A Modular and Flexible OPC UA Testbed Prototype for Cybersecurity Research

Sebastian Kraust, Peter Heller and Jürgen Mottok
Laboratory for Safe and Secure Systems (LaS³)
OTH Regensburg
93053 Regensburg, Germany
e-mail: {sebastian.kraust | peter2.heller | juergen.mottok}@oth-regensburg.de

***Abstract*—The security assessment of Industrial Control Systems (ICS) is becoming increasingly challenging due to their growing complexity and interconnectivity. Traditional penetration testing is often impractical in live environments due to the risk of operational disruption, making testbeds essential for evaluating security mechanisms, analyzing threats, and developing defense strategies. However, existing testbeds tend to be static and difficult to quickly adapt to a wide variety of scenarios. To address these limitations, we propose a modular and flexible ICS testbed that enables rapid reconfiguration of the testbed composition in order to test a wide variety of scenarios. Our open-source approach leverages containerized applications as building blocks, allowing users to create and modify the testbed with minimal effort. We show how to use the provided components to construct testbeds and how our approach can be used as a tool for accommodating penetration tests.**

Keywords-testbed; OPC UA; cybersecurity; penetration testing.

I. Introduction

The increasing complexity and interconnectivity of Industrial Control Systems (ICS) require extensive security assessments. This is no trivial task considering the rapidly evolving attack surface and the widespread use of devices and protocols without security features. Penetration tests offer a way to actively assess a system's security status but are not practical in live environments due to the risk of damaging equipment and endangering human life. Testbeds address this issue by providing a safe environment for evaluating security mechanisms, analyzing cyber-physical threats, and developing defense strategies. However, existing testbeds are often static in nature, tightly coupled to specific architectures, or require extensive effort to modify, making them unsuitable for adapting them to the specific needs for a given scenario. Most notably, this rigidity limits their usefulness for assessing threats to one's own system, which might be vastly different from a prefabricated testbed.

To address this issue, we introduce a novel ICS testbed prototype designed for maximum modularity and flexibility, enabling rapid restructuring and reconfiguration to accommodate various attack vectors and system configurations. The basic idea is to provide a set of building blocks in the form of containerized applications, from which a wide variety of testbed compositions can be created with minimal manual effort. By making the testbed components freely available, we hope to facilitate the sharing of problematic scenarios, insights, and custom testbed extensions within the research community.

In this paper, we first describe our design process along with the relevant literature in Section II. Then, we present the basic concept and components in Section III, and create and modify a basic testbed. We also show how containers can be swapped to enable security tests with certain vulnerabilities. We summarize the paper and provide an outlook for future work in Section IV.

II. Design Process and Related Work

Prior to introducing the testbed, we provide the relevant context regarding the project environment in which it was developed, and highlight topics and issues we encountered that influenced our design.

The idea for a new type of testbed was developed within the context of a research project focusing on improving the cyber resilience of critical infrastructure systems. Key elements include the exploration of new active and passive cyber security techniques for industrial environments powered by artificial intelligence. We conducted extensive literature research to find a suitable testbed that we could recreate. However, this turned out to be difficult due to lack of information or accessibility. As a result, we decided to develop our own environment to have total control over a system while launching realistic attacks. During this process, we first noticed an evident lack of literature that addresses testbed design in industrial environments. Most publications touch upon it only very briefly and instead cite a real system or a reference model as the inspiration, as shown in [1]. As a result, we decided to document our thought process during the development process.

The first task was to choose a common industrial protocol with known vulnerabilities, working exploits, and readily accessible toolkits and SDKs. Ultimately, OPC Unified Architecture (OPC UA) was selected as the primary protocol for a number of reasons: first, it is well defined in the open standard IEC 62541 [2] and comes with a plethora of security-relevant features, such as encryption, authentication, and certificate management. Second, it supports the creation of complex hierarchical system architectures, which are high-value targets for adversaries due to their highly interconnected monitoring and control devices. Lastly, OPC UA fits well into the context of our research project and has a significant share of usage in industry and research. We also realized that flexibility would play an important role during design, due to the great number of existing implementations of the protocol and the extensive configuration capabilities. Secondary protocols for simulating other legitimate applications in an ICS environment and additional noise are planned to be included in the future, but are not a focal point.

TABLE I. COMPARISON OF TESTBED FEATURES

Feature	MiniCPS [13]	DHALSIM [14]	MOTRA [12]
Focus	ICS network simulation, SDN	Impact analysis of ICS traffic events on physical process simulation	Penetration testing
Network fidelity	high	high	low (focus on OPC UA)
Physical process fidelity	low	high (EPANET, water-only)	low
Deployment	single host (Mininet)	single host co-sim	single/multi host (Docker)
Swap implementation/version	codebase modification	codebase modification	Docker images/tags

OPC UA provides a framework in which complex information can be modeled and accessed with standardized services [3][4]. The most basic building blocks are called *nodes*, which are used to construct more complex structures, such as hierarchical data types and objects. The entirety of all node instances is called the *address space*, which defines a standard way of structuring the nodes within in a tree-like fashion. This achieves a consistent way for servers to present data to clients. Initially, we focused on building prototypes of virtual devices for a custom water treatment testbed with different OPC UA software stacks and versions to enable certain vulnerabilities. For all these variants, we manually created all required nodes within the address space. Although we achieved near-identical behavior, the resulting tree structure was not exactly the same, making quick one-to-one replacements of devices during our penetration tests unfeasible. Furthermore, maintaining the different stack versions written in their respective programming languages required significant amounts of time and effort. Fortunately, OPC UA also allows the address space and custom extensions to be modeled using XML. Through the use of tools like the *OPC UA Model Compiler*, the source code for specific implementations can be generated automatically using a common modeling language. Most available stacks support these extensions; thus we can use a standardized way of building and maintaining devices by using XML-based models and configurations. This simplified the workflow and allowed us to verify a variety of vulnerabilities.

Developing interchangeable implementations was merely the initial phase in establishing a versatile framework. Subsequently, we needed to tackle their deployment. Container technology was the first solution to be considered as it allows packaging software to be developed and deployed independently of the underlying hardware. It can even be deployed on embedded platforms with built-in support for different platforms, such as x86 and ARM64. It also guarantees reproducible results, as it leaves no room for errors regarding software versions, installed tool chains, etc., and simplifies exchanging implementations by replacing containers. Other advantages include implicit versioning through tags, easy software sharing with the research community, and, to a certain extent, the ability to recreate testbeds and verify results independently. In addition, many network simulation tools, such as GNS3, natively support container integration, which facilitates the creation of complex architectures. Finally, the flexibility to replace any container through a real component makes it easy to expand from a virtual to a hybrid setup.

The ability to grow into a hybrid setup turns out to be very relevant, as commercial, proprietary products often use reference implementations as a basis. In 2021, OTORIO [5] released their latest research on OPC UA attack surface, mapping out supply chain dependencies for a number of major manufacturers. Based on the specification available from the standard body (IEC 62541 [2]), there have been different releases of the OPC UA Core Stack for public use. Before this, there have been different stacks (namely: .NET legacy, ANSI C legacy, JAVA legacy) that are not officially supported anymore. As OTORIO has shown, there is a significant relationship between the reference stack implementations and the selected OPC UA SDKs. The foundation reference implementations and core stacks have been partly used to design or build commercial and open source SDKs for products by different OEMs [5]. Due to the ability to include such products in a hybrid setup, we can also evaluate vulnerabilities in these proprietary stacks.

Finally, further aspects that we came across during development are configuration and bootstrapping issues. While the underlying protocol has been verified to be securely designed and audited by several bodies (BSI [6], Kaspersky [7], Claroty [8]), proper configuration, bootstrapping, and personnel training are still major issues [9]–[11]. Furthermore, certain implementations are characterized by incomplete feature sets and potentially confusing documentation. As a result, we started to vary the configurations in addition to the software stack itself.

Considering these issues in conjunction with the aforementioned lack of testbed design literature, we implement the penetration testing-focused methodology for deriving testbeds proposed by Kraust et al. [1] as a proof of concept. It introduces an iterative, protocol-agnostic approach that gradually builds up a complex testbed from individual devices. During each iteration, penetration tests including their respective

goals are defined and executed. The following iterations build on the knowledge gained, which allows the user to create more complex attacks over time. To be able to do this, the testbed must be modular in nature. As a consequence, we analyzed the OPC UA protocol in terms of its features and capabilities, which allowed us to extract the basic building blocks of a testbed centered around this protocol. We then used the so-called *Model Compiler* to translate XML files with the specified nodes of OPC UA devices into actual source code across different implementations. In this way, we were able to create applications with the same interface that use various software stacks underneath. These building blocks are currently available on Github [12]. We will explain the full extent of the available software in the following section. In this paper, we will use these building blocks to actually construct an exemplary water treatment testbed.

To conclude this section, we want to briefly address how this testbed concept distinguishes itself from other approaches in the ICS domain that also emphasize flexibility and reproducibility instead of using a static setup. For this comparison, we have chosen to use MiniCPS [13] and DHALSIM [14]. The former is a toolkit that extends Mininet (a network emulator) to emulate realistic ICS networks, providing a physical-layer API for coupling simulations. It was developed in response to the lack of generic simulation environments for cyber-physical systems (CPS), providing a framework that supports physical interactions and industrial protocols while placing a strong focus on software-defined networking (SDN). DHALSIM combines MiniCPS with the EPANET process simulator to achieve high-fidelity co-simulation of water distribution systems in order to study the impact of network anomalies and faults on the process. Table I summarizes the key features and highlights the differences compared to our approach. Although MiniCPS and DHALSIM can be used for cybersecurity analyses, this was not their primary design objective. Consequently, essential penetration testing features, such as quick and easy reconfiguration, multi-host setups, and swapping protocol implementations and software versions, require more time and effort. MOTRA was developed with these needs in mind, focusing on protocol-level interactions and semantics rather than high network-level fidelity.

III. The Testbed

The goal of this section is to introduce the reader to the overall testbed concept and to highlight how researchers can build a testbed tailored to specific scenarios. We divide our presentation into two parts: first, we introduce the overall concept and the building blocks. Second, we build up an exemplary testbed from scratch and show how the modular approach can be used to modify the system with minimal effort to perform a specific penetration test.

A. Concept and Components

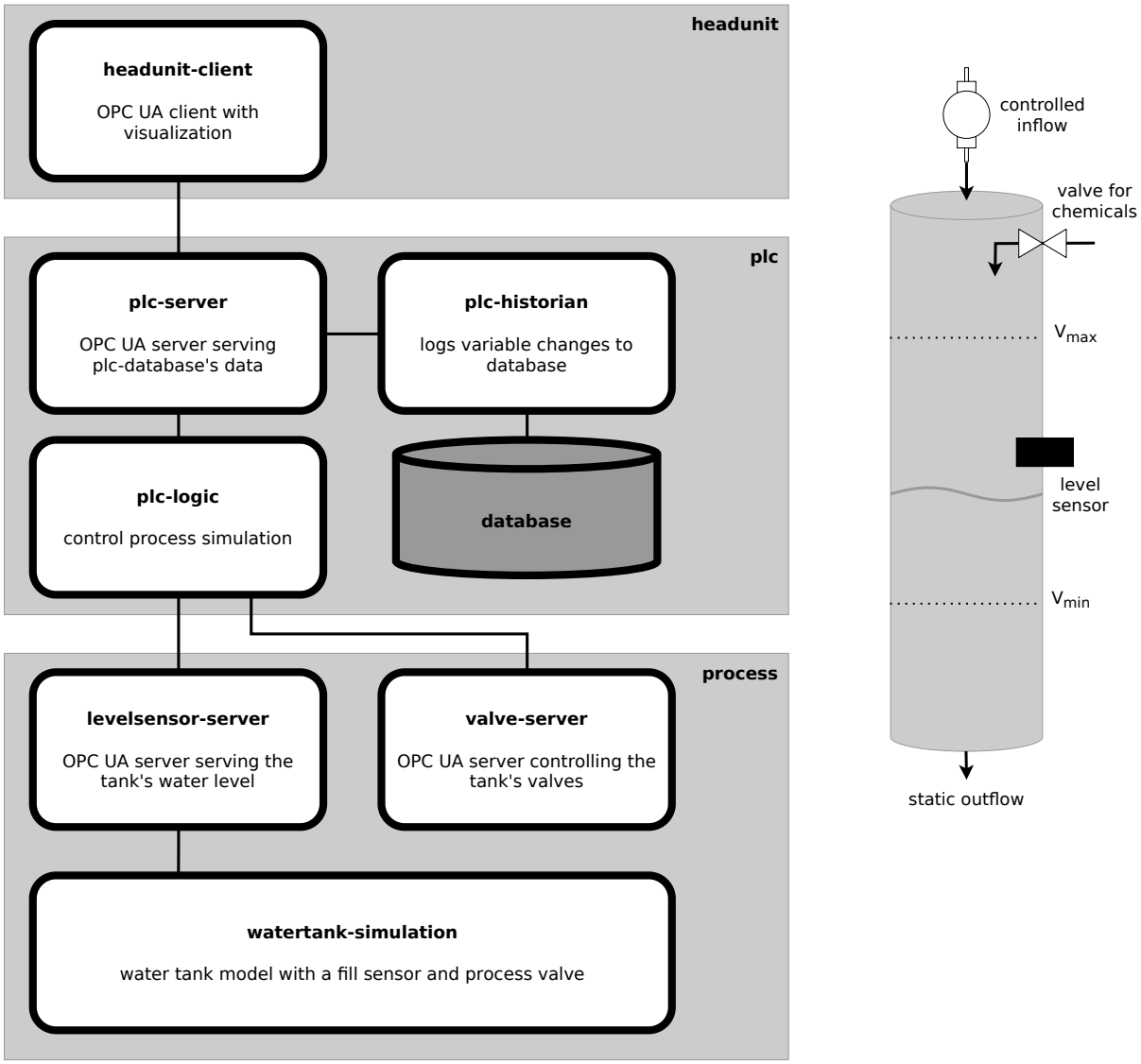
The concept of a flexible testbed through the use of containers allows users to test specific setups and configurations, and enables reproducible experiments across cybersecurity practitioners. This requires careful consideration of how to split testbeds into reusable chunks. During our initial tests, we often found ourselves in the situation of needing another already existing component, such as another sensor or actuator. As a result, packaging software units that perform a certain function was the obvious choice. The exemplary testbed used in this paper is shown in Figure 1a, where these units (hereinafter referred to as *Components*) are shown as white boxes. For the chosen OPC UA protocol, they can be derived in part from the protocol specification, such as discovery or global services.

In addition to the pure functionality, the next most relevant properties for penetration testing considerations are the underlying software stacks and the respective versions. Generally, components (e. g., the valve-server) can be realized using different implementations. Depending on the stack used, specific vulnerabilities exist and can be exploited. Closely related to this is the selection of the exact software version. As developers constantly patch their software to improve security, many real-world systems do not receive timely updates and continue to run with outdated versions. We account for this issue by allowing for the specification of a certain version. As a result, we package components according to these three parameters as separate containers, which is reflected in the suggested naming scheme in the following section. We decided to use Docker as the containerization solution, as it is freely available, well-known, and feature-rich.

Before explaining the usage of the suggested testbed, we provide a rough description of the workflow and the components shown in Figure 1a. Please note that unless stated otherwise, all components communicate using the OPC UA protocol. In principle, the functionality could be replicated using any other suitable protocol.

In essence, the testbed simulates a closed, single water tank system as shown in Figure 1b. It has a static outflow and an adjustable inflow through a pump. The water level is kept between an upper limit (V_{max}) and a lower limit (V_{min}) by activating the pump when the water level falls below V_{min} and deactivating it upon reaching V_{max}. The water level in the tank is monitored by a single fill level sensor. Depending on the current level, water purification chemicals are added by activating a valve. The control of the valve, logging, and monitoring functionalities are realized by the architecture shown in Figure 1a. In the following, we take a closer look at the individual components.

watertank-simulation - This component implements the simulation of the physical process of the tank in Python. This simplified version is derived from a real process of a water treatment facility. The current implementation features pump control to keep the water level between the allowed minimum and maximum. The pump is modeled to show PT2 behavior. The current tank fill level is reported to the levelsensor-server via an OPC UA client. For simplicity reasons, there is currently no feedback loop regarding the concentration of water treatment chemicals. The communication between the simulation and the connected servers can be made invisible



(a) Schematic representation of the testbed architecture

(b) Schematic representation of the simulated water tank system

Figure 1. Schematic representations of the testbed architecture and the simulated water tank system

to the host networks by using internal Docker networks while simultaneously decoupling the simulation and the server application code. The realism of the simulations was of lesser concern due to the testbed's focus on penetration testing.

levelsensor-server - This OPC UA server hosts the current sensor readings of the water level sensor. Depending on the requirements of the production network, the security configuration can be adjusted as needed. Another design consideration was the implementation of internal sensor value updates. We went with network-based communication for interacting with the simulation instead of using hard-coded callbacks in each custom server. This allows us to decouple the simulation from the server entirely, which simplifies replacing containers.

valve-server - The second OPC UA server allows the control of actuators in the system, which is currently just the valve for adding treatment chemicals. The valve status does not propagate back to the simulation, but we plan to extend the simulation to include this feature in the future. As for levelsensor-server, security features can be enabled as needed.

plc-logic - This component encapsulates the logic of activating and deactivating the valve depending on the current reading of the level sensor. It opens a connection to both the valve-server and the levelsensor-server and subscribes to changes in the water level variable. This triggers the latter to send a message to the logic client, where the reading is first written into a queue and evaluated asynchronously in another

```
services:
  headunit-dashboard:
    container_name: "headunit-dashboard"
    hostname: "headunit-dashboard"
    image: dashboard:latest
    environment:
      - SERVER_URI=opc.tcp://172.17.1.1:4840
    build:
      context: ${IMAGE_REPO_URL}#main:opcua/
          dashboard/python-opcua-asyncio/latest
    ports:
      - "8050:8050"
```

Figure 2. compose.yaml file for headunit

thread. Depending on the value, the valve position is changed by writing a new value to the valve-server. Any changes to either the water level or the valve position are also written to the plc-server. This allows other devices (the headunit in our case) to assess process data without having to directly interact and possibly interrupt the process level servers. Lastly, the water level thresholds for opening and closing the valves used by the logic component can be configured. For this purpose, it is informed if new values are written to the plc-server and adopts the values as soon as possible.

plc-historian - The historian acts as a recording mechanism for all relevant process parameters by writing them into a persistent database. The current implementation uses the file-based, lightweight SQLite database for simplicity reasons, which is made accessible to the container via volumes. The current implementation is not packaged as a container to allow easier replacement with the user's database of choice, and is therefore depicted in gray in Figure 1a. The historian subscribes to all relevant variables on the plc-server and is triggered upon receiving new values.

plc-server - This third OPC UA server allows systems of the upper layers to access process data for monitoring and planning purposes. In contrast to the production network servers, this instance simulates interactions with enterprise clients, e. g., encrypted and authenticated connections for administrative tasks or read-only connections for dashboards. Authorized users may also set certain properties that influence the simulation.

headunit-client - This client simulates a control station for visualizing and monitoring the underlying process through a web GUI. It also allows for setting certain process-relevant parameters, such as the threshold values. It connects to the plc-server via a secure connection.

Please note that these currently available components are implemented with different software stacks and versions. We only implemented what is currently needed for this proof-of-concept, but it is planned to add more containers. Another notable point is that we will add containers for network noise in the future, in order to simulate more realistic networks.

B. Building a Testbed

In this section, we present how the previously defined building blocks can be orchestrated and deployed. As we use Docker, the orchestration tool of choice is Docker Compose [15]. It allows the management of multi-container applications on a single host by using a declarative YAML file. It allows users to define services (containers), networks, and volumes that are then automatically configured and created upon starting the Compose application. Every physical device that is part of the testbed uses its own compose file. This gives the user maximum freedom in terms of distributing services across devices. We intentionally decided against using multi-host orchestration tools, such as Docker Swarm or Kubernetes, as this would introduce additional unwanted traffic that is normally not found in industrial environments.

```
services:
  plc-server:
    container_name: "plc-server"
    hostname: "plc-server"
    image: node-server:latest
    build:
      context: ${IMAGE_REPO_URL}#main:opcua/server/
          nodejs-node-opcua/latest
      args:
        NODESET_MODEL: "PLC.NodeSet2.xml"
      ...
    ports:
      - "4840:4840"
    networks:
      - plc-net

  plc-historian:
    container_name: "plc-historian"
    hostname: "plc-historian"
    image: historian:latest
    environment:
      - SERVER_URI=opc.tcp://plc-server:4840
    volumes:
      - /tmp/docker/database:/database
    build:
      context: ${IMAGE_REPO_URL}#main:opcua/
          historian/python-opcua-asyncio/latest
    networks:
      - plc-net
    depends_on:
      - plc-server

  plc-logic:
    container_name: "plc-logic"
    hostname: "plc-logic"
    image: plc-logic:latest
    environment:
      - PS_URI=opc.tcp://plc-server:4840
      - LSS_URI=opc.tcp://172.17.0.1:4840
      - VS_URI=opc.tcp://172.17.0.2:4840
    build:
      context: ${IMAGE_REPO_URL}#main:opcua/plc-
          logic/python-opcua-asyncio/latest
    networks:
      - plc-net
    depends_on:
      - plc-server

networks:
  plc-net:
    name: plc-net
    external: false
```

Figure 3. compose.yaml file for plc

```
services:
  levelsensor-server:
    container_name: "levelsensor-server"
    hostname: "levelsensor-server"
    image: node-server:latest
    build:
      context: ${IMAGE_REPO_URL}#main:opcua/server/
          open62541/latest
      args:
        NODESET_MODEL: "Tank.NodeSet2.xml"
      ...
    ports:
      - "4840:4840"
    networks:
      - levelsensor-net

  water-tank-simulation:
    container_name: "water-tank-simulation"
    hostname: "water-tank-simulation"
    image: water-tank-simulation:latest
    environment:
      - SERVER_URI=opc.tcp://levelsensor-server
          :4840/KRITIS3M/
    build:
      context: ${IMAGE_REPO_URL}#main:opcua/water-
          tank-simulation/python-opcua-asyncio/
          latest
    networks:
      - levelsensor-net
    depends_on:
      - levelsensor-server

networks:
  levelsensor-net:
    name: levelsensor-net
    external: false
```

Figure 4. compose.yaml file for process

To demonstrate the usage, we again consider the system shown in Figure 1a. Therein, we divided the system into three groups: headunit, plc, and process. This indicates a reasonable division of the testbed across different devices, which are Raspberry Pi 4's. The headunit could be assumed to be a monitoring workstation, the plc replicates the behavior of a real programmable logic controller, and the process encapsulates the interfaces with the low-level process devices. As a result, we would need a total of three compose files, which are presented below. Please note that the exact usage could deviate as the software matures, so please consult the online documentation for the latest version.

This first compose file in Figure 2 configures the services of the headunit device. The string for the *build* key points to path within the Github repository, which is comprised of the 4-tuple *protocol*, *component*, *library/stack*, and *version*. In this instance, the client is implemented using the latest version of the *asyncua* Python library. Our applications are configured by certain exposed environment variables (*environment*), in this case, the address of the plc-server on another physical device to which the client connects. As this application provides a graphical monitoring interface, it allows access via the host on port 8050 in this example.

The PLC application shown in Figure 3 is more complex, as

```
  tempsensor-server:
    container_name: "tempsensor-server"
    hostname: "tempsensor-server"
    image: node-server:latest
    build:
      context: ${IMAGE_REPO_URL}#main:opcua/server/
          open62541/latest
      args:
        NODESET_MODEL: "Temp.NodeSet2.xml"
      ...
    ports:
      - "4841:4840"
    networks:
      - levelsensor-net
```

Figure 5. compose.yaml file extension for additional sensor

it currently consists of three separate services that communicate over a private network *plc-net*. By using these networks, we expose ports only if it is necessary, and addressing within the network can be done by referencing the container names. The OPC UA servers in our implementation are able to load a number of different nodesets, depending on their task. This is specified by the *NODESET_MODEL* argument, and therefore avoids building the server anew every time the nodes change. Due to the fact that we are currently using SQLite, we have to mount the database file into the container using *volumes*.

Lastly, the production process application with the simulation is shown in Figure 4. It is demonstrated how internal container networks (in this example *process-net*) can be used to separate simulation-specific network traffic from the testbed-facing interfaces of the host system. Therefore, it isolates simulation and additional tools from the testbed system. It can also be seen that the same server implementation is used, but another nodeset is loaded upon startup. Please note that we omitted the valve server for the sake of clarity.

The described exemplary setup can be easily modified. For example, suppose that the system is upgraded by including a second sensor to measure the water temperature. To replicate this in the testbed, it is sufficient to modify the process compose file by adding a service as shown in Figure 5.

Modularity was a key requirement to support our concept for designing testbeds as proposed in [1]. By starting with a minimal setup initially and gradually adding functionality through additional containers, we can support a bottom-up approach when creating testbeds. This means that penetration tests are initially conducted in a relatively simple environment (e. g., only a server-client pair), and the following tests can build upon these layers of understanding. This is more feasible than coming up with complex scenarios straight away.

Another feature of the design is the redistribution of services between physical devices. This can easily be done by moving a service to another compose file and adjusting the environment variables if necessary. This is especially interesting for recording network data at specific nodes. By restructuring the setup, the desired connection can be exposed and recorded by inserting a network tap.

Lastly, adjusting the setup for a certain penetration test

```
plc-server:
  # change old version to new
  # build: "plc-server/node-opcua/latest"
  build: "plc-server/node-opcua/v2.73.0"
  ...
plc-server:
  container_name: "plc-server"
  ...
  build:
    # change old version to new
    # context: ${IMAGE_REPO_URL}#main:opcua/server
        /nodejs-node-opcua/latest
    context: ${IMAGE_REPO_URL}#main:opcua/server/
        nodejs-node-opcua/v2.73.0
  ...
```

Figure 6. modify image to enable vulnerabilities

is straightforward: first, the necessary implementation and version must be selected. This is as easy as searching for the relevant CVEs, and selecting the vulnerable software. Then, the image for the affected container is modified. For example, CVE-2022-21208 describes a Denial-of-Service attack against implementations using the node-opcua package. Before version 2.74.0, this causes the server to crash if an attacker continuously sends big chunks of data to the server. Simply modifying the version within the compose file enables the vulnerability, as shown in Figure 6.

We currently support the most common open-source implementations of OPC UA, namely open62541 (written in C), node-opcua (NodeJS), opcua-asyncio (Python), locka99/opcua (rust), and UA-.NETStandard (C#). Over time, we are planning to expand on the available stacks.

IV. Conclusion and Future Work

This paper proposes a modular and flexible testbed approach to facilitate easier and faster reconfigurations for penetration testing purposes. First, we put our approach into context by providing design considerations, relevant literature, and issues that we encountered. Next, we introduced our testbed approach by defining the building blocks of the modular design. Then, these were combined into an exemplary testbed. Lastly, we showed how the ability to quickly change the architecture, implementation, and configuration can be used to leverage penetration testing activities.

The next steps include open-sourcing of our testbed complete with an example configuration. We will also expand the number of available components in order to enable the modeling of more complex scenarios. We will use the knowledge gained to construct a testbed with a sufficient number of features to create an ICS dataset for intrusion detection research.

Acknowledgment

The presented work is part of the research project *KRITIS Scalable Safe and Secure Modules* (KRITIS³M), which is funded by the Project Management Jülich (PtJ) and the German Federal Ministry for Economic Affairs and Climate Action (BMWK) under funding code 03EI6089A.

References

[13] D. Antonioli and N. O. Tippenhauer, "MiniCPS: A Toolkit for Security Research on CPS Networks," in *Proceedings of the First ACM Workshop on Cyber-Physical Systems-Security and/or PrivaCy*, ser. CPS-SPC '15, New York, NY, USA: Association for Computing Machinery, 2015, pp. 91–100, ISBN: 978-1-4503-3827-1. DOI: 10.1145/2808705.2808715.

[14] A. Murillo *et al.*, "High-Fidelity Cyber and Physical Simulation of Water Distribution Systems. I: Models and Data," May 2023, Publisher: CISPA. DOI: 10.60882/cispa.25460440.v1.

[12] Laboratory for Safe and Secure Systems, "MOdular Testbed for Researching Attacks (MOTRA) - setups," [Online]. Available: https://github.com/Laboratory-for-Safe-and-Secure-Systems/motra-setups (Retrieved: 09/08/2025).

[1] S. Kraust, P. Heller, and J. Mottok, "Concept for designing an ICS testbed from a penetration testing perspective," in *2025 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, ISSN: 2768-0657, Jun. 2025, pp. 561–568. DOI: 10.1109/EuroSPW67616.2025.00071.

[2] OPC Foundation, "OPC UA Online Reference - Released Specifications," 2025, [Online]. Available: https://reference.opcfoundation.org/ (Retrieved: 09/08/2025).

[3] F. Pauker, T. Frühwirth, B. Kittl, and W. Kastner, "A Systematic Approach to OPC UA Information Model Design," *Procedia CIRP*, vol. 57, pp. 321–326, 2016, ISSN: 22128271. DOI: 10.1016/j.procir.2016.11.056.

[4] S. Friedl, C. von Arnim, A. Lechler, and A. Verl, "Generation of OPC UA Companion Specification with Eclipse Modeling Framework," in *2020 16th IEEE International Conference on Factory Communication Systems (WFCS)*, Apr. 2020, pp. 1–7. DOI: 10.1109/WFCS47810.2020.9114448.

[5] E. Jacob, "A Broken Chain: Discovering OPC UA Attack Surface and Exploiting the Supply Chain," Dec. 2021, [Online]. Available: https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-A-Broken-Chain-Discovering-OPC-UA-Attack-Surface-And-Exploiting-The-Supply-Chain.pdf (Retrieved: 09/08/2025).

[6] BSI, "OPC-UA Security Analysis," Bundesamt für Sicherheit in der Informationstechnik, Tech. Rep., 2022.

[7] P. Cheremushkin and S. Temnikov, "OPC UA Security Analysis," Kaspersky Lab, Security Analysis, 2018.

[8] Team82, "Exploring the OPC Attack Surface," 2020, [Online]. Available: https://claroty.com/team82/research/white-papers/exploring-the-opc-attack-surface (Retrieved: 09/08/2025).

[9] A. Erba, A. Müller, and N. O. Tippenhauer, "Security analysis of vendor implementations of the OPC UA protocol for industrial control systems," in *Proceedings of the 4th Workshop on CPS & IoT Security and Privacy*, ser. CPSIoTSec '22, New York, NY, USA: Association for Computing Machinery, Nov. 7, 2022, pp. 1–13, ISBN: 978-1-4503-9876-3. DOI: 10.1145/3560826.3563380.

[10] L. Roepert, M. Dahlmanns, I. Fink, J. Pennekamp, and M. Henze, "Assessing the security of OPC UA deployments," *Proceedings of the 1st ITG Workshop on IT Security*, May 11, 2020, Accepted: 2020-05-11T12:51:19Z ISBN: 9781698020358 Publisher: Universität Tübingen. DOI: 10.15496/publikation-41813.

[11] J. Polge, J. Robert, and Y. L. Traon, "Assessing the impact of attacks on OPC-UA applications in the Industry 4.0 era," in *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, Jan. 2019, pp. 1–6. DOI: 10.1109/CCNC.2019.8651671.

[15] Docker, "Docker compose manual," 2025, [Online]. Available: https://docs.docker.com/compose (Retrieved: 09/08/2025).