Better Safe than Sorry: Preventing Policy Violations through Predictive Root-Cause-Analysis for IoT Systems

Michael Norris
Pennsylvania State University
man5336@psu.edu

Syed Rafiul Hussain
Pennsylvania State University
hussain1@psu.edu

Gang Tan
Pennsylvania State University
gtan@psu.edu

***Abstract*—In an Internet of Things (IoT) environment, there are several way things can go wrong based on device activity. Poorly defined rules, conflicts between applications, physical interactions between devices, or unintentional interference by user behavior. Since these devices can have access to sensitive information or the capability to disrupt or harm physical elements in an environment, there is a strong motivation to protect confidentiality and integrity in IoT systems. In this paper we design IoTArmor, a novel Root-Cause-Analysis tool that uses machine learning models to select remediating actions that can prevent violations that would otherwise occur in the future. We assume violations have been predicted to occur and analyze the current system state to produce optimal fixes to prevent the violating behavior. Through this analysis, we can give accurate proposed fixes to prevent the violations, as well as detailed explanations to users as to why the fixes are effective. This methodology provides easily usable information to users about flaws in their environment, both in the current moment and in their overall application setup.**

I. Introduction

Internet of Things (IoT) is a fast developing field with a growing impact on a variety of environments, from small home deployments to city-wide smart devices. With the complexity of these devices and the software systems that manage them grow, there is an increased need to improve the safety of such systems to reduce harm to persons and property. The more complex IoT grows, the more difficult it is to predict potential problems created through through issues in the user setup or device interaction that can cause harmful behaviors.

Although many techniques in traditional computing systems have been adapted to secure IoT, unfortunately, none of them are capable of performing root cause analysis to help users perform corrective actions to increase safety. Static analysis [4], [6], [14] of IoT application code identifies safety problems in the code, but by itself does not provide user-understandable actions to correct those problems. Dynamic analysis [9], [7] can detect problems immediately before they occur at runtime, but also does not provide fixes. Additionally, only preventing problems as they are about to occur cannot address issues that are to happen in some future time (e.g., 10 minutes later). For this, other methods, such as ProvPredictor [15] and IoTSafe [5], predict problems in the future; however, same as previous systems, they do not provide analysis of possible fixes. Unfortunately, safety violations in this ecosystem are not only difficult to diagnose and resolve, but they also lead to errors or less effective solutions when addressed manually. This is due to the complex interactions between the physical and digital components of IoT systems and lack of in-depth understanding of the behavior of IoT devices. To improve the status quo, in this paper, we develop a system called IoTArmor to automatically provide viable fixes for future policy violations in IoT software environments. In particular, it **(1)** decides what fixes of the current runtime environment exist to prevent a safety violation that is likely to occur in the near future, **(2)** provides simple explanations for why they work, and **(3)** analyzes what effects each would have on the system to determine which one is the best.

To achieve these goals, IoTArmor employs a set of software engineering techniques, particularly a search algorithm for finding effective fixes. Before we can search for effective fixes, we must determine which of the numerous changes to the environment could potentially be used to prevent a policy violation. For our design, we consider viable changes to be device states that can be modified and applications that can be disabled. Therefore, our first step is to generate a set of *potentialChanges* made up of device states and applications that could be related to the violation. The core challenge in designing these tools is dealing with the large search space generated by considering all possible changes in IoT environments and their ramifications. To find a reduced list of potential changes, we perform a causal analysis on the individual behaviors necessary for violations to occur to find device states and applications that are related to the violation. Once a list of potential changes is found, the next search space is comprised of all combinations of these changes, with each combination representing an overall fix. Finding the best fix requires both addressing the challenge of efficiently estimating its overall effect, as well as determining from these effects which fixes are the best for preventing the violation without excessively disrupting the system. By performing a search on fixes to minimize a cost function that balances the violation probability with overall system changes, our design aims to find the least disruptive viable fix without requiring an exhaustive search of the expansive set of possible fixes.

For evaluation, we perform evaluations on physically deployed IoT environments. The first is a home environment we

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00238

set up manually. This is an IFTTT environment that contains 22 services and 40 applications with over 10,000 events over the course of 5 months. Across 12 policies, we were able to identify valid fixes 100% of the time, and the best-fix could be identified over 90% of the time. We also evaluate using a separate home environment dataset with different devices, as well as an agricultural environment dataset, which reports the recorded values of IoT devices deployed on a farm. Our evaluation shows IoTArmor was able to perform nearly as well in this environment, with only a moderate increase in latency accompanied by a minor decrease in accuracy finding effective fixes due to the additional complexity:

- We designed a search algorithm to find violation fixes by minimizing a cost function that uses values derived from predictive models to determine the effects of potential fixes on the violation and the system overall.
- We develop a causal analysis technique to narrow the large search space of system states down to potential fixes with strong causal connections to violating behaviors.
- We evaluated our methodology on a physical home environment and an agricultural environment and found IoTArmor effective in finding fixes and our analysis completed fast enough to allow for fixes to be implemented.

II. Overview

A. Policy Enforcement in IoT

Internet of Things (IoT) represents a diverse set of platforms that share the aim of linking smart devices and digital services to provide a mix of remote and autonomous control over a physical environment. Remote control is provided by allowing users to both monitor system information actively or through alerts and to send remote commands to change device states as desired. Autonomous control mostly functions off of rules that follow simple If-This-Then-That logic. There exist many IoT platforms of various popularity and functionality, such as Openhab, Apple Homekit, Wink, Android Things, KaaIoT, IoTivity, but all share this common goal and basic structure. The goal of enforcing policies in these environments is to ensure the live automated control never compromises the behavior the user intends.

Policy enforcement covers a wide range of possible policy types depending on the target environment and threat model. One such type of enforcement is access control, which focuses on restricting what entities have access to certain information or operations [20], [21]. IoTArmor instead focuses on policy types that enforce restrictions on the state of a system [4]. Specifically, a policy should be a collection of one or more statements about prohibited device behavior with optional contextual information. A statement can be about any device event or device state and then contains a condition under which it should be forbidden. The condition can be one of three types: **(1)** *present:* one or more other device states are present in the system, **(2)** *limit:* a limit on how many times it can occur, and a time-bound for the limit, and **(3)** *trigger*: state or event that triggered the behavior directly or indirectly. For example, a policy "Window Open forbidden when Location NOT Home" indicates the device state of an open window should only be allowed under the condition that location is home. To limit how often the device state “SMS Sent” to 5 times an hour, as user could create the policy “SMS Sent limited to 5 within 60 minutes”. For the final type, “SMS Message cannot be caused by Location *” says event SMS Message cannot be triggered by any Location state or event. With these three types of conditions, users can create policies that dictate the conditions on any behavior that can occur in the system and when it is permitted or forbidden.

We note that a policy violation can be prevented in two different ways, either by correcting the behavior that is causing the violation or correcting the root cause of that behavior. Static analysis [4], [6], [14] generally targets the latter, attempting to identify and remove potential root causes, such as problematic system design or buggy code. Dynamic analysis [9], [7] operates during runtime and uses information only available while the system is running to detect behavior that would cause violations and correct the runtime state. Our work is most similar to dynamic analysis, but instead of simply correcting the current runtime state as violation occurs we attempt to find partial root causes in terms of what device states and/or applications likely caused the behavior that ultimately causes the violation. Unlike static analysis, our goal is not to find a root cause that can permanently be removed from the system to prevent any future violations, but rather to identify states at runtime that can be altered so violations can be prevented ahead of time. This is useful in a number of situations, such as when the violating behavior is not directly under the control of the system or cannot be stopped immediately upon occurring, such as if a change in weather would cause a violation. In this way, rather than simply correcting violating states as they occur, our system identifies partial root causes, tracing backwards from the violating behavior to find the chain of state information that caused the behavior, but not necessarily always finding the ultimate root cause of in terms of system design or code implementation.

B. Motivating Example

To motivate the benefits of the Root-Cause-Analysis(RCA) analysis, we use a demonstrative example where both 1) future prediction is necessary to prevent policy violation and 2) finding the best possible solution to prevent these violations can be difficult for the user. Since users are generally not actively monitoring all autonomous IoT behaviors, they may not be able to intuit why a violation is likely and how best to fix it without additional guidance.

One complication with preventing violations is when the system does not have direct control over the violating behavior(s). For example, let us say that temperature exceeding a threshold could cause a violation in the system. While a system could use Heater of A/C to alter the temperature, it is very likely that this will take some time to accomplish, so the violation could very well happen before the change can

be make. Therefore, steps must be taken ahead of time to prevent it or change the environment such that it no longer causes a violation. Another potential issue is when a change alters the environment so the it is necessary must be aware of what effects the change will have on the environment overall. In this example, altering the temperature will likely affect other physical states like humidity or pressure. Additionally, these changes may also cause applications to fire that cause effects completely unrelated to the initial change or its physical effects. Complexity of finding a solution also increases when multiple changes are required to prevent the violation. For instance, temperature can be affected by a number of environmental components so may require several changes. Turning off the A/C may not be sufficient to prevent a falling temperature, it may also be necessary to close a window letting in cool air, or if the outside temperature is low enough it may also require turning on the Heater. Selecting a fix that will effectively prevent the violation but will have the least impact on the system has several complexities that also scale with how many devices and applications the user has, creating a search spaces that require advanced analysis to find fixes.

C. Challenges and Insights of IoTArmor

The process of finding the best manner in which to prevent a given violation has several challenges due to the unique features of IoT and the difficulty of measuring effects over time, and we will discuss how IoTArmor addresses them.

Finding viable fixes for preventing violations. Depending on the complexity of the system and how far into the future we want to measure the effects of a fix, determining all likely effects of the fix, including overall impact on violation probability, can be difficult to gauge. The immediate effects of a change can be reasonably determined through the knowledge of current state and application analysis, but what implications the implemented change has on the system as time passes are less tractable through static code analysis or dynamic predictions. For this reason, IoTArmor uses provenance information to determine how devices and services interact through applications, and utilizes history of behaviors to train models that track the interaction between different state elements. Using these models, we can estimate what effects a fix may have in the system, including factoring in both cascading effects and the impact of likely future events. By finding states with no high probability paths to violations, we can prevent the violations and limit the ability for noisy events or device interactions to interfere with the fix.

Addressing large search space of possible fixes The search space for possible fixes is large and grows extremely quickly with the number of devices and applications in the system. Without additional analysis, every possible change in a device's state or disabled application, as well as combination of multiple changes, is a possible fix. For this reason, we focus our design on being able to effectively explore this search space. First, we utilize causal analysis to identify those devices and applications that are directly or indirectly related to violating behaviors to prioritize their consideration. We then design a search algorithm based on Simulated Annealing [23] to search the space of possible fixes. This search starts from a random fix and can compare the quality of neighboring fixes to intelligently explore the space of possible fixes and converge on a fix that is effective against the violation.

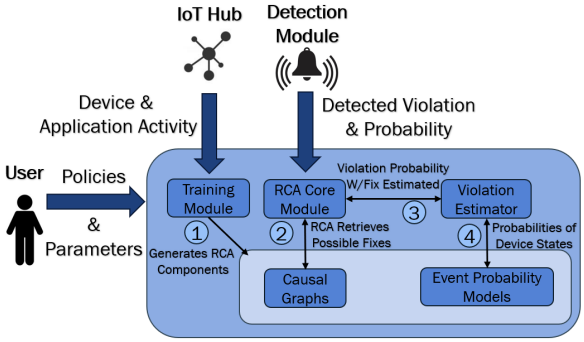


Fig. 1. IoTArmor's overall architecture

D. System overview

The overall design of IoTArmor is shown in Figure 1. We make some assumptions on the existence of several components to drive our system's behavior. We assume an IoT Hub or a similar component that can provide data on device states and applications, and also a detection module for detecting violations. Fortunately, the majority of IoT platforms provide very simple methods of accessing device states, though a separate tool will almost certainly be needed for violation detection. This can be by a technique like dynamic analysis [9], [7] or anomaly detection [1], though to fully leverage IoTArmor the module should predict violations some time before they happen [15]. With this existing variety of existing techniques, it should not be difficult for IoT platforms to integrate IoTArmor. Once IoTArmor is integrated, to use IoTArmor a user provides the policies to enforce as well as some simple model parameters to drive IoTArmor.

In order to find fixes for violations, we first train models on the Device and Application Activity provided by the IoT Hub. For the best results, this should contain a history of all events generated by smart devices and services and every time application code was run. For example, in IFTTT, logs can be generated each time an applet is run by creating an action to send a summary of what the applet did and the related events to a log. Once a large enough log of events is generated, IoTArmor's training module will use this activity to create a history of states to train on, and will use traces generated from that history to generate Causal Graphs and Event Probability Models①. The causal graphs contain edges between device states that training reveals to have causal relationships, meaning one state affects the probability of another state. Using these causal graphs, a model is trained for each possible event in the environment using traces that contain all devices that have a causal relationship with that state. Once these two components are fully generated, they will

be used by the RCA Core Module to find fixes for predicted violations.

When a violation is predicted, we require two pieces of information from the detection module; the probability of the violation and violation type. For the violation probability, the detection module should provide some confidence value of how certain it is the violation will occur. If no confidence level is provided, IoTArmor assumes that there is a 100% confidence. For violation type, our technique requires that the detection module provides which policy it violates. If multiple possible behaviors or states can violate the policy, accuracy can be improved if the specific behavior is provided; otherwise IoTArmor will determine from its own models.

The RCA Module uses the causal graphs to analyze what possible changes in an IoT system are likely to have an effect on violating behavior②. We consider two types of possible changes here: **(1)** changes of devices states and **(2)** disabling applications. To determine what device states may be altered, the RCA module analyzes the causal graphs to find all device states that have causal links to the violating behavior. Then, for each device state related through causal links, we add any application that can cause this behavior to our potential changes.

In order to find good fixes from all potential fixes, the RCA Module uses a search algorithm driven by a cost function to explore possible combinations of changes. For each explored fix, it uses predictive models to estimate the effects of applying the fix to the state ③. By applying the selected changes to the current state we generate a new state where the fix is implemented ④. Comparing predictions based on the new state with the unmodified state can determine the likely effects of implementing the fix. The effects on the violation probability and overall system state are used to determine the fix's cost. By reducing the cost, the search aims to converge on the fix with the least overall system impact but also high probability of preventing the violation. Currently, we assume the priorities of the user are that the violation is prevented with the minimal necessary change to their standard environment operation. Future work could allow for user input to determine other priorities, such as avoiding altering certain devices or conserving energy or implementing a method that does not place too much burden on the user. Additionally, while our design means IoTArmor will never predict conflicting changes in a fix, it is possible that insufficient or significantly noisy training data can cause selected fixes to be ineffective or potentially harmful. Because user behaviors and unknown devices are the greatest source of possible error and these fixes are not currently automated, a user should be able to determine if a suggested fix is likely harmful, but this would become a greater concern in future work automating such fixes.

III. Methodology

In this section, we discuss how we use causal analysis, probabilistic models, and a search algorithm to find viable fixes.

A. Generating Potential Fixes

We define a behavior to be either a state or an event in an IoT system. To find all potential fixes, IoTArmor iterates over all behaviors of devices in the system to find those that can cause violations to the input policy. This begins at behaviors directly related to the violated policy, and in order to determine what behaviors additional device states can affect those behaviors, we perform causal analysis using Chi-Square analysis [8] to find causal links between device states. To perform causal analysis, for each pair of device states in the system, the history provided by training data is used to generate a trace of the states of those two devices over time. We adapt base Chi-Square analysis to consider how often the occurrence of each device state is preceded by another device state. This is useful for determining the direction of the causal relation rather than simply a correlation. To determine if one device state is likely caused by the other, every time the target state is entered, whether the other device state was also entered in a recent time window is recorded. To determine this time window, we perform initial granger analysis on the two devices, which utilizes a similar window as a parameter known as lag. Granger analysis returns a certainty value for each lag with the conventional value of 5% being the lowest accepted value, and so the maximum lag that meets that criteria we use as our window. We make a determination using the Chi-Square equation comparing the expected value to the recorded value. Following Chi-Square general practice we accept the causal link if the p-value is less than 0.05. The calculation is performed in both directions for each device state pair, and when a causal link is accepted, we record the p-value to save the strength of the link and add the pair to an overall *causalGraph* which stores all causal links that involved device states that could affect violating behaviors.

For each behavior, IoTArmor performs analysis on *causalLinks* starting at the violating behavior and performing backwards reachability analysis. This analysis is performed by following directional causal links backward and transitively. The goal is to create a set *potentialChanges* that contains every device state that could be inserted into the current state to alter the vilation probability. For this reason, we first check if each reached state is present in the current state. If it is, that means it has an effect on violation probability itself, so any other state it is changed to could affect violation probability. Therefore, each possible alternate state of that device is added to *potentialChanges*. If the reached state is not in the current state, itself is added, since it is possible it has a negative causal relation to the behavior so inserting it in the system could prevent violation. After all violating behaviors have been explored, we additionally iterate over each state in *potentialChanges* to determine if any applications can result in those states, as disabling these applications could prevent actions that would cause that state. Each application that can is then also added to *potentialChanges* such that afterwards it includes all individual changes that can be made to the system to reduce the violation probability. Afterwards,

Device	Current State	Possible States
Temp Sensor	Low	High/Low
Window	Closed	Open/Closed
SmartLight	Off	On/Off
Heater	On	On/Off
Location	Home	Home/Work/Unknown

TABLE I
LIST OF ALL DEVICES USED IN METHODOLOGY EXAMPLE.

Application	Description
1	If Location is Home then Turn On SmartLight
2	If Temperature is Low then Turn on Heater
3	If Temperature is High then Open Window

TABLE II
LIST OF APPLICATIONS USED IN METHODOLOGY EXAMPLE.

IoTArmor performs a search over all combinations of these changes to find a good fix. These causal links are also how we provide the user with reasoning for why the fix will prevent the violation. By collecting the highest weight paths from the changes in the fix to the violating behaviors and reporting to the user the chain of interactions that prevents the violation.

We next illustrate using a simple example containing sample devices and applications. Assume we have a home environment with the devices in the states listed in Table I and the applications listed in Table II. In this environment, say we have the policy "Window should not be open while user location is not home", to keep the home secured against intruders while user is not home.

Now, let us assume we start in the state [Location:Home, Window:Closed, Temperature:Low, Heater:On] and our violation detection module has predicted that the policy will be violated with a 50% probability in the near future. The first step is to break down the policy into a set of behaviors that must occur for the violation. In this case, we get the set {Window:Open, NOT Location Home}. Since neither of these is true in the current state, we utilize the causal links found through the Chi-Square analysis [8]. These links can be represented in a causal graph in Figure 2, which shows what devices are causally linked and how strong the link is, represented by 1-*p* found during Chi-Square analysis. For each of our violating behaviors, we find all backward reachable device states by traveling in reverse along causal links, which reveals device states that have an effect on the behavior directly or indirectly.

In this example, "Window:Open" can be reached by almost all possible states, with the exception of those of SmartLight. Conversely, "NOT Location Home", meaning "Location:Work" or "Location:Away", does not have other states that can reach them. Thus, since "Window:Closed", "Temperature:Low", "Location:Home", and "Heater:On" are backward reachable and in the current state, all alternative states of each of those devices are added to the potentialChanges set, resulting in "Location:Work/Unknown", "Temperature:High", "Heater:Off", "Window:Open". After all device states are found, for each of these states, we check for any applications that cause that state in the system through actions. We find applications [App-2,App-3] from Table II could potentially be disabled or modified to prevent violation since they can change the state of devices in our causal set. App-1 has a trigger related to the causal set, but the action does not relate to any; so it is not included. We then move on to the next step which is searching for the best possible combination of changes to prevent the violation.

B. Searching for Fixes

Optimizing the search over the state space is essential because with even a moderate number of relevant devices the number of possible fixes becomes very large. Further, evaluating a fix requires making a prediction about the future effects of the fix, which incurs a non-trivial latency cost. For instance, in our experiments we found that, even in our fairly simple example setup, predictions up to 15 minutes in the future could take upwards of 1 second even on a relatively powerful server. Then, considering that the state space is made up of every possible combination of device states of all devices in the system, the number of fixes in an exhaustive search becomes massive. There are fortunately some factors that help restrict this space, such as causal analysis allowing us to remove some devices and device states from consideration. Additionally, states from the same device are mutually exclusive since they cannot both simultaneously be present. Despite this, even considering a simple set of 4 devices with 3 possible states each. We have to consider the fixes for each number of changes up to the number of devices, meaning how many devices are involved in the fix. Since we have $3 * 4$ device states, this can be roughly calculated using the sum of the combinations equation $\sum_{r=1}^{4} {}_{12}C_r = 793$. However, since states from the same device are mutually exclusive, any combination with two or more states from the same device must be removed, meaning we actually end up with 318 total fixes. Even if each evaluation takes only 1 second, an exhaustive search could take more than 5 minutes, greatly reducing the time left to prevent the violation and possibly making it impossible to do so. It is for this reason that substantial effort was directed at optimizing the search space and fix selection.

In order to discover viable fixes from the large state space of possible actions, we adopt the Simulated Annealing (SA) [23] algorithm to perform a guided search of possible fixes. Utilizing Simulated Annealing requires defining several components based on the features of the target search space; ① a definition of a state in the search space and a function for returning the neighboring states for a given state. ② a cost function that when applied to a state returns a value that shrinks proportionally to how closely the state matches an ideal solution, ③ a terminating condition for when the search stops looking for better solutions, and ④ a transition equation utilizing a temperature and a cooling schedule to determine how often a neighbor with a higher cost is accepted. With these four general components defined, an SA algorithm selects

a starting random state and calculates its cost, repeats the process of selecting a random neighbor from the neighboring states of the current state, and uses the cost function and transition equation to determine if it moves to the new state. The search ends if it meets the terminating condition or has no valid neighbors to explore.

For our search problem, we define SA nodes as fixes, which are sets of one or more changes to the system in the form of altering device states or disabling applications. A neighboring fix is one that the input fix can be mutated into with only a single action of either adding, removing, or replacing one of the existing changes in the input fix. So for our search, we first select an initial random fix from those that contain only one change. For our cost function, better cost is achieved by a fix through minimizing the probability of the violation, number of individual changes, and the overall change to the system. We will define the cost function in a later subsection. To select a neighboring node to explore, we simply consider every fix where only one change must be added, removed, or altered from the current fix to match, then we use a transition equation to determine whether the new node is accepted or rejected for exploration. We terminate search after some parameter (*searchLimit*) neighbors that fail to improve the cost.

In order to evaluate fixes, we first need to perform a prediction for the current state to get a baseline for what the future states of the system are likely to be. We perform a prediction by using the *eventModels* to predict the likely device states. For this work, we use a similar set of event models to the paper [15], meaning a set of specialized Markov Models, each of which predicts the probability of an individual device state at the next time step. By normalizing the results of these models for each device, we can get the probability of each overall state of the system at each time step. At each *timeStep*, the probability of any given device state is predicted by the associated model based on the current state and predictions leading up to that time-step and then stored in *baseStates*. Then, to get the cost of a fix, we generate a new current state by applying the changes in the fix, and then we predict how these changes will alter the likely future of the environment. We utilize our trained models to make predictions over time, as we did when getting the *baseStates*, but at each time-step, we calculate two metrics. First, we calculate the probability that a violation of the target policy has occurred at this state to *newViolationProb*. This is done by examining the target policy, the history, and the current state to determine if the required events have occurred for a violation to occur. Then, we compare the probability of each device state in this time step to its probability in *baseStates* and add the difference to *stateChange*. This means at the end of the new prediction, *newViolationProb* represents the likelihood of the violation after the fix is implemented and *stateChange* represents the overall impact of the fix on the state of the environment. Finally, the cost function is calculated and if it is an improvement it is recorded as the new *bestFix* and we reset our tracker for *searchLimit*. If not, we increment our *searchLimit* tracker, terminating if it reaches the limit. We then use our transition equation to determine if the search continues from current fix or the neighbor.

C. Designing a Cost Function

To check the effectiveness of each fix, we create a cost function to balance several desirable attributes of a fix in an IoT environment. For our fixes, we designate three features of a fix in IoT that determine how desirable it is in a user's environment; *Violation Improvement* (VI) is how much the fix reduces the probability of the violation, *Fix Size* (FS) is how many changes are in the fix, and *State Change* (SC) is the total effect the fix has on the likely future state of the system.

For each fix, we first calculate *VI*. To start with, the initial violation probability *VP* is provided by the module that predicted the violation initially. If it was predicted deterministically, this could be set to 100%. Otherwise, this is the confidence rating of the prediction module. When running the prediction algorithm with a fix applied to the state, it will return the new probability of the violation which we set to *VN*. To determine the quality of the result, we can compare this to VP. Since the goal is to minimize the cost, we calculate *VI* by VI = (VN/VP) A good fix would produce a *VN* that is small compared to *VP*, so *VN* is divided by *VP* to weigh improvements at a range of 0-1 and allow for penalizing increasing violation probability.

The fix size *FS* is simply the size of the number of changes in the fix. Next we must determine *SC*. In order to determine how much the fix affects the system, we implement our prediction algorithm in such a way that it tracks how much the probability of each element of the state changes over the predicted future. First, an initial run with the current state is necessary to get a baseline of how the state is predicted to appear at each time-step and record them. Then, when the prediction for each change is run, the algorithm will get the absolute value of the difference between the originally predicted state probabilities and the new predicted state probabilities. The total difference averaged across devices is used as *SC*.

Once all values are calculated, the final step before cost can be calculated is to determine how they should be weighted. This is necessary to prevent over-tuning towards one single feature. If features are naively multiplied together, a fix could optimize the cost function by reducing any feature to 0. For example, by simply not changing the state at all or possibly by disabling all activity to ensure no violations can occur. For this reason, *VI* and *SC* must have some non-zero minimum enforced, since they can naturally reach 0 easily. This minimum value we call *m* can be any value above 0 and may vary based on environmental factors or user parameters, and so it is best if this is tuned over the course of training. In this case since fix size is unlikely to rise past around 5 changes in a system, allowing only values above 0.1 for each keeps the function from overtuning towards either. Unlike the other two values that have a range of 1-2, *FS* has a range of 1 to the max number of unique fixes. For out design, to simplify fixes, we want to heavily penalize fixes with many changes, so instead of adjusting *FS* to the range 1-2, we simply left it

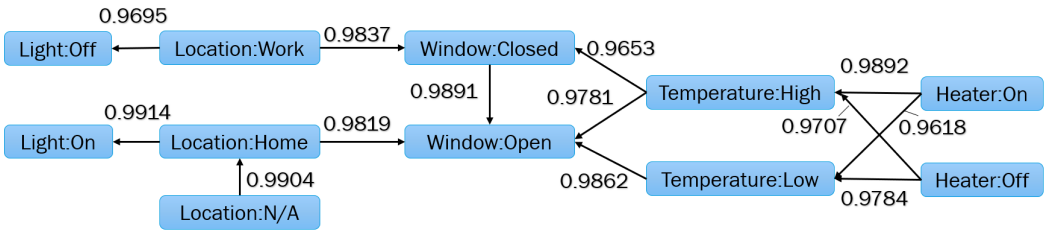


Fig. 2. Device state causal links established through the Chi-Square algorithm

as is. As such, our final cost function is Cost = min(VI,m) * (FS) * min(SC,m)

For example, lets say our original VP is 50%. For Heater, since in the original state the heater is on, we pass a modified state to the predictor to retrieve violation probability if the heater were off. The predictor returns that the violation will now occur with a 0.46% probability, meaning VI = 0.46/0.50 = 0.92 and that the overall state change is 21.16%. Therefore, the cost of fix *F1* is Cost(F1) = 0.46 * 1 * 0.23 = 0.2116 Since the violation probability is still high, the cost is not as small as it could be, and so we use our neighbor finding function to check another fix. For example, a neighbor could add the change to disable App2. So when we run with changes [Heater Off, App-2], *VI* = 3% and *SC* = 39%, because now App-2 does not turn the heater back on to cause the violation. Therefore, our cost function calculates cost of fix *F2* to be Cost(F2) = 0.03 * 2 * 0.39 = 0.0234, which is an improvement and selected as the next target fix. If it was not an improvement, we would need to use our transition equation to determine if exploration should be continued to this new fix.

In order to explore the fix set properly and avoid local minimums, the algorithm must be capable of exploring paths that may begin with an increase in cost. To do so, we follow Simulated Annealing to implement a transition equation to determine when a node with a higher cost should be explored. If the cost is lower, it will always be accepted, but if the new cost is greater than or equal to the current cost, we use the commonly accepted calculation for accepting new state *y* over current state *x*: A(x,y) = exp((Cost(x) - Cost(y))/T), where *T* is the current temperature of the search. This equation determines the probability based on how higher the cost is combined with a cooling effect determined by T (temperature). As the exploration continues, worse states should be accepted less often to converge eventually on a solution. We use a cooling function that divides some constant d by $\log(t)$, where d is generally 0.5 and t is how many nodes have been explored. In order to bias this function towards fixes that effectively lower violation probability (VP), we add current VP to the result; so while the current fix is ineffective we will also likely look for new nodes. This results in the temperature function $T(t) = VP + d/log(t)$ at round t.

IV. Evaluation

To show IoTArmor can achieve its stated goals, we performed evaluations to test (1) *Effectiveness:* can IoTArmor reliably determine what the best fix is for various policy violations and the likely effects of the fix? (2) *Efficiency:* can IoTArmor complete within a reasonable time?

A. Home Environment Evaluation

In order to test IoTArmor, we implemented it using the web-based IoT platform, IFTTT, utilizing applications installed on the platform to control physical devices and virtual services. Provenance data is collected whenever applications run and is used to train models that make predictions about whether violations occur. We deployed devices in a home to contain 7 physical devices, such as a heater and a temperature sensor, and also utilized 17 virtual services. We installed 40 applications from the recommended applications for services on IFTTT, and 4 custom services to represent malicious behavior and poorly-configured applications. Finally, we had 9 policies involving various devices and services across the types of policies enforced by this and other works. Evaluations were run on a Linux machine running Ubuntu 22.04 with 32 GB memory and running on eight 1.5MHz CPUs.

In order to evaluate how well IoTArmor can identify effective fixes for the environment, we first identify every time in our evaluation set where a violation occurs. For each violation, we use expertise of the smart home setup to perform an analysis of the causes between device states and applications and manually identify what fixes would best prevent the violation. We use these identified fixes as our ground truth during the evaluation. We then run IoTArmor's analysis on each violation to determine if it can find the best fix for the violation. We record accuracy based on how often the best fix can be found across different values for our parameters to determine the effectiveness of IoTArmor.

As demonstrated by our cost function in Section II-C, to determine the best fix, we aim for fixes that will have the highest chance of preventing the violation while also causing the least unrelated effects on the environment. Since we cannot know for certain exactly how the trace would have been modified had the fix been implemented at that time, we cannot verify exactly how much each change would affect the system empirically. However, given the deterministic nature of applications, and experience with the environment, identifying

which changes will have the lesser impact is feasible through manual analysis.

Parameters. IoTArmor has several parameters that can be adjusted to alter its priorities in choosing fixes. *Prediction length* determines how far ahead to look to determine the effects of fixes, both in how effective they are at preventing the violation and how much they affect the overall environment. The further ahead the violation is in the future, the more noise and cascading effects IoTArmor must consider, and therefore can increase the difficulty of optimizing the fix. *Search Cutoff Limit* determines how long the search algorithm is allowed to continue while failing to find neighbors with better fixes. The higher this value, the more complete the search of fixes will be and higher accuracy can be achieved, but it also drastically increases the latency of the analysis. While we performed experiments with the cutoff limit, we found results from *Search Restart Count* to be more interesting, and so present results from modifying prediction length and restart count in our evaluation. We select default values for our parameters that hold when that parameter is not being tested. By default we use our median prediction length of 15 minutes, our search halts after 10 nodes with no improvement as such a number means it is likely stuck or has found a good fix, and only performs 1 search for each violation to fairly judge other parameters.

Accuracy Over Improvements. We first perform an ablation study to show how the addition of different elements of our design improve over a typical naive selection of the simplest fix. The graph in Figure 3 shows the accuracy in finding the best possible fix for different approaches. At small prediction values, simpler approaches were reasonable at finding working solutions. In these cases, there was no consideration of long-term consequences of the fix and rarely required more than one change. However, the naive approach consistently failed to find working fixes when multiple changes were needed to prevent the violation. Additionally, once activity beyond the next step is considered, their working fixes were rarely the best fix, as often their solution would overly affect overall state down the line or would not prevent the violation from occurring soon again in the future. As we added do-cause analysis, the system was now able to recognize the overall impact of changes and sometimes select fixes with smaller footprints. Additionally, by actually checking the likely results of the changes, cases where multiple fixes were needed could be identified. Finally, implementing simulated annealing allowed for a more complete search of the space of possible changes, allowing the best fix to be found over 85% of the time even at the most extreme tested prediction ranges. This improvement is also notable for the fact that the fixes other methods failed to find are often the most complex, requiring changing multiple device states and/or disabling several applications. This is especially true as prediction length increases, which increases the effects of any changes made and the number of events that can cause violations. This does come at the cost of increased latency, but since the latency always remains significantly under the relevant prediction length, analysis is near-guaranteed to complete in time for the fix to be implemented.

Latency vs. Accuracy Next, we evaluate how accuracy can be improved by adjusting how many times the search is repeated. Since simulated annealing has a random component to the search in terms of where it starts and which neighbors it selects, there is a chance better solutions can be found by repeating with different starting positions. Performing additional searches comes at the cost of additional latency, but finding an optimal search cutoff paired with number of searches can increase the odds of finding the best possible fix. Therefore, we evaluate how this trade-off scales in Fig 4. In graph (a), we see how accuracy improves when IoTArmor is allowed to perform more random searches. We find this improvement is often in cases that require multiple different changes to provide an effective fix. The search then has difficulty finding the optimal solution that also has good results for the other variable(s) when the necessary intermediate steps are worse for those variables and increase overall cost. These local maximums can be difficult to escape, but by performing several searches from different starting points, the probability of success is increased.

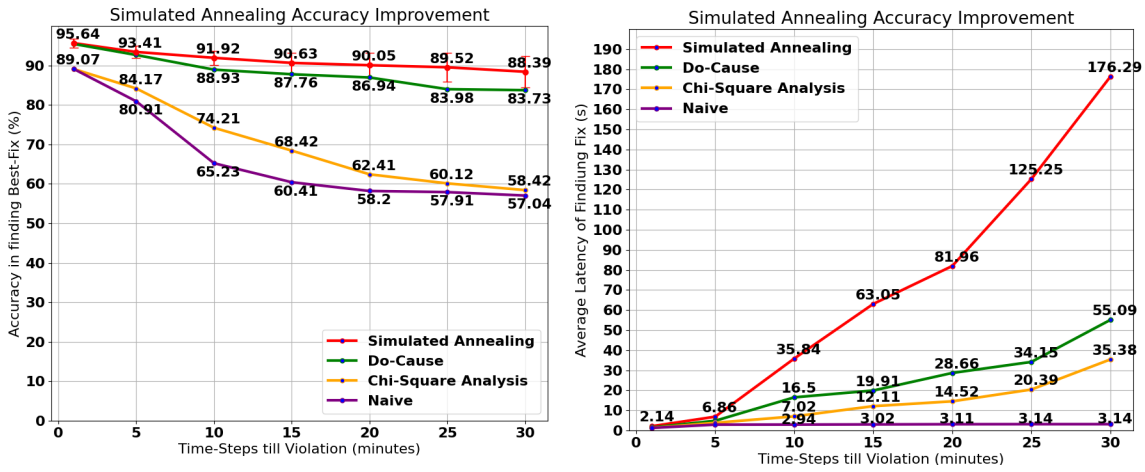
Effectiveness of fixes evaluation In order to determine whether the fixes generated by IoTArmor would indeed be effective at preventing violations, we performed evaluations on our physical environment by replicating states that led to violations and manually implementing the fixes IoTArmor had suggested. Instrumenting IoTArmor to autonomously implement these fixes is for future work, but through manually adjusting device states or disabling applications the effects of the fixes can be observed.

For a sample of our tested policies, we selected three states from the evaluation dataset at random that resulted in a violation occurring within the default time window. Then, the environment was manually forced into that state, by altering physical elements of the environment or forcing events into the system through custom applications. Then, similar methods were used to enforce the changes of the fix that IoTArmor provided for that state and then whether a violation occurs was observed. Overall we were able to validate the IoTArmor fixes for 6 out of our 9 policies. The rest 3 policies involving the temperature and heater devices could not be validated because of a recent hardware error that occurred in the temperature sensor that caused it to be unable to trigger events[1]

B. Evaluations on Public Datasets

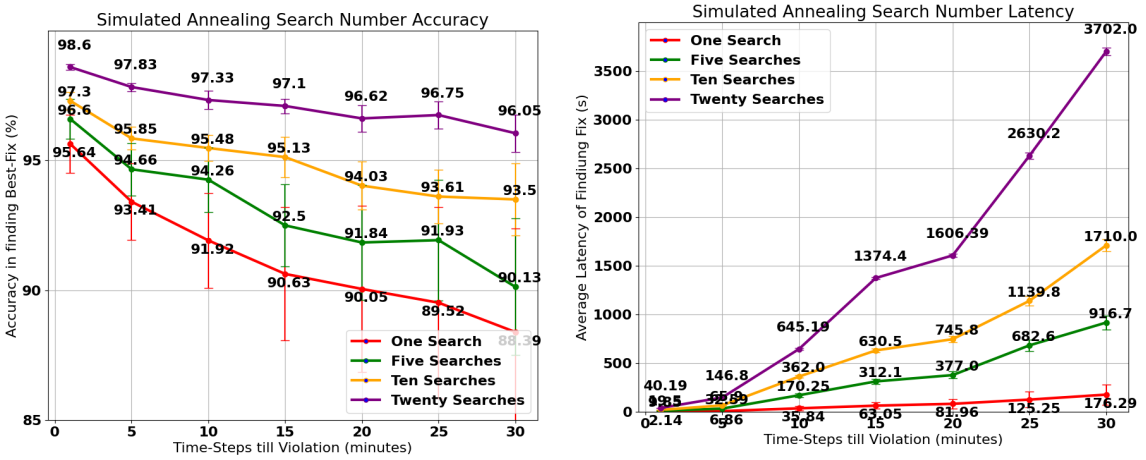
We also utilize existing public datasets to test our design with different devices and environments. In order to test with different devices, we use a dataset in TON_IoT [], which uses telemetry data to track IoT activity in a home environment both during normal operation and during attacks. The deployment featured 9 physical devices very different from our deployment, with the only overlap being the temperature

[1]We are buying a new temperature sensor and will provide the results in the next version. In all tested cases, no violation occurred within the time window after a fix was implemented. While for these evaluations we did not find a suitable metric for measuring how close to minimal impact the proposed fix was, the results at least indicate that the system will likely choose sufficient fixes to prevent violations even while attempting to minimize overall impact.



(a) (b)

Fig. 3. Accuracy and latency of finding the best-fix as prediction length increases across several stages of IoTArmor improvements compared with a simple Naive method.



(a) (b)

Fig. 4. Accuracy and latency of utilizing multiple simulated annealing searches starting from different initial fixes.

sensor from a thermostat. We performed the same training and evaluation as in our own deployment to test whether fixes could be identified. As in our original evaluation, IoTArmor was always able to identify fixes to suggest, though we found that the quality of selected fixes and accuracy dropped due to lack of provenance information and noise in the dataset. The first source of inaccuracies was that this dataset did not contain any information about applications or provenance showing the known cause of certain activity; so causal links could only be inferred from patterns observed in the event traces. In an environment where IoTArmor is deployed, for accuracy it should always have access to provenance information about what application causes an event. The dataset also had the complication of containing a high degree of noise, such as fridge temperature changing by tens of degrees in under a second. Despite these complexities, the accuracy on average stayed within 10% of the previous results. Comparing the results to those for our original evaluation shown in Figure **??**, the initial accuracy was 91.24%, dropping to 81.98% and finally going to 76.72% at 30 minutes. Additionally, as with prior evaluation, even when the absolute best fix is not found, the provided fix would still be useful, but may sometimes implement excessive changes or not quite guarantee no fault occurs.

For a different environment, we evaluated on an existing agricultural dataset. Specifically this was an environment of a farm that grows corn and soy beans with a weather station monitoring it nearby. Since this environment only contains sensors and does not contain applications that drive behavior, there is a more narrow range of information available to determine causes of behaviors and valid fixes. For these reasons we believe it is a valuable additional dataset to test on, as it shows how IoTArmor could be useful in a variety of environments.

Policy	Acc.
Soil Temperature not below 32°	97.83%
Soil VWC not above 20%	98.27%
Soil cond. not above 50% while soil VWC above 15%	94.08%
Humidity not above 60% while soil moisture above 22%	95.82%
Pressure not below 30 while temperature above 80	88.95%
Temperature not above 90°more than 3 daily	92.25%
Soil VWCb below 12% more than 2 times hourly	98.43%

TABLE III
AVERAGE DIFFERENCE IN COST FROM GROUND TRUTH TO SELECTED FIXES IN AGRICULTURAL VIOLATIONS.

For these evaluations, we created policies with the requirements that they ① were similar in type and complexity to our home policies, ② demonstrated reasonable violations that could happen in the environment, and ③ required non-trivial fixes that could be difficult to identify. To ensure ③ could be met, we also restrict what devices can be used in fixes.

With these restrictions in place, we first manually determined a ground truth dataset for violation fixes for each set of permitted devices, using the same method as previous experiments. Without applications and provenance information, this required more rigorous examination of the dataset and interactions shown by the activity and reasoning about logical effects of different changes. For example, when considering a violation to the policy "Soil conductivity cannot be above 50% while soil VWC is above 15%", the violation can be prevented by lowering conductivity or VWC in the soil. This means we want to determine which *can* be lowered, what effects that will have on the system overall, and also what changes are necessary to do so and what the effects of those will be. Since these answers are also heavily context dependent, each instance of a violation we use must be examined and a best estimated ground truth fix selected. In this example, the best ways to lower VWC is by raising the temperature via raising solar radiation, but this has a large impact on the system, and so lowering the conductivity by both lowering humidity and pressure is actually generally preferable. However, in another case the VWC is not high yet but will be going up soon due to rain, so by adjusting the wind we can prevent that change must easier. We perform analysis this way for each violation and generate our list of ground truth fixes.

The first observation we made about this dataset was, due to the properties of the devices, most notably the large number of numeric states, it was extremely difficult for IoTArmor to find the single best fix. However, upon comparing the effects of found fixes with manually identified fixes, we found it could quite often find a near-best fix. To accommodate for these close to correct fixes, we adjusted our evaluation to determine how close to our ground truths IoTArmor could get. For each violation, we calculated the average proportional difference in cost between the ground truth and the selected fix, as shown in Table III. We found that IoTArmor consistently found fixes that were almost exactly as effective as our selected ground truths, which shows IoTArmor is accurate, within the limits of our evaluation. Additionally, we performed this new analysis on our home environment and found it only performed marginally better than our agricultural analysis, meaning IoTArmor was about as good in the more complex setting at finding near-ideal fixes. With our method of generating ground truth and verifying the effectiveness depending on manual determination of fix effectiveness over an objective measure, a full physical deployment would be necessary to demonstrate fully verifiable effectiveness. Nevertheless, this evaluation shows that IoTArmor can operate and make reasonable determinations about different environments.

V. RELATED WORK

Provenance-based Analysis. The vast majority of Provenance work in IoT focuses on analyzing individual events generated by deployed devices. This can be done by verifying common links between devices to check if a device is lying [3], tracking the event through a multi-hop network [11], using device metadata to mitigate faulty behavior [2], or other techniques. While these types of techniques have proven very effective at verifying the source of an event is genuine, they do not consider what effects it has on the greater context of the environment.

While most research focuses on individual events, there are several existing works that aim to store and analyze provenance information collected across an entire system. DDIFT [19] and PDFC [17] aim to prevent improper information flow by using provenance information to enforce information flow policies. This is useful for ensuring confidentiality and integrity of data, but not protecting the actual physical state of IoT devices. There is also work on information and control flow enforcement through collected provenance in IoT [24], [17]. This is similar to ProvPredictor [15], however, they only consider a single sequence of why a certain event is occurring, not the overall system state. Moreover, they do not predict future policy violations, and existing policy enforcement techniques primarily assume that the fix to the violation is self-evident, in that there is a single preventable event or action that will stop a potential violation.

Root-Cause-Analysis. RCA is used across a number of domains in order to explore and explain the why and how of certain activity in systems. In traditional computing systems, RCA has been used to improve upon a wide array of cyber security techniques. By providing additional information and insights into what caused the target provided by the existing technique, RCA improves on such techniques as intrusion detection [10], anomaly detection [1], vulnerability detection and analysis [25], [26], and attack classification [13]. While several of these techniques have been applied in the IoT domain as well, they are so far limited in the scope of what they address. Intrusion and anomaly detection can help secure networks [18], [12], [27] or compromised devices [22], [16], but their primary way of stopping attacks is to simply block actions and isolate devices. This means attacks that use activities that cannot be blocked or isolated or that conduct their attacks through methods allowed in the system circumnavigate

their defense. Conversely, our technique does not depend on positively identifying a specific attacker, but rather working back from the effects of unsafe behavior to find if there is any possible way to prevent it.

VI. Limitations and Discussion

Autonomous Fix Limitation by Environment Ideally policy enforcement would require minimal input from the user and the violation could be prevented entirely autonomously by the tool, and if not could at least be fixed remotely through user input. Due to the diverse nature of IoT systems, however, this is unfortunately almost impossible to guarantee. The capabilities provided remotely are determined by what devices and services are integrated into the system, so whatever change is necessary may require actions the system has no control over. If no smart heater is present then temperature cannot be increased, or an open window may only have a sensor and no actuator to alter its state. For this reason, we did not focus our work on providing such autonomous solutions, as we placed more value on finding a strong technique that could be easily implemented in various environments. Future works could utilize or generate datasets containing information about typical IoT setups in various environments to generate fully autonomous policy enforcement.

VII. Conclusion

We propose IoTArmor, a system built on top of a policy enforcement tool designed to provide users with simple and logical steps to take to prevent violations in their environments, as well as explanations of why the violation is occurring and why the fixes will work. It allows users to not only prevent violations from occurring, but also give users insights into any vulnerabilities, faults, or misconfigured devices and applications present in their system. For this reason, we evaluate IoTArmor on a physical home and agriculture environments to demonstrate how it can find the best possible fixes a user could implement to prevent violations significantly before they happen.

Acknowledgment. The authors thank the anonymous ASE reviewers for their time and invaluable feedback to improve this work. This project has been supported by NSF under Grant No. CNS-1900873.

References

[1] A. Abdelkefi, Y. Jiang, and S. Sharma, "Senatus: an approach to joint traffic anomaly detection and root cause analysis," in *2018 2nd Cyber Security in Networking Conference (CSNet)*. IEEE, 2018, pp. 1–8.

[2] M. S. Aktas and M. Astekin, "Provenance aware run-time verification of things for self-healing internet of things applications," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 3, p. e4263, 2019.

[3] M. N. Aman, M. H. Basheer, and B. Sikdar, "A lightweight protocol for secure data provenance in the internet of things using wireless fingerprints," *IEEE Systems Journal*, vol. 15, no. 2, pp. 2948–2958, 2020.

[4] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated IoT safety and security analysis," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 147–158. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/celik

[5] W. Ding, H. Hu, and L. Cheng, "IoTSafe: Enforcing safety and security policy with real iot physical interaction discovery," in *28th Annual Network and Distributed System Security Symposium (NDSS)*. Virtual: The Internet Society, 2021.

[6] P. Ferrara, A. K. Mandal, A. Cortesi, and F. Spoto, "Static analysis for discovering iot vulnerabilities," *International Journal on Software Tools for Technology Transfer*, vol. 23, no. 1, pp. 71–88, 2021.

[7] M. Ficco, "Detecting iot malware by markov chain behavioral models," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE. Prague, Czech Republic: IEEE, 2019, pp. 229–234.

[8] T. M. Franke, T. Ho, and C. A. Christie, "The chi-square test: Often used and more often misinterpreted," *American journal of evaluation*, vol. 33, no. 3, pp. 448–458, 2012.

[9] J. Jeon, J. H. Park, and Y.-S. Jeong, "Dynamic analysis for iot malware detection with convolution neural network model," *IEEE Access*, vol. 8, pp. 96 899–96 911, 2020.

[10] K. Julisch, "Clustering intrusion detection alarms to support root cause analysis," *ACM transactions on information and system security (TISSEC)*, vol. 6, no. 4, pp. 443–471, 2003.

[11] M. Kamal *et al.*, "Light-weight security and data provenance for multi-hop internet of things," *IEEE Access*, vol. 6, pp. 34 439–34 448, 2018.

[12] P. K. Keserwani, M. C. Govil, E. S. Pilli, and P. Govil, "A smart anomaly-based intrusion detection system for the internet of things (iot) network using gwo–pso–rf model," *Journal of Reliable Intelligent Environments*, vol. 7, no. 1, pp. 3–21, 2021.

[13] M. Khosravi and B. T. Ladani, "Alerts correlation and causal analysis for apt based cyber attack detection," *IEEE Access*, vol. 8, pp. 162 642–162 656, 2020.

[14] Q.-D. Ngo, H.-T. Nguyen, V.-H. Le, and D.-H. Nguyen, "A survey of iot malware and detection methods based on static features," *ICT Express*, vol. 6, no. 4, pp. 280–286, 2020.

[15] M. Norris, P. McDaniel, S. R. Hussain, and G. Tan, "ProvPredictor: Utilizing provenance information for real-time iot policy enforcement," in *2nd EAI International Conference on Security and Privacy in Cyber-Physical Systems and Smart Vehicles (SmartSP)*, 2024, p. To appear.

[16] Y. Qiao, K. Wu, and P. Jin, "Efficient anomaly detection for high-dimensional sensing data with one-class support vector machine," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 1, pp. 404–417, 2021.

[17] X. Rong-na, L. Hui, S. Guo-zhen, G. Yun-chuan, N. Ben, and S. Mang, "Provenance-based data flow control mechanism for internet of things," *Transactions on Emerging Telecommunications Technologies*, vol. 32, no. 5, p. e3934, 2021.

[18] Y. K. Saheed, A. I. Abiodun, S. Misra, M. K. Holone, and R. Colomo-Palacios, "A machine learning-based intrusion detection for detecting internet of things network attacks," *Alexandria Engineering Journal*, vol. 61, no. 12, pp. 9395–9409, 2022.

[19] N. Sapountzis, R. Sun, and D. Oliveira, "Ddift: Decentralized dynamic information flow tracking for iot privacy and security," in *Workshop on Decentralized IoT Systems and Security (DISS)*. San Diego, California: NDSS, 2019.

[20] S. Sicari, A. Rizzardi, L. Grieco, G. Piro, and A. Coen-Porisini, "A policy enforcement framework for internet of things applications in the smart health," *Smart Health*, vol. 3, pp. 39–74, 2017.

[21] F. Siddiqui, M. Hagan, and S. Sezer, "Embedded policing and policy enforcement approach for future secure iot technologies," 2018.

[22] N. K. Thanigaivelan, E. Nigussie, R. K. Kanth, S. Virtanen, and J. Isoaho, "Distributed internal anomaly detection system for internet-of-things," in *2016 13th IEEE annual consumer communications & networking conference (CCNC)*. IEEE, 2016, pp. 319–320.

[23] P. J. Van Laarhoven, E. H. Aarts, P. J. van Laarhoven, and E. H. Aarts, *Simulated annealing*. Springer, 1987.

[24] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the internet of things," in *Network and Distributed Systems Symposium*. San Diego, California: NDSS, 2018.

[25] D. Xu, D. Tang, Y. Chen, X. Wang, K. Chen, H. Tang, and L. Li, "Racing on the negative force: Efficient vulnerability {Root-Cause} analysis through reinforcement learning on counterexamples," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4229–4246.

[26] C. Yagemann, M. Pruett, S. P. Chung, K. Bittick, B. Saltaformaggio, and W. Lee, "{ARCUS}: symbolic root cause analysis of exploits in production systems," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1989–2006.

[27] R. Zhao, G. Gui, Z. Xue, J. Yin, T. Ohtsuki, B. Adebisi, and H. Gacanin, "A novel intrusion detection method based on lightweight neural network for internet of things," *IEEE Internet of Things Journal*, vol. 9, no. 12, pp. 9960–9972, 2021.