Verifying Multi-Vendor IoT Deployments using Conditional Tables

Mubashir Anwar[1][0000−0003−1328−1916], Matthew Caesar[1][0000−0001−5955−9229], and Anduo Wang[2][0000−0002−1078−107X]

[1] University of Illinois Urbana-Champaign, Urbana IL, 61801, USA
{manwar,caesar}@illinois.edu

[2] Temple University, Philadelphia PA, 19122, USA
anduo.wang@gmail.com

Abstract. In recent years, IoT devices have seen widespread deployments in critical environments such as healthcare, military, and home security systems. These deployments often involve managing a heterogeneous collection of devices by non-expert users, increasing the risk of errors that could lead to significant monetary and physical damage. Unfortunately, such deployments face unique challenges such as incomplete visibility (e.g., need to interoperate with closed software systems), incompatibility (e.g., need to interact across different protocols and control logic created by different vendors), and management complexity (e.g., need to express complex and diverse intentions in ways that can be understood by inexperienced users). While system verification has been crucial in catching errors early in other domains, these challenges complicate its application in IoT. To perform verification under these conditions, we propose Pyotr, a system based on a mathematical framework from the theory of database systems called *incomplete databases*. Pyotr can integrate data from heterogeneous devices, perform data analysis under failures and uncertain knowledge, verify intended behavior using easy-to-use database-styled queries, and provide a generalized algebraic framework for reasoning about IoT systems in a rigorous and intuitive way. Our experiments on large IoT networks show that Pyotr can scalably answer complex queries on thousands of connected IoT devices within a few milliseconds.

Keywords: Formal verification · IoT · Conditional Tables.

1 Introduction

In recent years, Internet of things (IoT) has experienced a tremendous surge in growth. Today, there are over 15 billion IoT devices in the world, and this figure is expected to double by 2030 [38]. These devices find applications in vital sectors such as healthcare, military, and residential security. These domains involve extremely critical operations such as collection of sensitive data and automation of actuators capable of impacting the physical environment in dangerous ways. Failures and bugs in devices in these domains can have devastating consequences. For instance, a fault in a node in an IoT deployment for healthcare could lead to the leakage of critical patient information to unintended nodes or prevent it from reaching its intended destination (e.g., a doctor), posing risks to privacy and potentially endangering patients' lives. Similarly, incorrect control logic implemented by a home owner for security could result in issues like doors getting unlocked at unintended times or cameras failing to capture events when they should, rendering the house susceptible to intruders. As IoT devices become more integrated into critical domains, their capacity to inflict severe harm in both the digital and physical realms grows.

The awareness of the risks of automation in these critical systems is not new. When computers were first introduced in these domains, the dangers of errors in software were acknowledged. New debugging and verification techniques were introduced to make software more reliable. With the increasing influence of IoT and its capacity to introduce substantial risks, ensuring the safety and reliability of IoT systems has become a crucial imperative. However, IoT systems present unique challenges since they involve system and protocol heterogeneity, inexperienced users, and a lack of complete visibility into the devices:

System and protocol heterogeneity: IoT devices exhibit significant heterogeneity, making it hard to have a single framework that encompasses all formats. Many deployments involve devices from various vendors, each utilizing its own control language, protocols, management interface, and have differences in the granularity, units, and representation of collected data. Moreover, with high rate of innovation in IoT, the heterogeneity continue to grow, increasing the complexity of deployments. This not only makes it difficult to control IoT devices to work together, but also makes it difficult to manage them and ensure their safety. In a multi-vendor IoT deployment, understanding and reasoning about the collective behaviour of devices is difficult, making it hard to guarantee that the devices function as intended.

Inexperienced users: IoT devices are typically configured and administered by non-technical end users. They are used in places like enterprises, home environments, and in field applications such as agriculture where operators may have limited technical expertise. In these settings, users often lack the skills to program and manage the devices correctly, which can lead to errors in device programming and management. The absence of standardization aggravates this issue, as even if a user-friendly interface exists in one device, it may differ or be nonexistent in others. This lack of consistency complicates the user experience, adding another layer of difficulty when interacting with IoT systems. As IoT devices increasingly permeate various domains, they are being utilized by more and more individuals without formal training, making it challenging to guarantee proper configuration and effective troubleshooting. This raises serious concerns regarding safety and security of IoT deployments.

Lack of complete visibility: IoT deployments often include energy-constrained wireless devices vulnerable to failures and faults. They suffer complex failure modes such as Byzantine faults, communication faults, and timing failures. Moreover, the internal code of these devices is often hidden, making it harder to debug them. Under such challenging conditions, pinpointing the root cause of issues and fixing them is difficult. Unfortunately, we also lack comprehensive information about the faults, which makes it difficult to understand the situation. For instance, some devices in an IoT network may remain operational but are rendered inaccessible due to a variety of reasons (e.g., wireless interference, resource constraints, conserving battery by avoiding information transmission). Assuming the devices have failed would be premature, as they might still be functioning in accordance with our intentions. However, they might be malfunctioning or intentionally attempting to disrupt the system, potentially as a result of malicious user activity. Thus, troubleshooting the problem or ensuring that the system behaves as expected becomes challenging when we lack awareness of a portion of it. Given the critical nature of many IoT domains and the unreliability of methods to access all relevant information, it becomes crucial to model deployments even when certain details are inaccessible. The lack of complete visibility in IoT deployments, compounded by the presence of energy-constrained devices susceptible to various complex failure modes, poses a significant challenge to data accessibility, which is vital for ensuring the safety and reliability of IoT systems.

To tackle the challenges in heterogeneous IoT deployments, we introduce Pyotr, a user-friendly system for reasoning and verification of IoT deployments. Pyotr is based on the theory of *incomplete databases* [22] and uses database-styled semantics to represent the behaviour of IoT devices, which allows users to verify policies and ask "what-if" questions using intuitive queries. Database theory has long grappled with the issue of data inconsistencies and offers multiple user-friendly interfaces for inexperienced users. Furthermore, *incomplete databases* [22] have the capability to reason over missing information and integrate data from heterogeneous sources. These inherent qualities position them as excellent candidates to serve as a foundational basis for a reasoning system for IoT devices. At its core, Pyotr uses conditional tables [22] to represent data (e.g., collected from sensors) and rules (e.g., event condition-action (ECA) rules, forwarding rules). These tables effectively store incomplete information through conditions alongside regular data, making them well-suited for representing data and rules in IoT devices. Conditional tables can be queried using traditional relational algebra, allowing users to formulate questions and policies as database queries. A variety of user-friendly visual methods [25, 33, 37] have been proposed to query over databases, which makes Pyotr easy to use for non-expert users. Moreover, conditional tables can represent missing information and can be used for data integration in heterogeneous environments. Thus, they provide a generalized relational framework for reasoning about various properties of IoT devices, which can simplify development, analysis, and synthesis of other advanced functions for IoT systems. Pyotr is designed to efficiently store, manipulate, and reason over conditional tables. It extends concepts developed in [27] to the domain of IoT. To handle conditional tables, we have developed a query engine, a compiler to parse IoT rules and data, integrated different reasoning engines (e.g., Binary Decision Diagrams (BDD), Satisfiability Modulo Theories (SMT), Difference of Cube (DoC) [30]), and implemented query optimizations such as semi-naive evaluation. We have also modeled diverse use cases for verification in IoT deployments using conditional tables and conducted experiments on three scenarios. The results indicate that Pyotr can verify IoT deployments with minimal overheads. Pyotr also retains desirable scalability properties even in scenarios where portions of devices are inaccessible. Additionally, our evaluation emphasizes the significant impact of the chosen reasoning engine on the performance of conditional tables, illustrated through a comparison of three implemented engines.

The rest of the paper is structured as follows: Section 2 presents motivating examples of errors in IoT deployments, outlining their distinct challenges. In Section 3, a theoretical exploration of conditional tables is provided, accompanied by examples illustrating their application in modeling and verifying IoT deployments. Section 4 describes the architecture and implementation of Pyotr, including the optimizations that we made. Section 5 presents an evaluation of the performance and scalability of Pyotr for different IoT verification tasks. We discuss related work in Section 6 and conclude in Section 7.

2 Motivating Examples

In this section, we use examples to illustrate the necessity of verification in IoT devices, highlighting the distinctive challenges associated with it. Our examples are drawn from a Fire Suppression System in a factory and IoT networks in health monitoring. The next section shows how conditional tables can model and solve these problems.

2.1 Example 1: Configuration errors

Consider a Fire Suppression System installed in a factory, comprising a smoke sensor equipped with a connected light, linked to both an alarm and water sprinklers. The user configures the smoke sensor using event-condition-action (ECA) rules, as shown in the first half of Fig. 1. The user intends for the light to activate only during nighttime when smoke is detected, aiming to preserve battery life by preventing light activation during daylight hours. An alarm is designed to alert users upon the detection of any type of smoke, while water sprinklers automatically engage when smoke levels are elevated (e.g, at *high* level). However, there is a flaw in the control logic. The user desires the alarm to activate whenever noticeable smoke is detected, irrespective of the time of day. Although rules 3-5 seem to address this, rule 3 is limited to nighttime. This limitation could result in delayed fire response during daylight hours, posing a significant risk to factory employees. Even in this simple example, the bug is not easy to spot manually. Ideally, we want the user to ask a simple query such as "Does the alarm always sound whenever there is any noticeable smoke?".

2.2 Example 2: Integrating inconsistent sensor data

Now, let's extend the previous scenario by incorporating an additional smoke sensor from a different vendor into the example. With two sensors now in play, both contribute to the control of the alarm and sprinkler system. The second

```
/∗ Rules for the first device ∗/
IF smoke=low THEN light=off, alarm=off, sprinkler=off
IF time<20:00 THEN light=off
IF smoke=mid,time>20:00 THEN light=on, alarm=on, sprinkler=off
IF smoke=high,time>20:00 THEN light=on, alarm=on, sprinkler=on
IF smoke=high,time<20:00 THEN light=off, alarm=on, sprinkler=on

/∗ Rules for the second device ∗/
IF smoke=low THEN lights=off, alarm=off, sprinkler=off
IF time<20:00 THEN light=off
IF smoke=high,time>20:00 THEN light=on, alarm=on, sprinkler=on
IF smoke=high,time<20:00 THEN light=off, alarm=on, sprinkler=on
```

Fig. 1: Control rules for a Fire Suppression System

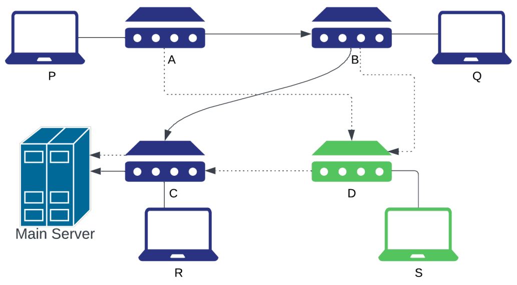


Fig. 2: Healthcare IoT network to monitor patients in critical conditions

smoke sensor is less accurate, and can only discern the presence or absence of smoke without distinguishing between *mid* and *high* smoke levels. Fig. 1 shows the rules for the second device. To verify the collective behavior of the two devices, establishing a standardized and accurate method for queries becomes important. However, addressing the inconsistency in the accuracy of the devices poses a challenge. For instance, how do we handle queries concerning *mid* levels of smoke when the second device lacks a conception of such a level? One straightforward solution involves encoding into our model that *high* levels of smoke for the second device should encompass both *mid* and *high* levels. However, implementing this requires a method that seamlessly incorporates such information when integrating inconsistent data from multiple devices.

2.3 Example 3: Reachability analysis involving inaccessible nodes

While the previous two examples show the potential for bugs in control logic of individual IoT devices, there are further issues that emerge in IoT deployments. Consider a healthcare system that monitors the vitals of all patients in critical conditions, as shown in Fig. 2. Information about the patients is regularly collected by monitoring devices (devices `A`, `B`, and `C`) and then sent to the main server in the hospital, which uses the data to decide if a patient needs attention. The monitoring devices are connected to each other via a reliable wireless connection, forming a mesh network. Relevant doctors are alarmed automatically via the main server if a patient needs urgent attention. Information about the patient is also accessible from a computer inside the room where the patient resides. For privacy, these computers (laptops P, Q, and R) only have access to the local patient's data, and do not have access to the data of other patients. Since this is sensitive data, it should only be allowed to traverse designated nodes. Moreover, since the data is critical to patient's health, the network should be resilient to failures. Thus, the state of the network is regularly verified by a centralized verifier (not shown in the figure) by collecting the data plane state of the network. Consider a scenario where a new bed is added to the hospital. Alongside it, a new monitoring device, D, and a computer, S, are added to the same room. With the new devices added, changes are made to the network. The old path to reach the main server is shown as solid arrows in Fig. 2, while the new path is shown with dotted arrows. The verifier now runs on the new state to make sure that sensitive data never reaches unintended locations and there are no loops or blackholes in the network. However, due to poor wireless signal, the verifier is unable to get the dataplane state of device A. How can the verifier then check for network invariants? If we assume the node has failed, we can proceed to verify the remainder of the network. However, what if the node has not failed but is merely inaccessible to us? This situation could result in sensitive traffic being forwarded to unintended nodes or even cause certain paths to become unreachable from the central server. In the worst-case scenario, we have to assume that the node may forward traffic to any of its connected nodes. Considering the frequent inaccessibility of IoT devices due to issues such as wireless connectivity problems, resource constraints, and battery-saving mechanisms, we need a reasoning method that remains effective even when there is missing information.

3 Pyotr: Modeling with conditional tables

In this section, we introduce conditional tables [22] and illustrate how they can be employed to model various problems in IoT deployments. We use examples from the preceding section to demonstrate their application.

3.1 Conditional Tables

Conditional tables [3,22] (c-tables for short) provide a strong representation system for incomplete information in relational databases. They support standard operations in relational algebra including projection, selection, union, join, and renaming. They were proposed to process data when some of it is missing, imprecise, or has inconsistencies. C-tables generalize relations by allowing values in relations to be unknown. Each unknown value is represented by a variable, which we call a conditional variable (c-variable for short). Valuation of c-variables (i.e., assigning each c-variable to a concrete constant) in a c-table results in a single instance of a relation. Thus, each c-table represents a set of possible instances that the relation can take. In addition to c-variables, each tuple t in the relation is associated with a *local condition*[3] $\phi(t)$, which is a mathematical formula that defines the set of possible values that the c-variable in that tuple can take. For a given valuation v of c-variables, a tuple appears in the result only if $v(\phi(t))$ is satisfiable.

[3] C-tables also include global conditions Φ_T over c-variables, which are just local conditions applied to every tuple in the relation T.

3.2 Querying over conditional tables:

Since c-tables support standard operations in relational algebra, they can be queried using most relational database query languages. Numerous efforts in the past have focused on enhancing the user-friendliness of database queries. Visual Querying Systems [25, 33, 37], for instance, have been suggested to enable the construction of queries through visual interfaces, proving to be effective tools for querying and analyzing databases. [7]. Techniques such as employing natural language to form database queries [28, 39, 41], query completion [18], and automatic query recommendation [12] have also been proposed to allow non-technical users to easily use databases. These approaches could potentially be employed to establish a user-friendly interface for conditional tables. As a preliminary step, Pyotr uses Datalog, a rule-based conjunctive query language, as the reasoning language for Pyotr. Datalog has found use in many applications such as declarative networking [2, 29], program analysis [20, 26, 35], network verification [15, 30], and big data analysis [34, 40]. Its adoption is attributed to its clear and simple syntax, offering a declarative abstraction for querying relational structures. Below, we provide a formal description [3, 27] of using a rule-based conjunctive query over incomplete databases:

Definition 1.1: Let $\mathbf{R}$ be a database schema. A rule-based conjunctive query over $\mathbf{R}$ is an expression of the form

$$H(u) :\text{-}\quad B_1(u_1), ..., B_n(u_n) \tag{1}$$

where $n >= 0$, B_i's are relation (predicate) names in $\mathbf{R}$, and H is a relation not in $\mathbf{R}$; and $\mathtt{u}$ and $\mathtt{u_i}$'s are free tuples that can either use constants in the attribute domain of $\mathbf{R}$ (denoted by **dom**) or variables. Let $\mathbf{dom}^C$ be the domain of c-variables, which contains both the constants and the c-variables in the attribute domain of $\mathbf{R}$. We use var(q) to denote all variables that appear in the query q. The subexpression $B_1(u_1), ..., B_n(u_n)$ is the body of the rule, and $H(u)$ is the head. A rule generates new facts by valuation of variables — a function (denoted by $\mathtt{v}$) from $\mathtt{var(q)}$ to $\mathbf{dom}^C$, whereby constants map to themselves or other c-variables. If one can find values that hold for the body, then one can derive the head. For conjunctive queries, the head is generated by performing a table join of all the tables in the body. Variable valuation finds assignments for the variables in a rule, whereby variables can map to both constants and c-variables. The meaning of a query over c-tables can be defined by *partial* variable valuations. Let $\mathtt{q}$ be a conjunctive query given by the foregoing rules, and let $\mathbf{I}$ be a database instance (i.e., c-table) of $\mathbf{R}$, the query result of $\mathbf{I}$ under $\mathtt{q}$ is:

$$\begin{aligned} q(\mathbf{I}) = \{v(u) | v \text{ is a } \textit{partial} \text{ valuation and} \\ v(u_i) \in \mathbf{I} \text{ for each } i \in [1, n]\}. \end{aligned} \tag{2}$$

The *partial* valuation function for c-tables is governed by the following rules:

1. Variables are assigned to constants or c-variables.
2. A constant $\mathtt{c}$ is assigned to either itself, or to a c-variable $\hat{x}$ if the constrain $\hat{\mathtt{x}} = \mathtt{c}$ does not contradict $\hat{x}$'s *local condition.*

Time	Smoke	Light	Alarm	Sprink	Condition
$\hat{t}$	$\hat{s}$	0	0	0	$\hat{s} = \texttt{LOW}$
$\hat{t}$	$\hat{s}$	1	1	0	$\hat{s} = \texttt{MID}\&\hat{t} > 20:00$
$\hat{t}$	$\hat{s}$	1	1	1	$\hat{s} = \texttt{HIGH}\&\hat{t} > 20:00$
$\hat{t}$	$\hat{s}$	0	1	1	$\hat{s} = \texttt{HIGH}\&\hat{t} <= 20:00$
$\hat{t}$	$\hat{s}$	0	0	0	DEFAULT

(a) Smoke Sensor 1

Time	Smoke	Light	Alarm	Sprink	Condition
$\hat{t}$	$\hat{s}$	0	0	0	$\hat{s} = \texttt{LOW}$
$\hat{t}$	$\hat{s}$	1	1	1	$\hat{s} = \texttt{HIGH}\&\hat{t} > 20:00$
$\hat{t}$	$\hat{s}$	0	1	1	$\hat{s} = \texttt{HIGH}\&\hat{t} <= 20:00$
$\hat{t}$	$\hat{s}$	0	0	0	DEFAULT

(b) Smoke Sensor 2

Alarm	Sound	Condition
0	0	
1	1	

(c) Alarm

Sprink	Water	Condition
0	0	
1	1	

(d) Sprinkler

Table 1: C-tables representing the control logic of sensors in fire suppression system

The result of a *partial* valuation is another c-table in which the relevant *local conditions* incorporate the restrictions imposed by the *partial* valuation function. Specifically, in the context of a table join, the *local condition* for each tuple in the resulting table is a conjunction of the joining conditions and the *local conditions* of all the corresponding joined tuples.

3.3 Models

In this section, we show how problems listed in Section 2 can be modeled and solved using conditional tables.

Example 1: Configuration errors Pyotr expresses rules of IoT devices as conditional tables. Most IoT devices use event condition-action (ECA) rules to configure the control logic. These rules consist of trigger-action pairs, often composed as If-This-Then-That (IFTTT) [21, 36] styled rules. Pyotr translates these rules into conditional tables. The triggers and actions become columns in a conditional table, and *local conditions* are used to represent sets of values for each trigger-action pair. Table 1a and 1b show the translation of the program for the smoke sensors in Fig. 1 as c-tables. The first two columns, *Time* and *Smoke* represent the triggers of the device. These triggers are specified after "IF" in IFTTT-styled rules. The next three columns, *Light*, *Alarm*, and *Sprink(le)* represent the actions of the device. A value of 0 indicates *off*, while a value of 1 indicates *on*. The condition column captures the *local conditions* of rules for both the triggers and the actions. The *DEFAULT* condition in the last row represents the negation of all previous conditions. This can be thought of as an "else" part, which applies when none of the other triggers apply. In this example, the light is linked to the smoke sensor and is directly governed by it. In contrast, the water sprinklers and alarm function as distinct devices with their individual control logic, stored as c-tables. Table 1c shows the logic of the alarm system: the trigger "alarm" from the smoke sensors control the speakers. The c-table for the sprinkler (Table 1d) is similar.

```
V(t,s,l,a,sp) :- Smoke(t,s,l,a,sp)[s >= mid], Alarm(a,0)
```

Listing 1: Datalog query to verify that alarm always sounds when the smoke levels are at or above *mid*

To verify intentions over this system, the database can be queried. For example, the query "Does the alarm always sound whenever there is any noticeable

Time	Smoke	Light	Alarm	Sprink	Condition
$\hat{t}$	$\hat{s}$	0	0	0	$\hat{s} = \texttt{LOW}$
$\hat{t}$	$\hat{s}$	1	1	0	$\hat{s} = \texttt{MID}\&\hat{t} > 20:00$
$\hat{t}$	$\hat{s}$	1	1	1	$\hat{s} = \texttt{HIGH}\&\hat{t} > 20:00$
$\hat{t}$	$\hat{s}$	0	1	1	$\hat{s} = \texttt{HIGH}\&\hat{t} <= 20:00$
$\hat{t}$	$\hat{s}$	0	0	0	$\hat{s} = \texttt{LOW}$
$\hat{t}$	$\hat{s}$	1	1	1	$\hat{s} = \texttt{HIGH}\&\hat{t} > 20:00$
$\hat{t}$	$\hat{s}$	0	1	1	$\hat{s} = \texttt{HIGH}\&\hat{t} <= 20:00$
$\hat{t}$	$\hat{s}$	0	0	0	DEFAULT

(a) Incorrect Data Integration

Time	Smoke	Light	Alarm	Sprink	Condition
$\hat{t}$	$\hat{s}$	0	0	0	$\hat{s} = \texttt{LOW}$
$\hat{t}$	$\hat{s}$	1	1	1	$\hat{s} = \texttt{HIGH}\&\hat{t} > 20:00$
$\hat{t}$	$\hat{s}$	0	1	1	$\hat{s} = \texttt{HIGH}\&\hat{t} <= 20:00$
$\hat{t}$	$\hat{s}$	1	1	1	$(\hat{s} = \texttt{HIGH} \vee s = \texttt{MID})$ $\&\hat{t} > 20:00$
$\hat{t}$	$\hat{s}$	0	1	1	$(\hat{s} = \texttt{HIGH} \vee s = \texttt{MID})$ $\&\hat{t} <= 20:00$
$\hat{t}$	$\hat{s}$	0	0	0	DEFAULT

(b) Correct Data Integration

Table 2: C-tables after integrating the two smoke sensors in the fire suppression system

smoke?" can be translated into the query as shown in Listing 1. The head of this query, V represents a violation of this intention. The query asks whether we can find tuples such that even when the value of the column smoke in the table *Smoke* is higher than *mid* (e.g. either *mid* or *high*) the alarm does not produce a sound (has value $= 0$, as shown as the last attribute of the *Alarm* relation). Additionally, we can extract the tuples that are responsible for this violation. The table violation will contain the last tuple of the smoke sensor, with the *DEFAULT* condition that (after simplification) translates to $\hat{s} = \texttt{MID}\&\hat{t} <= 20:00$. This tells the user exactly when (for what triggers) the sensor behaves unexpectedly: when the smoke is at level *mid* during day time. In this way, conditional tables allow users to verify expected device behaviors through simple SQL queries, making it straightforward to identify rule violations and unexpected responses in the system.

Example 2: Integrating inconsistent sensor data In this example, remember that there are two smoke sensors with differences in the granularity of smoke levels. This is an example of inconsistency in data. C-tables were originally proposed to deal with interoperability in heterogeneous data sources with inconsistencies by performing data integration. To analyze the behaviour of the system with the two sensing devices, a naive method would be to represent the combined behaviour as a single conditional table, as shown in Table 2a. However, this is incorrect, as it does not capture the incompatibilities in the accuracy of the two devices. Running the verification query from Listing 1 on this incorrect table would lead to the same output as before: a violation of the alarm policy, which is misleading. The correct way to integrate the data is to capture the fact that a smoke value of *high* for sensor 2 represents a value of either *high* or *mid* in terms of the accuracy of sensor 1. Such incompatibilities are easily fixed using conditional tables, since they can naturally represent incomplete information. A correct integration of control rules is given in Table 2b. The rules from second device denoting $\hat{s} = \texttt{HIGH}$ are replaced with the condition $\hat{s} = \texttt{HIGH} \vee \hat{s} = \texttt{MID}$, correctly taking care of the incompatibility. The query in Listing 1 does not show any violation, since sensor 2 is correctly programmed and it masks the bug in sensor 1. Conditional tables thus enable accurate integration of inconsistent sensor data, allowing users to capture device-specific variations and ensuring that verification queries reflect correct system behavior

Example 3: Reachability analysis involving inaccessible nodes The final example demonstrates the ease of modeling uncertainty with conditional tables. Pyotr can reason about devices even when one or more of them are inac-

Node	Destination	Source	Output	condition
A	main	$\hat{i_A}$	$\hat{o_A}$	$\hat{o_A} \in [B, D, P]$
B	main	A	D	
B	main	B	$\hat{o_B}$	$\hat{o_B} \in [Q, D]$
D	main	$\hat{i_D}$	C	$\hat{i_D} \in [A, B]$
D	main	D	$\hat{o_D}$	$\hat{o_D} \in [C, S]$
C	main	$\hat{i_C}$	main	$\hat{i_C} \in [A, B, D]$
C	main	C	$\hat{o_C}$	$\hat{o_C} \in [main, R]$

(a) Forwarding rules F

Node	Destination	Source	Path	condition
A	main	$\hat{i_A}$	$[\hat{o_A}]$	$\hat{o_A} \in [B, D, P], \hat{i_A} = A$
B	main	A	$[\hat{o_A}, D]$	$\hat{o_A} \in [B, D, P], \hat{i_A} = A, \hat{o_A} = B$
D	main	$\hat{i_D}$	$[\hat{o_A}, C]$	$\hat{o_A} \in [B, D, P], \hat{i_A} = A, \hat{i_A} = \hat{i_D}, \hat{o_A} = D$
C	main	$\hat{i_C}$	$[\hat{o_A}, C, main]$	$\hat{i_C} \in [A, B, D], \hat{o_A} = D, \hat{i_C} = A$
D	main	$\hat{i_D}$	$[\hat{o_A}, D, C]$	$\hat{i_D} \in [A, B], \hat{o_A} = B, \hat{i_D} = A, \hat{i_A} = \hat{i_D}$
C	main	$\hat{i_C}$	$[\hat{o_A}, D, C, main]$	$\hat{i_C} \in [A, B, D], \hat{o_A} = B, \hat{i_D} = A, \hat{i_C} = A$

(b) Partial results of R

Table 3: C-tables for reachability analysis.

cessible. The forwarding table of the example from Section 2.3 can be encoded as a c-table as shown in Table 3a. For instance, the third tuple conveys that when *Node* B receives a packet with *Destination* main and *source* B, it sends the packet to the *Output* node D and Q. It's noteworthy that despite being unable to access the forwarding state of node A, we can still incorporate any known information. For example, we know that the only devices in the range of (connected to) A are B, D, P. This is encoded in as conditions in the first tuple of the forwarding table.

```
R(n, "main", A, [o]) :- F(n, "main", A, o)
R(n, "main", A, p || [o2]) :- R(n, "main", A, p)[o2 ∉ p], F(p[-1],
    "main", A, o2)
```

Listing 2: Datalog query to perform reachability analysis

A reachability analysis of the network can be done by using the query shown in Listing 2[4]. The forwarding table is stored in relation F. The datalog program calculates paths from all nodes to the destination main, and stores them in table R. Partial results (Table 3b) reveal that for packets with source A and destination main, two possible paths exist: (i) A−>D−>C−>main (fourth tuple) and (ii) A−>B−>D−>C−>main (sixth tuple). From this result, we can see that these packets do not reach any other private computer. With c-tables, we were able to perform this analysis even when node A was inaccessible, allowing early verification of the network.

4 Pyotr: Architecture and Implementation

In this section, we outline the architecture and implementation of Pyotr, a system developed for the storage, management, manipulation, and querying of conditional tables, specifically tailored for the verification of IoT deployments. Pyotr is engineered to efficiently handle queries over incomplete databases while maintaining the semantics of database query languages that are grounded in relational algebra.

4.1 Architecture and Workflow

Fig. 3 shows the architecture and workflow of Pyotr. By utilizing a Database Management System (DBMS) for the storage and querying of tables, Pyotr capitalizes on the extensive advantages provided by these systems in the administration of databases, including storage on file systems, managing distributed computing, ensuring security, and performing query planning and optimization. Pyotr can profit from the efficiency and robustness of widely-used and highly

[4] The operator || is used to concatenate two lists. The attribute p[−1] represents the last value of the list p.

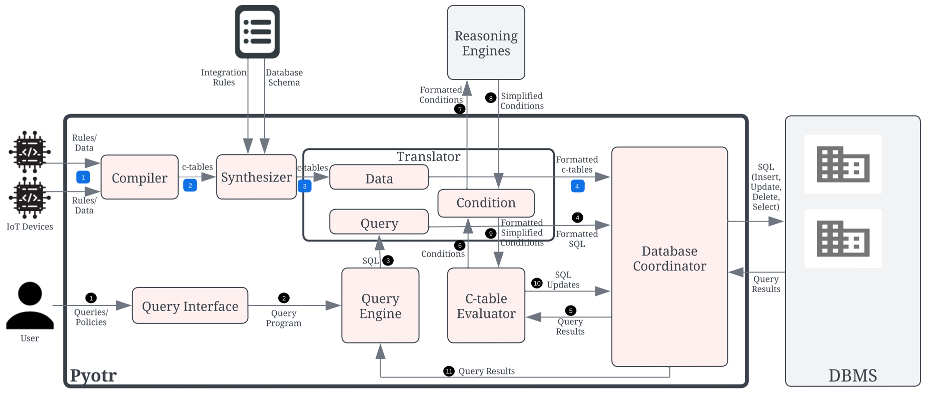


Fig. 3: Pyotr architecture and workflow

optimized DBMSes that already offer support for a diverse range of systems. Pyotr interacts with the DBMS through a *Database Coordinator*, which establishes and maintains a connection to the database. The *Database Coordinator* facilitates communication with the DBMS by using SQL queries to manage and retrieve data. Given that most popular DBMS systems support SQL queries, Pyotr is designed to be agnostic to the underlying DBMS. This flexibility allows Pyotr to seamlessly integrate with various DBMSes, providing versatility and adaptability across different environments.

Fig. 3 shows the workflow for both storing and querying conditional tables[5]. The blue squares mark the steps for storing rules and data obtained from IoT devices. The black circles mark the steps for querying conditional tables.

Workflow for storing c-tables 1, The rules (e.g., ECA rules, forwarding rules) or data (e.g., collected measurements from sensors) from each IoT device is sent to the *Compiler*, which converts them into c-tables. At this stage, there is a single c-table per device. 2, The c-tables are sent to the *Synthesizer*. The *Synthesizer* uses specified integration rules to integrate the c-tables. This is where inconsistencies between the devices are resolved. Additionally, there is an optional grouping of c-tables based on the provided database schema. This involves storing same attributes from different devices in a single table to simplify queries. For instance, data from two distinct temperature sensors can be grouped into a single c-table. 3, The *Synthesizer* forwards the integrated c-tables to the *Translator*. In this phase, the translator formats the tables according to the representation of c-variables in use. This step is essential because DBMS lacks the conception of c-variables. The details of various c-variable representations are explored in Section 4.2. 4, The *Database Coordinator* stores the formatted c-tables into the DBMS using SQL queries for data insertion.

Workflow for querying c-tables 1, User-specified queries and policies are transmitted to the *Query Interface*, where they are processed to generate a pro-

[5] For simplicity, we do not show the interaction between the *Database Coordinator* and the DBMS in the workflow steps.

gram in the specific database query language in use (e.g., datalog). 2, The program is sent to the *Query Engine*, where it is converted into SQL queries. These SQL queries use the selection operation to derive new tables (e.g., facts) and often involve table joins. 3, The SQL queries are sent to the *Translator*, where they are formatted based on the c-variable representation in use. 4, The *Database Coordinator* uses the formatted SQL to run the queries on the database. 5, *Database Coordinator* then sends the generated tables to the *C-table Evaluator*. The job of the *C-table Evaluator* is to handle the conditions in the tables. It extracts the conditions from the results and sends them to the *Translator*. 6, The *Translator* formats conditions based on the employed reasoning engine (e.g., Satisfiability Modulo Theories (SMT) solver) and the c-variable representation in use. This flexibility enables Pyotr to employ various reasoning engines for condition evaluation, with the integration of a new engine requiring modifications solely to the *Condition Translator*. 7, The formatted conditions are then transmitted to the active reasoning engine, which evaluates their satisfiability. In the case of satisfiable conditions, certain reasoning engines can simplify the conditions by identifying parts that constitute a tautology or are unsatisfiable, thereby accelerating future computations on that condition. 8, Simplified conditions undergo translation back into the format understandable by the DBMS through the *Translator*. However, this step is bypassed for unsatisfiable conditions and reasoning engines that do not perform simplification. In such cases, the determination of satisfiability is directly forwarded to the *C-table Evaluator* (step 9). 9, In the case of simplified conditions, the *C-table Evaluator* constructs SQL updates to modify the conditions within the database. Conversely, for unsatisfiable conditions, the *C-table Evaluator* generates SQL delete queries to remove the corresponding tuples. To enhance performance, all updates to one table are consolidated into a single batched SQL query. 10, The *Database Coordinator* executes the SQL queries received from the *C-table Evaluator* on the DBMS, thereby updating the tables generated in step 4. 11, The revised tables are then returned to the *Query Engine*. If a fixed point is achieved (i.e., when no new tuples are generated) the computation concludes, and the final results are dispatched to the user (not illustrated in the diagram). Alternatively, if a fixed point is not reached, steps 3-11 are iteratively repeated until convergence.

4.2 Implementation

In this section, we describe the important implementation details of Pyotr. As outlined in Section 3, we adopt datalog as the query language for Pyotr. The practical implementation of conditional tables poses several challenges: (1) effectively storing conditional tables in a database, (2) developing a custom datalog engine capable of executing datalog programs on conditional tables, and (3) incorporating a reasoning engine with the ability to evaluate conditions. While prototypes for conditional tables have been explored in academia [4, 16, 27], to the best of our knowledge, there is currently no stable implementation for conditional tables within any DBMS.

Storing Conditional Tables Pyotr uses PostgreSQL [17] as the underlying DBMS, since it is an open-source system used by millions of users and is amenable to modifications. To store conditions, we added a *condition* column to every new table created. We implemented the column as a list, where each element represents a condition, and the ultimate condition is a conjunction (logical *and*) of all elements. Given that the conjunction of conditions is frequently computed during c-table evaluation, the use of a list facilitates the rapid addition of new conditions. Global conditions are stored as a shared local condition across all tuples in the table. The type of the condition column is contingent upon the reasoning engine in use. For SMT solvers, we stored conditions as a list of text. For other reasoning engines, we used integer references to represent conditions.

A crucial factor influencing Pyotr's performance and the structure of SQL queries is the representation of c-variables in the database. Unlike constants, which can only take on a single value, c-variables can be valuated (i.e., assigned) to multiple constants. This poses a challenge because conventional database management systems (DBMSes) are designed to support normal constants and lack a built-in conception of c-variables. The manner in which c-variables are implemented directly impacts query performance; inefficient representations can lead to longer query execution times. Moreover, the representation of c-variables affects query translation, as the semantics of c-variables must be encoded into the query structure. This is further described in the next section. We experimented with multiple implementations of c-variables:

Text: In this implementation, we made each table column of "text" type. We defined some keywords (e.g., texts that begin with an underscore) to be interpreted as c-variables to distinguish them from regular textual constants. However, this approach meant that we had to use "text" datatype even when the underlying datatype was numeric. We found that querying over numeric columns was faster than "text" columns in PostgreSQL. This is likely to be true in other DBMS too, since integers are generally easier to optimize in databases. It is important to note that, in this solution, the underlying DBMS remained unaware of c-variables, and we managed the semantics of querying over c-variables externally by incorporating user-defined functions into queries for proper translation.

Modified datatypes: To fix the issues with the previous representation scheme, we made modifications to multiple datatypes (e.g. integer, text, and *inet*) in PostgreSQL to incorporate a built-in understanding of c-variables. We introduced defined keywords as c-variables and implemented the logic to handle them within the code for these datatypes in PostgreSQL. The advantage of this approach was the elimination of the need for query translation to handle c-variables, as the logic for their management was integrated into the DBMS itself. Despite our expectation of improved performance, our custom implementation turned out to be slower, especially for larger tables. The key limitation was the inability to support indexing over tables. Indexing in databases organizes information in columns using specialized data structures, leading to accelerated query execution. However, most indexing schemes require defining a partial order on the datatype. As we utilized custom keywords for c-variables, which lack a natural order among them, we were unable to index the tables. Consequently, this limitation resulted in slower query performance.

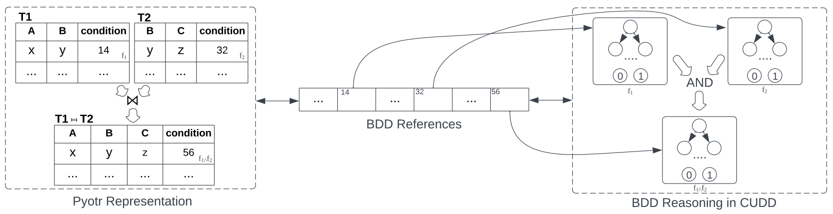


Fig. 4: Pyotr stores references to BDDs that represent conditions

Value Partitioning: In this scheme, we designated specific values (not keywords) within each datatype to represent c-variables. For instance, in the case of integers, we represented c-variables as negative integers. The scheme assumes that the designated values for c-variables never occur in the database as constants. While this approach required the external translation of queries for handling c-variables, it effectively addressed the performance issues associated with indexing and the text datatype. The substantial performance benefits observed led us to adopt this scheme as the default method for Pyotr.

Executing Queries The Datalog Engine is responsible for executing datalog programs on c-tables. To the best of our knowledge, there is no available datalog engine for c-tables. Our datalog engine expects two inputs: (i) a database schema encompassing tables, columns, column types, c-variables used, and the domain of columns, and (ii) a datalog program. A datalog program consists of a list of rules, each of which derives new facts for a particular table. The engine transforms each datalog rule into a selection SQL query, typically incorporating table joins. A join occurs for each atom in the body of the rule. For a table join between two tables, we append the conditions of the joining tables along with the corresponding *local conditions* to the *condition* column of the result. The SQL query is modified so that it can append the relevant conditions to the result. The query also needs to be translated to support c-variables. In the *value partitioning* and *text* schemes, we explicitly add conditions to allow multiple constants to be assigned to c-variables. For example, a simple query to select the tuples in the sensor table (Table 2b) where *Alarm* has a value of 1 can be written in SQL as *SELECT * FROM table WHERE Alarm=1*. For the *value partitioning* scheme in which c-variables are represented as negative integers, this query is converted into the query *SELECT * FROM table WHERE* $(Alarm = 1 \ or \ Alarm < 0)$. A key advantage of employing datalog as the query language is its support for recursive queries. The method of executing these recursive queries significantly influences system performance. A naive execution would continuously execute datalog rules until a fixed point is achieved, leading to substantial redundant generation of tuples. To address this, we implemented semi-naive evaluation [6], which avoids recomputing old tuples by utilizing only the generated tuples from the last iteration in each subsequent iteration. The transition to semi-naive evaluation resulted in a noteworthy improvement in performance.

Reasoning Engine The *C-table Evaluator* removes tuples that have unsatisfiable conditions in the generated table. It utilizes a reasoning engine to evaluate conditions. As we will see in Section 5, reasoning engines can be the most time consuming part of evaluating queries on conditional tables. Thus, the choice of the reasoning engine can determine the performance of Pyotr. The choice should be based on the problem. We tried three different reasoning engines:

Satisfiability Modulo Theories (SMT) solver: Utilizing an SMT solver is a common approach to evaluate the satisfiability of conditions. SMT solvers generalize the Boolean satisfiability problem for more complex conditions, involving diverse datastructures such as integers, lists, bitvectors etc. We used a well-known SMT solver, Z3 [31], for this purpose. In this scheme, we represent all conditions in a text format that can be understood by Z3 and store these conditions in the condition column. The IP addresses were represented as bitvectors.

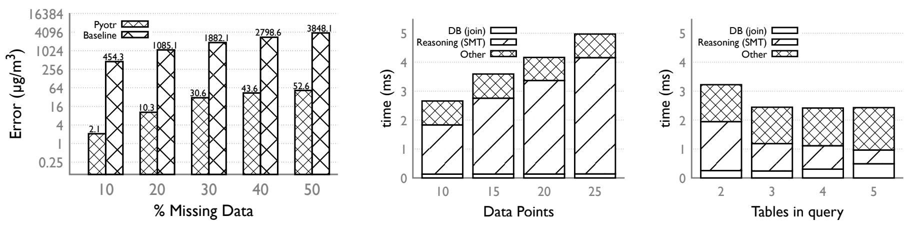
Difference of Cubes (DoC): While Z3 worked well for general conditions, it performed poorly when dealing with IP addresses. For this purpose, we tried a specialized encoding called Difference of Cube (DoC) from [30]. DoC uses ternary strings to represent IP addresses, and can efficiently represent dependencies using negation. For example, `1`$**\backslash$`10`$*$ concisely represents all packets that start with "1" excluding those that begin with "10." DoC supports optimizations for simplifying conditions that involve a large number of dependencies, making it efficient for data plane verification. For large verification tasks involving IP addresses, DoC performed better than SMT.

Binary Decision Diagram[6] (BDD): BDDs [5] serve as representations for boolean functions in the form of decision DAGs. To use BDDs, we translate conditions from Z3 format into boolean functions by employing a binary encoding of integers and IP addresses [9]. The implementation utilizes the CUDD library [23]. We opted for BDDs because of their efficient support for computing conjunctions, which is a frequent operation in the evaluation of conditional tables. The compact representation of boolean functions in BDDs allow for an optimized computation of *logical and*. Integrating BDDs into Pyotr posed a challenge, particularly in finding a suitable method to store them in a table. Since we cannot easily store the actual DAGs representing BDDs in a database, we opted to store only references to the constructed BDDs in the table. The BDDs themselves were stored in an array using a wrapper written in C for the CUDD library, as depicted in Fig. 4. This approach offered an additional advantage: the conditions were now represented as integers rather than lengthy strings (as in SMT implementation), resulting in faster database operations.

5 Evaluation

In this section, we evaluate Pyotr's performance across a range of conditions. First, we assess the benefits and overhead of Pyotr when handling missing information within deployments of heterogeneous devices. Next, we examine Pyotr's efficiency in cases where some nodes are entirely inaccessible. Finally, we showcase the potential of enhancing Pyotr's performance by utilizing different reasoning engines for different problems. We compare three implemented engines,

[6] Referring to Reduced Ordered Binary Decision Diagram [8] throughout paper.



(a) Distance from ground truth (b) Varying size of data (c) Varying queries

Fig. 5: Error and time taken for data analysis with missing data points.

namely SMT, DoC, and BDD. All experiments were conducted using a laptop with an Apple M1 Pro chip and 16GB of memory. For the representation of conditional variables in all experiments, we used the *value partitioning scheme*.

5.1 Verification with partly incomplete information

```
Ans(t, temp, mq4) :- Sensor1(t, temp, humidity, pressure)[temp > 35],
    Sensor2(t, mq4, mq7)[mq4 > 4]
```

Listing 3: Datalog query to check for high temperature and high gas leakage

We use a dataset [24] for environmental monitoring through a sensor network, encompassing eight different types of sensors. These sensors measure diverse environmental properties such as temperature, humidity, rainfall, gas leakages, and pollutants. The measurements are transmitted to a central hub at varying frequencies, resulting in instances where specific data may be absent during particular time periods. In our setup, the hub conducts user-defined checks on the received data at regular intervals and notifies the user if any of these checks fail. To address missing data, a basic method involves assuming that values remain the same as the last observation, which we refer to as the baseline approach. However, since environmental properties can change with time, a more refined method would be to account for potential changes since the last observation. Pyotr employs conditional tables to represent a range[7] around the last observed value for missing data. To assess the benefits of this approach, we compared Pyotr's method to the baseline by removing random data points from the dataset and evaluating the difference from the actual values. Such missing data can arise due to various factors, such as unreliable wireless connections and variations in the transmission frequency among different sensors. Fig. 5a illustrates the Euclidean error for one of the sensors in the dataset that measures inhalable particulate matter (expressed in $\mu g/m^3$). The error is significantly higher in the baseline approach, and grows with the rise in the percentage of missing points, which is attributed to the increased duration since the last observed value. The utilization of a range by Pyotr, as opposed to a single value, contributes to a significantly reduced error. We noticed a similar trend in all sensed properties in the dataset.

To measure the performance of Pyotr on the dataset, we run a query (Listing 3) that alarms when both the temperature and natural gas leakage levels are

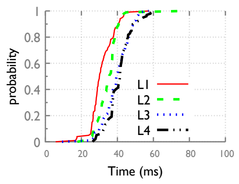
[7] We used the standard deviation of the sensed property as the range.

high. We varied the number of data points analyzed in each iteration, adjusting the frequency of data verification. When the verification frequency is lower, a larger batch of data points is analyzed. Fig. 5b shows the time taken by different components of Pyotr. Even for larger batches, Pyotr runs the query within ten milliseconds. The DB (join) represents the time taken by the selection query that performs the table joins. The reasoning time is the time taken by the SMT solver (Z3) to detect unsatisfiable conditions. Finally, the time categorized as *other* encompasses tasks related to managing conditional tables (e.g., deleting contradictory tuples, parsing datalog queries, etc.). While the database time remains nearly equal for tables of this size, the reasoning time increases due to the growing number of conditions to evaluate. Finally, we assess the impact of varying queries on performance by experimenting with different queries that involve varying numbers of tables. The evaluation time of databases is influenced by the number of tables involved in a conjunctive query, as more tables necessitate additional table joins. Counterintuitively, the reasoning time decreases when more tables are present in a query. This is because the inclusion of tables in a conjunctive query acts as a filter for results, resulting in fewer conditions to evaluate. Fig. 5c shows the time taken when the queries are varied while the number of data points to evaluate are fixed. Notably, while the database time increases, the reasoning time decreases. In general, reasoning time is governed by both the number of conditions to evaluate and the complexity of each condition.

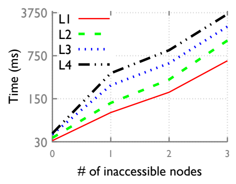
5.2 Verification with inaccessible nodes

Name	Pods	Nodes/Pod	Total Nodes
L1	28	12	496
L2	56	24	1664
L3	84	36	3504
L4	112	48	6016

(a) Details of different data center topologies used[8]



(b) Without inaccessible nodes

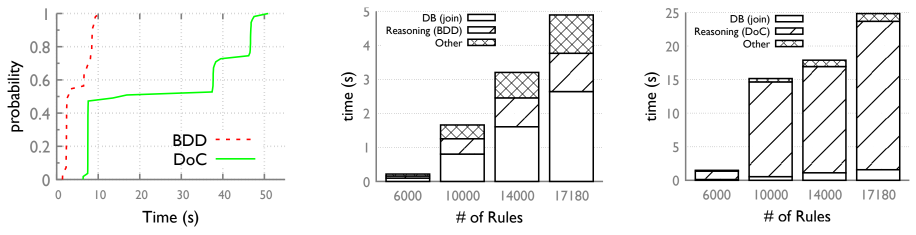


(c) With inaccessible nodes

Fig. 6: Details of datacenter topologies and time taken by Pyotr for reachability analysis on those topologies

In this section, we evaluate the performance of Pyotr when some nodes in a deployment are completely inaccessible. We consider a deployment of server room monitoring sensors within a data center, utilizing the backhaul network for communication. The network is centrally managed by a controller, and a verifier is integrated with the nodes to verify the reachability policies within the network and ensure that the sensed information is promptly collected and analyzed. Following each update, the devices transmit their forwarding state to the verifier. However, delays in sending updates may occur due to various factors, such as network congestion and crashes in the computation of forwarding information bases. A delay in the verification process could potentially result in the

[8] All topologies had 4 spines that connect all pods, which adds extra nodes.



(a) Total running time (b) BDD (c) DoC

Fig. 7: Calculation time for all reachable paths from random nodes in a campus network.

disconnection of sensors due to an erroneous update and could lead to a delay in reporting critical server room conditions on time. With conditional tables, we can perform verification even if there are inaccessible nodes in the network. For these nodes, we use any information we already have to model their forwarding behaviour. For example, in our experiments we assume that they can forward packets through any of their ports except the ingress port[9].

For our experiments, we use datacenter topologies from [19] and vary the number of nodes. The details of the considered topologies are given in Table 6a. We used a datalog program similar to the one shown in Listing 2 to calculate all paths[10] from random source nodes. Fig. 6b shows the time taken to perform reachability analysis on the entire network given a random source node over 100 runs. As the topology size increases, the time taken increases. However, even for the largest topology with more than 6000 nodes, the 90th percentile time is still less than 50ms, which is quick enough to be used in practice. We also did an experiment of an extreme case where we made some nodes completely inaccessible, even though this vastly increases the search space. Fig. 6c shows the average time taken for verification with different number of inaccessible nodes. As we increase the number of inaccessible nodes, the search space increases exponentially. However, our results show that even in large networks with 3 simultaneously inaccessible nodes, the reachability analysis still completes within a few seconds. These results show that Pyotr effectively handles real-time verification even with inaccessible nodes, supporting reliable network monitoring and timely issue detection in large networks.

5.3 Reasoning Engine Performance

In this experiment, we evaluate the performance of different reasoning engines implemented in Pyotr on large conditional tables. Just like the previous experiment, we perform a reachability query to calculate all paths in a network from a randomly selected node. However, instead of a datacenter network, we utilize the backbone of a campus network as our testing setup, leveraging a publicly available dataset [1]. Although this is a smaller network with 16 nodes, it is operated using traditional forwarding where all rules are pre-emptively installed. The network encompasses a total of 17,180 rules, which is much more than the

[9] Most switches do not allow forwarding back to the ingress port to avoid loops.

[10] In practice, we do not always need to compute all paths and can have different queries to catch particular violations (e.g. loops, blackholes, waypointing, firewalls).

number of rules in the previous experiment. Notably, these rules for the network also incorporate packet rewriting, adding an additional layer of complexity to the operational setup. In this experiment, most queries with the SMT engine could not complete. This can be attributed to two main factors: (i) the SMT engine lacks optimization for handling IP addresses, and (ii) string conditions become excessively large during intermediate computations, resulting in a significant increase in database evaluation time. Consequently, we present results solely for the DoC and BDD engines. Fig. 7a illustrates the overall time required to execute reachability queries on the network using random source nodes over 100 runs. The BDD engine outperforms the DoC engine significantly, primarily because BDD is faster than DoC at calculating conjunction of conditions — a crucial operation in the evaluation of conditional tables. The running time is dependent on the length and number of reachable paths. The path length dictates the number of iterations of the query, which further dictates the number of database operations and conditions to evaluate. This is why we observe a staircase-like cumulative distribution function, where each step corresponds to different path lengths.

Figures 7b and 7c provide a breakdown of the overhead caused by different components of Pyotr for BDD and DoC, respectively. It is important to note the difference in the y-axis scales between the two graphs. Total time for database operations and conditional table management is similar for both engines, but the BDD engine's total running time is mainly affected by database join operations, while DoC's is influenced by reasoning time. Varying the number of rules results in an increased running time, as more rules affect the length and quantity of reachable paths, impacting overall performance. These results underscore the significant impact of the choice of reasoning engine on Pyotr's performance.

6 Related Work

Prior efforts in debugging and verifying IoT devices have proposed the use of model-checking [13, 14, 32], program analysis [10], and dynamic testing [11] to catch bugs in control programs for IoT devices. Many of these tools borrow techniques directly from Software Engineering which have proven to work well for software systems in the past. While reusing proved-out techniques for verification seems like a natural way forward, IoT deployments pose some unique challenges that require further innovation. Prior verification efforts for IoT devices overlook device heterogeneity, the user-centric nature of IoT, and the missing information due to proneness to various faults in IoT devices. They target specific programming frameworks (e.g. Groove Programming Language) and do not focus on applicability in a heterogeneous deployment with different vendors and device types. Moreover, static-analysis and model checking tools do not provide a user-friendly environment for inexperienced users to articulate and validate their intentions. Lastly, prior tools assume that all relevant information about the deployment is available to them, which is not the case in IoT devices. The scarcity of comprehensive fault information complicates the understanding of IoT deployments, requiring a more nuanced approach to device behavior modeling and troubleshooting.

Pyotr also shares similarities with some of the tools designed for network verification. Notably, Faure [27] employed conditional tables to verify failures in ISP networks. In our case, we use conditional tables for a different purpose (verification of IoT deployments) and have implemented and evaluated a full-fledged system to facilitate querying over and support for conditional tables. Datalog has also found application in some network verification tools [15, 30]. However, none of these tools address all the listed challenges associated with IoT systems.

7 Conclusion

The growing prevalence of IoT devices in critical environments has underscored the need for methods that guarantee their safety and reliability. In this paper, we identified three challenges associated with the verification of multi-vendor IoT deployments: system and protocol heterogeneity, usage by inexperienced users, and the presence of incomplete information. Conditional tables offer methods for data integration to resolve incompatibilities, provide an easy-to-use database-styled interface for inexperienced users, and offer a natural approach to handling incomplete information. Through various examples, we demonstrated how problems in IoT deployments can be modeled and reasoned about using conditional tables. Finally, we implemented and evaluated a comprehensive system, Pyotr, designed to support verification in IoT deployments through conditional tables. In the future, we intend to integrate conditional tables into a database management system. This integration would minimize the overhead of external system management and enable optimizations like query planning and incremental evaluation over conditional tables. We envision that conditional tables could find application in reasoning and verification across other domains that face one or more of the identified challenges.

Acknowledgments. This work was supported by National Science Foundation Award CNS-1909450, CNS-2145242.

References

1. Stanford benchmark (2023), `https://bitbucket.org/peymank/hassel-public/src/master/hsa-python/examples/stanford/Stanford\_backbone/`
2. Abiteboul, S., Abrams, Z., Haar, S., Milo, T.: Diagnosis of asynchronous discrete event systems: Datalog to the rescue! In: Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. p. 358–367. PODS '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1065167.1065214, https://doi.org/10.1145/1065167.1065214
3. Abiteboul, S., Hull, R., Vianu, V. (eds.): Foundations of Databases: The Logical Level. Pearson, Boston, MA, USA (1995)
4. Abiteboul, S., Kanellakis, P., Grahne, G.: On the representation and querying of sets of possible worlds. ACM SIGMOD Record **16**(3), 34–48 (1987). https://doi.org/10.1145/38714.38724
5. Akers: Binary decision diagrams (1978). https://doi.org/10.1109/TC.1978.1675141

6. Bancilhon, F.: Naive Evaluation of Recursively Defined Relations, pp. 165–178. Springer New York, New York, NY (1986). https://doi.org/10.1007/978-1-4612-4980-1_17, https://doi.org/10.1007/978-1-4612-4980-1_17
7. Bauleo, E., Carnevale, S., Catarci, T., Kimani, S., Leva, M., Mecella, M.: Design, realization and user evaluation of the smartvortex visual query system for accessing data streams in industrial engineering applications. Journal of Visual Languages and Computing **25**(5), 577–601 (2014). https://doi.org/https://doi.org/10.1016/j.jvlc.2014.08.002, https://www.sciencedirect.com/science/article/pii/S1045926X14000652
8. Bryant: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **C-35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
9. Bryant, R.E.: Binary Decision Diagrams, pp. 191–217. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_7
10. Celik, Z.B., Fernandes, E., Pauley, E., Tan, G., McDaniel, P.: Program analysis of commodity iot applications for security and privacy. ACM Computing Surveys **52**(4), 1–30 (2019). https://doi.org/10.1145/3333501
11. Celik, Z.B., Tan, G., McDaniel, P.: Iotguard: Dynamic enforcement of security and safety policy in commodity iot. Proceedings 2019 Network and Distributed System Security Symposium (2019). https://doi.org/10.14722/ndss.2019.23326
12. Chatzopoulou, G., Eirinaki, M., Polyzotis, N.: Query recommendations for interactive database exploration (2009)
13. Ding, W., Hu, H., Cheng, L.: Iotsafe: Enforcing safety and security policy with real iot physical interaction discovery. Proceedings 2021 Network and Distributed System Security Symposium (2021). https://doi.org/10.14722/ndss.2021.24368
14. Fang, Z., Fu, H., Gu, T., Qian, Z., Jaeger, T., Hu, P., Mohapatra, P.: A model checking-based security analysis framework for iot systems. High-Confidence Computing **1**(1), 100004 (2021). https://doi.org/10.1016/j.hcc.2021.100004
15. Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., Millstein, T.: A general approach to network configuration analysis. In: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). pp. 469–483. USENIX Association, Oakland, CA (May 2015), https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel
16. Grahne, G., Onet, A., Tartal, N.: Conditional tables in practice. ArXiv **abs/1304.0959** (2013), https://api.semanticscholar.org/CorpusID:8798537
17. Group, P.G.D.: (Oct 2023), https://www.postgresql.org/
18. Guilly, M.L., Petit, J.M., Scuturici, V.M.: Sql query completion for data exploration (2018)
19. Guo, D., Chen, S., Gao, K., Xiang, Q., Zhang, Y., Yang, Y.R.: Flash: Fast, consistent data plane verification for large-scale network settings. In: Proceedings of the ACM SIGCOMM 2022 Conference. p. 314–335. SIGCOMM '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3544216.3544246, https://doi.org/10.1145/3544216.3544246
20. Hajiyev, E., Verbaere, M., de Moor, O.: codequest: Scalable source code queries with datalog. In: Thomas, D. (ed.) ECOOP 2006 – Object-Oriented Programming. pp. 2–27. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
21. Ifttt: Automate business and home (2023), https://ifttt.com/
22. Imielnski, T., Lipski, W.: Incomplete information in relational databases. In: Mylopolous, J., Brodie, M. (eds.) Readings in Artificial Intelligence and Databases, pp. 342–360. Morgan Kaufmann, San Francisco (CA) (1989). https:

//doi.org/https://doi.org/10.1016/B978-0-934613-53-8.50027-3, https://www.sciencedirect.com/science/article/pii/B9780934613538500273
23. Ivmai: The cudd package, https://github.com/ivmai/cudd
24. J J, J., Elumalai, P., S, O., N R, H., A, M.R.: Lora based wireless sensor network for environmental monitoring - dataset (2021). https://doi.org/10.21227/2g7j-e111, https://dx.doi.org/10.21227/2g7j-e111
25. Jin, C., Bhowmick, S.S., Choi, B., Zhou, S.: Prague: Towards blending practical visual subgraph query formulation and query processing. In: 2012 IEEE 28th International Conference on Data Engineering. pp. 222–233 (2012). https://doi.org/10.1109/ICDE.2012.49
26. Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. p. 1–12. PODS '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1065167.1065169, https://doi.org/10.1145/1065167.1065169
27. Lan, F., Gui, B., Wang, A.: Fauré: A partial approach to network analysis. In: Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks. p. 123–131. HotNets '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3484266.3487391, https://doi.org/10.1145/3484266.3487391
28. Li, J., Hui, B., Qu, G., Yang, J., Li, B., Li, B., Wang, B., Qin, B., Cao, R., Geng, R., Huo, N., Zhou, X., Ma, C., Li, G., Chang, K.C.C., Huang, F., Cheng, R., Li, Y.: Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls (2023)
29. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking. Commun. ACM **52**(11), 87–95 (nov 2009). https://doi.org/10.1145/1592761.1592785, https://doi.org/10.1145/1592761.1592785
30. Lopes, N.P., Bjørner, N., Godefroid, P., Jayaraman, K., Varghese, G.: Checking beliefs in dynamic networks. In: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). pp. 499–512. USENIX Association, Oakland, CA (May 2015), https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes
31. Microsoft: (Oct 2023), https://microsoft.github.io/z3guide/docs/logic/intro/
32. Nguyen, D.T., Song, C., Qian, Z., Krishnamurthy, S.V., Colbert, E.J., McDaniel, P.: Iotsan. Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (2018). https://doi.org/10.1145/3281411.3281440
33. Obaido, G., Ade-Ibijola, A., Vadapalli, H.: Generating sql queries from visual specifications (2019)
34. Seo, J., Guo, S., Lam, M.S.: Socialite: Datalog extensions for efficient social network analysis. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE). pp. 278–289 (2013). https://doi.org/10.1109/ICDE.2013.6544832
35. Smaragdakis, Y., Bravenboer, M.: Using datalog for fast and easy program analysis. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog Reloaded. pp. 245–251. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
36. Soares, D., Dias, J.P., Restivo, A., Ferreira, H.S.: Programming iot-spaces: A user-survey on home automation rules. Computational Science – ICCS 2021 p. 512–525 (2021). https://doi.org/10.1007/978-3-030-77970-2_39

37. Soylu, A., Giese, M., Jimenez-Ruiz, E., Vega-Gorgojo, G., Horrocks, I.: Experiencing optiquevqs: A multi-paradigm and ontology-based visual query system for end users. Univers. Access Inf. Soc. **15**(1), 129–152 (mar 2016). `https://doi.org/10.1007/s10209-015-0404-5`, `https://doi.org/10.1007/s10209-015-0404-5`
38. Vailshery, L.S.: Iot connected devices worldwide 2019-2030 (Jul 2023), `https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/`
39. Wang, B., Shin, R., Liu, X., Polozov, O., Richardson, M.: Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers (2021)
40. Wang, J., Balazinska, M., Halperin, D.: Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. Proc. VLDB Endow. **8**, 1542–1553 (2015), `https://api.semanticscholar.org/CorpusID:7191222`
41. Yu, T., Li, Z., Zhang, Z., Zhang, R., Radev, D.: Typesql: Knowledge-based type-aware neural text-to-sql generation (2018)