step (Lines 2 and 3) is to assign individuals (in our case system executions) to control and treatment groups. To continue our OpenAPS example, the control group consists of every run where no insulin was given at all, and the treatment group consists of every run where insulin was given at every time step. In a randomised control trial, participants would be randomly assigned to receive a particular treatment strategy before any data was collected. However, we cannot do this if we are working with pre-existing log data. Instead, we assign groups *a posteriori* based on which strategy a run starts out on. In Algorithm 2, the control group contains all runs that started the control strategy X; the treatment group contains all runs that started the treatment strategy X'.

In an ideal scenario, every individual would follow their assigned strategy exactly until either the outcome of interest (e.g. hypoglycaemia) has occurred or the study has ended. However, this is rarely the case, especially when working with pre-existing observational data. Even in a randomised control trial, some patients may need to switch treatment strategies for medical reasons (for example if their blood glucose level becomes dangerous), or may drop out of a study altogether (for example if they die, or their glucose monitor or insulin pump breaks). Additionally, we may simply not have sufficient control over the system to implement all of our desired interventions, for example if a particularly specific environmental intervention is required.

To tackle this, IPCW *censors* individuals from the data at the point where they deviate from their assigned strategy. For example, if an individual is placed in the "never give insulin" group,

Algorithm 2 IPCW estimation.

1: **function** IPCW(data, X, X', timesteps, baseline confounders, time-varying confounders)
2: controlGroup $\leftarrow$ $\{\text{PREFIXFOLLOWING}(r, X) \mid r \in \text{data} \wedge \text{STARTSTRATEGY}(r, X)\}$
3: treatmentGroup $\leftarrow$ $\{\text{PREFIXFOLLOWING}(r, X') \mid r \in \text{data} \wedge \text{STARTSTRATEGY}(r, X')\}$
4: **for** individual $\in$ controlGroup **do**
5: CALCULATEWEIGHT(individual, X, timesteps, baseline confounders, time-varying confounders)
6: **for** individual $\in$ treatmentGroup **do**
7: CALCULATEWEIGHT(individual, X', timesteps, baseline confounders, time-varying confounders)
8: **return** COXREGRESSIONHAZARDRATIO(controlGroup, treatmentGroup)
9: **function** CALCULATEWEIGHT(individual, strategy, timesteps, baseline confounders, time-varying confounders)
10: **for** $t \in [0..\text{timesteps}]$ **do**
11: $\text{individual.weight}_t = \frac{1-P(\text{DEVIATESFROM}(\text{individual,strategy})\mid\text{baseline confounders})}{1-P(\text{DEVIATESFROM}(\text{individual,strategy})\mid\text{baseline and time-varying confounders})}$

but actually receives a dose of insulin at time step 10, all remaining time steps of their data are removed from the study, but their first 9 time steps are still included in the control group. The end of the observation period represents the time at which all remaining individuals who have not yet experienced the outcome are censored from the data.

This censoring places a critical requirement on the data: to be able to calculate a causal effect estimate, we must have at least one individual that follows each treatment strategy either until the outcome of interest (e.g. hypoglycaemia) has occurred or the end of the observation period. If every individual is censored before this, then we cannot calculate an estimate. Since the probability of being censored tends to increase over time, this is a particular concern for long treatment strategies, and can be a major limiting factor when using pre-existing observational data, which we will discuss further when we come to our evaluation in Section 6.

While the control and treatment groups are mutually exclusive here, this is not always the case. For example, we may be interested in comparing two treatment strategies s_1 and s_2 that give insulin when the blood glucose level rises above some threshold, where the threshold for s_1 is lower than for s_2. Here, every individual that is given insulin for treatment strategy s_2 would also be given insulin following strategy s_1, so an individual may begin the study following *both* treatment strategies. In such cases, individuals are *cloned*, with one copy being placed in each of the two groups.

Having assigned groups and censored individuals, the next step is to calculate the weight for each individual at each time step (Lines 4-7). This involves training a statistical estimator for each strategy to estimate the probability of deviating from that strategy at each time step, based on the baseline confounders (e.g. M) and the combination of baseline and time-varying confounders (e.g. I). This weighting serves as the identification step here, allowing us to adjust for the time-varying confounding without introducing bias by controlling for those same variables as mediators.

Having calculated the weights for each group, the final step is to perform a weighted outcome analysis between the two groups. In CAUSALCUT, we do this using Cox regression [18] (Line 8). Where an Ordinary Least Squares regression model plots a cause X against an outcome Y, such that we can predict the change in Y arising from a particular intervention on X, the Cox model generates a survival curve that records the survival rate among individuals over time [31]. In our example, individuals "survive" as long as they have not entered a state of hypoglycaemia.

In this kind of analysis, *hazard ratios* (rather than ATEs) are commonly used to summarise treatment effects when comparing survival times. This represents the ratio between the hazards associated with each treatment strategy, which is the instantaneous risk of the event of interest happening to an individual, given that they have remained in the study to a given time. For example, individuals following the "always give insulin" strategy may be twice as likely to go hypoglycaemic within 60 time steps than those following the "never give insulin" strategy. Because this is a ratio, estimates are said to be significant if their confidence intervals do not contain one (rather than zero for ATE).

The weights from the previous step allow individuals who are censored to be represented by individuals with similar baseline and time-varying characteristics who remained in the study. Individuals that follow their treatment strategy for longer are assigned a higher weight. One potential issue here is that, where a lot of individuals are censored, the weights for those that remain in the study can become very large. This is unfortunately a fundamental limitation of the technique [31]. While this does not stop us from calculating an estimate, that estimate may not be accurate as it will depend on only a few individuals. Thus, as with any data-driven technique, the accuracy of an IPCW estimate depends on having sufficient data. That is, data where sufficiently many individuals follow their assigned treatment strategy to completion. In our evaluation in Section 6, we will examine what happens when this breaks down.

4 Test Minimisation with CausalCut

In Section 2, we defined the problem of test case minimisation in the CPS context: given a failure-inducing test t that achieves test goal γ, the aim is to find a minimal test with respect to t and γ. In this section, we present CausalCut, a technique that applies CI (as introduced in Section 3) to address this problem. By estimating the causal effect of interventions on failures from existing runtime data, CausalCut aims to reduce the number of new system executions required during minimisation.

4.1 Test Minimisation as a Causal Reasoning Problem

Testing CPSs is costly and often risky, making it infeasible to repeatedly re-execute different test permutations. The key insight behind CausalCut is to *reframe test minimisation as a causal reasoning problem*: if we can estimate the causal effect of an intervention on the failure, we may be able to remove it without re-running the (pruned) test.

For example, consider the testing of the APS (Section 2.1), which uses a glucose monitor, control algorithm, and insulin pump to regulate blood sugar. A fuzzer might generate a test sequence that forces the insulin pump to deliver multiple doses, eventually resulting in hypoglycaemia. The minimisation task is to determine which interventions (i.e. insulin doses) were truly necessary in causing the failure (i.e. hypoglycaemia). First, a lightweight causal graph (such as the one in Figure 8) is constructed using domain knowledge to capture relationships between variables such as meals, insulin doses, and glucose levels. Second, observational data from past APS executions is then used to estimate the causal effect of removing each intervention. If the effect is insignificant, the intervention can be safely pruned.

The CausalCut technique requires three things:

- a (possibly cyclical) *causal graph* of the system,
- an observational dataset of system executions (both normal and abnormal), and
- the ability to execute further tests (on the real system or a simulator).

The causal graph can be based directly on the process flow diagrams that are typically provided in the technical documentation for the CPS (e.g. Figure 2 for SWaT). The observational dataset

Algorithm 3 Process of causally minimising tests.

```
1: function phaseOne(test, dcg, data, confidence, baseline confounders, time-varying confounders)
2:     for interventionSet ∈ test do
3:         for intervention ∈ test do
4:             Try
5:                 test′ ← [i|i ∈ test ∧ i ≠ intervention]
6:                 hazardRatio ← IPCW(data, test, test′, timesteps, baseline confounders, time-varying confounders)
7:                 if (hazardRatio.ciLow < 1 < hazardRatio.ciHigh) then ▷ intervention estimated as not significant
8:                     intervention.score ← min(1 − hazardRatio.ciLow, hazardRatio.ciHigh − 1)
9:                     prunedInterventions.add(intervention)
10:            Catch Error ▷ Cound not estimate causal effect
11:                prunedInterventions.add(intervention))
12:            EndTry
13:            return prunedInterventions
14: function causalCut(test, dcg, data, confidence, baseline confounders, time-varying confounders)
15:    prunedInterventions ← phaseOne(test, dcg, data, confidence, baseline confounders, time-varying confounders)
16:    prunedInterventions.sortBy(score, timestep)
17:    minimisedTest ← [i|i ∈ test ∧ i ∉ prunedInterventions]
18:    while ¬revealsFault(minimisedTest) ∧ interventionsToAdd do ▷ Phase 2: Return interventions to the test
19:        minimisedTest ← prunedInterventions.pop()
20:    return minimisedTest
```

should ideally span both typical behaviour and a wide range of interventions, so that causal effects can be estimated with sufficient statistical confidence. As with any data-driven technique, the quality of the results depends on the quality and quantity of the available data. We discuss the ideal requirements later in this section, and evaluate in Section 6 how CausalCut performs when only non-ideal datasets are available. Finally, while the aim of CausalCut is to reduce the number of new executions, we still assume that the test engineer has the ability to run further tests on the CPS (or a simulation of it), since these are occasionally needed to validate the pruned test cases.

CausalCut proceeds in two phases:

(1) **Causal Effect Estimation.** Using IPCW (Algorithm 2), we estimate the causal effect of each intervention by comparing the original test against a variant with the intervention removed. Interventions with no significant effect are pruned.
(2) **Failure Preservation.** If pruning causes the test to lose its failure-inducing property (i.e. for the specific test goal), interventions are added back in order of their estimated significance until the same failure reappears.

Algorithm 3 summarises Phase 1 (lines 15-17) and Phase 2 (lines 18-19), which are elaborated on in the following subsections.

4.2 Phase 1: Causal Effect Estimation

The goal of phase 1 is to estimate the causal effect of each intervention in the failing test case on the failure, and remove those interventions that do not have a significant causal effect. As discussed in Section 3, estimating causal effects over time is much more complex than conventional CI since the same intervention can serve as both a confounder (which must be adjusted to remove bias) and a mediator (which must be left unadjusted to remove bias) at different points in time. As mentioned in Section 3, we tackle this problem using IPCW, but this estimates the effect of treatment strategies (i.e. entire test cases) rather than individual interventions.

To estimate the causal effect of each intervention i, we calculate the hazard ratio between the original test and the test with i removed (line 6). This produces the hazard ratio associated with i, and confidence intervals that can be used to determine the statistical significance of the estimate (line 7). If the hazard ratio is not statistically significant (i.e. contains 1), then i is removed from the test (line 9).

As with any data-driven technique, the performance of phase 1 is dependent on the quantity and quality of the available data. In particular, as mentioned in Section 3, we need to have data that satisfies the positivity assumption in order to estimate hazard ratios. This means that, to estimate the causal effect of intervention i, we need to have at least one run of t and at least one run of $t - \{i\}$. This is essentially the same data that is collected by Greedy as it performs minimisation. Of course, to estimate the causal effect *accurately*, we need several repeats of each configuration. Furthermore, an idealised dataset would contain several repeats of every subset of the interventions in t (including t itself) in order to account for interaction between interventions.

While Greedy collects its data by strategically executing the system, CausalCut works with pre-existing runtime data that does not follow any particular treatment strategy, so we are likely to have to censor many of the runs part-way through. However, this limits the amount of data we have to estimate the causal effects, especially for longer test cases where we are less likely to have observed all of the executions we require. One advantage we have here, though, is that the positivity check is trivial and can be performed a priori, enabling us to tell in advance how effective CausalCut is likely to be. If positivity is only satisfied for a small proportion of interventions, CausalCut is unlikely to be effective. Our evaluation in Section 6 investigates this in terms of both the amount of available data and the proportion of interventions for which we are able to calculate a causal effect estimate.

In an attempt to minimise censoring and maximise the amount of usable data, we here focus only on the interventions at the time points of the interventions in the test we are trying to minimise, and ignore all the others. For example, if we are trying to minimise a test in which the insulin pump gave a dose of insulin at time points 15, and 35, we would not censor a test in which a user also gave a dose at time 20. Even with this more relaxed censoring criterion, there still may be instances where we have no runs at all that follow one of the treatment strategies of interest. In such instances, we cannot calculate an estimate at all (line 10). Nevertheless, we still prune the intervention from the test (line 11). If it is significant, it will be added back in in phase 2. We will discuss the implications of this in Section 8.

It is also worth noting that the accuracy of the causal effect estimate of course depends on the accuracy of the causal graph. As with any model-based testing technique, a poorly specified model will lead to poor results. However, a major advantage of causal inference over traditional statistical modelling techniques is that the causal graph allows us to identify and remove sources of bias in the data, meaning that imbalanced and skewed data is automatically accounted for as long as the underlying causes are represented in the causal graph. It is this that allows us to work with pre-existing runtime data. Additionally it has been shown that causal inference is somewhat robust

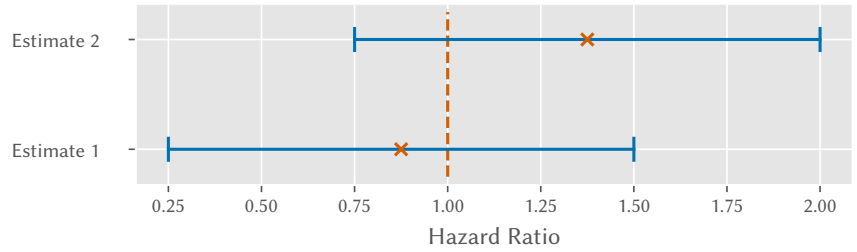


Fig. 5. Two different confidence causal effect estimates with their confidence intervals.

to misspecification [16]. While Clark et al. [16] showed this in a more traditional causal inference setting that does not take temporal feedback into account, the fundamental assumptions captured by a causal graph (namely that an edge $X \longrightarrow Y$ represents the possibility of X causing Y, and the absence of such an edge represents the fact that X definitely does not) remain the same here.

4.3 Phase 2: Re-adding Interventions

Given a sufficient amount of high-quality test data from which to make our estimations, phase 1 should produce a minimal, failure-inducing test. However, given the nature of CPSs, such data is unlikely to be available. Thus, phase 1 may be overaggressive in its pruning, resulting in a test that is not failure-inducing for the given test goal. The intention of phase 2 is to re-introduce pruned interventions to guarantee that the minimised test still induces the same failure as the original test (lines 18-19).

Our key insight is that we use the confidence intervals for the hazard ratio estimates from phase 1 to add interventions back starting with the least insignificant interventions first. When ordering the interventions to be returned to the test, our approach is to sort them by the minimum distance that one of their confidence intervals would have to move for the hazard ratio estimate to have been significant. To illustrate this, consider Figure 5 which shows the confidence intervals of two hazard ratio estimates. The confidence intervals of Estimate 1 are $[0.25, 1.5]$, and the confidence intervals of Estimate 2 are $[0.75, 2]$.

In Figure 5, Estimate 2 is somehow closer to being significant since its lower bound is only 0.25 away from 1 (which would then make it significant). By contrast, the upper bound of Estimate 1 is 0.5 away from 1, and its lower bound is 0.75 away from 1, which is even further. Thus, we would add Estimate 2 before Estimate 1 since their minimum distances are 0.25 and 0.5 respectively. This is the "score" that is being calculated in line 8 of Algorithm 3.

Where we do not have sufficient test data to calculate a causal effect estimate at all, we obviously do not have confidence intervals to help us order the interventions. In such cases, we simply default back to the naive greedy approach, adding them back in from earliest to latest. We do this only once all interventions for which we could estimate a hazard ratio have been added back in. In the worst case, where we were unable to estimate the hazard ratio for a necessary intervention right at the end of the test, we can end up needing to add back all of the interventions, resulting in the original test being returned.

Note that CausalCut is explicitly conditioned on the original test goal γ. During minimisation, candidate tests are only accepted if they remain failure-inducing for the same test goal, in accordance with Definition 9. If removing interventions causes the execution to satisfy a different test goal γ' (where $\gamma' \nRightarrow \gamma$), then the resulting test is rejected as a reduction. Handling such newly discovered failures with distinct goals lies outside the scope of the test-minimisation problem studied here, but

Algorithm 4 Combining CausalCut with Greedy postprocessing

```
function causalCutPlusGreedy(test, dcg, data, confidence, baseline confounders, time-varying confounders)
    prunedInterventions ← phaseOne(test, dcg, data, confidence, baseline confounders, time-varying confounders)
    prunedInterventions.sortBy(score, timestep)
    minimisedTest ← [i|i ∈ test ∧ i ∉ prunedInterventions]
    greedyPrunable ← minimisedTest
    while ¬revealsFault(minimisedTest) ∧ interventionsToAdd do          ▷ Phase 2: Return interventions to the test
        minimisedTest ← prunedInterventions.pop()
    for interventionSet ∈ greedyPrunable do                             ▷ Greedy postprocessing
        for i ∈ interventionSet do
            interventionSet ← interventionSet \ {i} ▷ Remove i from the test and see if it is still failure-inducing
            if testCase is no longer failure-inducing then              ▷ If not, put back i
                interventionSet ← interventionSet ∪ {i}
    return minimisedTest
```

represents a potential opportunity for future integration with test-generation or test-diversification techniques.

4.4 Hybrid Approach

CausalCut can easily be hybridised with the Greedy heuristic from Algorithm 1 by running it as a postprocessing step on the output of CausalCut to further reduce the test. However, there is a trade off here since, if the output of CausalCut is already minimal, the application of Greedy simply increases the cost of that minimisation without improving the end result. To manage this trade off, we only run Greedy on the interventions that were estimated in phase 1 to be causally significant. We do not consider interventions added during phase 2 of CausalCut since an execution has already been performed for each of these. We call this hybrid approach CausalCut+Greedy, and evaluate it alongside the "pure" CausalCut technique introduced above in Section 6.

5 Theoretical Analysis

In this section, we present a theoretical analysis of Greedy, DDmin, CausalCut, and CausalCut+Greedy in terms of their best and worst case pruning ability and executions required. From this we propose some guidelines as to the circumstances under which each technique would be the most cost efficient choice.

5.1 Best and Worst Cases

Recall from Definition 11 that minimising a failure-inducing test case t involves finding a minimal, failure-inducing test m that contains the $|m|$ necessary interventions to achieve test goal γ. As discussed in Section 2.5, both Greedy and DDmin are guaranteed to produce a 1-minimal result. This means that if a failure does not depend on interactions between interventions, they will always produce a test of length $|m|$. If there are multiple minimisations that contain $|m|$ interventions, it may not always be the case that $\text{Greedy}(t) = \text{DDmin}(t)$, but both techniques are deterministic, so will produce the same result each time they are run.

Table 1. Best and worst outcomes for the techniques for a test t containing $|m|$ necessary interventions with $m \sqsubseteq i \sqsubseteq t$ such that no intervention can be individually removed from i while still achieving γ.

	Pruning		Executions	
Technique	Best	Worst	Best	Worst
Greedy (no interaction)	$\|m\|$	$\|m\|$	$\|t\|$	$\|t\|$
Greedy (with interaction)	$\|m\|$	$\|t\|$	$\|t\|$	$\|t\|$
DDmin (no interaction)	$\|m\|$	$\|m\|$	$\leq 2\log_2(\|t\|)$	$\|t\|^2 + 3\|t\|$
DDmin (with interaction)	$\|m\|$	$\|t\|$	$\leq 2\log_2(\|t\|)$	$\|t\|^2 + 3\|t\|$
CausalCut	$\|m\|$	$\|t\|$	1	$\|t\| + 1$
CausalCut+Greedy	$\|m\|$	$\|t\|$	$\|m\| + 1$	$\|t\| + 1$

In terms of cost, Greedy will always perform $|t|$ executions, since it simply tries to remove each intervention in turn. The cost of DDmin depends on the distribution of the necessary and spurious interventions in t, and has been examined by Zeller and Hildebrandt [67]. In the best case, DDmin will require no more than $2\log_2(|t|)$ executions and has the same complexity as binary search. In the worst case, DDmin will require $|t|^2 + 3|t|$, which is considerably more than Greedy.

While interaction between interventions in a test case does not affect the number of executions required by either Greedy or DDmin, it can affect their performance since, as discussed in Section 2.5, both techniques are only guaranteed to produce a 1-minimal result. In fact, certain interactions can prevent Greedy from removing any interventions at all, even when most of the interventions are spurious, resulting in the original test t. While DDmin mitigates this by testing more combinations, it can also be affected.

Since CausalCut and CausalCut+Greedy use causal estimation, their performance will not be affected by interaction as long as the estimation model takes account of this [48]. Therefore, with or without interaction, the best case reduction is a minimal test case m containing $|m|$ interventions. Since their output is dependent on the availability of a sufficient amount of past execution data, they will produce suboptimal results if this is not available, even without interaction. In terms of executions, the best possible case for CausalCut is when phase 1 correctly estimates the causal effect of every intervention in t, resulting in a minimal test m. In this case, only a single execution would be required to confirm that m is failure inducing.

In the worst case, no interventions in t are estimated as significant during phase 1, meaning that every necessary intervention must be reintroduced during phase 2. CausalCut will perform one execution at the end of phase 1 as before, and then up to $|t|$ executions to reintroduce each necessary intervention. The result will contain all necessary interventions, plus any other interventions that were estimated as more significant than the last necessary intervention to be reintroduced, since phase 2 adds interventions in the order of their estimated significance. Where the original test t is already minimal, CausalCut will perform $|t| + 1$ executions[3] and produce t.

For CausalCut+Greedy, the best possible case is the same as for CausalCut. Again, one execution is required at the end of phase 1 to confirm that the minimisation is failure inducing, but CausalCut+Greedy then uses Greedy as an additional postprocessing step that performs one execution for each intervention estimated as significant. Thus, the best possible case here is $1 + |m|$ executions.

[3]If no intervention is estimated to be significant, the execution at the end of phase 1 may be skipped if the system can be assumed to require at least one intervention to fail, giving a worst case of $|t|$ executions.

The worst possible case is again the same as for CausalCut. When no interventions in m are estimated as significant during phase 1 of CausalCut, they must all be added during phase 2 by executing the system. Where no interventions are estimated as significant, there are no interventions eligible for pruning via the Greedy postprocessing step, so no additional executions are performed beyond the $|t| + 1$. If, during phase 1 of CausalCut, a set of spurious interventions s is estimated as significant, CausalCut performs $|s|$ fewer executions in phase 2, since the interventions are already in the test. However, the interventions in s are eligible for pruning via the Greedy postprocessing step (since they were estimated as significant during phase 1), so are removed from the test at a cost of $|s|$ executions. Thus, the most expensive case for CausalCut is still $|t| + 1$ executions.

5.2 Cost Efficiency

We now examine the circumstances under which CausalCut and CausalCut+Greedy are more cost efficient than Greedy, and provide some guidance on how to select the appropriate minimisation technique. This is highly dependent on the usage context, and especially on the cost involved in running a test. If the cost of each execution is negligible and there is no interaction between interventions, then Greedy is naturally the best choice as it is guaranteed to produce an optimal result. Conversely, if the objective is to significantly reduce the size of the test but a few remaining spurious interventions are acceptable, then CausalCut may be the better choice if there is sufficient execution data available.

To explore this trade off, we define a metric that combines both cost and pruning ability in terms of the number of executions required to perform minimisation and the number of spurious interventions that remain afterwards. For this, we frame the question of whether to remove an intervention as a binary classification problem — each intervention in the test is either necessary or not. In this context we define our cost-efficiency metric in terms of the confusion matrix of true and false positive and negative classifications. Here, we use the positive predictive value (PPV) [24]; the probability that an intervention in the reduced test case really is a necessary cause. For a reduced test case t', this is calculated as PPV $= {}^{|m|}/_{|t'|}$.

We then define our cost efficiency metric as PPV per execution, where x is the number of executions taken to minimise the test.

$$\text{Cost efficiency} = {}^{|m|}/_{|t'|x} \tag{1}$$

This spans between zero and one, with low values representing poor cost efficiency and 1 representing the best possible cost efficiency where a true minimal test case m is found using only a single execution. This captures the relationship between reduction and execution cost, with each having an even weighting, and gives us a solid basis upon which to compare the trade-off between the two. It is worth noting that alternative metrics that give greater weight to either dimension could easily be considered in cases where, for example, running executions is known to be particularly expensive.

Figure 6 shows the best and worst cases for each technique for tests up to length 25. For Greedy, PPV per execution is simply ${}^{1}/_{|t|}$, since the reduced test case will always be minimal at a cost of $|t|$ executions. However, as discussed above, where there is a test $m \sqsubseteq i \sqsubseteq t$ such that i is 1-minimal (Definition 12), Greedy will return i rather than m, leading to a lower PPV. The worst possible scenario is $|m| = 1, i = t$, as is the case for the example from Section 2.5, where the system fails for tests containing an odd number of interventions. In this case, the result is the same as for the worst case of CausalCut and CausalCut+Greedy but with one fewer execution, since CausalCut performs an extra simulation after phase 1 to check whether the reduced test is still failure-inducing.

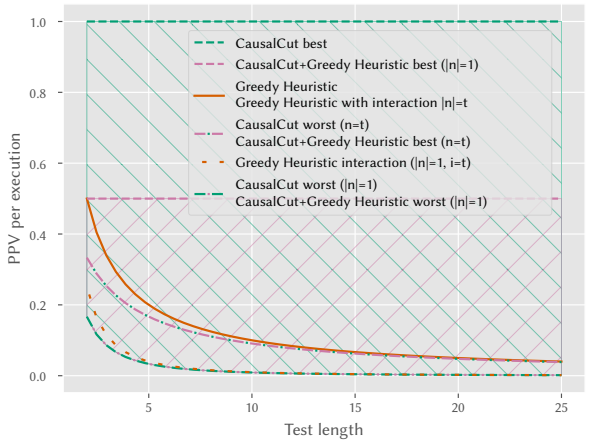


Fig. 6. The best and worst case for each technique by original test length.

CausalCut spans almost the entire cost efficiency range. In the best case, CausalCut achieves optimal cost efficiency by producing the minimal test m during phase 1. A single execution is then used to confirm that m is failure-inducing.

In the worst case, CausalCut produces the original test case t and uses $|t| + 1$ executions. The best possible situation here is $|m| = t$, meaning that the original test case is already minimal. This gives a PPV of 1 and a cost efficiency of $1/|t|+1$. The worst possible situation is that there is a single necessary intervention that is the last element of the test. This gives a PPV of $1/|t|$ and a cost efficiency of $1/|t|(|t|+1)$.

For CausalCut+Greedy, the best cost efficiency is the same as for CausalCut, namely that a minimal test m is identified during phase 1 with a single execution. In such cases, the Greedy postprocessing step then progressively lowers cost efficiency by performing $|m|$ executions, none of which can reduce the test, giving a cost efficiency of $1/|m|+1$. Thus, CausalCut+Greedy achieves its best cost efficiency of 0.5 for $|m| = 1$, regardless of original test length[4]. Where $m = t$, this gives a cost efficiency of $1/|t|+1$, the same as the "best worst" cost efficiency of CausalCut. The worst possible cost efficiency of CausalCut+Greedy is the same as for CausalCut, namely a single necessary intervention that is the last element of a test where no interventions are estimated significant, giving a cost efficiency of $1/|t|(|t|+1)$.

While the cost efficiency of CausalCut depends on the ability to correctly estimate the causal effects of the interventions in the test, Figure 6 shows that the potential for CausalCut to be more cost effective than Greedy increases with test length. We can use the cost efficiency formulae of the two techniques to explain this. For a reduced test t', CausalCut is more cost efficient than Greedy when $1/|t| < |m|/|t'|x$, which is equivalent to $|t'|x < |m||t|$. Since $|m|$, $|t'|$, and x all depend on several factors that are highly system-specific, we cannot give a precise probability of either tool being more cost effective for a particular test without further information. However, if a test contains fewer interventions that are necessary causes of a failure, these will each make a greater individual contribution to a given fault, making them easier to identify. Additionally, as discussed in Section 3, estimates for interventions that occur earlier in a test are more likely to be accurate as there tends to be more data available to calculate the estimates.

[4]Similar to above, CausalCut+Greedy could be optimised to skip Greedy postprocessing for single-element minimised tests from CausalCut, giving CausalCut+Greedy the capability of achieving optimal cost efficiency where $|m| = 1$. Cost efficiency for $|m| > 1$ would remain unchanged.

6 Evaluation

In order to empirically examine the trade-off between cost and quality of test minimisation, we pose the following research questions.

RQ1 (Cost efficiency): How does CausalCut perform in terms of cost efficiency?
RQ2 (Minimisation): To what extent can CausalCut minimise tests?
RQ3 (Executions): How many executions are required by CausalCut to minimise tests?

6.1 Methodology

6.1.1 Subject Systems. We have evaluated our technique using the OpenAPS and SWaT systems introduced in Section 2.1. OpenAPS has one sensor (a continuous glucose monitor) and one actuator (an insulin pump), and uses a control algorithm (oref0) to mimic the insulin-delivery behaviour of a healthy pancreas. SWaT is a fully operational, scaled-down six-stage water purification plant consisting of 36 sensors and 30 actuators, each with well-defined safe operating ranges. These systems are very different in nature, scale, and domain, but are both real cyber-physical systems that exhibit the testing challenges outlined in Section 2.1. Namely, tests are high-risk, expensive and time-consuming to run, and the systems can produce nondeterministic results.

The systems also differ in the resources that are available to facilitate research. OpenAPS has a configurable high-fidelity simulator [54] that can be used to run the system in arbitrary configurations without risking real users, thereby allowing us to collect a large amount of highly controlled test data. This makes it perfect for systematically investigating the relationships between the different features in this study. Conversely, SWaT provides a large set of real runtime data, but it is prohibitively expensive to collect additional data (note also that the existing simulator [10] neither exposes all variables of interest nor allows arbitrary intervention strategies required for causal analysis). This allows us to explore how CausalCut performs when faced with a fixed, limited dataset collected from real-world operation.

6.1.2 Baseline Approaches. We compared CausalCut and CausalCut+Greedy against Poskitt et al.'s Greedy heuristic [50] (Algorithm 1) and to Zeller and Hildebrandt's DDmin algorithm [67] using the implementation from [66]. As discussed in Section 2.5, both DDmin and Greedy are guaranteed to produce a 1-minimal result, meaning they represent more of a "gold standard" than a baseline in terms of pruning ability. However, this comes at a high cost: Greedy runs exactly one execution for each intervention in the original test, and DDmin can require even more than this in the worst case, which can become intractable for longer tests. Furthermore, where interventions do interact to induce a failure, their results may not be minimal.

6.1.3 Experimental Procedure. As detailed in Section 4, given a failure-inducing test case, CausalCut needs three things to perform reduction: a set of system executions, a causal graph, and the ability to perform further executions. The details of how we obtained these for each system are given in Section 6.2. We then ran CausalCut, CausalCut+Greedy, Greedy, and DDmin on each test case and recorded the information needed to answer each RQ. All experimental code and data are available as part of our replication package. Code is available at https://doi.org/10.15131/shef.data.32233797. Data is available at https://doi.org/10.15131/shef.data.30408154.

RQ1: Cost Efficiency. This RQ uses the cost efficiency metric from Equation (1) as an overall metric for how well CausalCut performs in terms of the number of interventions in the reduced tests, the number of spurious interventions remaining, and the "cost" of obtaining the reduction in terms of the number of executions. To answer this RQ, we consider how five features affect the performance of CausalCut and CausalCut+Greedy in comparison to Greedy and DDmin. These are the original test length, the proportion of necessary interventions in each test, the amount of

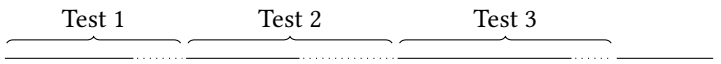


Fig. 7. Dividing continuous executions into segments to form separate tests. Solid line segments represent periods of safe blood glucose. Dashed line segments represent periods of hypo- or hyperglycaemia.

previous runtime data available, the proportion of interventions for which we could calculate a causal effect estimate, and the confidence intervals used to estimate causal significance. We then calculated the Spearman correlation between each feature and the cost efficiency to investigate which technique scales best, since our data was not normally distributed.

RQ2: Minimisation. To answer this RQ, we examine how the same five features considered for RQ1 affect the minimisation capability of CausalCut and CausalCut+Greedy in comparison to Greedy and DDmin in terms of the reduced test length. For this RQ, we additionally break CausalCut down into its two phases. Although the result of phase 1 is not guaranteed to be failure inducing, so is not a true reduction, this allows us to examine how often interventions needed to be reinstated during phase 2 of CausalCut. Since our original tests have different lengths with different numbers of necessary interventions, we normalise by the original test length so that this is a proportion rather than a raw number. As for RQ1, we calculated the Spearman correlation between each feature and the outcome to investigate which technique scales best.

RQ3: Executions. To answer this RQ, we examine how the same five features considered for RQ1 affect the number of executions required by CausalCut and CausalCut+Greedy in comparison to Greedy and DDmin. Again, we normalise by the original test length so that we are comparing the number of executions *per intervention in the original test.* This enables us to compare different tests more objectively. As for the previous RQs, we calculated the Spearman correlation between each feature and the outcome to investigate which technique scales best.

6.2 Experimental Setup

Having laid out our experimental procedure, we now present the details of how we obtained the failure-inducing test cases, system execution data, and causal graphs for each system.

6.2.1 Test Generation.

OpenAPS. As discussed in Section 2.1, OpenAPS has two obvious test goals: hypo- and hyperglycaemia. That is, the user's blood glucose level is below 4mmol/L or above 11mmol/L, respectively. A test is failure-inducing if either is achieved during the execution. Rather than generating tests ourselves, we instead obtained system executions from the OpenAPS Data Commons [42], which records data from real users of the OpenAPS system [21, 53] during normal operation over several months. Each of these long executions contains multiple instances of hypo- and hyperglycaemia (i.e. failures), so can be split into segments that represent stand-alone failure-inducing tests. To do this, we divided each execution into segments where the patient starts in a safe blood glucose range and ends in an unsafe one, as illustrated in Figure 7.

Having segmented the executions, we then extracted the test interventions using the tool published by Somers et al. [54]. Our interventions here are actions that the user performed themselves — either eating a meal or injecting themselves with an additional "top-up" dose of insulin called a *bolus*, commonly administered by users either before or after eating to compensate for the sudden influx of carbohydrates. In the OpenAPS data, the sizes of both meals and boluses are continuous. However, as discussed in Section 2, interventions need to be discrete so we discretised meals into three different sizes: snacks, light meals, and heavy meals. We treated boluses as a binary: either

a patient administered a bolus at a given time step or not. Having extracted and discretised the interventions, we used the simulator to execute each test to check that the failure was reproducible. Due to the simulator's long runtime, we did not consider tests covering more than 500 time steps as it would not have been feasible to collect sufficient test data for them to execute CausalCut.

As noted by Somers et al. [54], a significant majority of the OpenAPS Data Commons executions are not well formed and so could not be processed in this way due to missing values, sensor errors, and inconsistent formatting [23, 39]. Furthermore, users often forget to record meals and boluses, making it impossible for us to extract the test interventions from the executions. From the entire OpenAPS dataset, there was only one execution that was usable here[5]. However, this execution contains nearly 40,000 data points recorded at five minute intervals between June and October 2020, during which time the patient fluctuates between safe, low, and high blood glucose levels fairly regularly. In total, there were 481 safe $\longrightarrow$ unsafe segments, of which 70 represented failure-inducing tests for the simulator. On average, 86.19% of the interventions in the tests were bolus events, 8.18% were heavy meals, 4.35% were light meals, and 1.28% were snacks.

SWaT. Unlike OpenAPS, we were unable to execute the SWaT system in our evaluation as experiments on the real testbed are prohibitively expensive [34] and no suitable simulator is available. As a result, we could not readily rely on randomly generated tests (e.g. via fuzzing), since it would be difficult to determine in general whether they induce failures without system execution. Instead, we systematically constructed test cases to minimise by embedding known failure-inducing intervention sets within additional spurious interventions.

Specifically, we began with the tests from [50, Table II], which have been experimentally verified as optimal (Definition 10) for the physical SWaT system. These served as a gold standard: each defined a minimal set of interventions that induces a failure. To obtain *unminimised* tests, we injected additional 'spurious' interventions, chosen uniformly at random, randomly within each test. The resulting tests therefore contained the necessary interventions for inducing failures embedded within a potentially large number of irrelevant interventions, reflecting the kind of redundancy typically observed in tests generated by CPS fuzzing approaches such as Chen et al. [10]. This allowed us to evaluate minimisation on a second CPS domain (industrial water treatment) under realistic data constraints, albeit without system execution. Our goal was then to recover the minimal subsets responsible for each failure from the suite of unminimised tests.

Of the 33 minimal tests in [50], only 27 were usable here: five referenced variables that were unavailable in the SWaT dataset, and one had an outcome variable that remained within its safe range throughout the run. For each usable test, we generated four expanded variants, resulting in 108 unminimised test cases. In these test cases, 68.85% of interventions were on pumps, and 31.15% of interventions were on valves.

This construction allowed us to evaluate test minimisation in a setting where the ground-truth minimal intervention sets were known, enabling direct assessment of whether CausalCut correctly identifies the essential interventions. However, since we could not execute the system, it was not possible to validate whether each expanded test induces the original failure, nor could we rely on system reruns for Greedy or phase 2 of CausalCut. Instead, we employed a surrogate oracle that deems a test failure-inducing if it retains the interventions from the corresponding optimal test.

This introduced the implicit assumption that the injected interventions neither cause nor mask failures. Consequently, the setting favours Greedy and DDmin, since, as discussed in Section 2.5, they are then guaranteed to recover the optimal result. We discuss these threats to validity further in Section 8.5.

[5]Due to the data sharing agreement of this dataset, we unfortunately cannot share the ID of the patient that this execution came from.

Test Lengths. The number of interventions in each of the tests for both systems are shown in Table 2, together with their necessary interventions. For OpenAPS, we approximated this by performing the full combinatorial optimisation process on the output of the Greedy as it was not computationally feasible to perform the full combinatorial optimisation on the raw test cases. The longest test was for OpenAPS and consisted of 24 interventions, of which just eight were necessary. The longest test for SWaT had 12 interventions, three of which were necessary. This shows that most of the interventions in each fault-revealing test are spurious and can be removed without affecting the final outcome, as is the case with tests produced by current fuzzing techniques [50].

Table 2. Original and gold standard minimised test lengths for both studies.

		Original length (OpenAPS)												Original length (SWaT)									
		2	3	4	5	6	7	8	9	10	11	13	24	2	3	4	5	6	7	8	9	10	11	12
Necessary interventions	1	5	6	7	9	7	4	1	2	1	2			7	1	4	3	3	1	3			1	1
	2	1	1	2						1				9	11	6	4	2	2	4				1
	3		2		6	1							1		10	6	2	11	3	1	2	2	1	1
	4			2	1							1				2	3		2		1			
	5				2																			
	6											1												
	8												1											

6.2.2 Causal Graphs. The broad approach used to create causal graphs is described in Section 3.1. For our specific case studies we were able to take advantage of prior schematic diagrams or information embedded in source code, as described below.

OpenAPS. Figure 8 shows our causal graph for the OpenAPS system. The variables `stomach`, `blood insulin`, `blood glucose`, and `rate` are the variables recorded in the OpenAPS dataset (and the simulator from [54]), where `rate` is the *basal rate* of insulin delivered by the insulin pump. This is a continuous stream of insulin being delivered to the patient, with the amount being constantly reviewed by the control algorithm to mimic the behaviour of a healthy pancreas. This is different to the `bolus` intervention, which represents a larger one-off dose of insulin administered before or after eating to compensate for the sudden influx of carbohydrates [54]. M is a set of 11 *baseline confounders*, medical constants for each individual that define physical characteristics such as their sensitivity to insulin, as discussed in Section 3.5. The `snack`, `light meal`, and `heavy meal` nodes represent our three food intake interventions. The connections between nodes were formed by inspecting the source code of the simulator published by Somers et al. [54].

SWaT. For SWaT, we used Figure 2 directly, since the flow of water between the different nodes in the system can be thought of as the flow of causality. Our only modification is that, for each edge $X \rightarrow Y$ in the flow diagram, representing the unidirectional flow of water, we need to add a corresponding edge $X \leftarrow Y$ to represent the feedback cycle of causality, as illustrated in Figure 4a and "unfolded" as in Figure 4b. This is because the amount of water flowing through a particular sensor will depend on the state of the actuator immediately before (i.e. the *inflow*) and the actuator after (i.e. the *outflow*). Using Figure 2 without this modification would only have causality depending on the inflow, and thus bias the results.

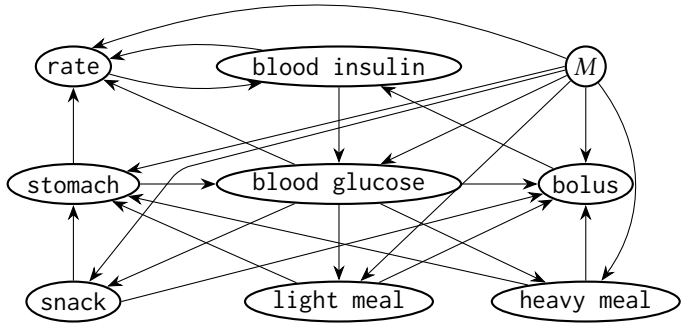


Fig. 8. Causal graph of the OpenAPS system and possible interventions.

6.3 Data Generation

OpenAPS. Recall from Section 4.2 that phase 1 of CausalCut uses a set of system executions to estimate the causal effects of the interventions in our failing tests. We had originally hoped to use the OpenAPS dataset for this but, as discussed above, the vast majority of the data is not sufficiently well-formed for this purpose. Instead, we collected a set of executions from the simulator [54]. The advantage of this is that we could collect arbitrarily many executions in a reasonable amount of time without endangering any human users, enabling us to systematically explore trends and relationships between variables. We discuss the threats to validity that arise from this in Section 8.5.

To generate our execution data, we took inspiration from the (guided) fuzzing setting of Poskitt et al. [50] used for testing cyber-physical systems. Fuzzers typically work by iteratively mutating tests in search of violations, thereby producing many similar, but not identical, failure-inducing tests. To simulate this process, we generated 5,000 mutations of each of our 70 failure-inducing tests by adding or removing one quarter of the interventions, independently at random. We ran our mutated tests with randomly chosen values for the constants that define a patient's physiological characteristics, as discussed in Section 3.5. We also included the original failure-inducing test executions in our dataset, which simulates the fuzzer "finding" these failures. This gave a total of 350,070 executions, which we ran for a maximum of 500 time steps, terminating earlier if a failure was observed.

To help mitigate for randomness, we ran our data collection procedure 5 times using different random seeds. Since each dataset is around 20GB and took around a week to collect using HPC at a cost of around $1,000 worth of CPU time[6] per dataset, we did not deem it feasible to collect more than 5 sets. We did not record detailed statistics of the runtime, memory, or CPU usage of these systems, but a typical run takes around two minutes on a standard desktop machine with 24GB of memory running Ubuntu 24.04.

SWaT. For SWaT, we used the "Attack V0" dataset from the SWaT data repository[7] [35], which contains data for 4 days recorded at 1 minute intervals. This data was collected from the physical SWaT system, not a simulator, so represents 4 physical days of runtime. As mentioned above, this time period contains data resulting from various attack scenarios, including some of those from

[6]We collected our data using the University of Sheffield's HPC system, which is free at point of use. This is an indicative figure provided by our internal management system.
[7]While the data sharing agreement does not allow us to share this data directly, it can be requested for academic purposes from [35].

[50] that were carried out one at a time with a suitable period between attacks for the system to return to normal levels[8].

As mentioned above, we did not have access to either the physical SWaT system or a suitable simulator, so could not collect any additional data ourselves. In some ways, this is advantageous as it allows us to evaluate the applicability of CausalCut using only real observational data. However, the FIT301 sensor remained in its safe range throughout the entire execution, so we could not investigate any test with FIT301 as an outcome as they are not failure-inducing with respect to this dataset.

Since the SWaT data is one single long execution, we divided it up into segments of 210 time steps to match the longest of our expanded tests. We started segments at intervals of 15 time steps since each intervention was carried out for 15 time steps, e.g. the first segment spans times 0 to 209, the second spans times 15 to 224, etc. This resulted in 449,920 segments. Due to the prohibitively high cost of running SWaT [34], we could not collect any additional executions. While Poskitt et al. [50] used a simulator for their investigation, this is not sufficiently configurable or high fidelity to be of use here, so we limited ourselves purely to the pre-existing executions.

6.3.1 Data Size and Suitability. Our datasets for both systems may seem extremely large, however, the fuzzer used by Poskitt et al. generated over 1.2M failure-inducing tests (Poskitt et al. [50, Table 1]). This represents an absolute best case as there will likely have been many more executions that did *not* result in a failure. Our own dataset is small by comparison (around a third of this for OpenAPS and under a half for SWaT) and covers many different configurations, so is realistic in terms of the numbers of executions collected by fuzzing tools.

It is also important to consider the *quality* of the data from the respective systems in terms of how well the assumptions necessary to calculate causal effect estimates (laid out in Section 3) are satisfied. Figure 9 shows the proportion of interventions for which it was possible to calculate a causal effect estimate (correct or not) for each data sample we considered in our evaluation. This shows a stark difference between the two systems. While we can calculate a causal effect estimate for almost every OpenAPS intervention using just 50 executions, we could not estimate over half of the SWaT interventions even when using the entire dataset. This will be reflected in the results for the two systems, since CausalCut and CausalCut+Greedy rely on being able to calculate causal effect estimates. Where this is not possible, a much greater reliance is placed on phase 2 of CausalCut, which can be both expensive and lead to poor minimisations. Thus, we expect CausalCut and CausalCut+Greedy to perform much better for the OpenAPS than for SWaT.

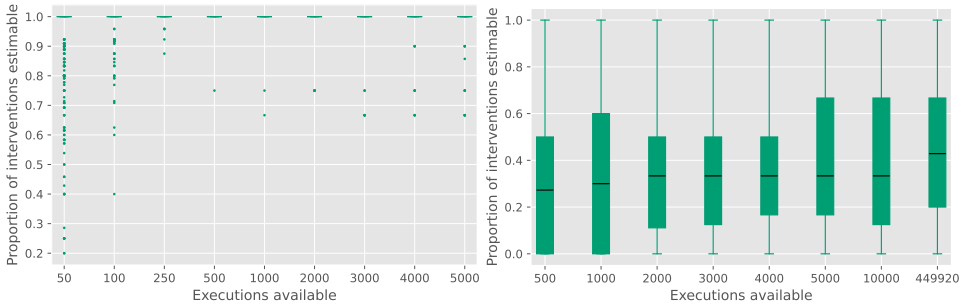
7 Results and Analysis

This section presents the results from our experiments, with a more detailed analysis being given in Section 8. For all three RQs, we found that the confidence intervals used in phase 1 of CausalCut had either a negligible or insignificant relationship on the final outcome. We therefore omit the plots from this section, along with plots for features in each RQ with negligible or insignificant effects. These form part of our replication package and can be seen in Section B.

7.1 RQ1: Cost Efficiency

Figure 10a shows how cost efficiency (as defined in Equation (1)) is affected by original test length, and the proportions of necessary and estimable interventions for the OpenAPS system. Overall, CausalCut achieved a mean cost efficiency 17% higher than Greedy, and a median cost efficiency

[8]Unfortunately it is not documented exactly which attacks were carried out, and the attacks in the SWaT data were carried out over a much longer timescale than [50], so it is impossible to directly map regions of the data to attacks from [50].



(a) Proportion of estimable interventions for OpenAPS.

(b) Proportion of estimable interventions for SWaT.

Fig. 9. Proportions of estimable data for the different data samples we considered in our evaluation.

of 8% higher. The mean and median cost efficiencies of CausalCut were 16% and 8% higher than DDmin respectively.

The leftmost plot of Figure 10a shows that all the techniques become less cost efficient as test length increases, with the significance of this being confirmed by Table 3. However, this negative correlation is weaker for CausalCut (r=-0.393) and CausalCut+Greedy (r=-0.474) than for Greedy (r=-0.960) and DDmin (r=-0.581). This indicates that our techniques scale better and can still be highly cost effective even for long tests (although they also conversely have the potential to be less cost effective for short tests). We will examine the trade-off between execution time and pruning ability in our answers to RQs 2 and 3.

Furthermore, the cost efficiency of Greedy is sometimes less than $1/|t|$ for tests of length 4 and 10. This indicates that certain test cases did rely on interactions between interventions to induce failures. Figure 10a also shows that DDmin can be even less cost efficient than Greedy, especially for shorter tests, although it is more cost efficient for tests of length ten, indicating that it is better able to cope with interaction.

The middle plot of Figure 10a tells a similar story in terms of proportion of the interventions in a test that are necessary. This plot shows a clear positive correlation for Greedy (r=0.636). The points for CausalCut and CausalCut+Greedy are more diffuse, and their respective correlations are much weaker because of this (r=0.199 and r=0.057). In particular, the figure shows that CausalCut is able to achieve high cost efficiencies even with a very low proportion of necessary interventions, although this is partially due to the fact that lower proportions of necessary interventions can only really occur for longer traces, for which we have just shown CausalCut to be more cost efficient than Greedy.

By contrast, the correlation for DDmin is actually negative (r=-0.135) although weak, indicating that it becomes slightly more cost effective as the proportion of necessary events goes down. This is not necessarily surprising since many of the necessary interventions in the OpenAPS tests occur towards the end of the tests, which fits well with the binary-search-like structure of the DDmin algorithm. As the proportion of necessary events reduces, DDmin becomes more likely to be able to find a "shortcut" where Greedy must still iterate through every intervention in the test.

The rightmost plot of Figure 10a shows the relationship between the proportion of estimable interventions and the cost efficiency for OpenAPS. To clarify, neither Greedy nor DDmin calculate estimates for any intervention; they are simply shown for comparison with CausalCut and CausalCut+Greedy. While Figure 10a clearly shows that high cost efficiencies were only achieved when the test contained a high proportion of estimable interventions, the points for both CausalCut

Table 3. Spearman r test results between each feature and cost efficiency.

		Original length		Necessary interventions		Executions available		CI alpha		Estimable	
	Technique	stat	p-value	stat	p-value	stat	p-value	stat	p-value	stat	p-value
OpenAPS	Greedy	-0.960	**0**	0.636	**0**	0	1.000	0	1.000	0.147	**2.34e-30**
	DDmin	-0.581	**0**	-0.135	**9.42e-26**	0	1.000	0	1.000	0.119	**2.09e-20**
	CausalCut	-0.393	**1.32e-221**	0.199	**5.24e-55**	0.083	**1.23e-10**	-0.026	**0.047**	0.196	**2.93e-53**
	CausalCut+Greedy	-0.474	**0**	0.057	**1.04e-05**	-0.049	**1.25e-04**	0.023	0.073	0.195	**5.66e-53**
SWaT	Greedy	-1.000	**0**	0.767	**4.98e-313**	0	1.000	0	1.000	0.387	**8.43e-59**
	DDmin	-0.739	**4.30e-279**	0.269	**3.33e-28**	0	1.000	0	1.000	0.260	**2.39e-26**
	CausalCut	-0.771	**2.93e-318**	0.597	**2.21e-156**	0.018	0.478	-0.012	0.624	0.555	**5.57e-131**
	CausalCut+Greedy	-0.787	**0**	0.549	**1.23e-127**	-0.010	0.674	-0.001	0.952	0.427	**1.76e-72**

and CausalCut+Greedy are rather diffuse, and correlations in Table 3 are relatively weak (r=0.196 and r=0.195), reflecting the fact that inaccurate as well as absent estimates lead to a poor cost efficiency.

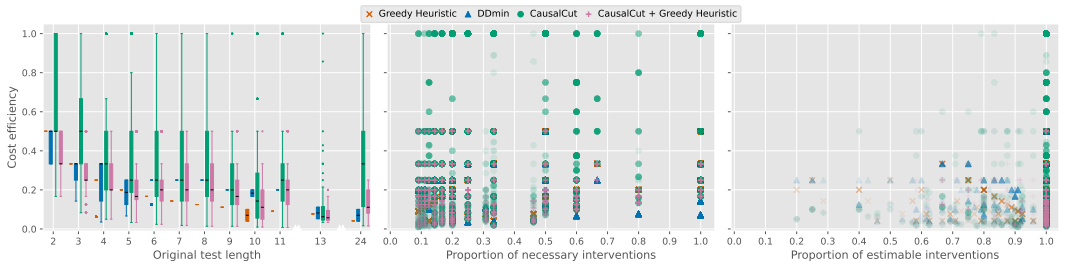
Somewhat surprisingly, Table 3 shows very little correlation between cost efficiency and the numbers of execution available to CausalCut (r=0.083). While we may have expected to see a moderate or even a strong positive correlation between the amount of data and the cost efficiency for both CausalCut and CausalCut+Greedy, the latter actually has a weak negative correlation (r=-0.049), indicating that additional executions can actually harm performance. As we will examine in our answers to RQs 2 and 3, this is because additional data can actually increase the length of the result of CausalCut, leading to more executions being performed by the Greedy postprocessing step of CausalCut+Greedy. We will discuss the reasons behind this in Section 8.

Figure 10b shows the same three relationships as Figure 10a for the SWaT system. The plots mostly indicate similar correlations as for OpenAPS, although CausalCut, CausalCut+Greedy, and DDmin tend to be much less cost efficient than for OpenAPS. CausalCut achieved a mean cost efficiency of just 3% higher than Greedy and a median cost efficiency 1.6% lower than Greedy, although it outperform DDmin by a mean of 12% and and a median 6%, and achieved optimal cost efficiencies for several tests.

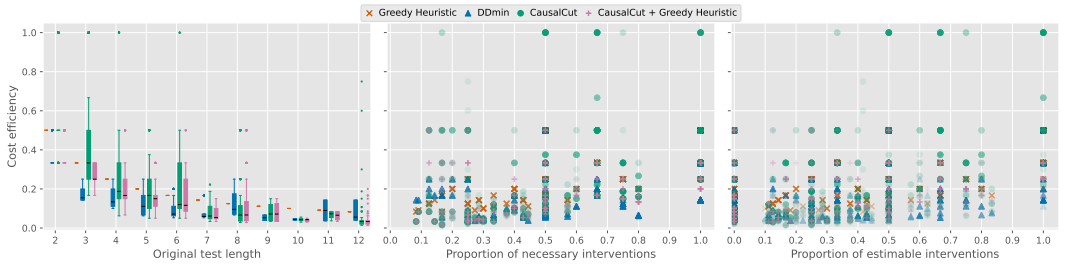
The explanation for the drop in performance from CausalCut and CausalCut+Greedy, as illustrated in Figure 9, is that the SWaT dataset does not enable us to calculate causal effect estimates at all for more than half of the interventions in the tests. Because of this, Table 3 shows a stronger correlation between proportion of estimable interventions and cost efficiencies (r=0.555 for CausalCut for SWaT in comparison to just 0.196 for OpenAPS). The lower threshold for achieving high cost efficiencies is explained by the fact that the tests for OpenAPS mostly have a very high proportion of estimable interventions anyway.

The drop in performance from DDmin is explained by the distribution of necessary and spurious interventions within the SWaT tests. Where the necessary interventions come mostly towards the end of the OpenAPS tests, they are much more evenly distributed in the SWaT tests. This structure is not well aligned with DDmin's search strategy, meaning it requires many more executions. This is reflected in the fact that the correlation between the proportion of necessary interventions and the cost efficiency of DDmin is now positive (r=0.269) as fewer necessary interventions leads to an increase in wasted effort.

Answer to RQ1: The main factor influencing the performance of CausalCut and CausalCut+Greedy in comparison to Greedy and DDmin here is the suitability of the available data. Where the dataset allows us to estimate causal effects for the majority of interventions (OpenAPS), CausalCut is the most cost effective. Where the dataset does not allow estimates to be



(a) Cost efficiency for OpenAPS.



(b) Cost efficiency for SWaT.

Fig. 10. Cost efficiency results.

calculated, DDmin is the most cost effective if the necessary interventions are clustered together at the end of the test, with Greedy being more cost effective if they are spread throughout the test. All the techniques studied here become less cost effective as test length increases and the proportion of necessary interventions decreases. The amount of data available and the confidence intervals used during phase 1 of CausalCut and CausalCut+Greedy have very little impact on the cost efficiency.

7.2 RQ2: Minimisation

The leftmost plots in Figures 11a and 11b show the relationship between original test length and tool-minimised test length for OpenAPS and SWaT, normalised by original test length. In addition to the main techniques, Figure 11 shows the number of interventions estimated as significant by phase 1 of CausalCut. As discussed in Section 4, these are not guaranteed to be failure-inducing, but it is interesting to examine how many interventions are added by each phase of CausalCut.

For OpenAPS, phase 1 of CausalCut resulted in a mean of 45.5% of interventions in the original test being estimated as significant, of which 76.65% were bolus interventions, 15.59% were heavy meals, 6.70% were light meals, and 1.06% were snacks. For SWaT, the proportion of events estimated as significant was just 27.3%, although this is in part due to the inability to calculate an estimate at all for many of the interventions. Of these interventions, 53.54% were on pumps and 46.46% were on valves. Phase 2 of CausalCut then reinstates a mean of 23.2% of interventions for OpenAPS (of which 79.39% were bolus interventions, 12.81% heavy meals, 6.71% light meals, and 1.08% snacks) and 54.6% for SWaT (of which 64.29% intervened on pumps and 35.71% intervened on valves). The Greedy postprocessing phase of CausalCut+Greedy then resulted in a mean reduction of 31.7% for OpenAPS and 64.7% for SWaT. Of these remaining events, for OpenAPS, 70.75% were bolus interventions, 18.32% were heavy meals, 9.83% light meals, and 1.10% snacks. For SWaT, 63.81% of

remaining interventions acted on pumps and 36.19% on valves. A full breakdown of the intervention types for each technique is given in Table 8 in Section B.

Table 4 shows that all techniques exhibit a negative correlation, indicating that we tend to get better minimisations for longer traces. This is explained by the fact that longer tests from both systems tended to contain a higher proportion of spurious interventions, as shown in Table 2. Since they are guaranteed to produce 1-minimal results, it is no surprise that Greedy and DDmin tend to produce the best results and have the strongest negative correlations (r=-0.614 and r=-647). However, Figures 11a and 11b show that CausalCut+Greedy can achieve comparable performance for OpenAPS. Indeed, for OpenAPS tests of length ten, CausalCut+Greedy even produced a better median result than Greedy as these tests involved interaction between variables, although the performance of DDmin was better still. While CausalCut tended to produce the longest minimisations, it did still manage to remove over half the spurious interventions (mean of 54% for SWaT and 52% for OpenAPS).

The middle plots of Figures 11a and 11b tell a similar story in terms of the proportion of necessary interventions. The plots show that Greedy and DDmin have a perfect positive correlation (r=1) for SWaT and an almost perfect correlation (r=0.967 and r=0.979) for OpenAPS. This clearly emphasises the fact that both produce an optimal results when there is no interaction, and that DDmin is less affected by interaction than Greedy. As mentioned in Section 6, the simple simulator we used for SWaT assumes no interaction between variables, which is why both techniques could achieve a perfect result every time.

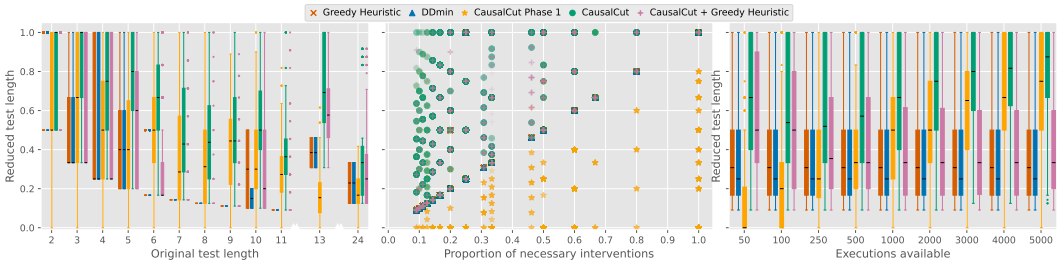
The rightmost plots of Figures 11a and 11b show the relationship between amount of available past execution data and tool-minimised trace length for OpenAPS and SWaT. Since Greedy and DDmin do not make use of this, their results remain constant, and there is precisely no correlation. Surprisingly, Table 4 actually shows a statistically significant *positive* correlation (r=0.239) here for CausalCut, indicating that additional execution data actually leads to *longer* tests. Although fairly weak, it is still strong enough to be seen in 11a, which shows that the best performance (i.e. the smallest tool-minimised test cases) was achieved with 250 executions for OpenAPS. This is due to the same phenomenon observed for CausalCut+Greedy in RQ1 that led to a negative correlation between data and cost efficiency, which we will discuss further in Section 8. CausalCut+Greedy has the monotonic relationship that we might expect, with a weak negative correlation (r=-0.153), indicating that more data tends to lead to shorter tests. Figure 11b and Table 4 show very little correlation for SWaT. This is explained by the fact that, as shown in Figure 9, less than half of the SWaT interventions could have a causal effect estimate calculated for them.

Table 4 shows negligible correlations between the estimable interventions and pruning. Since we are able to calculate effect estimates for the vast majority of interventions even with the smallest sample of the OpenAPS data considered here (50 executions), the ability to estimate interventions *accurately* plays much more of a role here. The correlation is stronger for SWaT (r=0.233) since the data does not allow us to calculate effect estimates at all for many of the interventions, meaning the ability to calculate accurate estimates is more correlated to the ability to calculate them at all.

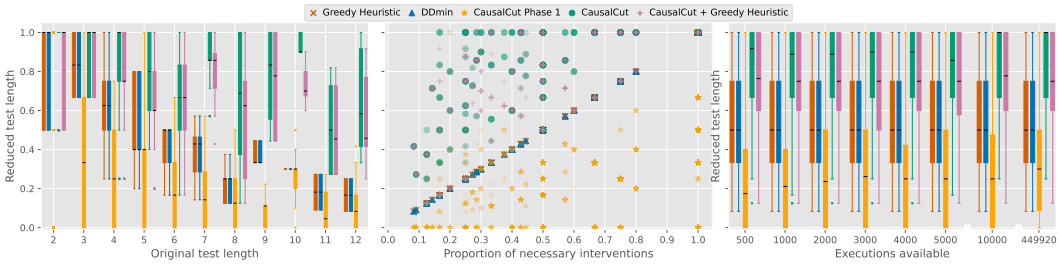
Answer to RQ2: While CausalCut tends to produce the longest minimisations, it still removed over half of the spurious interventions from traces of both systems, and can produce an optimal result if sufficiently high-quality data is available. While Greedy is guaranteed to produce a 1-minimal result, which is minimal when there is no interaction between interventions, CausalCut+Greedy can produce a better result where interventions do interact if suitable data is available. The pruning ability of DDmin is less affected by interaction than Greedy as it explores more combinations, although this comes at the cost of requiring more executions.

Table 4. Spearman r test results between each feature and normalised tool-minimised test length.

		Original length		Necessary interventions		Executions available		CI alpha		Estimable	
	Technique	stat	p-value	stat	p-value	stat	p-value	stat	p-value	stat	p-value
OpenAPS	Greedy	-0.614	**0**	0.967	**0**	0	1.000	0	1.000	0.046	**3.44e-04**
	DDmin	-0.647	**0**	0.979	**0**	0	1.000	0	1.000	0.056	**1.32e-05**
	CausalCut Phase 1	-0.266	**5.33e-98**	0.106	**1.33e-16**	0.604	**0**	-0.165	**5.98e-38**	0.191	**1.14e-50**
	CausalCut	-0.361	**1.09e-184**	0.554	**0**	0.239	**9.20e-79**	-0.058	**6.83e-06**	-0.010	0.438
	CausalCut+Greedy	-0.457	**1.09e-309**	0.841	**0**	-0.153	**5.05e-33**	0.035	**0.006**	-0.078	**1.13e-09**
SWaT	Greedy	-0.767	**4.98e-313**	1.000	**0**	0	1.000	0	1.000	0.361	**5.09e-51**
	DDmin	-0.767	**4.98e-313**	1.000	**0**	0	1.000	0	1.000	0.361	**5.09e-51**
	CausalCut Phase 1	-0.384	**7.24e-58**	0.363	**2.04e-51**	0.090	**2.84e-04**	-0.035	0.154	0.870	**0**
	CausalCut	-0.338	**1.59e-44**	0.582	**3.52e-147**	0.019	0.454	-0.006	0.804	0.233	**2.07e-21**
	CausalCut+Greedy	-0.383	**1.19e-57**	0.681	**1.36e-220**	-0.016	0.521	0.006	0.799	0.009	0.715



(a) Tool-minimised length for OpenAPS, normalised by original test length.



(b) Tool-minimised length for SWaT, normalised by original test length.

Fig. 11. Test minimisation results.

7.3 RQ3: Executions

The leftmost plots of Figures 12a and 12b show how the number of executions required to minimise tests scales with test length (again normalised by original length). It is here that CausalCut comes into its own. Where Greedy always performs one execution per intervention in the original test, CausalCut performs a mean of just 0.46 executions per intervention for OpenAPS and 0.79 for SWaT, making it much cheaper to run. DDmin tends to require the most executions here, requiring a mean of 1.08 executions per event for OpenAPS and 1.83 for SWaT.

As mentioned in Section 6.1.2, a typical execution of the OpenAPS simulator took around 15 seconds on a standard desktop machine, so CausalCut represents a saving of around seven seconds per intervention in comparison to Greedy or DDmin. Poskitt et al. [50] applied each SWaT intervention for 15 minutes, meaning that CausalCut represents a saving of just around 3 minutes per intervention in comparison to Greedy and close to 6 minutes in comparison to DDmin. More detailed analysis can be found in Section 8. As for the previous RQs, we get the best results from

Table 5. Spearman r test results between each feature and executions.

		Original length		Necessary interventions		Executions available		CI alpha		Estimable	
	Technique	stat	p-value	stat	p-value	stat	p-value	stat	p-value	stat	p-value
OpenAPS	Greedy	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
	DDmin	-0.576	**0**	0.939	**0**	0	1.000	0	1.000	0.056	**1.18e-05**
	CausalCut	-0.464	**0**	0.638	**0**	-0.323	**4.73e-146**	0.091	**1.16e-12**	-0.119	**1.73e-20**
	CausalCut+Greedy	-0.605	**0**	0.651	**0**	0.206	**1.33e-58**	-0.048	**1.86e-04**	0.033	**0.010**
SWaT	Greedy	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
	DDmin	0.133	**7.43e-08**	0.332	**7.12e-43**	0	1.000	0	1.000	-0.023	0.352
	CausalCut	-0.240	**1.42e-22**	0.379	**1.86e-56**	-0.061	**0.014**	0.026	0.293	-0.404	**2.04e-64**
	CausalCut+Greedy	-0.669	**1.27e-210**	0.764	**6.22e-309**	0.014	0.566	-0.005	0.842	0.331	**1.00e-42**

the longest traces, with Table 5 showing statistically significant negative correlations for both CausalCut and CausalCut+Greedy, although the absolute numbers of executions performed of course increases with test length.

The middle plots of Figures 19a and 19b show how the number of executions required to minimise tests scales with the proportion of necessary interventions. Though not immediately apparent in these plots, Table 5 shows moderate positive correlations for CausalCut and CausalCut+Greedy (r=0.638 and r=0.651). As with the previous RQs, we must take into consideration the confounding factor of trace length here since, as shown in Table 2, shorter traces tend to have a higher proportion of necessary interventions.

The rightmost plots in Figures 12a and 12b show a surprising relationship between the amount of available data and the executions required to perform minimisation. Where CausalCut shows the monotonic downward trend we would expect (r=-0.323 for OpenAPS and r=-0.061 for SWaT), the executions required by CausalCut+Greedy initially reduces, but then starts to increase again. As with RQ2, the best results for OpenAPS are for 250 executions, with the best results for SWaT appearing at 2000 executions. We discuss the reasons behind this in Section 8.

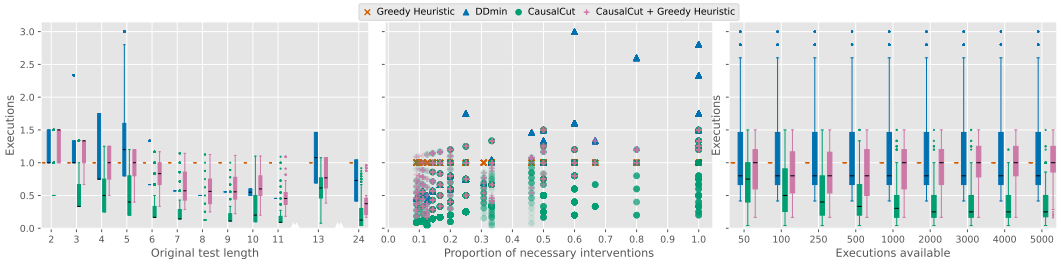
There is a symmetry here between the pruning ability of CausalCut and the number of executions required by CausalCut+Greedy. This same phenomenon is reflected in the positive correlations with the proportion of estimable interventions shown in Table 5 for both system although we omit the plots here as they are not particularly informative. They can be seen in the appendix. Again, we will discuss this further in Section 8.

Answer to RQ3: CausalCut tends to require around half the executions of Greedy to minimise tests, with the number of executions required by DDmin being greatly affected by the distribution of necessary interventions within a test case. The savings of CausalCut and CausalCut+Greedy become greater as tests get longer, with the number of executions required by CausalCut+Greedy tending to fall between CausalCut and Greedy once tests become sufficiently long. The number of executions required by CausalCut tends to fall as the amount of available data increases, whereas CausalCut+Greedy actually seems to require more executions once the amount of data passes a certain point.

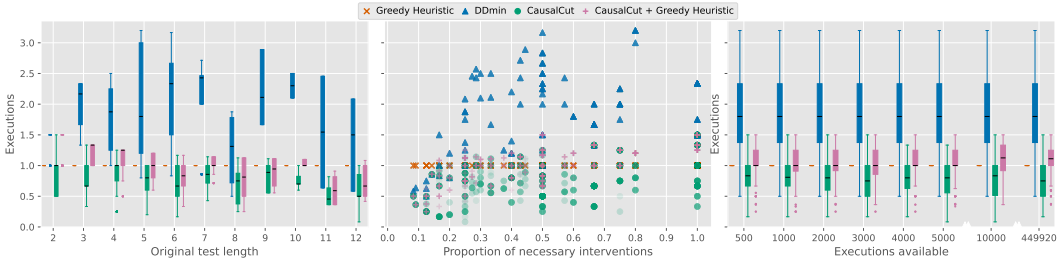
8 Discussion

8.1 Relationship Between Data and Executions

An interesting finding from our results in Section 7 is that there is not a monotonic relationship between the amount of test execution data available to CausalCut and the size of the minimised tests. In fact, CausalCut actually produces the smallest minimisations from a relatively small sample of data. This is because there is a trade-off between the two phases of CausalCut related



(a) Minimisation executions for OpenAPS, normalised by original test length.



(b) Minimisation executions for SWaT, normalised by original test length.

Fig. 12. Minimisation execution results.

to dataset size and the ability to estimate causal effects. Smaller datasets can lack the data needed to calculate causal effect estimates at all for later interventions, causing necessary interventions to be initially pruned during phase 1 and then reintroduced in phase 2, potentially along with spurious ones.

Conversely, larger datasets enable us to calculate causal effect estimates later in the test, but those estimates may not be correct, leading to spurious interventions being falsely retained due to seemingly significant effects. CausalCut+Greedy mitigates this by applying the Greedy heuristic as a postprocessing step after phase 2, enabling it to remove interventions that were falsely estimated as significant during phase 1. This explains the symmetry between the length of minimisations of Greedy and the number of simulations required by CausalCut+Greedy since only interventions added during phase 1 of CausalCut are eligible for Greedy postprocessing.

This raises the practical question of how many test executions should be supplied to enable CausalCut to produce the best possible minimisation. The exact number of executions required to achieve this is likely to vary significantly between systems, and possibly between individual tests. It will also depend on how well the available data meets the criteria described in Section 4.2, and the distribution of necessary interventions in the tests. We therefore cannot provide an exact (recommended) number of executions with which to run CausalCut, and an investigation into the factors that affect this is desirable future work.

However, we do remark that the distribution of necessary interventions in the tests of this study was rather left-tailed for OpenAPS. That is, the interventions necessary to induce a failure tended to occur later on in the test. We then obtain the shortest tests when we have enough data to estimate the causal effects of the interventions in the trace that make up a prefix consisting mostly of spurious interventions, but not so much data that interventions in this prefix are falsely estimated as significant. Following this logic, it is possible that larger amounts of data than we were able to collect for this evaluation may eventually lead to better results if it enabled accurate

estimation of interventions later in the test. In general, if the majority of necessary interventions happen in the second half of the test, then it is desirable to have sufficient data to estimate the causal effect of at least the first half of the test so that the spurious interventions are not added during phase 2 of CausalCut, thereby making them ineligible for Greedy pruning.

Our results for RQ3 show that for SWAT, where there is insufficient data to estimate the causal effect for a substantial proportion of interventions (see Figure 9), there is some evidence of a trade-off between the amount of data available and the number of executions required for minimisation. Whereas CausalCut clearly outperforms Greedy in both settings, this is not necessarily the case for CausalCut+Greedy, especially when the tests are shorter. It is however worth bearing in mind (as elaborated in Section 5.1) that in those cases where CausalCut+Greedy fails to outperform Greedy it will in the worst-case require $|t + 1|$ executions in comparison to the $|t|$ executions required by Greedy, so there is in the worst case a difference of a single execution.

8.2 Necessary Intervention Distribution and Interaction

The distribution of necessary interventions with a test has a significant effect on the performance of DDmin. For OpenAPS, the necessary interventions tended to occur towards the end of the tests. Because DDmin works by splitting tests in half, this structure is very favourable in terms of the number of executions required to perform the reduction. Where the necessary interventions are more evenly distributed throughout the test, as was true for SWaT, DDmin requires many more executions, although is still guaranteed to produce a 1-minimal result.

For OpenAPS, some of the tests of length 10 relied on interaction between necessary interventions to achieve the test goal, meaning that Greedy did not manage to fully minimise them. In this case, CausalCut+Greedy achieved a better reduction as the causal inference automatically accounted for this when estimating the contribution of each intervention. While DDmin was also able to better than Greedy here, this is not guaranteed (since both are only guaranteed a 1-minimal result) and cannot be predicted in advance.

8.3 SWaT Cost Efficiency

Our results for RQ1 were quite different for the two systems. For OpenAPS, CausalCut was the most cost effective, but Greedy had a higher median cost effectiveness for SWaT than either CausalCut or CausalCut+Greedy, despite the fact that it required the more executions for both systems. The explanation for this can be found in Figure 9, which shows that the SWaT data does not allow us to calculate causal effect estimates for more than half of the interventions. This places an increased reliance on phase 2 of CausalCut, which falls back to following the temporal order of interventions where no causal effect estimate can be calculated, and successively adds interventions to the test until it becomes failure-inducing.

While this result may give the impression that the extra executions required by Greedy are somehow "worth it", it belies the true cost of the technique. As mentioned in Section 1, SWaT can cost up to $350 (USD) per hour [34] to run. In their experiment Poskitt et al. [50] executed each intervention for 15 minutes. The longest test case in our evaluation consisted of 12 interventions. This equates to up to 3 hours of real-world time and $1,050 *per run*.

Recall that, to minimise a test case with 12 interventions, Greedy requires 12 executions. By contrast, CausalCut required a median of just 6. This represents a cost saving of $\$1,050 \times 6 = \$6,300$ and a time saving of 18 hours to remove a median of just over half of the spurious interventions (where Greedy removed them all). Depending on the circumstance, the extra cost required by Greedy may prove viable in the long term. For example if the reduced tests are going to be run regularly, then the initial investment of producing an optimal minimisation will eventually pay for itself. However, if the tests are not going to be run again, or may only be run once after a debugging

attempt and discarded if the fault has been resolved, then the testing budget may be better spent elsewhere.

8.4 Fine-grained Analysis of Removed Events

The interventions in a minimal test case can serve as a valuable indicator as to the mechanism of failure. While a detailed analysis of the failures observed during the course of our evaluation is highly system specific and falls outside the scope of this work, we note that our results for RQ2 indicate that the proportions of pruned events by category are broadly in line with the overall proportions in the unminimised test cases. A detailed breakdown can be found in Section B. No technique appears to systematically prune any individual category of intervention (e.g. removing all the pump interventions for SWaT, or only keeping large meals for OpenAPS). Thus, for our two systems, it appears that all interventions cause failures roughly as often as they appear in the tests rather than failures being caused by a few individual actuators.

8.5 Threats to Validity

The main threat to external validity is that we only evaluated our approach on two systems, SWaT and OpenAPS. This threat is somewhat mitigated by the fact that the two systems differ greatly in terms of their application and functionality, but the threat remains that our approach may not scale to other systems. Further evaluation using additional systems is left to future work, although it is worth noting that the underlying causal inference principles have already been theoretically and empirically shown to be generally applicable [6].

There are two main threats to internal validity. Firstly, we collected the test data for OpenAPS ourselves using a process designed to simulate the fuzzing approach typically taken when testing CPSs rather than using a real fuzzer. This was to save on computational and implementational cost, but does introduce the threat that our results may not apply to genuine fuzz data. To mitigate this threat, the second study of our evaluation applies CausalCut to SWaT using only real observational data. Along similar lines, the failure inducing tests for OpenAPS came from a single patient, meaning our results may not generalise to other patients. This threat is mitigated by the fact that the test lasted several months, during which time the patient experienced both hypo- and hyperglycaemia at different times of day, thereby providing a variety of scenarios.

Secondly, since the available failure-inducing tests for SWaT were already minimised, we introduced additional random spurious interventions. For practical and cost reasons, we were not able to run these new tests on SWaT to verify that they are still failure-inducing. Consequently, our results may not be entirely representative of the real SWaT system since we were not able to take account of any interactions between interventions which may have occurred. However, this can only understate the performance of CausalCut in comparison to Greedy since the causal effect estimates will be less reliable than they would otherwise be and, as stated in Section 2.5, Greedy is guaranteed to produce a truly minimal result where there is no interaction between interventions. We note that this threat is specific to the SWaT setting, and that our complementary evaluation on OpenAPS provides evidence of CausalCut's effectiveness in a substantially different CPS context.

9 Related Work

In this section, we discuss prior work on test case minimisation, causal inference in software testing, and testing CPSs.

9.1 Test Case and Test Suite Minimisation

Test case minimisation has been studied extensively as a means to reduce the cost and complexity of debugging and regression testing. A foundational contribution is DDmin [65, 67], which systematically simplifies failing inputs by removing parts of the test case and checking whether the failure still occurs. This idea inspired a range of reduction-based techniques that aim to produce minimal, failure-preserving test inputs. For instance, Misherghi and Su [45] introduced Hierarchical Delta Debugging, which generalises delta debugging to structured inputs such as XML and source code. Delta debugging has subsequently been adapted in multiple ways, e.g. to white box testing for HTML-generating web applications [3], transformation-based compiler testing [22].

In an attempt to reduce the cost of DDmin, Wang et al. [60] use probabilistic models to prioritise likely failure-inducing elements, although this relies on two key assumptions, which do not hold in our context. Monotony states that for tests t, t' and test goal γ, $t \not\vdash \gamma \implies \forall t' \sqsubseteq t.\, t' \not\vdash \gamma$; unambiguity states that $t \vdash \gamma \wedge t' \vdash \gamma \implies t \cap t' \vdash \gamma$. During our search for failure inducing tests for OpenAPS, we observed tests which violate both of these properties through the effects of eating interventions mitigating the effects of bolus interventions, and vice versa. While it may be possible to modify the underlying algorithm to relax these constraints, we leave this for future work.

To provide a better reduction than DDmin, Vince and Kiss [59] propose the DDmin$*$ algorithm, which iterates DDmin until no further improvement is observed. While this was shown to improve reduction by a further 48% in comparison to DDmin, this comes at a cost of additional system executions, which may be prohibitively expensive in the CPS context. Furthermore, the final result is still only guaranteed to be 1-minimal, so may be adversely affected when interventions interact.

In parallel, the regression testing community has explored test *suite* minimisation, which aims to select a minimal subset of test cases that satisfies a given coverage criterion [64]. Traditional test suite minimisation techniques often rely on greedy, heuristic, or metaheuristic algorithms and typically do not focus on preserving failure-inducing behaviours. To address this, several recent approaches shift the focus from code coverage to behavioural diversity, using similarity-based selection to retain test cases that are maximally dissimilar from one another. Notable examples include techniques based on big data clustering algorithms [19], reinforcement learning [29], and large language model embeddings [47]. However, the focus remains on removal of tests from a test suite rather than discovering the interventions which cause failures in individual test cases, meaning that these techniques are not directly applicable to our context.

The property-based testing tool QuickCheck [13] implements several minimisation (or *shrinking*) techniques. Shrinking not only removes unnecessary calls from the test case, but also simplifies arguments where possible based on their data type [32]. For example short lists and smaller integers are preferred. More recently, this minimisation has been leveraged to generalise test cases into equivalence classes based on their method calls and arguments [33]. These are used to prevent the generation of redundant test cases and instead direct test generation towards more diverse tests that discover entirely new failure modes.

There are two limitations that tend to apply to all of the above approaches. Firstly, there is the problem of expense; minimisation relies on the repeated execution of a partial test set, which can become prohibitively expensive when (as is the case with CPSs) they can take a long time to run. Secondly, their results can become brittle if there is nondeterminism in the system under test. By recasting test case minimisation as a causal reasoning problem, the use of CI by CausalCut directly addresses these problems.

9.2 Causality in Software Testing

CI techniques have been applied to fault localisation in traditional software testing, where the goal is to identify faulty program components. For instance, Baah et al. [4, 5] model fault localisation as a CI problem, using program dependence graphs and a linear model to estimate the causal effects of program statements on faults. Küçük et al. [38] build on this idea by transforming predicates into assignments and use machine learning models to estimate the true causal effect of program variables on failures, thereby mitigating confounding bias.

In addition to localising faults, causality-based approaches can be used to help explain *why* they occur. For instance, Johnson et al. [36] introduced 'Causal Testing', a method based on counterfactual causality, designed to assist Java developers in uncovering causal relationships linked to test failures by generating minimally varied inputs that do not trigger the observed fault. Inspired by counterfactual causality, Rößler et al. [52] proposed BugEx, a technique that systematically generates test cases to isolate failure causes by identifying execution features (e.g. branches taken and state predicates) that strongly correlate with failures. This helps developers understand the conditions under which a bug manifests. Giamattei et al. [27] propose an iterative test-generation approach, using Causal Discovery to infer a causal model, which can form the basis for subsequent test generation. Looking beyond monolithic systems, Wu et al. [62] proposed a CI-based approach for diagnosing microservice performance issues, evaluating six CI techniques across different anomaly scenarios to identify root causes and infer detailed explanations of service abnormalities, i.e. based on culprit metrics.

Causality has also been explored for software testing concerns beyond test generation, fault localisation and explanation. In their work on testing scientific software models, Clark et al. show how causal DAGs can enable metamorphic relationships to be checked against existing test-data (without requiring controlled test executions) [15, 16]. Going beyond traditional programs, Sun et al. [55, 56] propose a causality-based approach to repairing faults in neural networks, locating neurons that contribute the most to a defective decision, and adjusting the model parameters related to them until the resultant model behaves correctly. Sun et al. [57] apply a causal analysis tool to automatically identify the causes of faulty behaviours observed in failed autonomous vehicle test cases.

As is the case with all of these CI-based approaches, CausalCut takes advantage of the ability to analyse and reason counterfactually about prior test execution data. This is the first approach to apply these capabilities specifically to the task of test case minimisation.

9.3 Testing Cyber-Physical Systems

Testing CPSs presents a distinct set of challenges arising from their hybrid nature, real-time constraints, and complex interactions between discrete software control and continuous physical dynamics [40]. Moreover, test execution can be costly and time-consuming. These challenges have spurred an extensive line of research on efficient CPS testing and falsification strategies, including model-based [1], fuzzing [10], and simulation-based approaches [7].

In the context of simulation-based approaches, a significant proportion of recent CPS testing work has focussed on the domain of autonomous vehicle testing, using high-fidelity driving simulators such as CARLA [8]. Arcaini et al. [2] target simulator-based autonomous vehicle testing, and aim to minimise tests—which are traffic scenarios—by iteratively removing traffic participants that are not needed. Foster et al. [26] apply CI to regular test cases of driving agents in CARLA to identify faults without the need for controlled additional test inputs.

A prominent class of approaches leverages models built in environments like Simulink, often paired with test generation frameworks. If safety properties are formally specified (e.g. in signal

temporal logic [43]), various techniques can be used to search for violations within the model. For instance, S-TaLiRo [1] uses stochastic search to find simulation runs that minimise a global robustness metric. Yamagata et al. [63] also search for violations by minimising robustness, but use deep reinforcement learning to reduce the number of simulation runs required. However, these approaches typically require complete or high-fidelity models, which are rarely available in practice for complex, black-box CPSs.

In cases where models are unavailable, recent work has turned to black-box strategies such as fuzzing and search-based testing to explore the system's behaviour. For instance, Wijaya et al. [61] fuzzed sensor readings and actuator actions to obtain executions that can be used to train supervised classifiers for anomaly detection. Chen et al. [10] used a genetic algorithm to generate actuator manipulation sequences for SWaT that were predicted to lead to unsafe physical states. The algorithm aimed to find violating inputs as quickly as possible, often converging on the same (typically simplest) solution for a given objective, often including spurious manipulations introduced by optimisation noise. In follow-up work, Chen et al. [11] proposed an active learning-based fuzzing approach targeting network packets, but it continued to suffer from the same issue of irrelevant manipulations. To address this, Poskitt et al. [50] approximated the causal manipulations, defined equivalence classes over them, and guided the fuzzer to explore distinct (non-equivalent) regions of the input space. CausalCut builds on this line of work by offering a causality-driven approach to test case minimisation, avoiding costly re-executions by estimating causal effects from observational data.

10 Conclusion

CPSs present an interesting testing challenge. The behaviour of a CPS is dependent on the state of its environment, continually monitored by sensor readings and affected by its own actuator outputs. CPS executions can be longitudinal by nature, with a state-space that is the product of continuous and discrete inputs and outputs. These characteristics exacerbate the already challenging problem of test case minimisation.

We have shown how CI can be used to effectively minimise test cases in this context. State of the art approaches rely on an ability to run large numbers of follow-up test cases, which becomes intractable when test executions are resource-intensive. CI offers a work-around, by leveraging information from previous test executions to reason about the extent to which individual parts of a test-case contribute to a failure.

Our work proposed CausalCut, an algorithm that relies on CI to trim failing test cases. We also proposed a variant of this (CausalCut+Greedy), which feeds the output of CausalCut into the state-of-the-art minimisation approach by Poskitt et al. [50] (with the additional test executions that this entails).

In our experiments on two CPSs (the APS and SWaT), we have shown that CausalCut removes approximately half of the spurious interventions from test cases, and CausalCut+Greedy removes approximately 87%. Although these approaches do not tend to remove as many interventions as the baseline approaches, the key benefit lies in the reduced cost, which increases as traces become longer. For the SWaT system, CausalCut represented a cost saving of up to $6,300 while still removing over half of the spurious interventions.

One desirable line of future work would be to address the problem of estimating the causal effect of interventions that are towards the end of a test. This would enable the causal effects of more interventions to be calculated, so as to reduce the number of interventions added by phase 2 of CausalCut. One way to do this would be to reduce the overall time period of test cases by removing empty intervention sets altogether. A second approach would be to take advantage of notions of equivalence similar to those derived by Poskitt et al. [50] to reduce the number of

censored runs. Poskitt et al. define several notions of equivalence over tests and these all consider the capabilities utilised in a test (i.e. sensor and actuator manipulations). For the approach proposed in this paper, an alternative might be to define equivalences over tests in terms of the (subset of) interventions that could have had a causal effect. A single test would then lead to an equivalence class of tests, produced by varying (possibly adding or removing) interventions that have no causal effect on the failure. Naturally, any such approach might increase sensitivity to errors in the causal graph. Where it is still impossible to calculate a causal effect estimate, we could instead use more traditional associational machine learning techniques to estimate a hazard ratio. While this would result in biassed estimates, such estimates may be better than no estimate at all, and could provide a better ordering than simply chronological for reinstatement during phase 2. Finally, there is also scope to explore whether minimisation results depend on the types of interventions included in a run and what physical devices these correspond to. Any corresponding results would be specific to the system considered but there might be classes of systems that have similar patterns.

By demonstrating the feasibility of temporal CI in the context of CPS testing, this work has opened up a raft of potential applications in related fields where tests consist of input sequences, for example autonomous driving systems, or involve feedback between variables, for example agent-based models. This work also has the potential to feed into directed fuzzing techniques such as [9, 50], which tend to use simple linear models to predict the probability of failure. The application of causal reasoning could lead to more accurate estimations and more efficient use of data.

Acknowledgments

Foster, Walkinshaw, and Hierons are funded by the EPSRC CITCoM grant EP/T030526/1. For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY)[9] licence to any Author Accepted Manuscript version arising. We are grateful to Yuqi Chen for helpful discussions about the SWaT simulator.

References

[1] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. 2011. S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems. In *TACAS (Lecture Notes in Computer Science, Vol. 6605)*. Springer, 254–257.

[2] Paolo Arcaini, Xiao-Yi Zhang, and Fuyuki Ishikawa. 2022. Less is More: Simplification of Test Scenarios for Autonomous Driving System Testing. In *ICST'22*. 279–290. https://doi.org/10.1109/ICST53961.2022.00037

[3] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. 2010. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Trans. Software Eng.* 36, 4 (2010), 474–494.

[4] George K. Baah, Andy Podgurski, and Mary Jean Harrold. 2010. Causal inference for statistical fault localization. In *ISSTA*. ACM, 73–84.

[5] George K. Baah, Andy Podgurski, and Mary Jean Harrold. 2011. Mitigating the confounding effects of program dependences for effective fault localization. In *SIGSOFT FSE*. ACM, 146–156.

[6] Elias Bareinboim and Judea Pearl. 2016. Causal inference and the data-fusion problem. *Proceedings of the National Academy of Sciences of the United States of America* 113, 27 (2016), 7345–7352. https://www.jstor.org/stable/26470690

[7] Christian Birchler, Sajad Khatiri, Pooja Rani, Timo Kehrer, and Sebastiano Panichella. 2025. A Roadmap for Simulation-Based Testing of Autonomous Cyber-Physical Systems: Challenges and Future Direction. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 152 (May 2025), 9 pages. https://doi.org/10.1145/3711906

[8] CARLA Team. 2025. CARLA Simulator. https://carla.org/. Accessed: 2025-08-19.

[9] Yuqi Chen, Christopher M. Poskitt, and Jun Sun. 2018. Learning from Mutants: Using Code Mutation to Learn and Monitor Invariants of a Cyber-Physical System. In *2018 IEEE Symposium on Security and Privacy (SP)*. 648–660. https://doi.org/10.1109/SP.2018.00016

[9]Where permitted by UKRI a CC-BY-ND licence may be stated instead.

[10] Yuqi Chen, Christopher M. Poskitt, Jun Sun, Sridhar Adepu, and Fan Zhang. 2019. Learning-Guided Network Fuzzing for Testing Cyber-Physical System Defences. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, New York, 962–973. https://doi.org/10.1109/ASE.2019.00093

[11] Yuqi Chen, Bohan Xuan, Christopher M. Poskitt, Jun Sun, and Fan Zhang. 2020. Active fuzzing for testing and securing cyber-physical systems. In *International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA)*. Association for Computing Machinery, New York, 14–26. https://doi.org/10.1145/3395363.3397376

[12] Kwang Ting Cheng and A. S. Krishnakumar. 1993. Automatic functional test generation using the extended finite state machine model. In *International Design Automation Conference* (Dallas, Texas, USA) *(DAC)*. Association for Computing Machinery, New York, 86–91. https://doi.org/10.1145/157485.164585

[13] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (ICFP00)*. ACM, 268–279. https://doi.org/10.1145/351240.351266

[14] Andrew Graham Clark. 2023. *Establishing the Causal Foundations of Metamorphic Testing: A Novel Application of Causal Inference for Testing Computational Modelling Software.* Ph. D. Dissertation. University of Sheffield.

[15] Andrew G. Clark, Michael Foster, Benedikt Prifling, Neil Walkinshaw, Robert M. Hierons, Volker Schmidt, and Robert D. Turner. 2023. Testing Causality in Scientific Modelling Software. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 10 (nov 2023), 42 pages. https://doi.org/10.1145/3607184

[16] Andrew G. Clark, Michael Foster, Neil Walkinshaw, and Robert M. Hierons. 2023. Metamorphic Testing with Causal Graphs. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, New York, 153–164. https://doi.org/10.1109/ICST57152.2023.00023

[17] Sebastián Contreras, David Medina-Ortiz, Carlos Conca, and Álvaro Olivera-Nappa. 2020. A novel synthetic model of the glucose-insulin system for patient-wise inference of physiological parameters from small-size OGTT data. *Frontiers in bioengineering and biotechnology* 8 (2020), 195.

[18] D. R. Cox. 1972. Regression Models and Life-Tables. *Journal of the Royal Statistical Society Series B: Statistical Methodology* 34, 2 (Jan. 1972), 187–202. https://doi.org/10.1111/j.2517-6161.1972.tb00899.x

[19] Emilio Cruciani, Breno Miranda, Roberto Verdecchia, and Antonia Bertolino. 2019. Scalable approaches for test suite reduction. In *ICSE*. IEEE / ACM, 419–429.

[20] Thao Dang. 2011. Model-Based Testing of Hybrid Systems. In *Model-Based Testing for Embedded Systems*, Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman (Eds.). CRC Press. https://doi.org/10.1201/B11321-15

[21] Isabella Degen, Kate Robson Brown, and Zahraa S Abdallah. 2023. Studying insulin needs in Type 1 Diabetes by analysing the OpenAPS Data Commons. *International Journal of Population Data Science* 8, 3 (2023).

[22] Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpinski. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *PLDI*. ACM, 1017–1032.

[23] Andrea Facchinetti, Simone Del Favero, Giovanni Sparacino, Jessica R. Castle, W. Kenneth Ward, and Claudio Cobelli. 2014. Modeling the Glucose Sensor Error. *IEEE Transactions on Biomedical Engineering* 61, 3 (2014), 620–629. https://doi.org/10.1109/TBME.2013.2284023

[24] Tom Fawcett. 2006. An introduction to ROC analysis. *Pattern Recognition Letters* 27, 8 (June 2006), 861–874. https://doi.org/10.1016/j.patrec.2005.10.010

[25] Michael Foster, Andrew Clark, Christopher Wild, Farhad Allian, Robert Turner, Richard Somers, Nicholas Latimer, Neil Walkinshaw, and Robert M. Hierons. 2025. The Causal Testing Framework. *Journal of Open Source Software* 10, 107 (2025), 7739. https://doi.org/10.21105/joss.07739

[26] Michael Foster, Robert M. Hierons, Donghwan Shin, Neil Walkinshaw, and Christopher Wild. 2025. Using Causal Inference to Test Systems with Hidden and Interacting Variables: An Evaluative Case Study. In *EASE'25*. https://arxiv.org/abs/2504.16526

[27] Luca Giamattei, Roberto Pietrantuono, and Stefano Russo. 2023. Reasoning-Based Software Testing. In *International Conference on Software Engineering: New Ideas and Emerging Results* (Melbourne, Australia) *(ICSE-NIER)*. IEEE Press, New York, 66–71. https://doi.org/10.1109/ICSE-NIER58687.2023.00018

[28] Jonathan Goh, Sridhar Adepu, Khurum Nazir Junejo, and Aditya Mathur. 2016. A Dataset to Support Research in the Design of Secure Water Treatment Systems. In *Proc. International Conference on Critical Information Infrastructures Security (CRITIS 2016) (LNCS, Vol. 10242)*. Springer, 88–99. https://doi.org/10.1007/978-3-319-71368-7_8

[29] Sijia Gu and Ali Mesbah. 2025. Scalable Similarity-Aware Test Suite Minimization with Reinforcement Learning. *ACM Trans. Softw. Eng. Methodol.* (2025). https://doi.org/10.1145/3715008

[30] Fitash Ul Haq, Donghwan Shin, and Lionel Briand. 2022. Efficient online testing for DNN-enabled systems using surrogate-assisted and many-objective optimization. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 811–822. https://doi.org/10.1145/3510003.3510188

[31] Miguel A Hernán and James M Robins. 2020. *Causal Inference: What if*. Chapman & Hall/CRC, Boca Raton, FL.
[32] John Hughes. 2016. *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane*. Springer International Publishing, 169–186. https://doi.org/10.1007/978-3-319-30936-1_9
[33] John Hughes, Ulf Norell, Nicholas Smallbone, and Thomas Arts. 2016. Find more bugs with QuickCheck!. In *Proceedings of the 11th International Workshop on Automation of Software Test* (Austin, Texas) *(AST '16)*. Association for Computing Machinery, New York, NY, USA, 71–77. https://doi.org/10.1145/2896921.2896928
[34] iTrust Centre for Research in Cyber Security, Singapore University of Technology and Design. 2024. iTrust Labs overview. https://itrust.sutd.edu.sg/itrust-labs_overview. Accessed 2024-08-05.
[35] iTrust Centre for Research in Cyber Security, Singapore University of Technology and Design. 2024. SWaT dataset. https://itrust.sutd.edu.sg/itrust-labs_datasets. Accessed 2024-08-12.
[36] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2020. Causal testing: understanding defects' root causes. In *ICSE*. ACM, 87–99.
[37] Hyungsub Kim, Muslum Ozgur Ozmen, Antonio Bianchi, Z. Berkay Celik, and Dongyan Xu. 2021. PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, New York, 18 pages. https://doi.org/10.14722/ndss.2021.24096
[38] Yigit Küçük, Tim A. D. Henderson, and Andy Podgurski. 2021. Improving Fault Localization by Integrating Value and Predicate Based Causal Inference Techniques. In *ICSE*. IEEE, 649–660.
[39] Choong Ho Lee and Hyung-Jin Yoon. 2017. Medical big data: promise and challenges. *Kidney Res Clin Pract* 36, 1 (mar 2017), 3–11.
[40] Edward A. Lee. 2008. Cyber Physical Systems: Design Challenges. In *ISORC*. IEEE Computer Society, 363–369.
[41] Daniel Lehner, Jingxi Zhang, Jérôme Pfeiffer, Sabine Sint, Ann-Kathrin Splettstößer, Manuel Wimmer, and Andreas Wortmann. 2025. Model-driven engineering for digital twins: a systematic mapping study. *Software and Systems Modeling* 24, 5 (March 2025), 1339–1377. https://doi.org/10.1007/s10270-025-01264-7
[42] Dana Lewis. 2024. OpenAPS Data Commons. https://openaps.org/outcomes/data-commons. Accessed 2024-08-12.
[43] Oded Maler and Dejan Nickovic. 2004. Monitoring Temporal Properties of Continuous Signals. In *FORMATS/FTRTFT (Lecture Notes in Computer Science, Vol. 3253)*. Springer, 152–166.
[44] Aditya P. Mathur and Nils Ole Tippenhauer. 2016. SWaT: a water treatment testbed for research and training on ICS security. In *2016 International Workshop on Cyber-physical Systems for Smart Water Networks (CySWater)*. IEEE, New York, 31–36. https://doi.org/10.1109/CySWater.2016.7469060
[45] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical Delta Debugging. In *ICSE*. ACM, 142–151.
[46] Sheila F O'Brien and Qi Long Yi. 2016. How do I interpret a confidence interval? *Transfusion* 56, 7 (2016), 1680–1683.
[47] Rongqi Pan, Taher A. Ghaleb, and Lionel C. Briand. 2024. LTM: Scalable and Black-Box Similarity-Based Test Suite Minimization Based on Language Models. *IEEE Transactions on Software Engineering* 50, 11 (2024), 3053–3070. https://doi.org/10.1109/TSE.2024.3469582
[48] Judea Pearl. 2009. *Causality*. Cambridge university press, Cambridge.
[49] Judea Pearl and Dana Mackenzie. 2018. *The Book of Why*. Allen Lane.
[50] Christopher M. Poskitt, Yuqi Chen, Jun Sun, and Yu Jiang. 2023. Finding Causally Different Tests for an Industrial Control System. In *International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE)*. IEEE Press, New York, 2578–2590. https://doi.org/10.1109/ICSE48619.2023.00215
[51] James M. Robins and Andrea Rotnitzky. 1992. *Recovery of Information and Adjustment for Dependent Censoring Using Surrogate Markers*. Birkhäuser Boston, Boston, 297–331. https://doi.org/10.1007/978-1-4757-1229-2_14
[52] Jeremias Rößler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. 2012. Isolating failure causes through test case generation. In *ISSTA*. ACM, 309–319.
[53] Arsalan Shahid and Dana M Lewis. 2022. Large-scale data analysis for glucose variability outcomes with open-source automated insulin delivery systems. *Nutrients* 14, 9 (2022), 1906.
[54] Richard Somers, Neil Walkinshaw, Robert Hierons, Jackie Elliott, Ahmed Iqbal, and Emma Walkinshaw. 2024. Configuration testing of an artificial pancreas system using a digital twin. *STVR* (2024). https://doi.org/10.2139/ssrn.4732706
[55] Bing Sun, Jun Sun, Wayne Koh, and Jie Shi. 2024. Neural Network Semantic Backdoor Detection and Mitigation: A Causality-Based Approach. In *USENIX Security Symposium*. USENIX Association.
[56] Bing Sun, Jun Sun, Long H. Pham, and Tie Shi. 2022. Causality-Based Neural Network Repair. In *ICSE*. ACM, 338–349.
[57] Huijia Sun, Christopher M. Poskitt, Yang Sun, Jun Sun, and Yuqi Chen. 2024. ACAV: A Framework for Automatic Causality Analysis in Autonomous Vehicle Accident Recordings. In *ICSE*. ACM, 102:1–102:13.
[58] Hood Thabit and Roman Hovorka. 2016. Coming of age: the artificial pancreas for type 1 diabetes. *Diabetologia* 59, 9 (jun 2016), 1795–1805.
[59] Dániel Vince and Ákos Kiss. 2024. Evaluation of the fixed-point iteration of minimizing delta debugging. *Journal of Software: Evolution and Process* 36, 10 (June 2024). https://doi.org/10.1002/smr.2702

[60] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic Delta debugging. In *ESEC/FSE'21*. ACM, 881–892. https://doi.org/10.1145/3468264.3468625
[61] Herman Wijaya, Maurício Finavaro Aniche, and Aditya Mathur. 2020. Domain-Based Fuzzing for Supervised Learning of Anomaly Detection in Cyber-Physical Systems. In *International Conference on Software Engineering Workshops* (Seoul, Republic of Korea) *(ICSEW)*. Association for Computing Machinery, New York, 237–244. https://doi.org/10.1145/3387940.3391486
[62] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2021. Causal Inference Techniques for Microservice Performance Diagnosis: Evaluation and Guiding Recommendations. In *ACSOS*. IEEE, 21–30.
[63] Yoriyuki Yamagata, Shuang Liu, Takumi Akazaki, Yihai Duan, and Jianye Hao. 2021. Falsification of Cyber-Physical Systems Using Deep Reinforcement Learning. *IEEE Trans. Software Eng.* 47, 12 (2021), 2823–2840.
[64] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verification Reliab.* 22, 2 (2012), 67–120.
[65] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *ESEC / SIGSOFT FSE (Lecture Notes in Computer Science, Vol. 1687)*. Springer, 253–267.
[66] Andreas Zeller. 2024. *The Debugging Book*. CISPA Helmholtz Center for Information Security. https://www.debuggingbook.org/
[67] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

A Illustration of how to Compute the ATE on a Simplified OpenAPS example

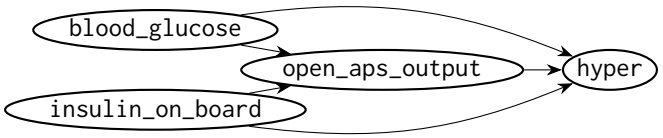


Fig. 13. A highly simplified Causal DAG inspired by our APS case study.

Figure 13 illustrates a causal DAG of an APS system. In this toy example, `blood_glucose`, and `insulin_on_board` represent the 'state' of the human user. These affect the ability of the APS to regulate the blood-glucose levels over several iterations (`open_aps_output`)[10]. This in turn has a causal effect on whether the user experiences hyperglycaemia (`hyper`).

Table 6. Sample execution data from the APS example (simplified for the purpuse of illustration).

open_aps_output 0 min, 2 max	**blood_glucose** 0 too low, 2 too high	**insulin_on_board** 0 none, 2 max	**hyper** 0 false, 1 true
0	0	1	0
1	2	1	1
0	1	2	0
2	2	0	0
0	1	1	0

Consider the sample execution data from the system in Figure 13, shown in Table 6 (again, the data is encoded in a categorical way for the sake of simplicity). From the DAG we can establish that the function for `open_aps_output` can be defined as follows: $f_{open\_aps\_output} =$

[10]A significant aspect of the simplification for this example is that the causal graph does not include cycles. In our actual OpenAPS causal graph we are forced to include cycles, which is why the use of the ATE no longer becomes appropriate, as discussed in Section 3.5.

$f(blood\_glucose, insulin\_on\_board)$. We can establish the function by applying a linear regression to the data, which yields:

$$open\_aps\_output = 0.5652 + 0.6087 * blood\_glucose - 0.6957 * insulin\_on\_board$$

Carrying out a linear regression for `hyper` gives us:

$$hyper = -0.25 + 0.5 * blood\_glucose - \ 0 * insulin\_on\_board - 0.25 * open\_aps\_output$$

Now, the tester might wish to establish the effect that a high initial blood glucose (i.e. $blood\_glucose = 2$) has on the likelihood of triggering hyperglycaemia ($hyper$). There exists a multitude of approaches and measures by which such effects can be calculated, and a detailed discussion of these can be found elsewhere [31, 48]. A common option is the Average Treatment Effect (ATE).

Informally, the ATE can be estimated by running the data (e.g. as given in Table 6) through the model twice. On the first occasion, we record the distribution of values for `hyper` when we fix $blood\_glucose = 2$[11]. On the second run, we simply record the distribution of values for $hyper$ without applying any constraints to the data. Taking the average of the difference between these distributions gives us the ATE.

Table 7. Results from runs of the causal model required to compute the ATE for treatment $blood\_glucose = 2$

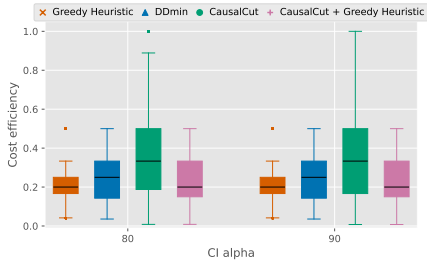
t	$blood\_glucose$	$insulin\_on\_board$	$open\_aps\_output$	$hyper$ (int)	$\mathbb{E}[hyper]$
1	2	1	1.0869	0.478275 (0)	0.5072625
	2	1	1.0869	0.478275 (0)	
	2	2	0.3912	0.6522 (1)	
	2	0	1.7826	0.30435 (0)	
	2	1	1.0869	0.478275 (0)	
	2	2	0.3912	0.6522 (1)	
0	0	1	-0.1305	-0.217375 (0)	0.2753792
	2	1	1.0869	0.478275 (0)	
	1	2	-0.2175	0.304375 (0)	
	2	0	1.7826	0.30435 (0)	
	1	1	0.4782	0.13045 (0)	
	2	2	0.3912	0.6522 (1)	

To apply this to our simplified example, Table 7 shows the results required for $blood\_glucose = 2$ when run through the inferred causal model. The first six rows show the results when we apply the treatment ($T = 1$), and the others show the results when the treatment is not applied ($T = 0$). The estimate of the ATE[12] can be achieved by subtracting the average result for `hyper` for $T = 1$ from the average for $T = 0$, which is $0.507 - 0.274 = 0.233$. In other words, setting $blood\_glucose = 2$ increases the value of `hyper` (before it is rounded to an integer) by 0.233.

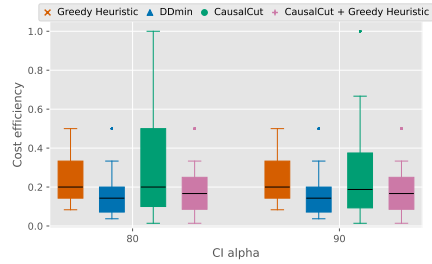
B Supplementary Results

[11] Fixing a value in this way is characterised by applying the 'do' operator in Pearl's do-calculus [48]

[12] It is called an 'estimated' ATE because the outputs are ultimately estimated by the inferred functions in F.

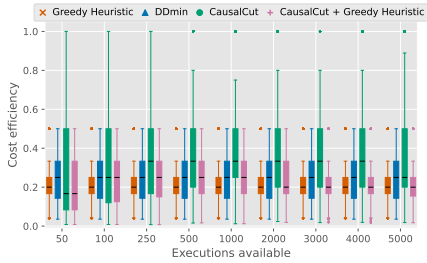


(a) Cost efficiency for OpenAPS.

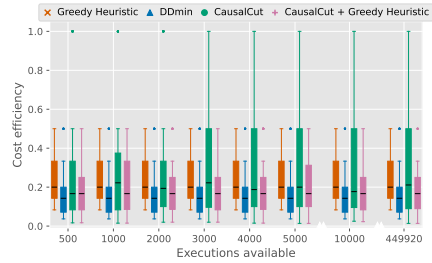


(b) Cost efficiency for SWaT.

Fig. 14. Cost efficiency for OpenAPS and SWaT by confidence intervals.

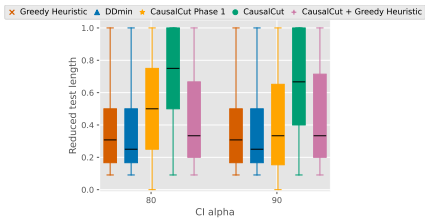


(a) Cost efficiency for OpenAPS.

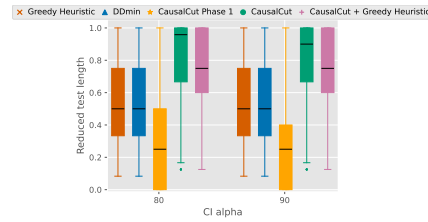


(b) Cost efficiency for SWaT.

Fig. 15. Cost efficiency for OpenAPS and SWaT by confidence intervals.

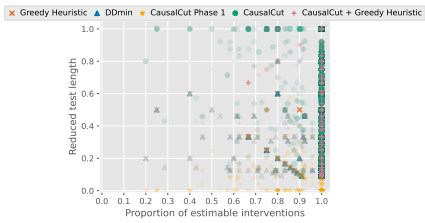


(a) Tool-minimised length for OpenAPS.

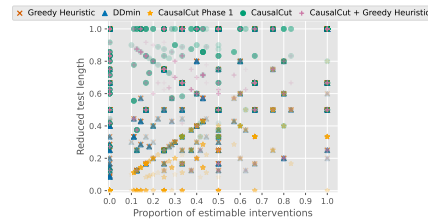


(b) Tool-minimised length for SWaT.

Fig. 16. Tool-minimised length for OpenAPS and SWaT by confidence intervals.

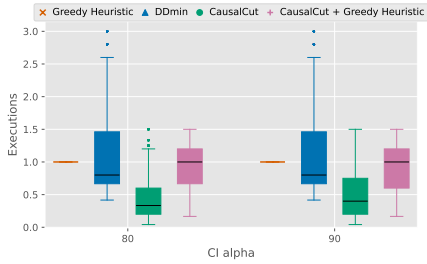


(a) Tool-minimised length for OpenAPS.

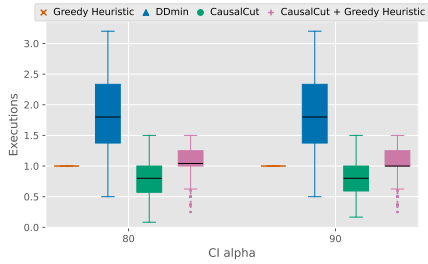


(b) Tool-minimised length for SWaT.

Fig. 17. Tool-minimised length for OpenAPS and SWaT by proportion of necessary interventions.

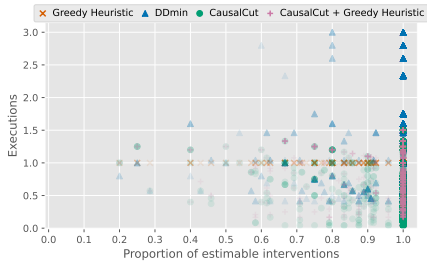


(a) Minimisation executions for OpenAPS.

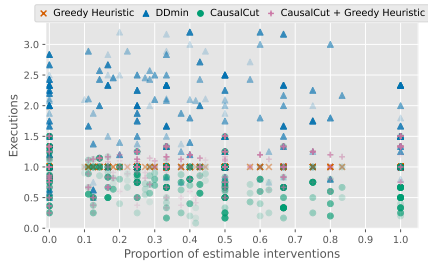


(b) Minimisation executions for SWaT.

Fig. 18. Minimisation executions for OpenAPS and SWaT by confidence intervals.



(a) Minimisation executions for OpenAPS.



(b) Minimisation executions for SWaT.

Fig. 19. Minimisation executions for OpenAPS and SWaT by proportion of necessary interventions.

Table 8. Test case interventions by category.

	OpenAPS				SWaT	
	Bolus	Light meal	Heavy meal	Snack	Pump	Valve
Original tests	86.19%	4.35%	8.18%	1.28%	68.85%	31.15%
Greedy	65.38%	11.54%	22.31%	0.77%	53.56%	46.44%
DDmin	64.57%	11.81%	22.83%	0.79%	53.56%	46.44%
CausalCut	79.39%	6.71%	12.81%	1.08%	64.29%	35.71%
CausalCut+Greedy	70.75%	9.83%	18.32%	1.10%	63.81%	36.19%
CausalCut Phase 1	76.65%	6.70%	15.59%	1.06%	53.54%	46.46%
CausalCut Phase 2 reinstated	79.39%	6.71%	12.81%	1.08%	64.29%	35.71%