Arming IDS Researchers with a Robotic Arm Dataset

Arpan Gujarati, Zainab Saeed Wattoo, Maryam Raiyat Aliabadi, Sean Clark, Xiaoman Liu,
Parisa Shiri, Amee Trivedi, Ruizhe Zhu, Jason Hein, and Margo Seltzer
University of British Columbia (UBC), Vancouver, Canada

***Abstract*—Industry 4.0 is rapidly transforming traditional manufacturing practices. Smart manufacturing technologies that automate research and development using a combination of robotic arms and domain-specific cyber-physical systems are at the core of this transformation. Unfortunately, dependence on networked communication increases the risk of security attacks, which must be mitigated using either platforms that are secure by design or intrusion detection and prevention systems. We report on an ongoing project to design and develop intrusion detection systems (IDS) for the Hein Lab, a smart manufacturing research lab in the chemical sciences domain. Designing effective IDS requires large datasets and high-quality, domain-specific benchmarks, which are difficult to obtain. To address this gap, we present the Robotic Arm Dataset (RAD), which we collected at the Hein Lab over a three-month period. We also present our non-intrusive tracing framework RATracer, which can be retrofitted onto any existing Python-based automation pipeline, and two sets of preliminary analyses based on the command and power data in RAD.**

***Index Terms*—robotic arms, intrusion detection, dataset**

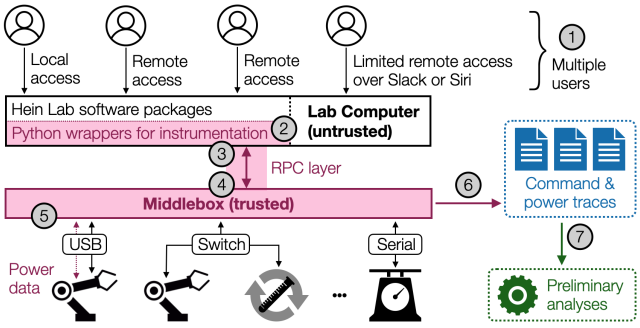


Fig. 1. Overview: ① Users program and automate experiments using the Hein Lab's software packages on the lab computer. ② Python wrappers instrument the software to intercept every device access. ③ An RPC client on the lab computer sends each device command to the middlebox. ④ An RPC server on the middlebox forwards each command to the target device, waits for its response, and sends the response back. ⑤ The middlebox also monitors the power consumption of robotic arms periodically. ⑥ Device commands, responses, and the power data are continuously logged. ⑦ Data mining.

I. Introduction

Industry 4.0, also known as the Industrial Internet of Things (IIoT), is being driven by fully automated smart manufacturing. These facilities need to be secured against attacks over the Internet as well as attacks arising from the use of off-the-shelf software. Defense in depth is crucial to ward off zero-day attacks. For example, while host- and network-based intrusion detection systems (IDS) help minimize unauthorized access, additional safeguards are necessary to prevent automation tools such as robotic arms from damaging expensive property or harming the humans working in the same physical space.

Our work addresses the challenge of designing domain-specific safeguards for a smart manufacturing research laboratory in the chemical sciences domain. Specifically, we are collaborating with researchers at the Hein Lab [15] – a state-of-the-art research lab at the University of British Columbia that blends advanced robotics with synthetic organic chemistry – to secure their cyber-physical systems (CPS) infrastructure, which they use for automating organic chemistry experiments.

The Hein Lab uses a single *lab computer*, accessed locally or remotely, to programmatically control all CPS devices. We introduce a trusted *middlebox* between the lab computer and the CPS devices that need not be connected to the Internet; the middlebox accepts only a restricted set of commands (Fig. 1). Our goal is to use the middlebox as the last level of defense by deploying effective safeguards on it (*e.g.*, alerts, anomaly detection, rule-based IDS, more complex behavioral-based IDS) that understand the "language" in which the software on the lab computer communicates with the automation tools.

However, designing effective safeguards, even for a smaller state space consisting only of commands, requires high-quality domain-specific datasets and benchmarks. Unfortunately, such datasets are limited, and those that are available, *e.g.*, AURSAD by Leporowski et al. [33, 34], do not apply to real-world scenarios such as ours (§II). Robot arm simulators [1, 2] offer a different approach to dataset generation, but real-world deployments are often heterogeneous, consisting of one or more robotic arms together with an assortment of smart devices. It is nontrivial at best, and impossible at worst, to integrate a collection of device-specific simulators to model an environment as complex as the Hein Lab's setup.

This paper: We summarize our experience working on the first phase of our collaboration with the Hein Lab. Our goal is to make available a robotic arm dataset that can support research in multiple areas. To date, we have focused on anomaly-based intrusion detection, which is a particularly promising approach because, **(i)** there do not exist databases of known attacks, and **(ii)** there is insufficient accumulated experience to produce a collection of rules likely to capture all attack scenarios.

First, we present RATracer, a non-intrusive robotic arm tracing framework that can be retrofitted onto any existing Python-based automation pipeline without significant effort from programmers and researchers (§III). RATracer is published with the Python Package Index by the name *niraapad* [19].

Second, we open-source all traces collected by RATracer as the Robotic Arm Dataset (RAD) (§IV) [22]. The dataset includes **(i)** command/response data from multiple robotic arms

engaged in a variety of software-controlled chemical synthesis workflows; **(ii)** data documenting the interaction between these robotic arms and other smart devices used in the workflows; and **(iii)** detailed power data recorded from each of the joints in one of the robotic arms (the Universal Robots UR3e).

Third, we show that command sequences in RAD can be interpreted as a language, allowing us to use natural language processing (NLP) techniques to model experimental procedures (§V) [6]. Our preliminary analyses use this approach to classify different procedure types and identify anomalous procedure runs in a small subset of supervised data within RAD.

Finally, we analyze power data measurements from the UR3e to demonstrate that such side channels offer tremendous potential in identifying both command parameters, such as robot arm velocities, and external contexts, such as payload weights (§VI) [7]. These results are encouraging, because such data can be collected independently of Hein Lab's software infrastructure and without RATracer-like frameworks.

II. Related Work

RAD is not the first open-source robotic arm dataset. Leporowski et al. [33, 34] recently presented a similar time-series dataset based on automated screw-driving operations, carried out using the UR3e and an OnRobot screwdriver. Likewise, Narayanan and Bobba [36] focused on a triangle-shaped laser cutting application using a six-axis Yaskawa Motoman MH5 robotic arm. We are also not the first to examine side-channels, such as power profiles, as important data sources that reveal the workings of a robotic arm. Pu et al. [38], Duman et al. [27], and Khan et al. [32] studied power, acoustic, and electromagnetic signals arising from robotic arms, respectively. Our focus on heterogeneous, real-world, end-to-end workflows sets RAD and this paper apart from prior work.

Wu *et al.* [3, 42, 43] address intrusion detection in smart manufacturing, using an extensive testbed consisting of a 3D printer, CNC milling machine, heating chamber, conveyor, and three robotic arms for moving, welding, and assembly. However, while their testbed is comparable to ours in complexity, each of their case studies focuses on a single device, *e.g.*, evaluating a weakened 3D printing object, a manipulated CNC milling process, or speed attack on the robotic arms.

In theory, simulators can generate RAD-like datasets, as suggested in prior work by Zuo et al. [44] and Vijayan et al. [41]. However, no integrated simulation framework exists today that can provide real-world data on the interactions between heterogeneous devices. Simulators also cannot generate side channel data. For example, we observed significant discrepancies between the power data collected from the UR3e robot and that collected from its simulator [2].

Therefore, to the best of our knowledge, this is the first effort of its kind in understanding the nature of data that originates from real-world automation processes spanning multiple heterogeneous components. Although RAD, RATracer, and the analyses are presented in the context of Hein Lab's chemical synthesis experiments, the ideas, techniques, and inferences generalize to other domains.

III. RATracer

Recall from §I the middlebox-based setup that we deploy in the Hein Lab. We focus on chemical synthesis experiments spanning six different CPS devices: **(i)** four-axis N9 robot arm from North Robotics; **(ii)** six-axis UR3e robot arm from Universal Robots; **(iii)** C-Mag HS 7 magnetic stirrer and heater from IKA; **(iv)** 100-240V, 50/60Hz Fisherbrand Mini-Centrifuge from Fisher Scientific; **(v)** Cavro XLP 6000 syringe pump from Tecan; and **(vi)** Quantos balance for solid dosing from Mettler Toledo [5, 16–18, 24, 25]. Henceforth, we refer to these as N9, UR3e, IKA, Centrifuge, Tecan, and Quantos, respectively. Since both N9 and Centrifuge are controlled via N9's controller box, we treat them as a single device called the C9. Similarly, we consider the Arduino-controlled stepper motor used for z-axis control in Quantos to be a part of Quantos.

Prior to our deploying the middlebox, all the devices in the Hein Lab were connected directly to the lab computer over Ethernet or USB (see Fig. 2). The lab computer hosts the Hein Lab's Python packages, typically one for each device, which expose an intuitive and uniform high-level programming API [12]. This enables lab members to easily program large and complex automation processes spanning heterogeneous devices. Our RATracer framework incorporates three main design features: *interception points* (*i.e.*, where the lab computer and device communication is intercepted and traced), *programmer-friendly tracing*, and *distributed implementation*.

RATracer intercepts all communication at the boundary between low-level third-party software packages (*e.g.*, Python's networking interfaces for TCP [21] and serial communication [20], UR3e's *urx* package [8]) and high-level Python libraries from the Hein Lab [12], which obviates the need to rely on third-party software. Intercepting at boundary points such as *class FtdiDevice* [11], which are used by many devices to interface with the proprietary Windows FTDI driver, also allows for seamless extension to accommodate other similar devices.

For tracing, our key priority is programmer friendliness: We want to make it possible to enable RATracer with minimal modification. Ideally, the main experiment script requires only a single import statement, *e.g.*, *import ratracer.backends*, and allows for optional customization with a few lines of tracing configuration (see below). We achieve this ideal by using Python's support for dynamic class modification (also known as *monkey patching*). Specifically, we *virtualize* each class on the data collection boundary. Higher-layer classes interact with the virtualized classes. Each virtualized class executes logic from the original class implementation and additionally logs all class-level (static) and object-level accesses (including responses from modules) to a MongoDB instance or a *.csv* file. Fig. 3 (top) gives an overview of this approach.

We map the tracing process described above to a distributed system by introducing a middlebox. We implement two different modes using the gRPC remote procedure call framework. In DIRECT mode, the middlebox simply collects trace data; in REMOTE mode, the middlebox both captures trace data and sends commands to the robots. DIRECT mode is useful for

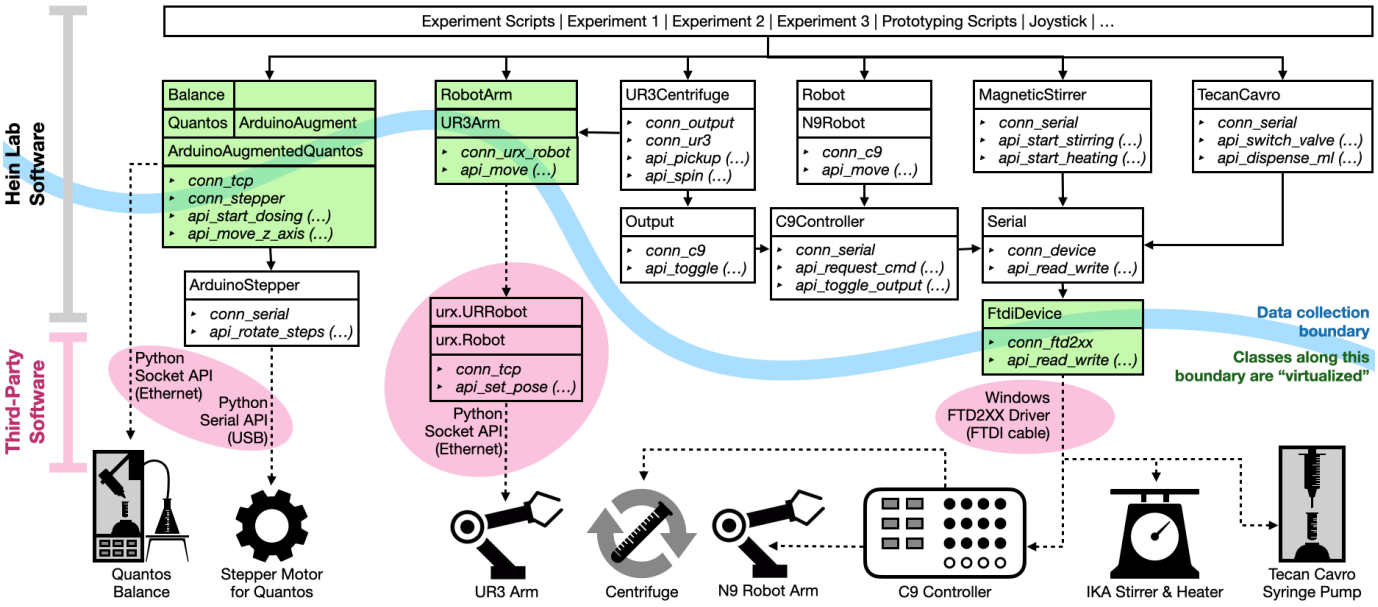


Fig. 2. An overview of the Hein Lab's software architecture [12]. Each box illustrates a Python class, its parent classes (if any), a connection variable, and its main APIs, *e.g.*, *class ArduinoAugmentedQuantos* inherits from *class Quantos(Balance)* and *class ArduinoAugment*, *conn_tcp* denotes a socket connection used to communicate with Quantos, *conn_stepper* denotes an instance of a lower-level *class ArduinoStepper*, and *api_start_dosing(...)* and *api_move_z_axis(...)* denote respective APIs for dosing and motor control [13]. Boxes shaded in green denote classes whose API accesses we trace, *e.g.*, *class FtdiDevice* [11]. Ovals shaded in pink denote third-party software, *e.g.*, Python's Socket [21] and Serial packages [20].

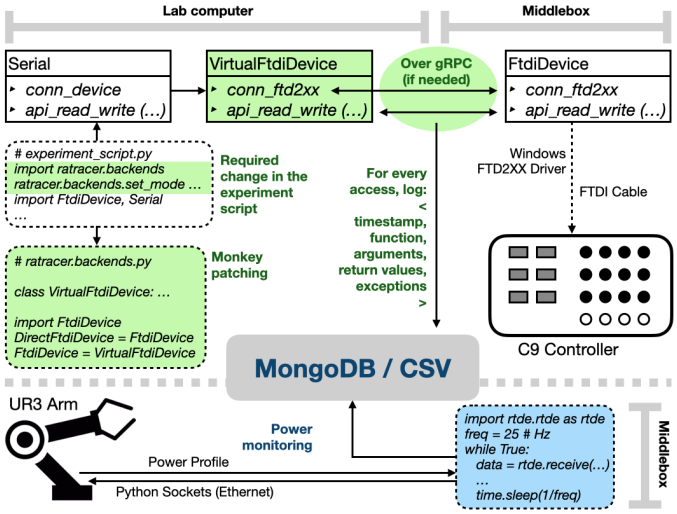


Fig. 3. (Top) Tracing in RATracer using *class FtdiDevice* as an example. (Bottom) Periodic monitoring of power data from the UR3e robot arm.

verifying that RATracer is operating correctly, before allowing it to interpose between the lab computer and the robotic arm. REMOTE mode is useful for actually deploying the IDS in a secure environment. RATracer also allows configuring some devices for DIRECT tracing and some for REMOTE tracing. Such hybrid configurations are convenient in practice, because we can immediately incorporate new devices while their communication issues with the middlebox (*e.g.*, cabling or TCP connection problems) are sorted out by IT.

Finally, we add a simple Python module to RATracer, which collects power monitoring data from the UR3e robot arm at 25 Hz (Fig. 3, bottom). We currently do not collect power data from other devices, since they do not provide similar APIs.

Overall, RATracer is a simple and extensible tracing framework. To trace a new device that relies on a different networking stack, we need to identify a Python class in the stack to virtualize, add a virtual class mimicking the class's method signatures to *backends.py* (this step can be automated), and then add the necessary import statements to the main script. Other languages could be supported using appropriate language bindings. RATracer is versatile, as demonstrated by integrating it with different device-specific libraries (Fig. 2).

RATracer is also non-intrusive; Hein Lab researchers have been using our distributed setup seamlessly for weeks while prototyping new experiments and running fully automated experiments. Nonetheless, we evaluated RATracer's performance by comparing the response times of N9 commands in DIRECT and REMOTE modes, when the commands were invoked via continuous button presses on a joystick.[1] We summarize the results for six different button press sequences in Fig. 4.

[1]RAD also enables end-to-end performance evaluation of network stacks that connect the robot arms with the lab computer. For example, we rented a high-end F16s v2 Windows Server instance on Azure [4] that is identical to our middlebox (16 vCPUs and a 32GB RAM) and replayed the DIRECT mode joystick traces by emulating N9 commands in the cloud server. The average response times ($\sim$60 ms), as shown in Fig. 4, are an order of magnitude higher than DIRECT and REMOTE modes ($<$10 ms), but are still an order of magnitude lower than common robot arm movements (seconds), indicating that cloud deployment is within the realms of feasibility.

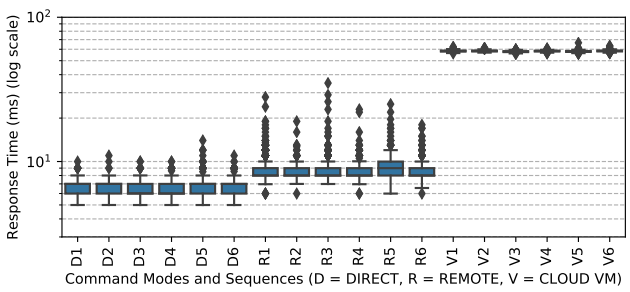


Fig. 4. Response time box plots for N9's ARM command. The y-axis ranges from 3 to 100 ms. Boxes denote the interquartile-range (IQR) between quartiles Q1 and Q3; lower and upper whiskers denote points Q1 - 1.5 IQR and Q3 + 1.5 IQR; and outliers denote response times smaller and greater than the whiskers. Outliers beyond 100 ms are not shown.

The results indicate that REMOTE mode increases average response time by around 2 ms, and occasionally, the latency exceeds 30 ms. As robot arm movements are on the order of seconds, this overhead is negligible, in general. Even for joystick procedures, which are finer grain, the quality-of-service provided by RATracer remained intact.

IV. Robotic Arm Dataset

We present a brief summary of our Robotic Arm Dataset (RAD). Our online documentation contains an exhaustive list of features and their explanations [23].

The dataset is divided into two parts: **(i)** *command dataset*, which contains information curated from the traces collected by RATracer when it intercepted Hein Lab's software stack; and **(ii)** *power dataset*, which refers to the power monitoring data collected by RATracer directly from UR3e.

The command dataset traces the communication between Hein Lab scripts and five automation devices: C9 (N9 and Centrifuge), UR3e, IKA, Tecan, and Quantos. As of this writing, the command dataset includes 128,785 trace objects, where each trace object corresponds to a single command instance, and each command instance corresponds to one of the 52 different command types. Fig. 5(a) illustrates the command-wise distribution of all trace objects.

The dataset was collected over a three-month period during which Hein Lab researchers executed several procedures, including many short scripts for prototyping or for trying out new libraries. We did not supervise all procedures that were run, except for a few that we analyse in §V. Specifically, we supervised a total of 25 procedure runs across four types of procedures. **P1:** Automated Solubility with N9 (5 runs); **P2:** Automated Solubility with N9 and UR3e (4 runs); **P3:** Crystal Solubility (4 runs); and **P4:** Joystick Movements (12 runs) Among these, we mark three as *anomalous*, since they resulted in crashes between a robot arm and another device. The remaining 22 are marked *benign*; these executed either successfully or were stopped by the lab operator (*e.g.*, if the operator put the wrong set of vials next to the robot arm, they would terminate the process on the lab computer). All commands from supervised experiments are labeled accordingly whereas all other commands are labeled "unknown procedure".

Procedures **P1-P3** are a novel set of modular, closed-loop solubility screening techniques proposed by the Hein Lab [9, 10, 40]. In a typical run, the robot arm iteratively increases the amount of solvent in the solid until image analysis determines that the solid has dissolved. Each run varies based on the solids and solvents used, resulting in variations in the observed sequences of commands. P4 corresponds to a user operating the joystick to control the N9 to perform tasks such as lifting a vial, uncapping a vial, and placing the vial in the Quantos (we used these for evaluating RATracer's performance in Fig. 4).

The power dataset contains 122 physical properties that are collected every 40 ms, using the UR3e's real-time monitoring API. As a result, even though our power dataset contains more than 40 million entries, the majority of these correspond to quiescent periods. We store only a small fraction of the entries that belong to quiescent periods (*i.e.*, we store quiescent period entries only on days with some activity).

Each entry in the power dataset measures a set of physical properties, *e.g.*, velocity, acceleration, current, moment, and speed, for each of the UR3e robot's specific joints. The analyses in §VI focus on joint-specific current values that were collected across multiple procedure runs of type **P2**, and two other simple procedures: **P5**, UR3e movements with different velocities, and **P6**, UR3e movements with different payload weights [14].

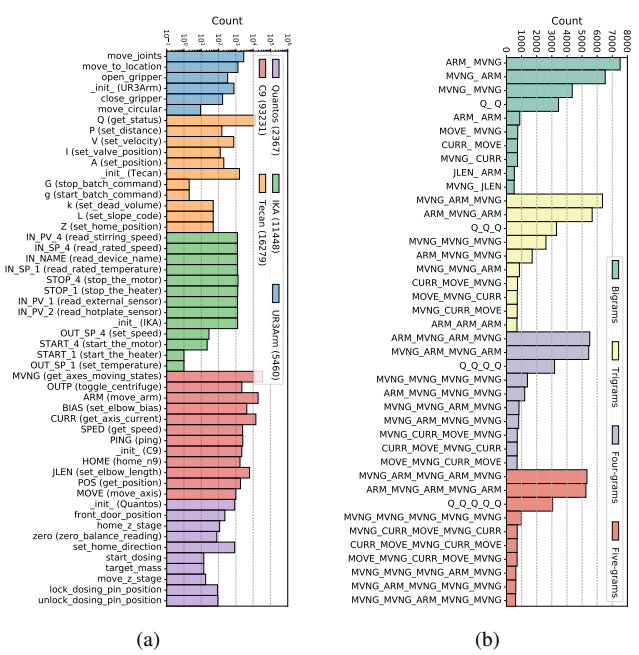
V. Command Dataset Analysis

We collected the command dataset for the purpose of building an intrusion detection system, however, other use cases are also possible, e.g., *program synthesis*, generating a sequence of low-level commands from a high-level specification, and *specification mining*, deriving a high-level program specification from low-level commands. All of these use cases are premised on the assumption that the dataset contains identifiable underlying patterns. To evaluate this assumption, and to gather other useful insights from the dataset, we use data mining and NLP techniques that are easily interpretable, specifically n-gram, TF-IDF, and perplexity analyses. We use these techniques to answer the following questions – **RQ1:** Can we identify Hein Lab's different scientific procedures in the RAD? **RQ2:** Can we identify unexpected variations in procedures in the RAD? – which can guide future development of more sophisticated (*e.g.*, ML-based) IDS. All our code is open source, implemented in Python using *pandas* [35] and *sklearn* [37].

A. RQ1: Identifying Procedures

We begin by considering a simple *n-gram model*[2] to ask if certain sequences of commands repeat more regularly than others. Fig. 5(b) shows the distribution of n-grams for $n \in \{2, 3, 4, 5\}$. The results indicate that, as in natural language, some sequences occur more frequently than others. Additionally, we find that all the runs of a specific procedure have similar n-gram frequencies.

[2]An *n-gram* is a contiguous sequence of n items from a given sample of text or speech [26]. In our case, an n-gram refers to a sequence of n commands.



(a) (b)

Fig. 5. **(a)** Command-wise distribution of trace objects in the command dataset. For non-intuitive command names, we provide their human-readable versions in parentheses. The total number of command instances observed for each device appears in the legend along with the device name. **(b)** Top ten bigrams, trigram, four-grams, and five-grams in RAD, including those with repeated commands. The `Q` command is the Tecan command for `get_status` (a).

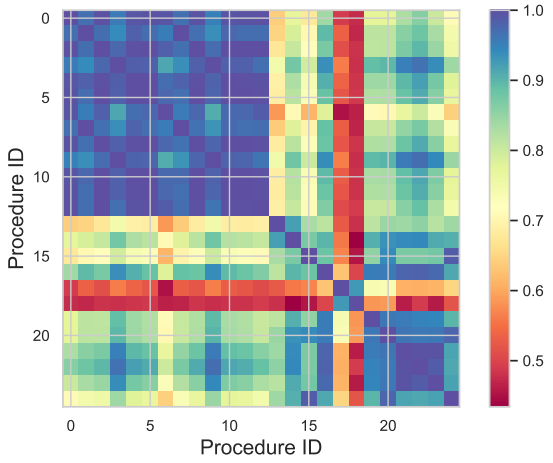


Fig. 6. Pair-wise similarity scores based on TF-IDF for 25 supervised runs of procedures **P1**-**P4**. IDs 0-11 correspond to Joystick Movements (**P4**), 12-16 correspond to Automated Solubility with N9 (**P1**), 17-20 correspond to Automated Solubility with N9 and UR3e (**P2**), and 21-24 correspond to Crystal Solubility (**P3**). A similarity score of 1.0 (dark blue) indicates highest similarity.

Next, we use *term frequency–inverse document frequency* (TF-IDF), to identify unique fingerprints for each procedure. TF-IDF quantifies the importance of a word to a document *relative* to the word's importance in the entire corpus of documents [39]. TF-IDF assigns weights to each command to give more weight to commands that appear more frequently in a procedure than is expected, given their frequency in the entire dataset.

We use the following procedure to compute pairwise similarities between each procedure in our dataset: **(i)** count the number of times each command appears in a procedure run; **(ii)** divide each count by the total number of commands in the procedure run, so that the normalized counts in each procedure run sum to one; **(iii)** use TF-IDF to scale each normalized count to the significance of the command in the procedure; and **(iv)** compute pairwise similarity scores using *cosine similarity* between the TF-IDF vectors generated in (iii) for all procedure runs (the higher the score, the greater the similarity).

Fig. 6 shows the 625 pairwise similarity scores for the 25 procedures. The dark blue region in the upper left quadrant indicates that the Joystick Movement procedures (**P4**) are all quite similar. This is unsurprising, because the joystick API that is part of Hein Lab's software distribution translates each button press into a continuous stream of specific commands that are repeated until the button is released, thereby giving all traces of **P4** a distinct flavor. Interestingly, although procedure 12 is an Automated Solubility with N9 procedure (**P1**), it is more similar to the joystick procedure runs than to other **P1** procedures. The metadata in our dataset shows that procedure 12 used the joystick for a substantial time to move N9 to its starting position, unlike other **P1** procedures. Additionally, procedure 12 stopped midway due to a shortage of solid and executed none of the Quantos and Tecan commands that are part of the automated solubility procedures but not part of the joystick procedures (*e.g.*, `start_dosing`, `target_mass`, `Q`, `V`). The remaining **P1** procedures (*i.e.*, 13-16) exhibit moderately high similarity among themselves (mostly above 0.8). Procedure 16 is an anomaly, because the Quantos front door crashed with the robot. However, before crashing the procedure initiated Quantos commands such as `start_dosing` and `target_mass`, so it is still similar to the other Automated Solubility procedure runs.

Procedures 17-20 correspond to the Automated Solubility with N9 and UR3e experiment (**P2**). Among these, procedures 17 and 18 have low similarity (around 0.58) with procedures 19 and 20 but a high similarity (more than 0.9) with each other. This is particularly interesting, because the metadata shows that run 17 is an anomaly but run 18 is not. We determined that both stopped executing about one-tenth of the way into the experiment, which accounts for their similarity. However, in 17, because Quantos' front door crashed into UR3e, we marked it as an anomaly, whereas in 18, since a lab researcher had mistakenly used the wrong gripper configuration, we did not mark it as an anomaly. Procedure runs 19 and 20 were complete, normal executions.

Procedures 21-24 are Crystal Solubility experiments (**P3**). All runs exhibit high pairwise similarity scores (ranging between 0.9 and 0.99), even though Procedure run 22 is labeled an anomaly. The robot arm crashed with the Tecan at the *end* of the experiment. As the anomaly occurred at the end of the procedure, run 22 is more similar to the other procedures of the same type than the anomalous run for procedure **P2**.

In the preceding analysis, we considered only commands

and not their parameters. Despite this limitation, these TF-IDF-based similarity scores do distinguish one procedure from another (**RQ1**). More importantly, the analysis suggests that the dataset captures sufficient information to sometimes detect anomalies *after the fact*, e.g., the anomalous P2 experiments; the next section investigates a potentially better technique that could be adapted to real time detection.

B. RQ2: Identifying Unexpected Variations

While the analysis in §V-A suggests that anomaly detection is possible, it does not provide a real-time solution. We next demonstrate how we can capture both command frequencies and command orderings in the training dataset to identify unexpected procedures.

Given a training dataset consisting of N_v valid command sequences S_1, S_2, ..., S_{N_v}, and a new command sequence S_{new}, we want to compute the probability with which S_{new} is likely to occur. This probability is a *likelihood function* for S_{new}, which we can use to classify if S_{new} is an anomaly. Suppose that S_{new} consists of commands $c_1, c_2, \ldots, c_{|S_{new}|}$. Using the training dataset, we first compute the n-gram probability $P(c_i \mid c_{i-n+1}, \ldots, c_{i-1}))$ for each $i \in \{n, |S_{new}|\}$, *i.e.*, the probability with which c_i follows $c_{i-n+1}, \ldots, c_{i-1}$ in the training dataset. For example, the bigram probability of command sequence S_{new} is defined as $P(c_2|c_1) \times P(c_3|c_2) \times \ldots \times P(c_{|S_{new}|}|c_{|S_{new}|-1})$. To account for varying procedure lengths, we need to normalize the likelihood function. We compute the *perplexity score*, which is the normalized *inverse* probability of S_{new}; as this is an inverse of the probability, a lower perplexity score suggests a normal or benign trace and a higher perplexity score suggests an anomaly. The perplexity score is defined as $(\prod_{i=1}^{|S_{new}|} 1/P(c_i|c_{i-n+1}, \ldots, c_{i-1}))^{1/|S_{new}|}$.

We use the 25 supervised procedure runs in RAD to generate both training and test data using *5-fold cross validation*. That is, we **(i)** shuffle all 25 procedure runs and divide them into five groups of five; **(ii)** hold one group as a test set and use the other four groups as a training set; and then **(iii)** evaluate the perplexity score of each procedure in the test set. We repeat steps (ii)-(iii) five times using each of the five groups as our test set. Each time, we compute three sets of perplexity scores using bigram, trigram, and four-gram probabilities. Finally, we cluster the computed perplexity scores into two classes, *anomalous* and *benign*, using the *Jenks natural breaks optimization* technique [31]. Table I summarizes various metric scores for evaluating each of the three models.[3]

For anomaly detection, we desire high *recall* (*true positives / (true positives + false negatives)*), because running anomalous procedures could be disastrous. Our recall across all three models is 1.0, indicating that all the models correctly classify the three anomalies. From bigram to trigram, the number of true negatives increases while the number of false positives decreases, resulting in the improvement of *accuracy*, *precision*, and *F1-score* metrics. The performance slightly degrades between trigrams and four-grams, suggesting that selecting an ideal model size is nontrivial.

[3]In Table I, *weighted accuracy* is computed by assigning a higher weight (2×) to the true positive count over the true negatives count.

TABLE I

Metrics	Bigram	Trigram	Four-gram
Accuracy	64%	84%	80%
Weighted accuracy	67.85%	85.71%	82.14%
Precision	0.25	0.43	0.38
F1 score	0.4	0.6	0.54
True positives (negatives)	3 (13)	3 (18)	3 (17)
False positives (negatives)	9 (0)	4 (0)	5 (0)

These experiments show that perplexity scores can be used to classify unexpected procedure variations (**RQ2**). However, our models raise too many false positives. We are guardedly optimistic that a larger and more varied training dataset will let us produce models that retain perfect recall, while reducing the number of false positives.

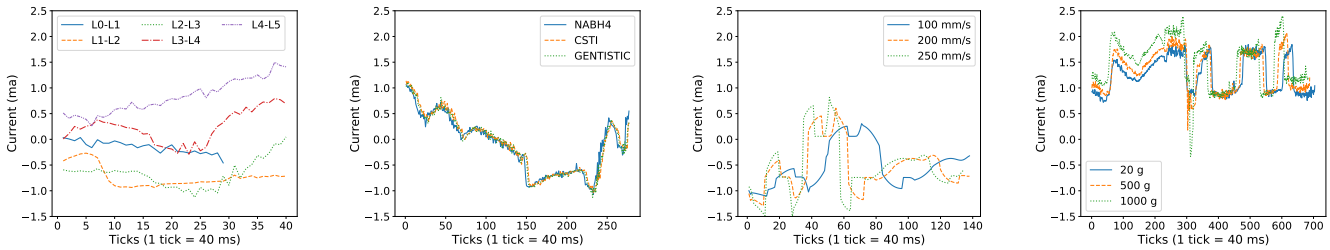
VI. Power Dataset Analysis

The previous section suggests that we can likely build an intrusion detection system by analyzing the communication stream between the lab computer and the devices. However, capturing that data requires a RATracer-like infrastructure and the ability to modify the existing software infrastructure. While we were able to do this for the Hein Lab, deploying such an integrated system might not always be feasible. We collected a power dataset to allow us to answer the question, **RQ3**, "Can we use power monitoring to identify the same kinds of patterns identified via command tracing to facilitate anomaly detection in an unobtrusive fashion?"

Side-channel data such as power consumption can be collected unobtrusively by attaching probes at power outlets or even on the robot body itself. However, in our prototype, we use RATracer and UR3e's real-time monitoring API as a proxy for such probes. While it is possible for us to gather power data from a collection of robotic arms, during the course of our data collection, the UR3e was the only robot arm with a power monitoring API, so our dataset contains joint-specific current profiles for each of the six joints in the UR3e only.

We conducted three sets of controlled experiments using procedure types **P2**, **P5**, and **P6** (see §IV for details).

P2 (Automated Solubility with N9 and UR3e) includes a sequence of 58 commands, a majority of which are UR3e move commands. Fig. 7(a) shows five portions of joint 1's current trace, corresponding to five different instances of the `move_joints()` command. Each instance moves the robot arm from location L_i to L_{i+1}, for $i \in \{0, 1, 2, 3, 4\}$. We observe that the current trace for each command instance is unique and that these unique patterns remain identical across multiple iterations of this experiment. These results suggest that while the command type alone does not correlate with the current profile, the robot arm trajectories identified by the command type *and its arguments* (*e.g.*, location endpoints) correlate strongly with the current profile. We test this hypothesis by repeating the current experiment varying



(a) Different locations (b) Different solids (c) Different velocities (d) Different weights

Fig. 7. UR3e joint 1 current profiles for four different scenarios. **(a)** The five curves show the current profile for five different instances of `move_joint()`, each of which moves the robot arm from location L_i to L_{i+1} for $i \in \{0, 1, 2, 3, 4\}$. Each curve is a subset of the time series of currents from the procedure **P2**. **(b)** The time series of current measurements from a subset of procedure **P2** during with the UR3e picks up a vial from the storage rack, places it in the Quantos, and then returns to the home position. The three curves correspond to selecting three different solids. **(c)** The time series of currents from executions of procedure (**P5**) moving the arm at different velocities. **(d)** The time series of currents from procedure (**P6**) with the arm carrying different weights.

the solid used in procedure **P2**. As shown in Fig. 7(b), the current profiles do not vary as the solid changes (the Pearson correlation coefficient exceeds 0.97), supporting the claim that the variation in the power consumption is due to the specific robot arm trajectories.

Next, we investigate the effect of robot arm velocities on the current usage. We execute a procedure **P5** where the robot arm is moved between two specified locations with varying linear velocities while keeping all other arguments (*e.g.*, angle and positions) constant. Fig. 7(c) illustrates the current traces on joint 1 for velocities 100 mm/s, 200 mm/s, and 250 mm/s. The current traces have similar shapes in each configuration, *i.e.*, same number of peaks, similar slopes and gradients, and the amplitudes are proportional to the velocity. However, the curve for 100 mm/s is "stretched", because at low velocity, the robot arm requires more time to move from one location to another.

Finally, we execute **P6**, where the robot arm moves payloads of different weights from one location to another. Fig. 7(d) illustrates the current traces on joint 1 for weights 20 g, 500 g, and 1000 g. As expected, lifting heavier objects draws more power. Typically, weights are not specified as part of command arguments; they are simply an artifact of the object lifted by the arm. A power-based IDS can detect varying weights in the power profile, while a command-based IDS would need additional information to make such a determination.

While the results shown here are for only one of the six UR3e joints, we observe similar correlations in the current profiles collected from other five joints. This data provides compelling evidence that the power traces do reveal important information about the particular procedure being run and features such as velocity and payload weights, the latter of which is unavailable from other means.

VII. Conclusion and Future Work

Security is of grave concern in Industry 4.0 and IIoT deployments. The risk of security attacks is potentially high, because device controllers and lab workstations are routinely connected to the Internet. In the worst case, if the CPS devices in these deployments are maliciously controlled, they can harm people and/or their surroundings. This is particularly concerning in domains such as chemical sciences, where CPS devices such as robot arms are surrounded by potentially lethal chemicals, and a catastrophic outcome is just one misstep away.

We are collaborating with the Hein Lab, a research lab that studies methods to fully automate chemical synthesis procedures, to design and deploy a multi-level defense system for the CPS devices in their lab. Using RATracer, we showed the feasibility of deploying a middlebox-based design that intercepts all communication between the lab computer and the CPS devices. We open-source our Robotic Arm Dataset (RAD), which is the first of its kind, as it captures automation experiments spanning multiple heterogeneous automation devices. Finally, we presented two sets of preliminary analyses based on the command and power data in RAD to infer procedure types, command parameters, and experimental contexts.

However, we are still a long way from deploying a full-fledged IDS in the Hein Lab. As the number of devices grows from five to fifty (as is expected in the Hein Lab), a single middlebox will not suffice. While a single middlebox can easily scale to tens of devices, we expect space and cabling issues to be a more significant challenge. Expansion will therefore require, potentially, a distributed architecture with multiple middleboxes in smaller form factors.

Modeling robot commands as a language and using NLP techniques for intrusion detection is a new approach, with no precedent in the robotics literature. We therefore need to investigate a wider array of techniques that are best suited for our dataset and objectives. Our immediate goals are to bring command arguments into the fold, find ways to automatically generate labels, and evaluate models such as *long short-term memory* [30] (which have been successfully deployed to model time series data in many domains).

We also plan to conduct a more comprehensive study of side channels by monitoring the power usage of all devices in the lab. Recent results [28, 29] showing how to classify device-specific power usages from the main power usage inside a room are encouraging. Finally, while RAD is novel, we need to generate many more anomalous traces for testing, or for benchmarking other IDS. However, doing so in a manner that does not destroy equipment remains an open question.

ACKNOWLEDGEMENTS

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and the UBC Science STAIR Grant. We also acknowledge the contributions of other Hein Lab members, Veronica Lai, Tara Zepel, Daniel Griffin, Jonathan Reifman, Shad Grunert, Lars P.E. Yunker, Sebastian Steiner, Henry Situ, Fan Yang, and Paloma L. Prieto, to the development of the automated and crystal solubility procedures.

REFERENCES

[1] "Kortex Gazebo," https://github.com/Kinovarobotics/ros_kortex/tree/noetic-devel/kortex_gazebo.
[2] "UR3 Unity Simulation," https://github.com/tonydle/ur3_unity_sim.
[3] "Intrusion detection system for cyber-manufacturing system."
[4] "Fsv2-series," https://docs.microsoft.com/en-us/azure/virtual-machines/fsv2-series.
[5] "Fisher Scientific," https://www.fishersci.com.
[6] "Command Data Analysis," https://github.com/ubc-systopia/dsn-2022-rad-artifact/tree/main/analysis/Dataset_CommandAnalysis.
[7] "Power Data Analysis," https://github.com/ubc-systopia/dsn-2022-rad-artifact/tree/main/analysis/Dataset_PowerAnalysis.
[8] "Python Library to Control the Robots from Universal Robots," https://github.com/SintefManufacturing/python-urx.
[9] "Code for Hein Lab's Automated Solubility Experiment," https://gitlab.com/heingroup/robotic_security_experiments/-/blob/master/n9/experiments/main_solubility.py.
[10] "Code for Hein Lab's Crystal Solubility Experiment," https://gitlab.com/heingroup/robotic_security_experiments/-/blob/master/n9/experiments/main_crystal_solubility_profiling.py.
[11] "Serial Library that uses the FTDI Driver for More Reliable Serial Communications," https://gitlab.com/ada-chem/ftdi_serial.
[12] "Code repositories for the Hein Group at the University of British Columbia," https://gitlab.com/heingroup.
[13] "Quantos Python API Wrapper," https://gitlab.com/heingroup/mtbalance.
[14] "Code for UR3e Movements with Different Payload Weights," https://gitlab.com/heingroup/robotic_security_experiments/-/blob/master/ur/tests/ur_different_weights.py.
[15] "Hein Lab," http://heinlab.com/.
[16] "IKA," https://www.ika.com.
[17] "Mettler Toledo," https://www.mt.com.
[18] "North Robotics," https://www.northrobotics.com.
[19] "A Library that Routes all Communication to the N9 and UR3 Robots via a Secure Middlebox," https://pypi.org/project/niraapad/.
[20] "pySerial's documentation," https://pythonhosted.org/pyserial/.
[21] "`socket` – Low-level networking interface," https://docs.python.org/3/library/socket.html.
[22] "RosyChem Lab Robotic Arm Dataset," https://github.com/ubc-systopia/dsn-2022-rad-artifact.
[23] "Robotic Arm Dataset (RAD) Features Description," https://github.com/ubc-systopia/dsn-2022-rad-artifact/blob/main/docs/RAD_Description.pdf.
[24] "Tecan," https://www.tecan.com.
[25] "Universal Robots," https://www.universal-robots.com.
[26] P. F. Brown, V. J. Della Pietra, P. V. Desouza, J. C. Lai, and R. L. Mercer, "Class-Based n-gram Models of Natural Language," *Computational linguistics*, vol. 18, no. 4, pp. 467–480, 1992.
[27] T. B. Duman, B. Bayram, and G. İnce, "Acoustic Anomaly Detection Using Convolutional autoencoders in Industrial Processes," in *14th International Conference on Soft Computing Models in Industrial and Environmental Applications*, 2019.
[28] J. Froehlich, E. Larson, S. Gupta, G. Cohn, M. Reynolds, and S. Patel, "Disaggregated End-Use Energy Sensing for the Smart Grid," *IEEE Pervasive Computing*, vol. 10, no. 1, pp. 28–39, 2010.
[29] S. Gupta, M. S. Reynolds, and S. N. Patel, "ElectriSense: Single-Point Sensing Using EMI for Electrical Event Detection and Classification in the Home," in *12th ACM International Conference on Ubiquitous Computing*, 2010.
[30] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[31] G. F. Jenks, "The Data Model Concept in Statistical Mapping," *International Yearbook of Cartography*, vol. 7, pp. 186–190, 1967.
[32] H. A. Khan, N. Sehatbakhsh, L. N. Nguyen, R. L. Callan, A. Yeredor, M. Prvulovic, and A. Zajić, "IDEA: Intrusion Detection through Electromagnetic-Signal Analysis for Critical Embedded and Cyber-Physical Systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 3, pp. 1150–1163, 2019.
[33] B. Leporowski, D. Tola, C. Hansen, and A. Iosifidis, "AURSAD: Universal Robot Screwdriving Anomaly Detection Dataset," *arXiv preprint arXiv:2102.01409*, 2021.
[34] ——, "Detecting Faults during Automatic Screwdriving: A Dataset and Use Case of Anomaly Detection for Automatic Screwdriving," in *Towards Sustainable Customization: Bridging Smart Products and Manufacturing Systems*. Springer, 2021, pp. 224–232.
[35] W. McKinney, "Data Structures for Statistical Computing in Python," in *9th Python in Science Conference*, 2010.
[36] V. Narayanan and R. B. Bobba, "Learning Based Anomaly Detection for Industrial Arm Applications ," in *4th ACM Workshop on Cyber-Physical Systems Security and PrivaCy*, 2018.
[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
[38] H. Pu, L. He, C. Zhao, D. K. Yau, P. Cheng, and J. Chen, "Detecting Replay Attacks against Industrial Robots via Power Fingerprinting," in *18th Conference on Embedded Networked Sensor Systems*, 2020.
[39] G. Salton and C. Buckley, "Term-Weighting Approaches in Automatic Text Retrieval," *Information Processing & Management*, vol. 24, no. 5, pp. 513–523, 1988.
[40] P. Shiri, V. Lai, T. Zepel, D. Griffin, J. Reifman, S. Clark, S. Grunert, L. P. Yunker, S. Steiner, H. Situ *et al.*, "Automated Solubility Screening Platform Using Computer Vision," *Iscience*, vol. 24, no. 3, p. 102176, 2021.
[41] A. Vijayan, H. Singanamala, B. Nair, C. Medini, C. Nutakki, and S. Diwakar, "Classification of Robotic Arm Movement using SVM and Naïve Bayes Classifiers," in *3rd IEEE International Conference on Innovative Computing Technology*, 2013.
[42] M. Wu, "Intrusion Detection for Cyber-Physical Attacks in Cyber-Manufacturing System," Ph.D. dissertation, Syracuse University, 2019.
[43] M. Wu and Y. B. Moon, "Alert Correlation for Detecting Cyber-Manufacturing Attacks and Intrusions," *Journal of Computing and Information Science in Engineering*, vol. 20, no. 1, p. 011004, 2020.
[44] Y. Zuo, W. Qiu, L. Xie, F. Zhong, Y. Wang, and A. L. Yuille, "CRAVES: Controlling Robotic Arm with a Vision-based Economic System," in *32nd IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019.