Deriving a Cost-Effective Digital Twin of an ICS to Facilitate Security Evaluation

Ron Bitton[1], Tomer Gluck[1], Orly Stan[1], Masaki Inokuchi[2], Yoshinobu Ohta[2], Yoshiyuki Yamada[2], Tomohiko Yagyu[2], Yuval Elovici[1], and Asaf Shabtai[1](✉)

[1] Department of Software and Information Systems Engineering, Ben-Gurion University of the Negev, Beersheba, Israel
shabtaia@bgu.ac.il

[2] Security Research Laboratories, NEC Corporation, Minato, Japan

Abstract. Industrial control systems (ICSs), and particularly supervisory control and data acquisition (SCADA) systems, are used in many critical infrastructures and are inherently insecure, making them desirable targets for attackers. ICS networks differ from typical enterprise networks in their characteristics and goals; therefore, security assessment methods that are common in enterprise networks (e.g., penetration testing) cannot be directly applied in ICSs. Thus, security experts recommend using an isolated environment that mimics the real one for assessing the security of ICSs. While the use of such environments solves the main challenge in ICS security analysis, it poses another one: the trade-off between budget and fidelity. In this paper we suggest a method for creating a digital twin that is network-specific, cost-efficient, highly reliable, and security test-oriented. The proposed method consists of two modules: a problem builder that takes facts about the system under test and converts them into a rules set that reflects the system's topology and digital twin implementation constraints; and a solver that takes these inputs and uses 0–1 non-linear programming to find an optimal solution (i.e., a digital twin specification), which satisfies all of the constraints. We demonstrate the application of our method on a simple use case of a simplified ICS network.

Keywords: Industrial control systems
Supervisory control and data acquisition · Penetration test
Non linear integer programming

1 Introduction

Supervisory control and data acquisition (SCADA) is user to refer to a range of industrial control systems (ICSs) which assist in overseeing complex industrial processes. SCADA systems are used in a long list of industrial applications and processes in facilities including electricity generation plants, chemical plants, manufacturing plants, water and sewage treatment facilities, and industries such

© Springer Nature Switzerland AG 2018

as the transportation industry. SCADA systems have gained increasing popularity, and industries have become heavily dependent on these systems for collecting data from industrial processes in order to control and monitor their operations to ensure that they are functioning properly. A failure in a SCADA system or one of its elements may result in a failure of the industrial process being controlled. In some cases those systems are life critical, and thus a successful attack on them can jeopardize thousands of people's lives [12,18]. Because of this, the foremost design considerations of such systems have always included a high level of reliability and availability. In general, modern SCADA systems are comprised of a communication infrastructure and the following major elements:

The programmable logical controller (PLC) is one of the main components of the SCADA system. Field devices, e.g., sensors and controllers, send signals and status updates to the PLC and receive operational commands from the PLC, usually without the direct involvement of a human operator. The PLC is also responsible for reflecting the field device state to remote devices (e.g., HMI).

The engineering workstation (EWS) is a computer workstation used to update the PLC software and program the PLC logic.

The human machine interface (HMI) is a computer workstation that makes the industrial process controlled by a SCADA system accessible to a human operator. The operator can monitor processes (e.g., the HMI may display the current water level at an automated reservoir) and send commands to the field devices through an HMI (e.g., stopping the operation of a pump).

Sensors are used in order to reflect the state of an industrial element (e.g., wind speed in a wind tunnel) or the environment (e.g., air temperature). The information from sensors is used by the PLC to control the industrial elements.

Communication infrastructure includes switches, cables, wireless receivers, etc. Contemporary SCADA systems are able to use Ethernet and TCP/IP infrastructure in order to achieve connectivity; legacy SCADA systems rely on older technologies and communication protocols. SCADA components communicate by utilizing standard SCADA protocols, such as DNP3 and IEC 61850, or proprietary vendor-specific protocols, such as S7 and variants of Modbus.

Additional components such as **controllers and actuators**, **databases** which store historical information (i.e., Historian), and **security elements** such as Firewall and one-way traffic devices can also be found in a typical SCADA system.

SCADA systems, especially legacy SCADA systems, are inherently insecure. Initially they were designed and built using specialized and proprietary protocols, implemented by old software and hardware which were rarely patched [11]. Security measures such as anti-viruses and encryption are usually not considered in ICSs. These security measures are not capable of identifying and defending against ICS-specific attacks (e.g., attacks against SCADA protocols such as Modbus) and might harm the availability of the system, which is one of its most important requirements [7,14].

The use of SCADA systems in critical infrastructures makes them desirable targets for attackers. Attacks on such systems have been increasing in recent years. As demonstrated by the Stuxnet worm, and more recently by the TRITON malware, a successful SCADA attack can have serious impact on a nation's economy, safety, and stability. For this reason, continuous security evaluation of ICSs is crucial for mitigating cyber-attacks.

Penetration testing (pen-testing) [1] is a commonly used security measure. The goal of pen-testing is to detect weaknesses in the network such as hosts running vulnerable software, misconfiguration of network components or security countermeasures, usage of default passwords for login services, etc.

The security evaluation of an ICS is quite different from the security evaluation of a typical enterprise network. Typical pen-testing activity focuses on an enterprise's IT environment, especially IT components that can be exploited via the Internet. These kinds of tests usually represent a small part of a typical security evaluation of an ICS [1]. Pen-testing for ICSs mainly focuses on the industrial components (e.g., Historian, HMI, PLC, and sensors) which communicate over dedicated industrial protocols (e.g., Modbus, DNP3). These components and protocols were originally developed for serial communication based on the assumption that ICSs are isolated from the IT environment (and thus not connected to the Internet); therefore, security properties such as authentication and encryption are usually not implemented in these protocols. Currently, industrial protocols are commonly transmitted over TCP/IP; in addition, many ICSs are connected to the Internet, thus making them easy targets for attackers.

The fact that SCADA systems are implemented in critical infrastructures also makes it difficult to evaluate their security. A typical pen-testing activity (for a non-industrial environment) is usually executed within the enterprise network, however this cannot be done in the case of an ICS. Pen-testing activities involve intrusive actions such as port scanning (e.g., using Nmap) and vulnerability assessment (e.g., using OpenVAS or a Zeus scanner), which may crash industrial components and therefore cannot be directly executed in operational industrial environments. Given this, security experts have suggested the construction of a dedicated testbed for evaluating the security of an ICS [5,7,9].

A testbed is an isolated environment which contains a generic implementation of the architecture of the system under test and allows safe execution of penetration tests. The creation of a testbed requires significant investment of funds and effort. Therefore, an efficient testbed should be able to mimic a variety of ICS setups [5]. For this reason, most testbeds are not designed to represent a specific ICS environment, but are more generic so as to be able to address the needs of multiple facilities in the same industry. Keeping the testbed generic can compromise the fidelity (i.e., the requirement that a testbed should represent the system under inspection as accurately as possible) [17].

In this paper, we introduce a new automated method for inferring the specification of a *digital twin* that is designed to facilitate the security evaluation of a specific industrial environment. In contrast to testbeds, which are generic, a digital twin is a replica of a specific ICS; i.e., a model that consists of all of the components from the original industrial environment. Each replicated component can either be implemented as a digital clone (e.g., by using simulation or virtualization software), or alternatively can be physically installed in the twin model. The components that are implemented within the digital twin, as well as the level of implementation of each component, defines the specific security tests that can be conducted on the digital twin (e.g., a digital twin without the HMI implemented does not support the execution of security tests on the HMI). The primary benefit of using a digital twin, as opposed to a testbed, is that it reliably represents the real industrial environment. In other words, the results of a pen-test conducted on the digital twin genuinely reflect the expected results of conducting the same test in the real environment.

One of the most challenging tasks in the process of creating a digital twin is determining the implementation level (specification) of its components. The implementation level of the components in the digital twin directly affects the overall cost of establishing the digital twin as well as the degree to which it reflects the industrial environment (fidelity). For example, a twin model that is completely identical to the real industrial environment (i.e., a physical clone) has the highest fidelity (as it allows the execution of all possible tests), but implementing such a model is extremely expensive. We present a method for deriving the specification of a cost-effective digital twin that is specifically designed to facilitate the security evaluation of a specific industrial environment. The proposed method models the problem of deriving the digital twin for a specific industrial environment as an optimization problem. The optimization problem attempts to maximize the impact of the digital twin under strict budget constraints (i.e., allowing the execution of the most important penetration tests for improving the security of the industrial environment).

The contributions of this paper are as follows:

- We introduce the concept of creating a cost-effective digital twin that is specifically designed to facilitate the security evaluation of a specific industrial environment.
- We propose a method that is based on a constrained optimization problem, specifically, 0–1 non-linear programming, for deriving the configuration of the digital twin model of a specific industrial environment.
- We demonstrate the application of our proposed method on a simplified thermal power plant architecture.

2 Related Work

In order to conduct penetration testing on ICS networks, the use of a testbed has been proposed. A testbed is an isolated environment that simulates the operation of some real system.

According to a recent survey conducted by Qassim *et al.* [15] testbed implementation approaches can be categorized as follows:

Physical implementation: refers to a physical clone of the components. This approach reflects the industrial environment at the highest degree. However, physical implementation of all of the components of a specific factory is in most cases, not feasible because of the high costs of such implementation. As a result, the majority of physical testbeds are more generic, aimed at being able to address the needs of multiple facilities in the same industry, rather than specific facilities.

Virtualization/emulation software: eliminates the software's dependency on the hardware. Virtualization/emulation software enables the establishment of large-scale testbeds, while requiring less hardware, thereby reducing the implementation costs. This approach enables the testing of software components and protocols, but it does not enable the testing of hardware components. In addition, by eliminating software and hardware dependencies, some of the penetration tests may not provide the expected results as tests performed in the real environment.

Software simulation: designed to simulate the inputs, outputs, and behavior of real components (e.g., temperature sensor). This approach can provide large-scale implementation at a low cost, however, it provides very low fidelity. Therefore, the main usage of simulation software is to enable the testing of other components (e.g., to feed a virtual or physical component with simulated inputs/outputs).

To avoid the high costs (as described above), as well as the maintenance involved in a physical replication testbed, many researchers chose to implement their testbed using the simulation, virtualization, or hybrid approaches.

Genge *et al.* [4] and *Lemay et al.* [8] presented testbeds for assessing the security of ICS networks. Both works suggested the combination of emulated and simulated components in order to reduce implementation costs. Lemay *et al.* [8] provided the following methodology for component implementation: the components that are relevant to the test objectives should be emulated; components that directly interface with the emulated components should be implemented as closely as possible to real life; the remaining components can be implemented at any level, and can even be simulated.

Unlike Genge *et al.* [4] and Lemay *et al.* [8], *Gao et al.* [3] and Green *et al.* [5] suggested the integration of physical devices in their testbeds.

Alves *et al.* [2] also addressed the discrepancies between different implementation levels. They established physical and virtual gas pipeline testbeds and showed that the testbeds behave differently under a denial of service attack, and behave similarly under a man-in-the-middle attack.

A digital twin is a concept from the product life-cycle management (PLM) domain introduced by Grieves *et al.* [6]. It is a virtual representation of a specific physical product. The idea behind this concept is that the digital twin should be linked to the physical product throughout the product's life-cycle and constantly mirror it. By doing so, the digital twin enables the prediction of the future behavior and performance of the real product.

Unlike the previously mentioned works that suggested general testbed architectures, we propose an *adaptive* method for deriving the configuration of a cost-effective digital twin for a specific industrial environment. The cost-effective digital twin defines the implementation level of the different industrial components (physical implementation, virtualization/emulation software, and simulation software) to allow the evaluation of the desired security tests.

3 Cost-Effective Digital Twin for ICS

In this section, we present an adaptive method for deriving a digital twin specification for a given ICS, under strict budget constraints. The proposed method maximizes, within the budgetary limitations, the impact of the digital twin. The impact of a digital twin is evaluated by the number and types of security penetration tests that it supports. On one hand, each test has its own *benefit* i.e., security-wise, one test might be more important than another. On the other hand, each test has its own cost. The cost of a test is determined by the costs of the participating components (i.e., the direct cost of implementing them in the digital twin), as well as the test's execution costs (e.g., security expert's time/salary). Note that a component might be required for multiple security tests.

Similar to the creation of testbeds, we consider three types of implementation levels for each element: *physical*, *virtualization/emulation*, and *software simulation*. The output of the proposed method specifies the digital twin configuration, i.e., which components of the ICS should be implemented and at which implementation level.

Our proposed method models the problem of deriving a cost-effective digital twin as a 0–1 non-linear programming problem. Such problems optimizes a non-linear target function (e.g., the overall benefit of the tests supported by the digital twin), while being subjected to multiple related constraints (e.g., budget limits).

3.1 Notations

In order to formally describe the problem and the method's inputs, we define the following notations.

General ICS Environment Information. The set of possible ICS components is denoted by

$$C = \{c_1, \ldots, c_{n_C}\}.$$

For example, $C = \{PLC, EWS, Historian, PC, \ldots\}$.

We also define the following subsets of C:

- $N \subset C$ - ICS component communicating over IP
- $M \subset N$ - ICS components running modern operating systems (e.g., desktops, Web servers, HMI, EWS, Historian)
- $NC \subset N$ - Network components (e.g., router, switch, and firewall)
- $F \subset C$ - Field devices (e.g., generator and boiler)

- $D \subset N$ - ICS components which are part of the direct control layer (e.g., RTU and PLC)
- $S \subset M$ - ICS components which are part of the supervision layer (e.g., HMI, EWS, and Historian)

General Test Specification. The set of all possible tests is denoted by

$$T = \{t_1 \dots t_{n_T}\}$$

A list of possible tests for the penetration testing of electric utilities based on the NESCOR methodology [16] is presented in Appendix C.

We denote the execution of test t_i on component c_j by $t_i(c_j)$. For each test $t_i \in T$ we specify three types of prerequisites in order to be able to execute t_i on c_j: device implementation requirements (DIR), environment implementation requirements (EIR), and prerequisite test (PT).

- $DIR(t_i, c_j)$, $t_i \in T \wedge c_j \in C$ - denotes the minimal implementation level of a tested component c_j, which enables the execution of test t_i in the digital twin. For example, disassembling an embedded device (test $t_{4.1.1}$ in Appendix C) cannot be performed on either an emulated or simulated device, thus a physical implementation of the component in the digital twin is essential for executing this test. The formal representation of this requirement is as follows (p stands for *physical*):

$$DIR(t_{4.1.1}, f \in F) = f^p$$

- $EIR(t_i, c_j)$, $t_i \in T \wedge c_j \in C$ - denotes the minimal implementation level of components that communicate with c_j and are required for executing t_i. For example, in order to perform functional analysis (test $t_{4.2.1}$ in Appendix C) on $d \in D$, such as a PLC, one must emulate the components that communicate with the PLC from the direct control group, such as other PLCs (denoted by D_d), and from the supervisory control group, such as HMI (denoted by S_d). In addition, there is a need to simulate field devices that communicate with the PLC (denoted by F_d). The formal representation of these requirements is as follows (e stands for *emulation*, and s stands for *simulation*):

$$EIR(t_{4.1.1}, d \in D) = \{D_d^e, S_d^e, F_d^s\}$$

- $PT(t_i, c_j)$, $t_i \in T \wedge c_j \in C$ - represents the dependencies between tests; for example,

$$PT(t_{4.2.5}, f \in F) = \{t_{4.2.3}, t_{4.2.2}, t_{4.2.1}\}$$

indicates that tests $t_{4.2.3}$, $t_{4.2.2}$, $t_{4.2.1}$ should be executed first in order to execute test $t_{4.2.5}$ on f.

Using these three types of requirements, we define the set of test dependencies (TD) for executing test $t_i \in T$ on a component $c_j \in C$, as follows:

$$TD(t_i, c_j) = <DIR(t_i, c_j), EIR(t_i, c_j), PT(t_i, c_j)>$$

- **Example I: Device disassembly.** In order to enable the disassembling of a field device $f \in F$ ($t_{4.2.1}$), the digital twin model must physically implement f. Thus, the test dependencies for device disassembly of field devices $f \in F$ are as follows:

$$TD(t_{4.2.1}, f \in F) = <f^p, \emptyset, \emptyset>$$

- **Example II: Endpoint fuzzing.** Endpoint fuzzing ($t_{4.2.5}$) is a pen-testing activity that could be executed on an emulated or physical device. However, it is not possible to perform fuzzing without understanding the tested interface and without capturing and analyzing the communication with the interface. For these reasons, capture analysis ($t_{4.2.3}$), communication capture ($t_{4.2.2}$), and interface functional analysis ($t_{4.2.1}$) are prerequisite tests for endpoint fuzzing. In addition, in order to perform this test the digital twin must also emulate the direct control devices which communicate with the tested device. Thus, the test dependencies for fuzzing a field device $f \in F$ are as follows:

$$TD(t_{4.2.5}, f \in F) = <f^e, \{D_f^e\}, \{t_{4.2.3}, t_{4.2.2}, t_{4.2.1}\}>$$

Specific ICS Environment Information. The specific ICS environment (for which we would like to derive the digital twin definition) is denoted as follows:

- $E = \{e_1, \ldots, e_{n_E}\}$ - the set of elements in a specific ICS environment, e.g., e_i is a specific PLC in the ICS.
- $Communication = \{<e_i, e_j> \mid e_i, e_j \in E\}$ - the set of links between elements in the specific ICS environment, as was observed in the ICS's network, e.g., $<e_i, e_j>$ indicates that a communication was observed between element e_i and element e_j.
- $Topology = <E, Communication>$ - the topology of the specific ICS, which consists of the set of elements (E) and their communication links ($Communication$).
- $I = \{p, v, s\}$ - the set of possible implementation levels of an element in E where p stands for physical replica, v for virtualization, and s for simulation.
- $role : E \rightarrow C$ - a function that maps an element in the specific ICS environment to its type, e.g., $role(e_1) = PLC$ indicates that element e_1 is an instance of a PLC in the ICS.

In addition, we define the specific environment dependencies (ED) as follows:

$$ED(t_i, e_j), t_i \in T \wedge e_j \in E$$

Unlike the test dependencies (TD), the environment dependencies (ED) are derived for a specific ICS environment, e.g., the following expression: $ED(t_m, e_n) = \{{e_2}^v, {e_4}^s, {e_5}^s\}$ indicates that in order to execute test t_m on the specific element e_n, the digital twin must contain the following: a virtual (or higher) implementation of element e_2 and at least a simulation of elements e_4, and e_5.

According to the proposed method the main prerequisite for deriving the digital twin is the topology of the specific ICS environment. Typical ICS environments are extremely complex and may change over time; thus, acquiring the environment information is not a trivial task. There are several tools and methods that can be used to collect the required information, including the ICS blueprints which usually contain the architecture design of the specific ICS environment, as well as passive monitoring tools such as the GRASSMARLIN that are able to extract information from the live (or recorded) network traffic (including IP addresses, operating system of components, vendors, and component types).

Costs, Benefits and Budget

- $cost : E \times I \rightarrow \mathbb{R}$ - a function that maps a specific implementation of an element to its cost, e.g., $cost(e, p) = 650$ indicates that a physical implementation of element e in the digital twin costs \$650.
- $benefit : T \times E \rightarrow \mathbb{R}$ - a function that defines the benefit of executing a test on an element, e.g., by setting the $benefit(t, e)$ to b, the asset owner indicates that the benefit of executing test t on element e is b; where, a high b value will increase the probability that this test will be supported by the digital twin model (by setting the $benefit(t, e)$ to ∞, the asset owner can force the algorithm to derive a digital twin which support this test). The benefit of a test is assigned according to the importance of the test (the significance of the expected findings) and the element being tested.
- $Budget \in \mathbb{R}$ - the overall budget assigned to create the digital twin.

3.2 Proposed Method

The proposed method consists of the following three main modules (see Fig. 2 in Appendix E):

The **Data Processor** is responsible for integrating the general test dependencies (i.e., TD) and the topology of a specific industrial environment (i.e., $Topology$), in order to derive the list of environment dependencies (i.e., ED) of the specific industrial environment.

The **Problem Builder** is responsible for translating the information provided for the specific industrial environment (e.g., budget and test dependencies) to a non-linear maximization problem.

The **Solver** solves the non-linear maximization problem in order to derive the specification of the cost-effective digital twin.

The input to the proposed method includes the following:

ICS Architecture: the specification of the architecture of the industrial environment for which the digital twin is created. The specification includes: system topology

($Topology$) i.e., a description of the elements in the system (E) and their communication patterns ($Communication$); the role of each element ($role(e)$);

the cost for each possible implementation of the elements ($cost(e, i)$); and the benefit of executing tests on elements ($benefit(t, e)$).

Budget: ($Budget$) the overall budget allocated for the creation of the digital twin.

Test Specification: includes the set of possible tests $T = \{t_1 \dots t_{n_T}\}$ and the set of test dependencies $TD(t_i, c_j), t_i \in T \wedge c_j \in C$.

3.3 Data Processor

The Data Processor derives the set of environment dependencies (ED) by analyzing the following inputs: (1) a general specification of test dependencies (TD); (2) the specific topology of the industrial environment under test ($Topology = <E, Communication>$); and (3) an element in the environment ($e \in E$). This is done according to the process presented in Algorithm 1.

Given the inputs, the Data Processor initially adds the appropriate device implementation requirement (DIR) to the environment dependencies (lines 10–12). Then, for each environment implementation requirement $r \in EIR$ it adds the elements in the ICS that communicate with e and are of the type specified in r (lines 13–16). Finally, it recursively adds the environment dependencies of the prerequisite tests (lines 19–21). The output of the procedure are the environment dependencies for executing t on e, which are specific for the particular ICS architecture.

3.4 Problem Builder

The Problem Builder represents the digital twin inference problem as a 0–1 non-linear programming problem. The non-linear integer programming problem focuses on the optimization of a non-linear target function, while satisfying a set of non-linear constraints (that are represented as algebraic equations) [10]. The non-linear integer problem is formally defined as follows:

$$\begin{aligned}
&\min / \max f(x) \\
&s.t\ g_i(x) \leq b_i, i = 1, \dots, m \quad h_i(x) = y_i, j = 1, \dots, k \\
&x \in X, X \subset \mathbb{Z}^n\ and\ X\ is\ a\ finite\ set
\end{aligned}$$

where $f(x)$ is the target function that we wish to maximize (or minimize), and the constraints are represented by $g_i(x)$ and $h_i(x)$.

A 0–1 non-linear programming problem is a special case of the non-linear integer programming problem, in which x can either be 0 or 1. In this section, we describe how we define the target function ($f(x)$) and the constraints ($g_i(x)$), in order to represent the digital twin specification inference problem as a 0–1 non-linear programming problem.

The specification of a given digital twin model is defined by the variables of the 0–1 non-linear programming problem, which are denoted as follows:

$$X = <x_1^s, x_1^e, x_1^p, \dots, x_{n_E}^s, x_{n_E}^e, x_{n_E}^p>. \tag{1}$$

Algorithm 1. Data Processor

```
1: **Inputs:**
2:     ${TD(t,c) | t \in T \wedge c \in C}$
3:     $Topology \leftarrow <E, Communication>$
4:     $e \in E$
5: **Precondition:**
6:     $role(e) \in C$
7: **Initialize:**
8:     $ED \leftarrow \emptyset$
9: **function** ProcessData($TD(t,c)$, $e$ ,$Communication$)
10:    $DIR \leftarrow GetDeviceImplementationRequirement(TD(t,c))$
11:    $i \leftarrow GetImplementationLevel(DIR)$
12:    $ED \leftarrow ED \cup e^i$
13:    $EIR \leftarrow GetEnvironmentImplementationRequirement(TD(t,c))$
14:    **for each** $r \in EIR$ **do**
15:        $i \leftarrow GetImplementationLevel(r)$
16:        $C \leftarrow GetIndustrialControlSubGroup(r)$
17:        **for each** $<j,k> \in Communication | j = e \wedge role(k) \in C$ **do**
18:            $ED \leftarrow ED \cup k^i$
19:    $PT \leftarrow GetPrerequisiteTests(TD(t,c))$
20:    **for each** $t^* \in PT$ **do**
21:        $ED \leftarrow ED \cup Process(TD(t^*,c), e, Communication)$
22:    **return** $ED$
```

Each variable indicates whether a specific element e is implemented as i within the digital twin as defined by Eq. 2.

$$X = \{x_e^i \mid i \in I,\ e \in E\} \tag{2}$$

These variables can be equal to 0 (zero) or 1 (one), and thus the first set of constraints is:

$$x_e^i \in \{0, 1\} \tag{3}$$

where $x_e^i = 1$ indicates that element e is implemented in the digital twin as i, and $x_e^i = 0$ indicates that element e is not implemented in the digital twin as i.

Equation 4 presents the *implementation constraint*, which ensures that an element e is implemented as either *simulated*, *virtualized*, *physical*, or not implemented at all. The number of implementation constraints is equal to the number of elements in the given ICS (i.e., n_E).

$$x_e^p + x_e^v + x_e^s \leq 1 \tag{4}$$

In order to ensure that the overall cost of the digital twin implementation does not exceed the allocated budget, we define the *cost constraint* presented in Eq. 5.

$$\sum_{x_e^i \in X} cost(x_e^i) \leq Budget \tag{5}$$

Each assignment for X defines a possible configuration of the digital twin, where a valid assignment satisfies all of the defined constraints.

Given the above constraints, the target function (defined in Eq. 6) is designed to maximize the impact of the digital twin model.

$$\max\left(\sum_{e\in E, t\in T} benefit(t,e)\cdot \prod_{e_j^i\in ED(t,e)} x_{e_j}^i\right) \tag{6}$$

where the impact of a given digital twin model (defined by the assignment X) is defined as the sum of all of the benefit values for the tests in T that can be executed on X. As can be seen, the benefit value is added only if all of the dependencies of a test are satisfied.

3.5 Solver

A 0–1 non-linear programming problem is NP-hard [13]. In small environments the solution for this problem can be determined by applying a brute force approach, i.e., for each possible assignment for X, first check whether it satisfies all of the constraints; if all of the constraints are satisfied, compute the value of the target function, and finally, select a valid assignment that provides the maximal value.

The time that it will take for the brute force approach to provide the optimal result is significant as it grows exponentially by the number of components. Given n components, and m security tests, and three implementation levels (real, emulated, simulated), the time complexity for the brute force algorithm is as follows:

$$O(3^n\cdot nm\cdot n) \tag{7}$$

where, 3^n represents all of the possible implementation of a components, nm represents the maximum tests per component, and n is the calculation of the cost per implementation state. The exponential time complexity makes the brute force algorithm unsuitable for large ICS environments (more than 20 components). For example, executing the brute force approach on the simple ICS environment presented in Appendix E in Fig. 3, which consisted of 14 components, takes three minutes when using a standard personal computer. In future work, we plan to develop and evaluate different heuristics which are on average sub-exponential (but may not provide the best setup for the digital twin.)

4 Demonstration

In this section, we demonstrate the application of the proposed method on a simplified ICS environment of a thermal power station with one boiler and two generators.

4.1 Description of the Tested ICS Environment

The simplified environment (illustrated in Appendix E, Fig. 3) consists of an enterprise network, a supervision layer, a direct control layer, and field devices.

The enterprise network contains an IT client and an IT server, which are connected to the supervision layer through a firewall that filters improper packets. The supervision layer consist of the following components which monitor and control the direct control components:

- **Historian.** Responsible for logging all events occurring during the process. To do so, the historian periodically queries the PLCs for their states (via Modbus/TCP in the case of PLC-1, or S7comm in the case of PLC-2).
- **Human machine interface (HMI).** Provides a human-friendly interface for interacting with the field devices. In order to report the field devices' states and alarms to the operator, the HMI periodically queries the PLCs, as the Historian does (via Modbus/TCP in the case of PLC-1, or S7comm in the case of PLC-2). Moreover, the HMI enables the operator to remotely change field devices' parameters.
- **Engineering Work Station (EWS).** Enables the operator to change the PLCs' configurations and logic. The EWS has all of the required programming and configuration software installed. It communicates with the PLCs and HMI through the S7comm protocol when such updates occur.

The supervision layer's components are connected to the direct control devices through a switch. The direct control components include:

- **Two Siemens S7-300 PLCs.** These components directly control the field devices. PLC-1 controls both the boiler (BLR) and one of the generators (GEN-1). It can turn the boiler's heater on or off, change the generator's rotation speed, and start or stop its operation. PLC-2 controls only GEN-2 and can perform the two latter actions as well. The PLCs are connected to the supervision layer via the switch (SW-2), and communicate with each other via the S7comm protocol.
- **Remote Terminal Unit (RTU).** This component is connected directly to the PLCs and enables the operator to manually change the field devices' parameters and present their current states and alarms.

The field devices include the components that physically perform the process. This simplified environment contains two generators (GEN-1 and GEN-2) and one boiler (BLR).

4.2 Security Test Specifications

For the demonstration, we followed the pen-testing methodology presented by the National Electric Sector Cybersecurity Organization Resource (NESCOR) [16]. This methodology provides guidelines for executing penetration tests on smart grid systems. Although the NESCOR methodology is specifically designed for smart grid systems such as advanced metering infrastructure (AMI), wide-area monitoring, protection and control (WAMPAC), and home area network (HAN), it provides an extensive list of pen-testing activities that can be applied on other types of ICSs.

The various testing activities presented in their methodology are classified into four categories: *embedded device penetration tasks*, which address the physical attack vector against field devices; *network communication penetration tasks*, which address the exploitation of devices through network protocol manipulation; *server application penetration tasks*, which address testing applications that are running on the control servers; and, *server operating system penetration tasks*, which address testing of the operating system of the control servers.

Execution of the pen-tests presented in the NESCOR methodology on a digital twin in which not all of the components are physically implemented is not trivial, because, as described in Sect. 3.2, the execution of some activities in a digital twin may depend on a specific set of requirements (denoted by DIR, EIR, and PT).

We thoroughly analyzed more than 80 penetration tests presented in the NESCOR methodology and defined the three types of requirements for each test. The complete set of tests and requirements is summarized in Appendix C. For our demonstration we select the following five tests: Device Disassembly (4.1.1), Interface Functional Analysis (4.2.1) Communication Capture (5.2.1), Fuzzing (5.2.4), Application Fingerprinting (7.1.1), and Application Functional Analysis (7.1.2).

4.3 Implementation Cost Description

In the proposed method we considered three types of implementation levels: *physical*, *virtual*, and *simulation*.

Obviously, not all of the components can be implemented by all type of implementations, and some physical devices may not have an emulated/virtual version. In addition, the pricing of different implementation levels is not the same for different vendors. For example, a physical SIEMENS PLC can cost from hundreds of dollars to thousands with an average cost of about \$2500 for the S7300 models[1]; a license for S7-Plcsim software, which can be used for emulating a SIEMENS PLC or HMI costs \$700[2]; and using third party tools to simulate a PLC can be less expensive (e.g., *awlsim*[3] is free of charge, with costs just for the setup time).

For simplicity, in our demonstration we assumed that a physical implementation of a device would have the highest cost and a simulation-based implementation the lowest. Specifically, as presented in Appendix B, a physical implementation is ten times more expensive than virtualization, which is three times more expensive than simulation. In addition, an equal benefit for all tests i.e., $benefit(t, e) = 1 \ \forall t \in T \wedge e \in E$ was assumed. It should be mentioned that these assumptions do not affect the construction of the problem or its solution by using 0–1 non-linear methods; therefore we believe that these assumptions are plausible.

[1] http://www.isgautomation.com/siemens-simatic-s7-300-plc-6es7.html.
[2] https://www.steinerelectric.com/p/siemens-simatic-s7-s7-plcsim-v5-4-floating-lic/429647.
[3] https://github.com/mbuesch/awlsim.

4.4 Results

The creation of the cost-effective digital twin model for the specific ICS environment starts with processing the generic test specification (TD) and the specific topology inputs ($Topology = <E, Communication>$). This is done by applying the data processing algorithm presented in Algorithm 1 on each combination of element $e \in E$ and test $t \in T$. The output of this algorithm produces 42 different tests (presented in Appendix D), each of which includes a set of environment dependencies (ED). The environment dependencies are specific to the ICS environment described in Sect. A.

Next, given the specific budgetary limitations, we apply the Problem Builder module and create the 0–1 non-linear programming problem (A formal representation of the problem is presented in Appendix A). We implemented a naive brute force algorithm to find the optimal configuration for a given budgetary limitation.

We conducted an experiment in which we derived the configuration of a digital twin model for different budgetary limitation values, while considering all of the tests presented in Appendix D (a total of 42 tests).

The results of this experiment are presented in Fig. 1. As expected, the higher the available budget the higher the impact of the digital twin.

In this figure, it can also be seen that when $Budget = \$3700$, all of the elements are implemented as virtual devices; in this case, the digital twin model supports 36 tests of the 42 possible tests. The remaining tests require physical implementation of various elements; in order to support all of the tests, the budget required is \$23500 (while the total cost of the industrial system is \$40000).

The results show a logarithmic increase of the benefit (impact) with the increase in the available budget.

5 Conclusions and Future Work

We present a method for deriving the specification of a digital twin for an ICS for the purpose of security analysis. The resulting specification is a cost-effective representation of the ICS under test that provides the high fidelity required for executing a given set of security tests. The method is designed as a three step process. First, the Data Processor derives the ICS's environment dependencies from its topology and the tests' dependencies. Then, the Problem Builder uses the ICS's architecture, tests' dependencies, and budgetary limitations to create a 0–1 non-linear programming problem representation. Finally, the Solver applies a search algorithm to find the best solution for the problem, i.e., finds the digital twin specification with the highest impact and an affordable cost (i.e., its implementation cost does not exceed the specified budget). To demonstrate the application of the proposed method, we used a simplified structure of a thermal power station and the NESCOR pen-testing methodology to define the tests and their requirements.

In future work we plan to evaluate the method on more realistic environments from a diverse range of industries and propose a heuristic algorithm for finding a

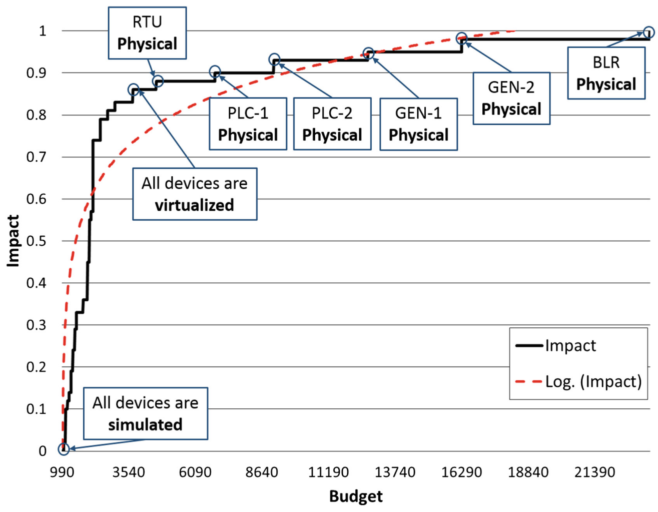

Fig. 1. The trade-off between the budget and the impact of the digital twin computed for the simple thermal power station.

near-optimal solution (digital twin setup) with sub-exponential time complexity. In addition, we plan to extend the solution to support different pricing strategies for the various implementations, such as software bundles with contribution margin-based pricing. We also plan to (1) add new types of constraints, e.g., constraints that take the physical space available within the digital twin that will be implemented (e.g., a small room or an open space) into account; (2) consider implementations of multiple elements as virtual or simulations on the same machine; and (3) handle identical setups in an industrial environment (e.g., if two similar production lines are implemented, there is no need to test both of them). Finally, a general knowledge base of possible tests and their test dependencies should be researched and established.

A Formal Representation

1. $C = \{PC, Server, Switch, Firewall, EWS, HMI, Historian, PLC, RTU, Generator, Boiler\}$
2. $N = \{PC, Server, Historian, HMI, EWS, PLC\}$
3. $M = \{PC, Server, Historian, HMI, EWS\}$
4. $NC = \{Switch, Firewall\}$
5. $F = \{Generator, Boiler\}$
6. $D = \{PLC, RTU\}$
7. $S = \{EWS, HMI, Historian\}$
8. $E = \{IT - Client, IT - Server, SW - 1, FW - 1, EWS - PC, HMI - PC, SW - 2, Hist - PC, PLC - 1, PLC - 2, RTU - 1, GEN - 1, GEN - 2, BLR\}$

9. $Communication = \{<IT-Client, IT-Server>, <IT-Server, IT-Client>, <Hist-PC, PLC-1>, <Hist-PC, PLC-2>, <HMI-PC, PLC-1>, <HMI-PC, PLC-2>, <EWS-PC, HMI-PC>, <EWS-PC, PLC-1>, <EWS-PC, PLC-2>, <PLC-1, GEN-1>, <PLC-1, BLR>, <PLC-2, GEN-2>, <RTU-1, PLC-1>, <RTU-1, PLC-2>\}$
10. $T = \{4.1.1, 4.2.1, 5.2.1, 5.2.4, 7.1.1, 7.1.2\}$.
11. $$role(e) = \begin{cases} PC, & e = IT-Client \\ Server, & e = IT-Server \\ Switch, & e \in \{SW-1, SW-2\} \\ Firewall, & e = FW-1 \\ EWS, & e = EWS-PC \\ HMI, & e = HMI-PC \\ Historian, & e = Hist-PC \\ PLC, & e \in \{PLC-1, PLC-2\} \\ RTU, & e = RTU-1 \\ Generator, & e \in \{GEN-1, GEN-2\} \\ Boiler, & e = BLR \end{cases}$$
12. The *cost* function is defined in Appendix B.

B Implementation Costs of the ICS Components (USD)

	p	v	s
$IT-Client$	1000	100	30
$IT-Server$	4000	100	30
$SW-1$	3000	300	90
$SW-2$	3000	300	90
$FW-1$	4000	400	120
$EWS-PC$	1000	100	30
$HMI-PC$	1000	100	30
$Hist-PC$	1000	100	30
$PLC-1$	2500	250	75
$PLC-2$	2500	250	75
$RTU-1$	1000	100	30
$GEN-1$	4000	400	120
$GEN-2$	4000	400	120
BLR	8000	800	120

C Specification of Penetration Testing Activities Based on NESCOR Methodology

Category	Subcategory	ID	Name	T	DIR	EIR	PT
Embedded Device	Electronic Component	4.1.1	Device Disassembly	$f \in F$	f^p	$\{\}$	
		4.1.2	Circuit Analysis	$f \in F$	f^p	$\{\}$	4.1.1
		4.1.3	Datasheet Analysis	$f \in F$	f^s	$\{\}$	4.1.2
		4.1.4	Dumping Embedded Data	$f \in F$	f^p	$\{\}$	4.1.3
		4.1.5	Bus Snooping	$f \in F$	f^p	$\{\}$	4.1.3
		4.1.6	String Analysis	$f \in F$	f^p	$\{\}$	4.1.4,4.1.5
		4.1.7	Entropy Analysis	$f \in F$	f^p	$\{\}$	4.1.4,4.1.5
		4.1.8	Systematic Key Search	$f \in F$	f^p	$\{\}$	4.1.4,4.1.5
		4.1.9	Data Decoding	$f \in F$	f^p	$\{\}$	4.1.6,4.1.7,4.1.8
		4.1.10	Embedded Hardware Exploitation	$f \in F$	f^p	$\{\}$	4.1.9
		4.1.1	Device Disassembly	$d \in D$	d^p	$\{\}$	
		4.1.2	Circuit Analysis	$d \in D$	d^p	$\{\}$	4.1.1
		4.1.3	Datasheet Analysis	$d \in D$	d^s	$\{\}$	4.1.2
		4.1.4	Dumping Embedded Data	$d \in D$	d^p	$\{\}$	4.1.3
		4.1.5	Bus Snooping	$d \in D$	d^p	$\{\}$	4.1.3
		4.1.6	String Analysis	$d \in D$	d^p	$\{\}$	4.1.4,4.1.5
		4.1.7	Entropy Analysis	$d \in D$	d^p	$\{\}$	4.1.4,4.1.5
		4.1.8	Systematic Key Search	$d \in D$	d^p	$\{\}$	4.1.4,4.1.5
		4.1.9	Data Decoding	$d \in D$	d^p	$\{\}$	4.1.6,4.1.7,4.1.8
		4.1.10	Embedded Hardware Exploitation	$d \in D$	d^p	$\{\}$	4.1.9
	Technician Interface	4.2.1	Interface Functional Analysis	$f \in F$	f^e	$\{D^e\}$	
		4.2.2	Communication Capture	$f \in F$	f^e	$\{D^e\}$	4.2.1
		4.2.3	Capture Analysis	$f \in F$	f^e	$\{D^e\}$	4.2.2
		4.2.4	Endpoint Impersonation	$f \in F$	f^e	$\{D^e\}$	4.2.3
		4.2.5	Endpoint Fuzzing	$f \in F$	f^e	$\{D^e\}$	4.2.3
		4.2.6	Exploitation	$f \in F$	f^e	$\{D^e\}$	4.2.4,4.2.5
		4.2.1	Interface Functional Analysis	$d \in D$	d^e	$\{D^e, F^s, S^e\}$	
		4.2.2	Communication Capture	$d \in D$	d^e	$\{D^e, F^s, S^e\}$	4.2.1
		4.2.3	Capture Analysis	$d \in D$	d^e	$\{D^e, F^s\}$	4.2.2
		4.2.4	Endpoint Impersonation	$d \in D$	d^e	$\{D^e, F^s, S^e\}$	4.2.3
		4.2.5	Endpoint Fuzzing	$d \in D$	d^e	$\{D^e, F^s, S^e\}$	4.2.3
		4.2.6	Exploitation	$d \in D$	d^e	$\{D^e, F^s, S^e\}$	4.2.4,4.2.5

(continued)

(*continued*)

Category	Subcategory	ID	Name	T	DIR	EIR	PT
	Firmware Binary	4.3.1	Disassembly	$f \in F$	f^p	$\{\}$	
		4.3.2	Code Analysis	$f \in F$	f^p	$\{\}$	4.3.1
		4.3.3	Exploitation	$f \in F$	f^p	$\{\}$	4.3.2
		4.3.1	Disassembly	$d \in D$	d^p	$\{\}$	
		4.3.2	Code Analysis	$d \in D$	d^p	$\{\}$	4.3.1
		4.3.3	Exploitation	$d \in D$	d^p	$\{\}$	4.3.2
Network	Protocol Analysis	5.2.1	Communication Capture	$n \in N$	n^e	$\{N^e\}$	
		5.2.2	Cryptographic Analysis	$n \in N$	n^e	$\{N^e\}$	5.2.1
		5.2.3	Unknown Protocol Decoding	$n \in N$	n^e	$\{N^e\}$	5.2.2
		5.2.4	Fuzzing	$n \in N$	n^e	$\{N^e\}$	5.2.1
		5.2.5	Exploitation	$n \in N$	n^e	$\{N^e\}$	5.2.4
Server OS	Information Gathering	6.1.1	DNS Interrogation	$m \in M$	m^e	$\{\}$	
		6.1.2	Port Scanning	$m \in M$	m^e	$\{\}$	
		6.1.3	Service Fingerprinting	$m \in M$	m^e	$\{\}$	6.1.2
		6.1.4	SNMP Enumeration	$m \in M$	m^e	$\{\}$	6.1.3
		6.1.5	Packet Sniffing	$m \in M$	m^e	$\{M^e, D^e\}$	6.1.4
		6.1.2	Port Scanning	$n \in NC$	n^e	$\{\}$	
		6.1.3	Service Fingerprinting	$n \in NC$	n^e	$\{\}$	6.1.2
		6.1.5	Packet Sniffing	$n \in NC$	n^e	$\{\}$	6.1.4
	Vulnerability Analysis	6.2.1	Unauthenticated Vulnerability Scanning	$m \in M$	m^e	$\{\}$	6.1.4
		6.2.2	Authenticated Vulnerability Scanning	$m \in M$	m^e	$\{\}$	6.1.4
		6.2.3	Vulnerability Validation	$m \in M$	m^e	$\{\}$	6.2.1,6.2.2
		6.2.4	Packet Capture Analysis	$m \in M$	m^e	$\{M^e, D^e\}$	6.1.5
		6.2.1	Unauthenticated Vulnerability Scanning	$n \in NC$	n^e	$\{\}$	6.1.4
		6.2.2	Authenticated Vulnerability Scanning	$n \in NC$	n^e	$\{\}$	6.1.4
		6.2.3	Vulnerability Validation	$n \in NC$	n^e	$\{\}$	6.2.1,6.2.2
		6.2.4	Packet Capture Analysis	$n \in NC$	n^e	$\{\}$	6.1.5
	Exploitation	6.3.1	Identify Attack Avenues	$m \in M$	m^e	$\{M^e, D^e\}$	6.1,6.2
		6.3.2	Vulnerability Exploitation	$m \in M$	m^e	$\{M^e, D^e\}$	6.3.1
		6.3.3	Post Exploitation	$m \in M$	m^e	$\{M^e, D^e\}$	6.3.2
		6.3.1	Identify Attack Avenues	$n \in NC$	n^e	$\{\}$	6.1,6.2
		6.3.2	Vulnerability Exploitation	$n \in NC$	n^e	$\{\}$	6.3.1
		6.3.3	Post Exploitation	$n \in NC$	n^e	$\{\}$	6.3.2
Server Applications	Application Mapping	7.1.1	Application Fingerprinting	$m \in M$	m^e	$\{\}$	
		7.1.2	Functional Analysis	$m \in M$	m^e	$\{M^e, D^s\}$	7.1.1
		7.1.3	Process Flow Modeling	$m \in M$	m^e	$\{M^e, D^s\}$	7.1.2
		7.1.4	Request/Resource Mapping	$m \in M$	m^e	$\{M^e, D^s\}$	7.1.3
	Application Discovery	7.2.1	Default Configuration Testing	$m \in M$	m^e	$\{\}$	
		7.2.2	Authentication Testing	$m \in M$	m^e	$\{M^e, D^s\}$	
		7.2.3	Session Management Testing	$m \in M$	m^e	$\{M^e, D^s\}$	7.2.2
		7.2.4	Authorization Testing	$m \in M$	m^e	$\{M^e, D^s\}$	7.2.3
		7.2.5	Business Logic Testing	$m \in M$	m^e	$\{M^e, D^s\}$	
		7.2.6	Code Injection Testing	$m \in M$	m^e	$\{M^e, D^s\}$	
		7.2.7	Denial of Service Testing	$m \in M$	m^e	$\{M^e, D^s\}$	
		7.2.8	Client-Side Code Testing	$m \in M$	m^e	$\{M^e, D^s\}$	
	Application Exploitation	7.3.1	Identify Attack Avenues	$m \in M$	m^e	$\{M^e, D^s\}$	7.1,7.2
		7.3.2	Vulnerability Exploitation	$m \in M$	m^e	$\{M^e, D^s\}$	7.3.2
		7.3.3	Post Exploitation	$m \in M$	m^e	$\{M^e, D^s\}$	7.3.3

D Environment Dependencies

ID	Test	Element	The list of environment dependencies
1	5.2.1	IT-client	$\{IT-client^e, IT-server^e, SW-1^e\}$
2	5.2.4	IT-client	$\{IT-client^e, IT-server^e, SW-1^e\}$
3	7.1.1	IT-client	$\{IT-client^e\}$
4	7.1.2	IT-client	$\{IT-client^e, IT-server^e\}$
5	5.2.1	IT-server	$\{IT-server^e, IT-client^e, SW-1^e\}$
6	5.2.4	IT-server	$\{IT-server^e, IT-client^e, SW-1^e\}$
7	7.1.1	IT-server	$\{IT-server^e\}$
8	7.1.2	IT-server	$\{IT-server^e, IT-client^e\}$
9	5.2.1	SW-1	$\{SW-1^e, IT-server^e, IT-client^e, SW-2^e, FW-1^e\}$
10	5.2.4	SW-1	$\{SW-1^e, IT-server^e, IT-client^e, SW-2^e, FW-1^e\}$
11	5.2.1	SW-2	$\{SW-2^e, SW-1^e, FW-1^e, HMI-PC^e, EWS-PC^e, PLC-1^e, PLC-2^e\}$
12	5.2.4	SW-2	$\{SW-2^e, SW-1^e, FW-1^e, HMI-PC^e, EWS-PC^e, PLC-1^e, PLC-2^e\}$
13	5.2.1	EWS-PC	$\{EWS-PC^e, PLC-1^e, PLC-2^e, SW-2^e\}$
14	5.2.4	EWS-PC	$\{EWS-PC^e, PLC-1^e, PLC-2^e, SW-2^e\}$
15	7.1.1	EWS-PC	$\{EWS-PC^e\}$
16	7.1.2	EWS-PC	$\{EWS-PC^e, PLC-1^s, PLC-2^s\}$
17	5.2.1	HMI-PC	$\{HMI-PC^e, PLC-1^e, PLC-2^e, SW-2^e\}$
18	5.2.4	HMI-PC	$\{HMI-PC^e, PLC-1^e, PLC-2^e, SW-2^e\}$
19	7.1.1	HMI-PC	$\{HMI-PC^e\}$
20	7.1.2	HMI-PC	$\{HMI-PC^e, PLC-1^s, PLC-2^s\}$
21	5.2.1	Hist-PC	$\{Hist-PC^e, PLC-1^e, PLC-2^e, SW-2^e\}$
22	5.2.4	Hist-PC	$\{Hist-PC^e, PLC-1^e, PLC-2^e, SW-2^e\}$
23	7.1.1	Hist-PC	$\{Hist-PC^e\}$
24	7.1.2	Hist-PC	$\{Hist-PC^e, PLC-1^s, PLC-2^s\}$
25	4.1.1	PLC-1	$\{PLC-1^p\}$
26	4.2.1	PLC-1	$\{PLC-1^e, RTU-1^e, BLR^s, GEN-1^s, HMI-PC^e, EWS-PC^e, Hist-PC^e\}$
27	5.2.1	PLC-1	$\{PLC-1^e, RTU-1^e, HMI-PC^e, EWS-PC^e, Hist-PC^e, SW-2^e\}$
28	5.2.4	PLC-1	$\{PLC-1^e, RTU-1^e, HMI-PC^e, EWS-PC^e, Hist-PC^e, SW-2^e\}$
29	4.1.1	PLC-2	$\{PLC-2^p\}$
30	4.2.1	PLC-2	$\{PLC-2^e, RTU-1^e, GEN-2^s, HMI-PC^e, EWS-PC^e, Hist-PC^e\}$
31	5.2.1	PLC-2	$\{PLC-2^e, RTU-1^e, HMI-PC^e, EWS-PC^e, Hist-PC^e, SW-2^e\}$
32	5.2.4	PLC-2	$\{PLC-2^e, RTU-1^e, HMI-PC^e, EWS-PC^e, Hist-PC^e, SW-2^e\}$
33	4.1.1	RTU-1	$\{RTU-1^p\}$
34	4.2.1	RTU-1	$\{RTU-1^e, PLC-1^e, PLC-2^e\}$
35	5.2.1	RTU-1	$\{RTU-1^e, PLC-1^e, PLC-2^e\}$
36	5.2.4	RTU-1	$\{RTU-1^e, PLC-1^e, PLC-2^e\}$
37	4.1.1	GEN-1	$\{GEN-1^p\}$
38	4.2.1	GEN-1	$\{GEN-1^e, PLC-1^e\}$
39	4.1.1	GEN-2	$\{GEN-2^p\}$
40	4.2.1	GEN-2	$\{GEN-2^e, PLC-2^e\}$
41	4.1.1	BLR	$\{BLR^p\}$
42	4.2.1	BLR	$\{BLR^e, PLC-1^e\}$

E Illustrations

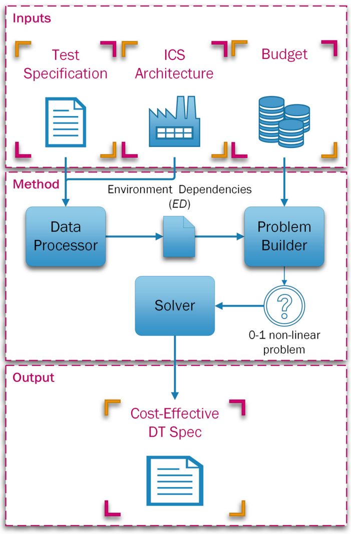

Fig. 2. An illustration of the proposed method.

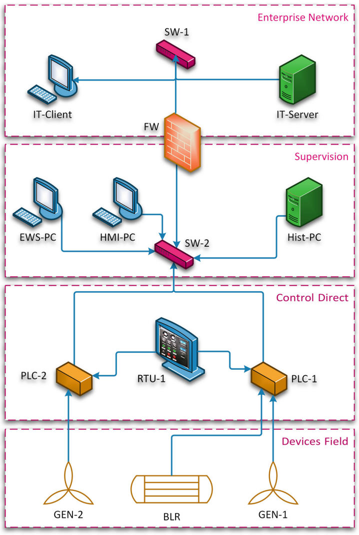

Fig. 3. Simple thermal power station environment.

References

1. Cyber security assessment of industrial control systems - a good practice guide. Technical report, Centre for the Protection of National Infrastructure, April 2011
2. Alves, T., Das, R., Morris, T.: Virtualization of industrial control system testbeds for cybersecurity, pp. 10–14. ACM
3. Gao, H., Peng, Y., Dai, Z., Wang, T., Jia, K.: The design of ICS testbed based on emulation, physical, and simulation (EPS-ICS testbed). In: 2013 Ninth International Conference on Intelligent Information Hiding and Multimedia Signal Processing, pp. 420–423. IEEE (2013)
4. Genge, B., Siaterlis, C., Fovino, I.N., Masera, M.: A cyber-physical experimentation environment for the security analysis of networked industrial control systems. Comput. Electr. Eng. **38**(5), 1146–1161 (2012)
5. Green, B., Lee, A., Antrobus, R., Roedig, U., Hutchison, D., Rashid, A.: Pains, gains and PLCs: ten lessons from building an industrial control systems testbed for security research. In: 10th USENIX Workshop on Cyber Security Experimentation and Test (CSET 2017). USENIX Association, Vancouver (2017)
6. Grieves, M., Vickers, J.: Digital twin: mitigating unpredictable, undesirable emergent behavior in complex systems. In: Kahlen, F.-J., Flumerfelt, S., Alves, A. (eds.) Transdisciplinary Perspectives on Complex Systems, pp. 85–113. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-38756-7_4

7. Holm, H., Karresand, M., Vidström, A., Westring, E.: A Survey of Industrial Control System Testbeds. Springer, Cham (2015)
8. Lemay, A., Fernandez, J., Knight, S.: An isolated virtual cluster for SCADA network security research. In: Proceedings of the 1st International Symposium for ICS & SCADA Cyber Security Research, p. 88 (2013)
9. Leszczyna, R., Egozcue, E., Tarrafeta, L., Villar, V.F., Estremera, R., Alonso, J.: Protecting industrial control systems-recommendations for Europe and member states. Technical report (2011)
10. Li, D., Sun, X.: Nonlinear Integer Programming, vol. 84. Springer, Cham (2006)
11. McLaughlin, S., Konstantinou, C., Wang, X., Davi, L., Sadeghi, A.-R., Maniatakos, M., Karri, R.: The cybersecurity landscape in industrial control systems. Proc. IEEE **104**(5), 1039–1057 (2016)
12. Mitchell, R., Chen, I.-R.: A survey of intrusion detection techniques for cyber-physical systems. ACM Comput. Surv. (CSUR) **46**(4), 55 (2014)
13. Murray, W., Ng, K.-M.: An algorithm for nonlinear optimization problems with binary variables. Comput. Optim. Appl. **47**(2), 257–288 (2010)
14. Nazir, S., Patel, S., Patel, D.: Assessing and augmenting SCADA cyber security: a survey of techniques. Comput. Secur. **70**, 436–454 (2017)
15. Qassim, Q., et al.: A survey of SCADA testbed implementation approaches. Indian J. Sci. Technol. **10**, 26 (2017)
16. Searle, J.: NESCOR guide to penetration testing for electric utilities. Technical report, National Electric Sector Cybersecurity Organization Resource (NESCOR)
17. Siaterlis, C., Genge, B.: Cyber-physical testbeds. Commun. ACM **57**(6), 64–73 (2014)
18. Stouffer, K., Falco, J., Scarfone, K.: Guide to industrial control systems (ICS) security. NIST Spec. Publ. **800**(82), 16 (2011)