BayesImposter: Bayesian Estimation Based .bss Imposter Attack on Industrial Control Systems

Anomadarshi Barua
Department of Electrical Engineering and Computer Science
University of California, Irvine
California, USA.
anomadab@uci.edu

Lelin Pan
Department of Electrical Engineering and Computer Science
University of California, Irvine
California, USA.
lelinp@uci.edu

Mohammad Abdullah Al Faruque
Department of Electrical Engineering and Computer Science
University of California, Irvine
California, USA.
alfaruqu@uci.edu

ABSTRACT

Over the last six years, several papers used memory deduplication to trigger various security issues, such as leaking heap-address and causing bit-flip in the physical memory. The most essential requirement for successful memory deduplication is to provide identical copies of a physical page. Recent works use a brute-force approach to create identical copies of a physical page that is an inaccurate and time-consuming primitive from the attacker's perspective.

Our work begins to fill this gap by providing a domain-specific structured way to duplicate a physical page in cloud settings in the context of industrial control systems (ICSs). Here, we show a new attack primitive - *BayesImposter*, which points out that the attacker can duplicate the .bss section of the target control DLL file of cloud protocols using the *Bayesian estimation* technique. Our approach results in less memory (i.e., 4 KB compared to GB) and time (i.e., 13 minutes compared to hours) compared to the brute-force approach used in recent works. We point out that ICSs can be expressed as state-space models; hence, the *Bayesian estimation* is an ideal choice to be combined with memory deduplication for a successful attack in cloud settings. To demonstrate the strength of *BayesImposter*, we create a real-world automation platform using a scaled-down automated high-bay warehouse and industrial-grade SIMATIC S7-1500 PLC from Siemens as a target ICS. We demonstrate that *BayesImposter* can predictively inject false commands into the PLC that can cause possible equipment damage with machine failure in the target ICS. Moreover, we show that *BayesImposter* is capable of adversarial control over the target ICS resulting in severe consequences, such as killing a person but making it looks like an accident. Therefore, we also provide countermeasures to prevent the attack.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**.

KEYWORDS

PLCs, DLL file, bayesian estimation, adversarial control

ACSAC '22, December 5–9, 2022, Austin, TX, USA

ACM ISBN 978-1-4503-9759-9/22/12.
https://doi.org/10.1145/3564625.3564638

ACM Reference Format:
Anomadarshi Barua, Lelin Pan, and Mohammad Abdullah Al Faruque. 2022. BayesImposter: Bayesian Estimation Based .bss Imposter Attack on Industrial Control Systems . In *Annual Computer Security Applications Conference (ACSAC '22), December 5–9, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3564625.3564638

1 INTRODUCTION

Historically, Industrial Control Systems (ICSs) follow the ANSI/ISA 95 model [65], where *disconnected* computer systems and *isolated* sensor frameworks were used to screen various operations and tasks in lower *levels* of the *automation pyramid* [20]. As we enter the fourth industrial revolution [51] (Industry 4.0), the ANSI/ISA95 model is going under different transformations. These transformations include the vertically/horizontally *interconnected* and *decentralized* ICSs in all levels of the *automation pyramid* for flexible monitoring and control. The decentralization of ICSs in Industry 4.0 adds fuel to movement to the Industrial Internet of Things (IIoT) trend, where *cloud servers* and *virtualization* [74] play an important role by providing easy-to-access automation platforms.

In Industry 4.0, Infrastructure-as-a-Service (IaaS) enables Programmable Logic Controllers (PLCs) to connect with clouds [48]. Moreover, to support multiple PLCs and supervisory platforms, today's ICSs use multiple Virtual Private Servers (VPSs) in a single cloud platform [38]. The cloud server has memory deduplication feature enabled [33], which is a *widespread optimizing* feature present in today's cloud servers to support virtualization. In this typical ICS platform, the user sends control programming and supervisory commands from VPSs using cloud protocols (i.e., MQTT, AMQP) to PLCs [49]. The cloud protocol's software stack has a specific DLL file, which transports these commands and is located in the server computer. We call this specific DLL file as *target control DLL* file.

In this paper, at first, we show that *the .bss section* of the target control DLL file of cloud protocols transports the critical control commands from VPSs to PLCs (i.e., lower level of the automation pyramid). Next, after identifying the target control DLL file, we introduce the *Bayesian estimation* by which an attacker can recreate or fake the memory page of the .bss section of the target control DLL file. We name the fake .bss section[1] as the *.bss imposter* and denote the attack model by *BayesImposter*.

The intuition behind *BayesImposter* is that as ICSs can be expressed as state-space models [35], our *BayesImposter* exploits the *Bayesian estimation* technique to accurately predict the current state of the industrial controller. As control commands are directly

related to the current states of the industrial controller, after estimating the states, the attacker can also estimate the control commands from the estimated states. As the .bss section contains the control commands, hence, the attacker can successfully recreate the .bss section using the estimated control commands. We show that our proposed *Bayesian estimation* results in less memory and attack time to recreate the page of the *.bss imposter* compared to the brute force approach demonstrated in recent works [19, 29, 58, 62].

After recreating the fake .bss section, *BayesImposter* uses the underlying memory deduplication feature enabled in the cloud to merge the page of the fake .bss section with the legitimate .bss section. In this way, the attacker can locate the memory address of the fake .bss section in the host machine and can use a malicious *co-located VPS* to trigger a bit-flip in the page of the .bss section using the Rowhammer bug [19, 29, 58, 62] of the host machine. As the .bss section contains the control commands, this paper shows that a bit flip in this section may cause corruption or even change the actual command. This method can be termed as *false command injection*. The injected false commands propagate from VPSs to the PLCs and may cause an unplanned behavior with catastrophic machine failure in the target ICS. It is worthwhile to mention here that, as *BayesImposter* has more control over the recreation of a fake .bss section, our attack is capable of *adversarial control* over the target ICS from a *co-located VPS* on the same cloud. To the best of our knowledge, *BayesImposter* is the first work that successfully merges the idea of *Bayesian estimation* of the state-space models of ICSs with the memory deduplication and the Rowhammer bug in cloud settings in the context of ICSs.

Technical Contributions: Our contributions are:

- We are the first to point out how the .bss section of the target control DLL file of cloud protocols can be exploited by using memory deduplication in modern ICSs.
- We are the first to introduce Bayesian estimation to recreate the .bss section. Our attack requires less memory and time compared to the brute force approach used in recent works [19, 29, 58, 62].
- We create a real-world scaled-down factory model of a practical ICS, which has an automated high-bay warehouse from *fischertechnik* [6]. We use an industrial-grade PLC with a part# SIMATIC S7-1500 [12] from Siemens to create the automation platform and connect the PLC to clouds using industry-standard cloud protocols.
- We evaluate *BayesImposter* in our factory model considering five variants of industry-standard cloud protocols and show the adversarial control to generalize our attack model in cloud settings.

The demonstration of our work is shown in the following link: https://sites.google.com/view/bayesmem/home.

2 BACKGROUND

2.1 Connecting PLCs with clouds

IIoT enables PLCs to upload the acquired data directly to clouds [64]. PLCs are connected to clouds normally in two ways: using an adapter or directly using a standard protocol. Standard cloud protocols, such as MQTT and AMQP support *bidirectional* and *event-based* data transmission between PLCs and upper managements.

[1]In this paper, *the .bss section* means the .bss section of the target control DLL file of cloud protocols; unless otherwise mentioned.

The upper management can modify control functions of PLCs in run-time by flashing new *control programs* to PLCs from clouds.

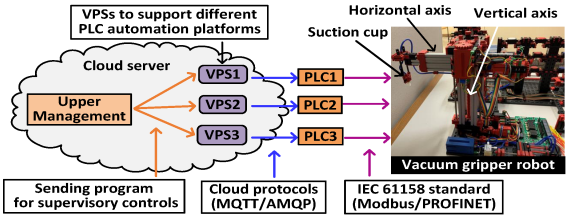

Figure 1: Different components of an ICS in cloud settings.

2.2 Programs for supervisory controls

The IEC 61131 programming standard [72] is used for control programming of PLCs. Control programs can be broadly divided into three categories: (i) programs for basic functions, (ii) programs for supervisory controls, and (iii) programs for critical time-constraint functions (e.g., security and real-time response, etc.). Traditionally, all these three categories of control programs were implemented in PLCs in industrial premises. However, with the new trend in Industry 4.0, nowadays, only the programs for critical time-constraint functions are implemented in PLCs. Programs for basic functions and supervisory controls are not implemented in PLCs; rather, they are implemented in clouds or in web-server. For example, basic functions and supervisory control programs are outsourced as web services to a cloud or to a server for class C33 PLC controller [49]. *This gives more flexibility to upper managements as they can change programs remotely in run-time to tackle abruptly changing situations.*

2.3 Use of VPSs with PLCs

ICSs are becoming more complex in Industry 4.0. ICSs often need to support multiple automation platforms that may conflict with each other. Moreover, multiple PLC controllers and supervisory platforms may need multiple software packages that may require multiple operating systems. Also, introducing web servers and clouds to ICSs increases the necessity of using multiple private servers. As using multiple separate physical machines to support multiple automation platforms or operating systems or private servers is one of the available solutions, industries evidently use VPSs to reduce the number of required physical machines to reduce cost [63]. Moreover, modern cloud platforms offer cheap access to VPSs by sharing a single server among multiple operating systems on a single server machine using virtualization software [11].

2.4 A motivational example of an ICS

A motivational example is shown in Fig. 1 where we consider an automated high-bay warehouse as our example ICS. It has a vacuum gripper robot, which stores objects in the storage rack of the warehouse using a suction cup and moves along the horizontal and vertical axis. We elaborate more on this in Section 7.1 while demonstrating our attack model. Here, multiple PLCs having different platforms are supported by a cloud using multiple VPSs. Upper management located in the cloud send programs for supervisory controls from VPSs to PLCs using cloud protocols (i.e., MQTT/AMQP). PLCs communicate with the underlying sensors and controllers using IEC 61158 standard protocols (e.g., Modbus, PROFINET, etc.). Given this background, an attacker can perturb

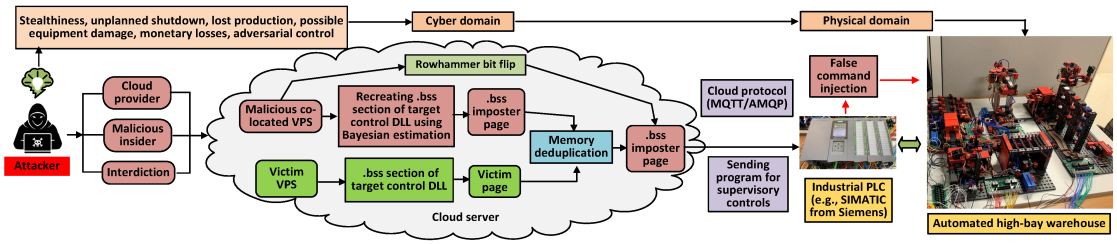

Figure 2: Different components of our attack model - *BayesImposter* on industrial control systems in cloud settings.

the supervisory control commands (i.e., false command injection) in our example ICS and remotely hamper its normal operation using our attack model - *BayesImposter*.

2.5 Memory deduplication

Memory deduplication is a process that merges identical pages in the physical memory into one page to reduce redundant pages having similar contents. It is a widely used feature in cloud servers allowing multiple VPSs to run on less allocated memory in a single physical machine. The amount of redundant pages can be as high as 86% [30] and memory deduplication can save up to 50% of the allocated memory in the cloud server [42]. This feature is available in Windows 8.1, Windows Server 2016, 2019, and 2022 and Linux distribution. Windows Servers have it as Data Deduplication [3] and Linux distributions have it as Kernel Samepage Merging (KSM), which is implemented in Kernel-based Virtual Machine (KVM) (see Appendix 11.5, 11.6, and 11.7 for more detail on this topic).

3 ATTACK MODEL

Fig. 2 shows the attack model - *BayesImposter* in cloud settings. The essential components of *BayesImposter* are described below.

(i) Target system: We consider an infrastructure [39] where PLCs are connected with a cloud for maintenance and control programming, and multiple Virtual Machines (VMs) acting as VPSs are located in the same cloud to support multiple automation platforms. As multiple VPSs in the same cloud share the same hardware, an attacker can exploit the shared hardware from a co-located VPS.

(ii) Attacker's capabilities: Let us consider a scenario where a user gives commands from his proprietary VPS to a PLC to do control programming and supervisory controls.

• **.bss imposter:** A few specific DLL files (i.e., target control DLL) of the cloud protocols transport these commands from VPS to PLCs. These DLL files are organized into different *sections*. Each section can be writable or read-only and can encapsulate executable (i.e., code) or non-executable (i.e., data) information. The section, which encapsulates uninitialized data, is known as *.bss* section. The .bss section of the target control DLL contains control programming and supervisory control specific information/data, which are mostly *boolean* type coming from the user as commands. This .bss section is page-aligned in virtual memory as well as in physical memory. Let us denote this as *victim page*. If an attacker can recreate the *victim page*, the attacker can use this *recreated victim page* (a.k.a., *.bss imposter page*) to trigger memory deduplication.

• **Bottleneck:** To recreate the *victim page*, the attacker needs to guess all the initialization values of uninitialized variables of the .bss section. As there could be hundreds of control variables present in the .bss section, this is **almost impossible** for the attacker to successfully guess the *victim page* and recreate it following the brute force approach adopted in recent works [19, 29, 58, 62]. The brute force approach was successful in [19, 29, 58, 62] because they only guessed a specific 32-bit data to recreate a *victim page*. To guess hundreds of variables in the .bss section, the brute force approach could require hundreds of hours. Moreover, the attacker may need to spray the physical memory with terabyte amount of recreated pages to initiate a successful attack in the brute-force approach.

• **Solution:** Thankfully this challenge can be handled by using *BayesImposter*. The intuition behind *BayesImposter* is that if an attacker knows the state-space model of the ICS, the attacker can estimate the boolean and non-boolean control commands because the control commands are directly correlated with the current states of an ICS. As the .bss section transports the control commands, the estimation of the control commands helps the attacker to successfully guess the control variables present in the .bss section leading to a successful recreation of the *victim page* (i.e., *.bss imposter page*).

• **Memory deduplication + Rowhammer:** After recreating the .bss imposter page using our *BayesImposter*, the attacker can initiate memory deduplication to merge the *victim page* with the attacker's provided *.bss imposter page*. In this way, the attacker maps the *victim page* in his address space to initiate the Rowhammer attack on the *.bss imposter page* from his address space. It can flip bits in the *.bss imposter page* and change values of control commands.

(iii) Outcomes of the attack: As the .bss section contains important data dedicated to control programming and supervisory controls, the bit flips in the .bss section may lead to potential failure in ICSs. It can cause an unplanned shutdown, possible equipment damage, catastrophic machine failure, monetary losses, or even can kill a person but making it looks like an accident in the target ICS.

(iv) Attacker's access level: Our attack requires the deployment of a malicious co-located VPS in the cloud where the victim VPS resides. As public clouds are not common in ICSs, the clouds in ICSs can be either private or hybrid. The access needed to private or hybrid clouds can be possible in at least three scenarios.

In the first scenario, the attack can be originated from the cloud provider targeting the VPS of cloud users [61]. As cloud providers provide software, platform, and infrastructure as service [16], they have physical access to target clouds where the victim VPS resides.

In the second scenario, a malicious insider [31, 75], which can be a disgruntled employee, can use his insider knowledge of the system to deploy the malicious co-located VPS. A similar incident is found in the literature where a disgruntled ex-employee of an ICS posted a note in a hacker journal indicating that his insider knowledge of the system could be used to shut down that ICS [69].

The third scenario is interdiction, which has been rumored to be used in the past [17, 67, 73] and has been recently proven to be practically feasible [70]. In this scenario, during interdiction, a competitor can intercept the installation of VPS in clouds while providing service and may deploy the malicious VPS.

(v) Stealthy attack: The authorities may not be aware of the co-located malicious VPS and would possibly not detect the source of our attack. In this sense, our attack is stealthy and can alter the normal behavior of PLCs in ICSs while remaining unidentified.

(vi) Attacker's cost: Most of these specific DLLs are available as open-source, and very few are proprietary. To acquire the open-source DLL files, the attacker has a zero cost. To acquire the DLL files of the proprietary cloud protocols, the attacker just needs to buy a basic commercial license that may cost a minimum of $100 [1]. Moreover, most proprietary cloud protocols have a free evaluation for few days, and the attacker can also use this free evaluation period to access the .bss section of the target control DLL.

4 .BSS SECTION OF TARGET CONTROL DLL

To recreate the .bss imposter page, the attacker first needs to find the target control DLL file of cloud protocols (i.e., MQTT, AMQP) that transports the control commands from the VPS to PLCs.

4.1 Target control DLL file

Mostly, the name of the target control DLL file depends upon the cloud protocol's implementation variants. For example, the name of a popular implementation of MQTT cloud protocol is Mosquitto, and the target control DLL file for this variant to access by the attacker is mosquitto.dll. We do an exhaustive search and tabulate five popular variants of MQTT and their target control DLL files in Table 1. The same approach is equally applicable to other cloud protocols. The DLL files are located in the parent directory of the installation folder in the cloud.

Table 1: Target control DLL file of cloud protocol variants

Sl.	Cloud protocol variants	Target control DLL
1	EMQ X Broker [4]	erlexec.dll
2	Mosquitto [9]	mosquitto.dll
3	MQTT-C [10]	mqtt_pal.dll
4	eMQTT5 [5]	MQTT_client.dll
5	wolfMQTT [13]	MqttMessage.dll

4.2 Format of target control DLL files

In 64-bit Windows, DLL files follow Portable Executable 32+ (PE32+) format. In high level, PE32+ has a number of *headers* and *sections* (Fig. 4). The header consists of DOS header, PE header, optional header, section headers, and data directories. These headers have *Image base Address* and relative virtual address (RVA) of every section that tells the dynamic linker how to map every section of the DLL file into physical memory. There are different sections placed after headers in DLL. Among different sections in DLLs, we want to mention four sections, namely .rdata, .data, .text, and .bss sections. The .rdata section contains string literals, the .data section contains global/static initialized variables, the .text section contains machine code of the program, whereas the .bss section contains zero-initialized variables. It is important to note that all these sections are *page-aligned* [60]. This means that these sections must begin on a multiple of a page size in both virtual and physical memory. These sections of DLL files are mapped to pages in physical memory after the *base-relocation* [60]. The base-relocation is randomized, and the *ASLR technique* is used to map these sections to pages in physical memory at *load time* by the operating system.

4.3 Reasons for choosing the .bss section

The intention of the attacker is to find a section in the DLL file that has less entropy, which leads to a successful guess of the section. As the .rdata, the .data, and the .text sections consist of different unknown data and addresses, the pages in physical memory corresponding to these three sections have higher entropy. Hence, the estimation of these pages by the attacker requires large memory and time [19] that is not computationally feasible.

On the other hand, we examine that the .bss section of a target control DLL file of cloud protocols (i.e., MQTT, AMQP) is responsible for transporting control programming and supervisory control-related data, which are static except a new control command is issued. The .bss section contains different uninitialized global/static variables. They are also known as *tag values* and are organized in a tag table. The tag table is typically placed in the .bss section.

An example of the tag values: We use a real-world testbed of an automated high-bay warehouse from *fischertechnik*. The warehouse is connected with a SIMATIC S7-1500 PLC from Siemens. The PLC communicates with the cloud using a TIA portal [7] through the MQTT cloud protocol Mosquitto. A snippet of tag values in the tag table sent from the TIA portal to the SIMATIC PLC are shown in Fig. 3. A complete list of the tag values is provided in the following link: https://sites.google.com/view/bayesmem/home.

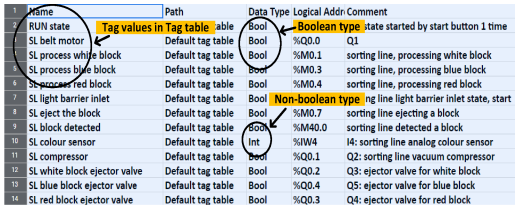

Figure 3: Tag values in tag table of the TIA portal.

If we analyze the tag values in tag tables (Fig. 3), we can observe that tag values correspond to particular states of the target ICS, e.g., the position of a vacuum gripper robot in the warehouse. Most of the tag values are boolean, and very few of them are other data types. The initialization of tag values to either *0 or 1* or non-boolean values in .bss section depends on states of the target ICS and increases entropy. Therefore, it provides a challenge to the attacker to successfully recreate the .bss section. Thankfully, this challenge can be handled by using the Bayesian estimation of specific command data in the .bss section. This process is discussed in the next section.

5 BAYESIAN ESTIMATION OF .BSS SECTION

We first mathematically model ICSs using the Bayesian estimation and then use the model to recreate the .bss imposter page.

Proposition 1- State-space model of an ICS: An ICS is dynamic in nature and can be expressed as a discrete-time state-space model [35]. Therefore, a control system in ICS can be expressed by a state vector x_k, which is a parameter of interest, and a measurement vector y_k, which is the measurement for x_k at discrete-time index k (see Fig. 4). The terms x_k and y_k can be expressed as:

$$x_k = f_{k-1}(x_{k-1}, q_{k-1}) = p(x_k|x_{k-1}) \quad (1)$$

$$y_k = h_k(x_k, r_k) = p(y_k|x_k) \quad (2)$$

where q_{k-1} and r_k are state noise and measurement noise vector respectively, and they are mutually exclusive. Please note that both x_k and y_k are stochastic processes, and Eqn. 1 implies that current state x_k at time index k depends on only the previous state x_{k-1} at time index $k-1$ (i.e., Markov process). We implement the state space model of ICS in lines 2-3 of our *BayesImposter* algorithm 1.

Source of the data to create the state-space model: To create the state-space model and to estimate x_k and y_k, the main challenge for the attacker is to gather the previous states, $x_{1:k-1}$ and previous measurements, $y_{1:k-1}$. *The attacker can gather $x_{1:k-1}$ and $y_{1:k-1}$ from OPC tags, historian data, specific PLC block information, or network traffic [31]. Moreover, as mentioned in Section 3, the cloud provider, or a malicious insider, or an interdiction method can make it possible to get $x_{1:k-1}$ and $y_{1:k-1}$ from these sources.* The attacker can use $x_{1:k-1}$ and $y_{1:k-1}$ to create a probabilistic graphical model - Bayes net, which is a directed acyclic graph describing how a joint density can be factorized. The Bayes net also illustrates conditional dependencies among all the states in the ICS (Fig. 4).

The tag values located in the .bss section are directly related to the current states (x_k) and measurements (y_k). Therefore, *BayesImposter* has the following two parts:

Part 1. Estimation of the current states (x_k) and measurements (y_k) of the state-space model.

Part 2. Estimation of tag values from the estimated x_k and y_k.

5.1 Estimation of states and measurements

At first, we define the univariate and multivariate ICS to provide background on the design space of the state-space model of ICSs.

Definition 1 (Univariate ICS). We define an univariate ICS as where each state x_k has only a single measurement quantity y_k at any time step k.

Definition 2 (Multivariate ICS). We define a multivariate ICS as where each state x_k has multiple (i.e., n number) measurement quantities, $[y_k^1, y_k^2, ..., y_k^n]$ at any time step k.

Practically speaking, an ICS is a mixture of univariate and multivariate state-space models. Therefore, the main challenge for the attacker is to satisfactorily estimate the current state x_k and measurement y_k for both univariate and multivariate ICSs. To handle this challenge, we bring Propositions 2 and 3 to estimate x_k and y_k for a univariate ICS and Propositions 4 and 5 for a multivariate ICS.

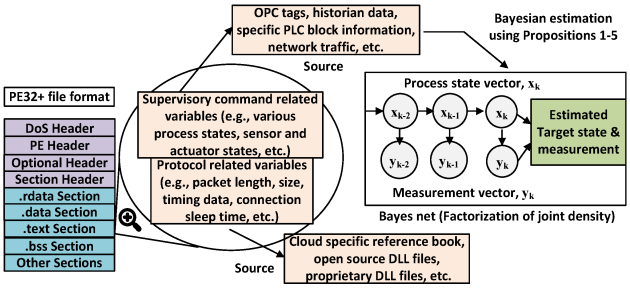

Figure 4: An overview of duplicating the .bss section of the target control DLL file.

Proposition 2: *BayesImposter* can predict the current state x_k at time k if the attacker has information only on the previous state x_{k-1} and previous measurements $y_{1:k-1}$, by using the Chapman-Kolmogorov equation. Here, $y_{1:k-1}$ consist of all previous measurement data $[y_1\ y_2\ ...\ y_{k-1}]$ up-to time $k-1$.

Explanation of Proposition 2: Let us give an example to clear this concept. Let us denote the *states* of a suction cup of the vacuum gripper robot in our example warehouse as x_k at time k. Let us consider the suction cup can be in one of two states, $x_k \epsilon$ {ON, OFF}. The activation of the suction cup in each state depends on the position of the horizontal and vertical axis of the vacuum gripper robot (see Fig. 1). The position measurement can be expressed by y_k at time k. If the attacker knows previous state x_{k-1} of the suction cup and previous position measurements $y_{1:k-1}$, then the attacker can use these data to accurately estimate the current state x_k at time k by using Eqn. 3 (i.e., Chapman-Kolmogorov equation). The L.H.S of Eqn. 3, $p(x_k|y_{1:k-1})$, is a conditional estimation of current state x_k, while previous measurements $y_{1:k-1}$ are given. The R.H.S of Eqn. 3 depicts that $p(x_k|y_{1:k-1})$ is a function of previous state, x_{k-1}, that is an indication of Markov process. The Proposition 2 is implemented in lines 6-7 of our *BayesImposter* algorithm 1.

$$p(x_k|y_{1:k-1}) = \int p(x_k|x_{k-1})p(x_{k-1}|y_{1:k-1})dx_{k-1} \quad (3)$$

An example: The name of a specific tag value in the .bss section of the mosquitto.dll is `suctionstate`, which corresponds to the state information $x_k \epsilon$ {ON, OFF} of the suction cup of our example automated high-bay warehouse. After estimating the state x_k using Eqn. 3, the attacker can initialize the tag value to 0 or 1 of the variable `suctionstate` in the .bss section. If the .bss section contains multiple uninitialized tag values originating in the VPS, the attacker can use a similar technique to successfully estimate all uninitialized tag values and can recreate the .bss section.

Proposition 3: *BayesImposter* can predict the current measurement y_k if the attacker has information on current state x_k.

Explanation of Proposition 3: It is important to note that along with state information x_k, the .bss section transports current measurement y_k from VPSs to PLCs. The importance of sending measurement information y_k from VPSs to PLCs is explained below.

An example: In the automated high-bay warehouse, a solenoid is present in the suction cup of the vacuum gripper robot that is turned on/off if the position of the horizontal and vertical axis is above/below a threshold position. Let us denote this threshold position by S_θ. If the threshold position is required to be changed by the upper management located in the cloud, the VPS can send a new threshold position S_k^θ to overwrite the previous value S_{k-1}^θ. The new threshold position S_k^θ is equivalent to the current measurement y_k, which depends on the current state x_k of the suction cup. Therefore, the current measurement, $y_k = S_k^\theta$, can be calculated using the Naive Bayes estimation equation as below:

$$p(y_k = S_k^\theta|x_k) = \frac{p(x_k|y_k = S_k^\theta) \times p(y_k = S_k^\theta)}{\sum_{y_k} p(y_k)p(x_k|y_k)} \quad (4)$$

Here, the likelihood term, $p(x_k|y_k = S_k^\theta)$, is calculated from the frequency distribution of the measurement y_k for the state x_k. The frequency distribution is calculated from the OPC tags and the historian data (Fig. 4). The prior probability, $p(y_k = S_k^\theta)$, is

the probability that the parameter takes on a particular value S_k^θ, prior to taking into account any new information (i.e., current state x_k). If the probability of the estimation, $p(y_k = S_k^\theta | x_k)$, is below a cut-off value (K_c), *BayesImposter* discards that estimation and picks another $y_k = S_k^\theta$ to test in Eqn. 4. By this way, the attacker can use *BayesImposter* to estimate any measurement quantity y_k at time step k. It is noteworthy that if the current state x_k is unknown, *BayesImposter* can use the Proposition 2 to calculate the current state x_k first, and then use the Proposition 3 to calculate $p(y_k|x_k)$ using Eqn. 4. The Proposition 3 is implemented in lines 9-17 of our proposed *BayesImposter* algorithm 1.

Proposition 4: If multiple (i.e., n) measurement quantities, [y_k^1, y_k^2, y_k^3,..., y_k^n], at a time step k, jointly contribute to estimate any state x_k, *BayesImposter* uses the joint probability of multiple measurement quantities, $p(y_k^1 \cap y_k^2 \cap y_k^3 \cap ..... \cap y_k^n)$, in Eqn. 3.

Explanation of Proposition 4: Let us assume that each state x_k in a multivariate ICS has n number of measurements at every time step. For example, at state x_1, the ICS has $y_1^1, y_1^2, y_1^3, ..., y_1^n$ measurement values; at state x_2, the ICS has $y_2^1, y_2^2, y_2^3, ..., y_2^n$ measurement values and so forth. Let us denote the joint probability of n number of measurement values at state x_k by $Y_k = p(y_k^1 \cap y_k^2 \cap y_k^3 \cap ..... \cap y_k^n)$. Eqn. 3 is modified in the following way to accommodate the joint probability of measurement values.

$$p(x_k|Y_{1:k-1}) = \int p(x_k|x_{k-1}) p(x_{k-1}|Y_{1:k-1}) dx_{k-1} \tag{5}$$

where joint probability of measurement values from time step 1 to $k-1$ is denoted by $Y_{1:k-1}$. The Proposition 4 is implemented in lines 20-22 of our proposed *BayesImposter* algorithm 1.

An example: From the explanation of the Proposition 2, we know that the suction cup can have any one of the following two states: $\{ON, OFF\}$, depending upon the position of the horizontal and vertical axis of the vacuum gripper robot. In multivariate ICS, instead of having a single position value for a particular state, the horizontal and vertical axis could have multiple position values within a range. For example, a position within 0 cm to 10 cm of the horizontal axis could trigger the state to ON from OFF. If there are n measurement values within the position range of 0 cm to 10 cm, *BayesImposter* uses Eqn. 5 to estimate the next state x_k.

Proposition 5: If multiple (i.e., n) measurement quantities, [y_k^1, y_k^2, y_k^3, ..., y_k^n], at a time step k, present in a multivariate ICS, *BayesImposter* finds y_k that gives the highest probability in Eqn. 4.

Explanation of Proposition 5: The Proposition 5 is an extension of the Proposition 3 for multiple number of measurement values $[y_k^1, y_k^2, y_k^3, ..., y_k^n]$, at a current state x_k. To estimate a measurement value from multiple measurement values, *BayesImposter* plugs in most frequent values from the distribution of measurement values $[y_k^1, y_k^2, y_k^3, ..., y_k^n]$ in Eqn. 4 with an intention to maximize the left hand side of Eqn. 4. For example, if the threshold position in the explanation of Proposition 3 has multiple values $S_k^{\theta 1}$, $S_k^{\theta 2}$,...,$S_k^{\theta n}$ for current state x_k, we can write Eqn. 4 as below.

$$\max_{\forall y_k} \{p(y_k|x_k)\} = \max_{\forall y_k} \{\frac{p(x_k|y_k) \times p(y_k)}{\sum_{y_k} p(y_k) p(x_k|y_k)}\} \tag{6}$$

where $y_k \epsilon \{S_k^{\theta 1}, S_k^{\theta 2}, ..., S_k^{\theta n}\}$. The $\max_{\forall y_k}$ is the function that maximizes $p(y_k|x_k)$ for all y_k that is implemented using an iterative approach in lines 24-34 of the proposed *BayesImposter* algorithm 1.

Algorithm 1: BayesImposter Algorithm.

```
Input: Previous measurements, y_{1:k-1} and states x_{1:k-1} up to k-1
Output: Current measurements, y_k and states, x_k at k step
1  for k ← 1 to k-1 do // Proposition 1 for state-space model
2      Collect y_{1:k-1} and x_{1:k-1} information of ICS
3      Create state-space model: x_k = p(x_k|x_{k-1}) & y_k = p(y_k|x_k)
4      if ICS is univariate then
5          for Each unknown x_k do // Proposition 2 for x_k
6              Find p(x_k|y_{1:k-1}) for every x_k
7              Select x_k having the highest p(x_k|y_{1:k-1})
8          for Each unknown y_k do // Proposition 3 for y_k
9              if x_k is known then
10                 Find p(y_k|x_k) for every x_k
11                 if p(y_k|x_k) > cut-off K_c then
12                     Select the y_k as the estimation
13                 else
14                     Discard the estimated y_k
15             else
16                 Find x_k first using Proposition 2
17                 Then use Proposition 3
18     if ICS is multivariate then
19         for Each unknown x_k do // Proposition 4 for x_k
20             Find joint probability Y_k = p(y_k^1 ∩ y_k^2 ∩ ..... ∩ y_k^n)
21             Find p(x_k|Y_{1:k-1}) for every x_k
22             Select x_k having the highest p(x_k|Y_{1:k-1})
23         for Each unknown y_k do // Proposition 5 for y_k
24             if x_k is known then // max_{∀y_k} function
25                 Find p(y_k^1|x_k) for y_k ϵ {y_k^1, y_k^2, .., y_k^n}
26                 max ← p(y_k^1|x_k)
27                 for Every y_k ϵ {y_k^2, y_k^3, .., y_k^n} do
28                     Find p(y_k|x_k)
29                     if p(y_k|x_k) > max then
30                         max ← p(y_k|x_k)
31                 Select max as the y_k for given x_k
32             else
33                 Find x_k first using Proposition 2
34                 Then use Proposition 5
```

5.2 Tag values from the estimated x_k and y_k

It is mentioned earlier in section 4 that the .bss section contains different uninitialized global/static tag variables. They can be broadly divided into two categories, namely the control programming or command related variables and protocol related variables (Fig. 4).

Estimation of control commands from x_k and y_k: After estimating x_k and y_k, the next challenge is to look for the corresponding control commands from the estimated x_k and y_k. It can be done in two ways. *Firstly*, most control commands are the direct values of x_k and y_k that are already estimated by *BayesImposter*. For example, from the Proposition 2, the threshold position S_k^θ is equal to the estimated measurement y_k in the .bss section. *Secondly*, rest of the control commands are estimated from OPC tags and specific PLC information (Fig. 4) using the estimated x_k and y_k. For example, the value of `suctionstate` $\epsilon \{ON, OFF\}$ corresponding to 0 or 1 can be found from specific PLC information (see Section 5.3).

Estimation of protocol related variables: The protocol-related variables are specific to cloud protocols and hence, are fixed and initialized at the load time of the control DLL file. The attacker can get the list of all the protocol-related variable names and their values from the reference book of a specific cloud protocol. As mentioned

in Section 3, most of the target control DLLs are available as open-source, and very few are proprietary, which are accessible by a basic commercial license (cost less than $100 [1]).

5.3 Entropy in the .bss section

The size of the specific control variable used in the .bss section can be a maximum of 64 bits in a 64-bit machine. Therefore, we have an entropy of 2^{64} possible values. For example, the tag variable `suctionstate` ideally could have 2^{64} values. But, in real-world implementation, the control variables are problem-specific and they have very few key values, which are also problem specific. Therefore, as mentioned in Proposition 2, the state variable, `suctionstate`, has two possible key values: {ON, OFF}. So, the entropy of the `suctionstate` is not 2^{64}; instead, the entropy is only two. Moreover, these key values are declared in the header files of the program codes, and programmers, as a good practice, generally use user-defined data types, such as *Enumeration (enum)* type to declare these key values. The use of enum data type by the programmer makes the declared control variable (e.g., `suctionstate`, etc.) more predictable. For example, after careful examination of control-related application codes that are running on top of cloud protocols, we find the following code snippet that supports our observation:

```
enum statepool {0,1};
enum statepool suctionstate;
```

This indicates that the values of ON/OFF is 0 or 1. In this way, the attacker can specifically know the tag values in the .bss section to recreate the .bss imposter page.

6 MEMORY DEDUPLICATION+ROWHAMMER

So far, we have discussed how the attacker can recreate the .bss imposter page using *BayesImposter*. Now, we discuss how the attacker uses the memory deduplication + Rowhammer bug to trigger a bit flip in the recreated .bss imposter page to corrupt control commands. As recent works [19, 29, 58, 62] have already provided details on the memory deduplication + Rowhammer bug, we will not repeat the same details here. We refer to Appendix 12 for more details. Instead, we provide advantages of our approach over [19, 29, 58, 62]. Let us briefly discuss the memory deduplication + Rowhammer first.

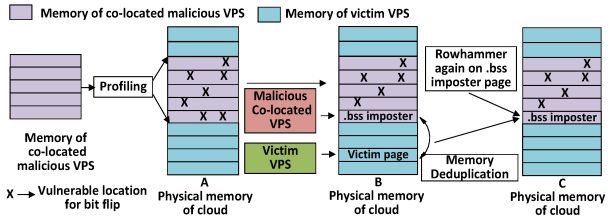

Figure 5: (A) Profiling the memory of cloud. (B) Placing *.bss imposter page* in the vulnerable location. (C) After memory deduplication, *victim page* is backed by the *.bss imposter page* and the Rowhammer causes bit flips in the *.bss imposter page.*

Brief overview: Memory deduplication merges identical pages located in the physical memory into one page. Rowhammer [45] is a widespread vulnerability in recent DRAM devices in which repeatedly accessing a row can cause bit flips in adjacent rows.

Memory deduplication thread (i.e., KSM) running in the host cloud hypervisor (i.e., KVM in Linux) maintains stable/unstable trees in a red-black tree format to keep track of the pages having identical contents in memory. If the *.bss imposter page* arrives first in the memory provided from the co-located malicious VPS, the node of the red-black tree will be updated first with the *.bss imposter page*. Therefore, if the *victim page* comes later from the victim VPS, the *victim page* is merged with the *.bss imposter page*, and the *victim page* shares the same memory location of the *.bss imposter page*. In this way, the attacker can control the memory location of the *victim page* and can trigger a Rowhammer on that page.

The first step to initiate Rowhammer is to find the *aggressor/victim* addresses in the physical memory of the running system. This step is named as **profiling**. The *aggressor* addresses are the memory locations within the process's virtual address space that are hammered, and the *victim* addresses are the locations where the bit flips occur (Fig. 5(A)). From the profiling step, the attacker knows the aggressor rows for the vulnerable memory locations. After placing the *.bss imposter page* in one of the vulnerable locations, the attacker hammers again on the aggressor rows (Fig. 5(C)). This results in bit-flips in the *.bss imposter page* that in effect changes the control commands in the .bss section of the target control DLL.

6.1 Advantages of BayesImposter

6.1.1 ***No first precedence and two copies of target pages.*** To ensure that the *.bss imposter page* arrives first in the memory, the attacker's VPS should start first before the victim VPS. This is known as the first precedence. Recent works [19, 29, 58, 62] use this technique along with creating two copies of target pages to place the *.bss imposter page* in the red-black tree before the target *victim page*. These techniques require more control over the victim VPS and may not be feasible in practical ICSs. For example, the attacker may not know when the victim VPS is started.

Thanks to the Bayesian estimation of the *victim page*. Referring to Section 5, if the attacker can predict the current states (x_k) and measurements (y_k), this means that he actually can predict the *victim page* before time k. As the attacker has the predicted *victim page*, the attacker can provide this predicted *victim page* to the memory deduplication thread at any time. Hence, the attacker does not need to start his VPS before the victim or does not need to create two copies of the target pages in our attack model. This makes our attack model more practical and reliable in the context of ICSs.

6.1.2 ***BayesImposter provides simpler profiling step.*** Recent works [19, 29, 58, 62] activate the *large pages* [55] in VPS to exploit the double-sided Rowhammering. However, *large pages* may not be explicitly turned on in the victim VPS. Therefore, double-sided Rowhammering may not be feasible in the context of ICSs [66]. Therefore, *BayesImposter* uses the random address selection approach for profiling the bit-flippable memory locations.

In this approach, *BayesImposter* allocated a 1 GB block of memory using a large array filled with doubles. A value of $1.79769313486231 \times 10^{308}$ is stored as double that gives 1 in memory locations. Next, the attacker randomly picks virtual aggressor addresses from each page of this large memory block and reads 2×10^6 times. Then the attacker moves to the next page and repeats the same steps. As the attacker can know the number of memory banks of the running system from his VPS, he can calculate his chance of hammering addresses in the same bank. For example, in our experimental setup, the machine has 2 Dual Inline Memory

Modules (DIMMs) and 8 banks per DIMM. Therefore, the machine has 16 banks, and the attacker has a 1/16 chance to hit aggressor rows in the same bank. Moreover, the attacker hammers 4 aggressor rows in the same iteration that increases the chance of having successful Rowhammering.

7 ATTACK MODEL EVALUATION

7.1 Automated high-bay warehouse testbed

We prepare a testbed to evaluate *BayesImposter* on a practical ICS. We choose a scaled-down model of an automated high-bay warehouse (AHBW) from *fischertechnik* connected with a vacuum gripper robot (VGR), multiprocessing oven (MPO), and sorting line (SL). The process begins first in MPO with a workpiece placed in the oven feeder. The processed workpiece from the MPO is then sent to SL using a conveyor belt. The SL sorts the workpiece depending upon color and stores it in the storage location. Next, the VGR uses its suction cup to hold the workpiece and transports it from the storage location to the pre-loading zone of the rack feeder of the AHBW. Then the rack feeder stores the workpiece in the warehouse. A video demonstration of the factory system is given here: https://sites.google.com/view/bayesmem/home.

The AHBW is connected with a SIMATIC S7-1500 PLC from Siemens using 32 input/output ports and 8 analog input ports. The PLC communicates with the cloud using a TIA portal through the MQTT cloud protocol Mosquitto. The cloud server runs on Intel CPU i7-6900K with 8 cores and 64GB of DDR3 RAM. We use Ubuntu Server 14.04.2 LTS x86_64 as the cloud server, which has a Kernel-based Virtual Machine (KVM). Memory deduplication is implemented as Kernel Samepage Merging (KSM) in KVM. The KVM is kept at its default configuration. The parameters for KSM (see Appendix 11.6) are also kept at their default settings. *All VPSs run with Windows 10* [8] and have 2 GB of main memory. *The idea of BayesImposter is equally applicable to the Linux VPSs with .so file [19] of cloud protocols.* The victim VPS is using MQTT to communicate with the PLC using TIA portal. The testbed is shown in Fig. 6.

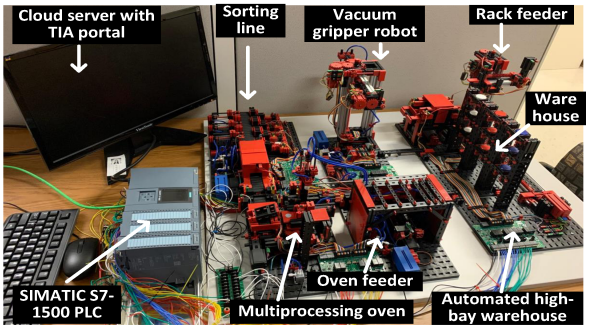

Figure 6: A small scale real-world testbed of automated high-bay warehouse to evaluate *BayesImposter*.

7.2 Estimation accuracy of *BayesImposter*

A practical ICS could have hundreds of states (x_k) and measurement values (y_k). Let us mathematically formulate this first.

Proposition 6: If an ICS has M state variables (x_k) and each state variable has N probable states, N^M combinations are possible among state variables and probable states. Similarly, If an ICS has P measurement variables (y_k) and each measurement variable has Q probable values, Q^P combinations are possible among measurement variables and probable values.

After counting, we find that our testbed - automated high-bay warehouse has $M = 420, N = 3, P = 160, Q = 4$. We find that the estimation accuracy for next states or next measurements using Propositions 1-5 of our *BayesImposter* algorithm is ~91%. It means that *BayesImposter* can estimate the next state or measurement variables within 1/0.91 = 1.09 attempt.

Table 2: Estimation accuracy of *BayesImposter*.

Estimating state variables x_k	Estimating measurement variables y_k
90.2%	91.47%

7.3 Recreating the .bss imposter page

The automated high-bay warehouse testbed has $M = 420$ state variables (x_k) in total, and each state has an average of $N = 3$ probable states. The brute-force approach gives $3^{420} \approx 2.4 \times 10^{200}$ combinations according to the Proposition 6. Moreover, this ICS in hand has also $P = 160$ measurement variables (y_k) in total, and each variable has an average of $Q = 4$ probable values. The brute-force approach gives $4^{160} \approx 2.13 \times 10^{96}$ combinations. In combined, there are $2.4 \times 10^{200} + 2.13 \times 10^{96} = 2.4 \times 10^{200}$ combinations are possible for the ICS in hand. For a 4KB page size, this may require $(4 \times 2.4 \times 10^{200})$ KB $= 9.6 \times 10^{194}$ GB of guessed pages. In other words, the attacker may need to spray 9.6×10^{194} GB pages in the physical memory for successful memory deduplication that is not possible in terms of time and memory. It is not possible to accommodate 9.6×10^{194} GB pages in one attempt of the attack, and the attacker may require thousands of attempts to spray the memory with the guessed pages. In contrast, as *BayesImposter* has an estimation accuracy of ~91% (see Section 7.2), it does not require to guess N^M or Q^P combinations; instead, it can guess states and measurement variables in $1/0.91 = 1.09$ attempt. Therefore, most of the time, *BayesImposter* requires only one or two pages (because of ~91% accuracy) of size 4KB to spray in the physical memory.

The victim VPS in our example ICS has a 2 GB main memory, and it takes ~13 minutes to scan all the pages of main memory in a single attempt (see Section 7.7). And, out of 2 GB of memory, we can spray 1.2 GB with the guessed pages at each attempt (i.e., remaining 0.8 GB for operating systems and other applications). Therefore, brute force requires $(9.6 \times 10^{194})/1.2 = 8 \times 10^{194}$ attempts, whereas *BayesImposter* requires only a 1.09 attempt. As each attempt takes ~13 minutes, *BayesImposter* requires only ~13 minutes compared to $9.6 \times 10^{194} \times 13$ min. $= 2 \times 10^{194}$ hours of brute force approach which is not feasible. This reduction of attempts also reduces the attack time (see Section 7.7). As the attack time for *BayesImposter* is significantly low compared to a brute force approach, *BayesImposter* gives more control over the ICS from the attacker's perspective. Table 3 shows the memory and time requirements for brute-force and *BayesImposter* approaches.

Table 3: Attack time of *BayesImposter*

BayesImposter		Brute force	
Guessed page	Time	Guessed page	Time
4KB or 8KB	13 min.	9.6×10^{194} GB	2×10^{194} Hr.

7.4 Attacking the vacuum gripper robot (VGR)

As mentioned in Section 7.1, the VGR uses its suction cup to transport the workpiece from the SL to the rack feeder of the AHBW. The solenoid present in the suction cup is turned on/off if the position of the horizontal and vertical axis of the VGR is above or below a threshold position. The threshold position is a measurement value (i.e., y_k) and can be estimated by *BayesImposter*. The correct value of the threshold position where the suction cup is turned off (release the workpiece) is 2 cm. The estimated value of the threshold position is also calculated as 2 cm using *BayesImposter* at a particular state (i.e., moving from SL to AHBW). After the successful estimation of the threshold position with all other tag values of the *victim page* using the same *BayesImposter*, the attacker can recreate the .bss imposter page. Now, the attacker initiates the memory deduplication + Rowhammer attack and arbitrarily causes a bit-flip in the .bss imposter page. A demonstration of the attack is shown in Fig. 7, which indicates the location of the occurred bit-flip in the victim row. *(0 0 1 7 3c97 0)* means address of channel 0, dimm 0, rank 1, bank 7, row 3c97, column 0 in DRAM with a row-offset 0743, which has a byte value *f7* after the bit-flip; however, byte expected according to fill pattern is *ff (i.e., all erased)*. The victim byte *f7* is the upper byte of the threshold position being corrupted that changes the 2 cm threshold position to 2050 cm. This causes an out-of-range value for the VGR resulting in a wrong drop-off location of the workpiece other than the rack-feeder. This may result in possible equipment damage or even can kill a person if the attacker drops the workpiece on a target person. A video demonstration of this attack is given here: https://sites.google.com/view/bayesmem/home

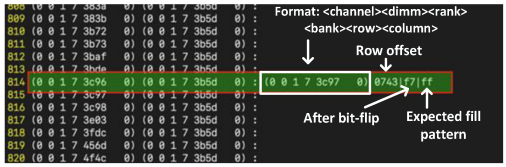

Figure 7: Bit-flip in the .bss imposter page.

7.5 Adversarial control using BayesImposter

As the attacker knows the physical location of a tag value in the tag table of the .bss imposter page, he can target a particular tag value and initiate an adversarial control over that tag value. For example, the attacker can cause a bit-flip of `suctionstate` from 1 → 0 and can adversarially drop the workpiece from the suction cup when it is not supposed to drop the workpiece (Fig. 8). This may result in possible equipment damage or even can kill a person if the attacker drops the workpiece on a target person. This adversarial control makes *BayesImposter* stronger compared to [19, 29, 58, 62].

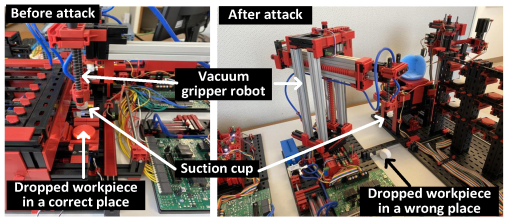

Figure 8: Dropping workpiece using adversarial control.

7.6 Profiling time in our testbed

Fig. 9 evaluates the profiling time (see Section 6) for different number of VPSs in the cloud. *BayesImposter* takes ~51.45 seconds to complete single-sided Rowhammer for each target row. We searched for vulnerable locations for the Rowhammer in the memory space, and Fig. 9 shows that to get ~20000 vulnerable locations, ~100 hours are required. With the increase of VPSs, this profiling time increases due to more memory pressure in the system memory. Fig. 9 shows the profiling time for 1, 3, and 6 VPSs in the same cloud.

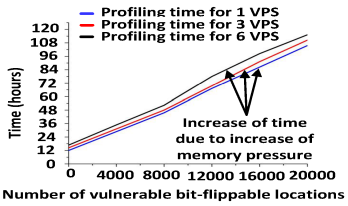

Figure 9: Profiling time for different number of VPSs.

7.7 Attack time

Here, we define attack time as how much time it takes to cause a bit flip in the .bss section. Attack time is the summation of the memory deduplication time and the Rowhammer implementation time. The exact time required for memory deduplication can be calculated using the timing side-channel [29]. However, roughly, the maximum time for memory deduplication is the time needed to scan all the memory of the co-located VPSs in the cloud. Here, for simplicity, we assume that deduplication happens within this maximum time frame, and hence, we consider this maximum time as the memory deduplication time. The memory deduplication time depends upon the parameters `pages_to_scan` and `sleep _millisec`. In default configuration, `pages_to_scan` = 100 and `sleep _millisec` = 20. Therefore, Linux/KSM can scan 1000 pages/second, which results in a total scan time of almost 5 minutes per 1GB of main memory [56]. As the victim VPS has a main memory of 2 GB, it should take approximately 10 minutes to scan all the pages in the main memory of a VPS. In our testbed, the memory deduplication takes approx. 13 minutes, and the Rowhammering process takes approx. 51.45 seconds to complete a single-sided Rowhammer for each target row. Therefore, after summing up these two figures, the total attack time is approximately 13 minutes and 52 seconds for 1 target VPS.

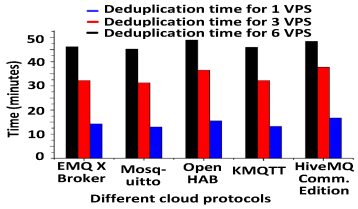

Figure 10: Deduplication time for different protocols.

Fig. 10 shows the memory deduplication time for five variants of MQTT cloud protocol for 1, 3, and 6 VPSs. This figure indicates that all five variants of the cloud protocol give almost equal deduplication time. As the addition of a VPS increases the scannable memory locations, the deduplication time increases with the number of co-located VPS in the cloud. The Rowhammer implementation time for a target row is almost the same for all five protocol variants.

7.8 Evaluation for different cloud protocols

As our attack model does not require any software bug present in the implementation of cloud protocols, state-of-the-art variants of cloud protocols should be vulnerable to our attack model. To support this claim, we implement a total of five variants of the MQTT protocol in our testbed and find that all are equally vulnerable, which proves the generalization of our attack model in ICSs.

Table 4: Cloud protocol variants vulnerable to *BayesImposter*

Sl.	Cloud protocol variants	Vulnerability
1	EMQ X Broker [4]	✓
2	Mosquitto [9]	✓
3	MQTT-C [10]	✓
4	eMQTT5 [5]	✓
5	wolfMQTT [13]	✓

8 DEFENSE

The following mitigations should be adopted against *BayesImposter*.

Increasing entropy in the .bss section: To prevent the attack, we increase entropy in the .bss section. This is done using a random variable as a signature in the .bss section. The attacker requires a significant amount of memory and time to break this signature variable [19] as this variable is not a part of the state variable. This approach is also effective against a malicious insider.

Securing cloud server from the malicious VPS: Any unauthorized cloud provider or personnel, or visitor should not access the cloud server without the presence of authorized personnel. Periodic screening by an authorized person needs to be carried out to look for any unauthorized co-hosted VPS. Any unnecessary or suspicious co-located VPS should be considered as a security breach and should be immediately contained in the cloud.

Turning off the KSM: To prevent memory deduplication, KSM can be turned permanently off. KSM is off by default in recent Linux kernel [2]. However, the KSM service, which is included in the `qemu-kvm` package, is turned on by the KVM host in the cloud setting. We turn off the KSM using the `ksm/ksmtuned` services in the KVM host. However, turning off the KSM may increase memory usage in clouds. Therefore, it is not favorable where memory workloads are high in cloud settings [43].

Preventing Rowhammer in DRAM: The next way to prevent *BayesImposter* is to prevent the Rowhammer in DRAM. While the built-in error-correcting codes (ECCs) can prevent single bit-flip in 64-bit words [32], it may not be enough where the Rowhammer causes multiple bit-flips [15, 50]. While only modern AMD Ryzen processors support ECC RAM in consumer hardware, Intel restricts its support to server CPUs [40]. One method to prevent Rowhammer is to increase (e.g., double) the refresh rate in DRAM chips [57]. This can reduce the probability of multiple bit-flips in DRAM, but causes more energy consumption and more overhead in the memory [34, 45]. Another method is to probabilistically open adjacent or non-adjacent rows, whenever a row is opened or closed [44]. An introduction of a redundant array of independent memory (i.e., RAIM) [54], and ANVIL [18] in the server hardware can make the Rowhammer attack infeasible. Moreover, replacing older chips with DDR4 having Target Row Refresh (TRR) capability can prevent single-sided and multi-sided Rowhammer attack on cloud networks [47]. However, [36] shows that DDR4 can also be compromised using TRR-aware attacks.

9 RELATED WORK

Attacks on ICSs: The attacks on ICSs can be broadly classified as attacks on physical hardware (e.g., PLCs, control modules, etc.), attacks on communication networks, and attacks on sensing side.

Abbasi et al. [14] demonstrated an attack on PLCs by exploiting pin control operations of certain input/output pins resulting in abnormal hardware interrupt in PLCs. Garcia et al. [37] presented a malware-PLC rootkit that can attack PLCs using the physics of the underlying systems. Bolshev et al. [28] showed an attack on the physical layer (i.e., analog-to-digital converter), resulting in false data injection into PLCs. Spenneberg et al. [68] developed a worm - PLC Blaster, that independently searches any network for S7-1200v3 devices and attacks them when the protective mechanisms are switched off. *Compared to our attack model, these attacks on PLCs lack the presence of adversarial control over PLCs and do not provide any means of stealthiness with respect to the monitoring entity.*

Klick et al. [46] showed that internet-facing controllers act as an SNMP scanner or SOCKS proxy, and their protocols can be misused by an adversary to inject false codes into PLCs, which are not directly connected to the internet. Basnight et al. [26] presented an attack on firmware exploiting communication protocols of PLCs. Beresford et al. [27] discovered vulnerabilities in Siemens S7 series communication protocol and showed a replay attack on ICSs. *Compared to these attacks, our attack model does not need any vulnerabilities in the communication protocol and does work without any presence of software bugs at any level of the system.*

Barua et al. [21–25], Liu et al. [52], and McLaughlin et al. [53] showed *false data injection* attack on different sensing nodes of ICSs leading to abnormal behaviour of the underlying system. *Compared to these attacks, our attack model is capable of false command injection from a remote location with adversarial control in ICSs.*

Attacks using memory deduplication and/or Rowhammer: Bosman et al. [29] demonstrated memory deduplication based exploitation vector on Windows using Microsoft Edge. Barresi et al. [19] exploited the memory deduplication in a virtualized environment to break ASLR of Windows and Linux. This attack uses brute force to duplicate the target page in the memory. Razavi et al. [62] provided Flip Fleng Shui (FFS) to break cryptosystems using both the memory deduplication and Rowhammer. **There are fundamental differences between our work and [19, 29, 62].** ***First****, our attack model exploited the .bss section of cloud protocols that is more impactful and realistic in ICSs.* ***Second****, our attack uses the Bayesian estimation to duplicate the target page compared to the brute force approach in [19, 29, 62]. This results in significantly less memory usage (i.e., in KB compared to GB) and time (i.e., in minutes compared to hours) to duplicate the target page. This makes our attack model more feasible.* ***Third,*** *our attack model demonstrates adversarial control over the target ICS that is absent in [19, 29, 62].*

Seaborn et al. [66] exploited CPU caches to read directly from DRAM using the Rowhammer bug. Gruss et al. [41] used cache eviction sets and Transparent Huge Pages (THP) for a successful double-sided Rowhammer. Tatar et al. [71] used Rowhammer attacks over the network to cause bit-flips using Remote DMA (RDMA). *Compared to these works, our work uses memory deduplication to skip the knowledge of physical memory location and uses single-sided Rowhammer on the target cloud memory. Moreover, our*

attack does not require any RDMA to happen that makes our attack more flexible in the context of ICSs.

10 CONCLUSION

We present an attack model-*BayesImposter* that can hamper the availability and integrity of an ICS in cloud settings. We are the first to point out how the .bss section of the target control DLL file of cloud protocols is vulnerable in ICS. *BayesImposter* exploits the memory deduplication feature of the cloud that merges the attacker's provided .bss imposter page with the victim page. To create the .bss imposter page, *BayesImposter* uses a new technique that involves the *Bayesian estimation*, which results in less memory and time compared to recent works [19, 29, 62]. We show that as ICSs can be expressed as state-space models; hence, the *Bayesian estimation* is an ideal choice to be combined with the memory deduplication in cloud settings. We prepare a scaled-down model of an automated high-bay warehouse using SIMATIC PLC from Siemens and demonstrate our attack model on this practical testbed. We show that our attack model is effective on different variants of cloud protocols, and does not need any vulnerabilities in the cloud protocol, and works without any presence of software bug in any level of the system that proves a generalization of our attack model. We show that *BayesImposter* is capable of *adversarial control* that can cause severe consequences through *system demage*. Therefore, our attack is impactful, and the countermeasures should be adopted to prevent any future attack like ours in ICSs.

ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation (NSF) under awards CMMI-1739503 and ECCS-2028269. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] [n.d.]. *Affordable MQTT Broker Pricing.* https://www.bevywise.com/mqtt-broker/pricing.html.
[2] [n.d.]. *Chapter 7. KSM.* https://docs.fedoraproject.org/en-US/Fedora/18/html/Virtualization_Administration_Guide/chap-KSM.html.
[3] [n.d.]. *Data Deduplication Overview.* https://docs.microsoft.com/en-us/windows-server/storage/data-deduplication/overview.
[4] [n.d.]. *EMQ X Broker.* https://www.emqx.io/downloads#broker.
[5] [n.d.]. *eMQTT5.* https://github.com/X-Ryl669/eMQTT5.
[6] [n.d.]. *Factory Simulation 24V.* https://www.fischertechnik.de/en/service/elearning/simulating/fabrik-simulation-24v. (Accessed: 03-22-2022).
[7] [n.d.]. *How to use TIA Portal Cloud.* https://new.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal/highlights/tia-portal-cloud.html.
[8] [n.d.]. *Linux kernel 2.6.32, Section 1.3. Kernel Samepage Merging (memory deduplication).* https://kernelnewbies.org/Linux_2_6_32#Kernel_Samepage_Merging_.28memory_deduplication.29.
[9] [n.d.]. *mosquitto.* https://mosquitto.org/.
[10] [n.d.]. *MQTT-C.* https://github.com/LiamBindle/MQTT-C.
[11] [n.d.]. *Siemens: How to connect to a PLC with TIA Portal in a Virtual Machine.* https://web.awc-inc.com/siemens-how-to-connect-to-a-plc-with-tia-portal-in-a-virtual-machine/.
[12] [n.d.]. *SIMATIC S7-1500.* https://cache.industry.siemens.com/dl/files/914/59191914/att_86487/v1/s71500_cpu1516_3_pn_dp_manual_en-US_en-US.pdf.
[13] [n.d.]. *wolfMQTT.* https://github.com/wolfSSL/wolfMQTT.
[14] Ali Abbasi and Majid Hashemi. 2016. Ghost in the plc designing an undetectable programmable logic controller rootkit via pin control attack. *Black Hat Europe* 2016 (2016), 1–35.
[15] Barbara Aichinger. 2015. DDR memory errors caused by Row Hammer. In *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–5.
[16] S Annapoorani, B Srinivasan, and GA Mylavathi. 2018. Analysis of various virtual machine attacks in cloud computing. In *2018 2nd international Conference on Inventive Systems and Control (ICISC)*. IEEE, 1016–1019.
[17] J.R. Appelbaum, L. Poitras, M. Rosenbach, C. Stöcker, J. Schindler, and H. Stark. 2013. Inside TAO : documents reveal top NSA hacking unit. *Der Spiegel* (29 12 2013).
[18] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. 2016. ANVIL: Software-based protection against next-generation rowhammer attacks. *ACM SIGPLAN Notices* 51, 4 (2016), 743–755.
[19] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R Gross. 2015. {CAIN}: Silently Breaking {ASLR} in the Cloud. In *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)*.
[20] Christoph Jan Bartodziej. 2017. The concept industry 4.0. In *The concept industry 4.0*. Springer, 27–50.
[21] Anomadarshi Barua and Mohammad Abdullah Al Faruque. 2019. *The Hall Sensor Security*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–4. https://doi.org/10.1007/978-3-642-27739-9_1652-1
[22] Anomadarshi Barua and Mohammad Abdullah Al Faruque. 2020. Hall Spoofing: A Non-Invasive DoS Attack on Grid-Tied Solar Inverter. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 1273–1290.
[23] Anomadarshi Barua and Mohammad Abdullah Al Faruque. 2020. Special session: Noninvasive sensor-spoofing attacks on embedded and cyber-physical systems. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 45–48.
[24] Anomadarshi Barua and Mohammad Abdullah Al Faruque. 2022. A Wolf in Sheep's Clothing: Spreading Deadly Pathogens Under the Disguise of Popular Music. In *29th ACM Conference on Computer and Communications Security (CCS)*.
[25] Anomadarshi Barua and Mohammad Abdullah Al Faruque. 2022. PreMSat: Preventing Magnetic Saturation Attack on Hall Sensors. In *International Conference on Cryptographic Hardware and Embedded Systems (TCHES 2022)*.
[26] Zachry Basnight, Jonathan Butts, Juan Lopez Jr, and Thomas Dube. 2013. Firmware modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection* 6, 2 (2013), 76–84.
[27] Dillon Beresford. 2011. Exploiting siemens simatic s7 plcs. *Black Hat USA* 16, 2 (2011), 723–733.
[28] Alexander Bolshev, Jason Larsen, Marina Krotofil, and Reid Wightman. 2016. A rising tide: Design exploits in industrial control systems. In *10th {USENIX} Workshop on Offensive Technologies ({WOOT} 16)*.
[29] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 987–1004.
[30] Chao-Rui Chang, Jan-Jan Wu, and Pangfeng Liu. 2011. An empirical study on memory sharing of virtual machines for server consolidation. In *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*. IEEE, 244–249.
[31] Seungoh Choi, Jongwon Choi, Jeong-Han Yun, Byung-Gil Min, and HyoungChun Kim. 2020. Expansion of {ICS} testbed for security validation based on {MITRE} atT&Ck techniques. In *13th {USENIX} Workshop on Cyber Security Experimentation and Test ({CSET} 20)*.
[32] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. 2019. Exploiting correcting codes: On the effectiveness of ECC memory against Rowhammer attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 55–71.
[33] Yuhui Deng, Xinyu Huang, Liangshan Song, Yongtao Zhou, and Frank Z Wang. 2017. Memory deduplication: An effective approach to improve the memory system. *Journal of Information Science and Engineering* 33, 5 (2017), 1103–1120.
[34] Philip G Emma, William R Reohr, and Mesut Meterelliyoz. 2008. Rethinking refresh: Increasing availability and reducing power in DRAM for cache applications. *IEEE micro* 28, 6 (2008), 47–56.
[35] Bernard Friedland. 2012. *Control system design: an introduction to state-space methods*. Courier Corporation.
[36] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TRRespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 747–762.
[37] Luis Garcia, Ferdinand Brasser, Mehmet Hazar Cintuglu, Ahmad-Reza Sadeghi, Osama A Mohammed, and Saman A Zonouz. 2017. Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit.. In *NDSS*.
[38] Omid Givehchi, Jahanzaib Imtiaz, Henning Trsek, and Juergen Jasperneite. 2014. Control-as-a-service from the cloud: A case study for using virtualized PLCs. In *2014 10th IEEE Workshop on Factory Communication Systems (WFCS 2014)*. IEEE, 1–4.
[39] Thomas Goldschmidt, Mahesh Kumar Murugaiah, Christian Sonntag, Bastian Schlich, Sebastian Biallas, and Peter Weber. 2015. Cloud-based control: A multi-tenant, horizontally scalable soft-PLC. In *2015 IEEE 8th International Conference on Cloud Computing*. IEEE, 909–916.
[40] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. 2018. Another Flip in the Wall of Rowhammer Defenses. In *2018*

IEEE Symposium on Security and Privacy (SP). 245–261. https://doi.org/10.1109/SP.2018.00031
[41] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer. js: A remote software-induced fault attack in javascript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 300–321.
[42] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C Snoeren, George Varghese, Geoffrey M Voelker, and Amin Vahdat. 2010. Difference engine: Harnessing memory redundancy in virtual machines. *Commun. ACM* 53, 10 (2010), 85–93.
[43] Gangyong Jia, Guangjie Han, Joel JPC Rodrigues, Jaime Lloret, and Wei Li. 2015. Coordinate memory deduplication and partition for improving performance in cloud computing. *IEEE Transactions on Cloud Computing* 7, 2 (2015), 357–368.
[44] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. 2014. Architectural support for mitigating row hammering in DRAM memories. *IEEE Computer Architecture Letters* 14, 1 (2014), 9–12.
[45] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 361–372.
[46] Johannes Klick, Stephan Lau, Daniel Marzin, Jan-Ole Malchow, and Volker Roth. 2015. Internet-facing PLCs-a new back orifice. *Blackhat USA* (2015), 22–26.
[47] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. 2020. RAMBleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 695–711.
[48] Reinhard Langmann and Leandro F Rojas-Peña. 2016. A PLC as an Industry 4.0 component. In *2016 13th International Conference on Remote Engineering and Virtual Instrumentation (REV)*. IEEE, 10–15.
[49] Reinhard Langmann and Michael Stiller. 2019. The PLC as a smart service in industry 4.0 production systems. *Applied Sciences* 9, 18 (2019), 3815.
[50] Mark Lanteigne. 2016. How rowhammer could be used to exploit weaknesses in computer hardware. *SEMICON China* (2016).
[51] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. 2014. Industry 4.0. *Business & information systems engineering* 6, 4 (2014), 239–242.
[52] Yao Liu, Peng Ning, and Michael K Reiter. 2011. False data injection attacks against state estimation in electric power grids. *ACM Transactions on Information and System Security (TISSEC)* 14, 1 (2011), 1–33.
[53] Stephen McLaughlin and Saman Zonouz. 2014. Controller-aware false data injection against programmable logic controllers. In *2014 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. IEEE, 848–853.
[54] Patrick J Meaney, Luis Alfonso Lastras-Montaño, Vesselina K Papazova, Eldee Stephens, JS Johnson, Luiz C Alves, James A O'Connor, and William J Clarke. 2012. IBM zEnterprise redundant array of independent memory subsystem. *IBM Journal of Research and Development* 56, 1.2 (2012), 4–1.
[55] Microsoft. [n.d.]. *Large-Page Support*. https://docs.microsoft.com/en-us/windows/win32/memory/large-page-support.
[56] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. 2013. XLH: More effective memory deduplication scanners through cross-layer hints. In *2013 USENIX Annual Technical Conference USENIX ATC 13)*. 279–290.
[57] Onur Mutlu. 2017. The RowHammer problem and other issues we may face as memory becomes denser. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 1116–1121.
[58] Marco Oliverio, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2017. Secure Page Fusion with VUsion: https://www. vusec. net/projects/VUsion. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 531–545.
[59] Nicholas Petreley. 2004. Security report: Windows vs linux. *The Register* 22 (2004), 1–24.
[60] Matt Pietrek. 1994. Peering inside the PE: a tour of the win32 (R) portable executable file format. *Microsoft Systems Journal-US Edition* 9, 3 (1994), 15–38.
[61] Noëlle Rakotondravony, Benjamin Taubmann, Waseem Mandarawi, Eva Weishäupl, Peng Xu, Bojan Kolosnjaji, Mykolai Protsenko, Hermann De Meer, and Hans P Reiser. 2017. Classifying malware attacks in IaaS cloud environments. *Journal of Cloud Computing* 6, 1 (2017), 26.
[62] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip feng shui: Hammering a needle in the software stack. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 1–18.
[63] Jarno Ruotsalainen. 2018. *Hardening and architecture of an industrial control system in a virtualized environment*. Master's thesis.
[64] Anam Sajid, Haider Abbas, and Kashif Saleem. 2016. Cloud-assisted IoT-based SCADA systems security: A review of the state of the art and future challenges. *IEEE Access* 4 (2016), 1375–1384.
[65] Bianca Scholten. 2007. *The road to integration: A guide to applying the ISA-95 standard in manufacturing*. Isa.
[66] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat* 15 (2015), 71.
[67] Bill Snyder. 2014. Snowden: The NSA planted backdoors in cisco products. *InfoWorld* 15 (2014).
[68] Ralf Spenneberg, Maik Brüggemann, and Hendrik Schwartke. 2016. Plc-blaster: A worm living solely in the plc. *Black Hat Asia* 16 (2016), 1–16.
[69] Jonathan Stidham. 2001. Can hackers turn your lights off: The vulnerability of the US power grid to electronic attack. *SANS Institute InfoSec Reading Room* (2001).
[70] Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, Amir Moradi, and Christof Paar. 2017. Interdiction in practice—Hardware Trojan against a high-security USB flash drive. *Journal of Cryptographic Engineering* 7, 3 (2017), 199–211.
[71] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 213–226.
[72] Michael Tiegelkamp and Karl-Heinz John. 1995. *IEC 61131-3: Programming industrial automation systems*. Vol. 14. Springer.
[73] Lonneke Van der Velden. 2015. Leaky apps and data shots: Technologies of leakage and insertion in NSA-surveillance. *Surveillance & Society* 13, 2 (2015), 182–196.
[74] Yuping Xing and Yongzhao Zhan. 2012. Virtualization and cloud computing. In *Future wireless networks and information systems*. Springer, 305–312.
[75] Ercan Nurcan Ylmaz, Bünyamin Ciylan, Serkan Gönen, Erhan Sindiren, and Gökçe Karacayılmaz. 2018. Cyber security in industrial control systems: Analysis of DoS attacks against PLCs and the insider effect. In *2018 6th International Istanbul Smart Grids and Cities Congress and Fair (ICSG)*. IEEE, 81–85.

11 APPENDIX

11.1 Automation pyramid

The automation pyramid is a graphical representation of the layers of automation within a typical industry (Fig. 11). It has five different levels of integrated devices. The name of the five levels and their components are briefly described below :

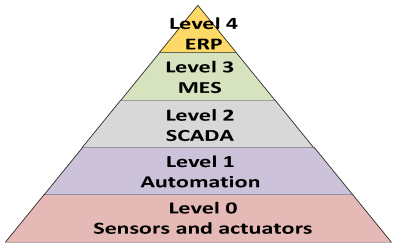

Figure 11: Automation pyramid in a typical Industry.

Level 0 - Sensors and actuators: This is the bottom level of the automation pyramid and comprises wide variety of sensors and actuators including measurement instruments, communication protocols, and actuators.

Level 1 - Automation: This level is made up with different controllers, such as PLCs, proportional-integral-derivative.

Level 2 - SCADA: This level consists of data acquisition system, human-machine interface, monitoring interfaces, etc.

Level 3 - MES: This level has management execution system (MES) for monitoring the entire process.

Level 4 - ERP: This level is made up with enterprise resource planning (ERP) which is responsible for the integrated management of main business processes.

11.2 PLCs and Industry 4.0

As Programmable Logic Controllers (PLCs) are one of the key ingredients of ICSs, Industry 4.0 drives new approaches in the PLC design [48]. Historically, PLCs were originally designed to support three main concepts, namely programmability, reliability and, real-time

response. Different programmable platforms, such as microprocessors, FPGAs, Hard Processor Systems (HPS) are chosen to support programmability in PLCs, as these hardware are programmable in run time in onsite industrial premises following the IEC 61131 key programming standard. Moreover, the standard IEC 61131 is developed in such a way to ensure reliability and real-time response by treating PLCs as logically independent with its own, individual configuration.

An architecture like this may provide predictable outcomes with a low likelihood of failure, but on the flip-side, it turns out to be progressively lumbering when confronted with developments in IIoTs that require noteworthy adaptability. The IIoTs require the cooperation of individual PLCs on a much deeper level. Moreover, individual PLCs likewise need to work considerably more closely with each other within the industry and remotely, to the web-server and cloud, for instance.

11.3 PLCs interface for basic web technologies

Todays PLCs have an interface that can be connected to a web-server via a device gateway. The device gateway is integrated into the existing PLC controllers that can support web-compatible protocol required for communication with the IP network. The web-server can connect to the PLC controller using HTML pages that enables a browser-based communication and diagnosis of the PLCs. The web-server can read and write control variables and collect measurement data from PLCs, with restrictions. Sometimes, this web-server is referred to as a "thin server" having enough computing resources to support local client/server network architecture.

11.4 Implemented protocols

Different protocols exist in different layers of ICSs. Typically IEC 61158 standard protocols are used in communication between PLCs and sensors. Here PLCs act as master, and sensors act as slaves. IEC 61158 standard contains a total of nine protocols: Fieldbus, Common Industrial Protocol (CIP), PROFIBUS/PROFINET, P-NET, WorldFIP, INTERBUS, HART, CC-Link, and SERCOS. These same protocols can be used between PLCs (master) and cloud adapters (slave). RS-232 or RS-485 based Fieldbus has multiple variants. Modbus and DNP3 are two of the most popular variants. They are widely adopted as a de facto standard and has been modified further over the years into several distinct variants. Moreover, Ethernet-based protocols, such as PROFINET, CC-LINK, SERCOS have lower latency than the Fieldbus protocols. Hence, these are preferred over Fieldbus in today's ICSs.

As already discussed in Section 2.2, the program for basic functions and supervisory controls are implemented in clouds or in web-server. These control programs are implemented using service functions in PLC controllers. A standardized protocol named Device Protocol for Web Services (DPWS) enables service-based access to PLC controllers. As mentioned earlier in Section 2.1, MQTT and AMQP are used to communicate with PLCs from clouds using an IoT gateway.

11.5 Memory deduplication and KVM

Memory deduplication or content-based page sharing is a process that combines/merges identical pages in the physical memory into one page. When the same/similar operating systems or applications are running in co-located VPSs, lots of redundant pages with same contents are created on the host system. The amount of redundant pages can be as high as 86% depending on the operating system and workload [30], and about 50% of the allocated memory can be saved through memory deduplication [42]. Memory deduplication is a feature in Windows 8.1, Windows 10, and Linux distribution. Due to more reliability, high security, stability, and less cost, Linux is more preferable over Windows in ICSs [59]. That is why here we consider Linux as our implementation platform for memory deduplication, and the idea is similarly applicable to Windows as well. Let us consider that the cloud in our discussion of ICS runs in the Linux platform. To allocate multiple VPSs in the same cloud, Kernel-based Virtual Machine (KVM) has been introduced in the Linux kernel since 2.6.20. Memory deduplication is implemented as Kernel Samepage Merging (KSM) in KVM. Next, we discuss how KSM is used in our attack model to merge the duplicated .bss section.

11.6 Kernel Samepage Merging (KSM)

When a VPS is started, a process named `qemu-kvm` of the KVM hypervisor allows KSM to merge identical pages in the memory. KSM has a specific daemon named `ksmd` that periodically scans a specific region of the physical memory of an application. The daemon `ksmd` can be configured in `sysfs` files in /sys/kernel/mm/ksm location. The `sysfs` files contain different configurable parameters. Among them, we need to mention two parameters: `pages_to_scan`, and `sleep_millisec`. The parameter `pages_to_scan` defines how many pages to scan before `ksmd` goes to sleep, and `sleep_millisec` defines how much time `ksmd` daemon sleeps before the next scan. If `sleep_millisec` = 500, and `pages_to_scan` = 100, then KSM scans roughly 200 pages per second. These numbers depend upon workload and are configured by the cloud provider accordingly. The values of `sleep_millisec` and `pages_to_scan` have a significant influence on the *attack time*. This is discussed in Section 7.7.

11.7 KSM data structure

The daemon `ksmd` periodically scans registered address space and looks for pages with similar contents. KSM reduces excessive scanning by sorting the memory pages by their contents into a data structure, and this data structure holds pointers to page locations. Since the contents of the pages may change anytime, KSM uses two data structures in *red-black* tree format, namely unstable tree and stable tree. Moreover, there are three states of each page in the memory: frequently modified state, sharing candidate yet not frequently modified state, and shared/merged state. The page which is frequently modified is not a candidate to be loaded in a stable or unstable tree of KSM. The page which has similar contents yet not frequently modified (i.e., unchanged for a period of time) is a candidate to be loaded in unstable tree first. The pages in the unstable tree are not write-protected and liable to be corrupted as their contents are modified. The stable tree contains pointers of all shared/merged pages (i.e., ksm pages), and these pages are sorted by their contents in the stable tree. Each page in the stable tree is write-protected. Hence, whenever any process tries to write in the merged/shared page of the stable tree, a private copy of the page corresponding to that particular process is created first and

mapped into the page-table-entry (PTE) of that particular process. Then the process writes in that private copy of the page. This is known as copy-on-write (CoW). As CoW involves the creation of a private copy of the shared/merged page of the stable tree first and then writes to that private page, CoW operation is expensive. Therefore, this takes a longer time compared to a write to a regular page. In other words, a longer write time on a page probably indicates that the page is already merged/shared in the stable tree by `ksmd` daemon. This longer write time in CoW process works as a side channel [29] and provides an indication that the page is already merged with another page having similar contents.

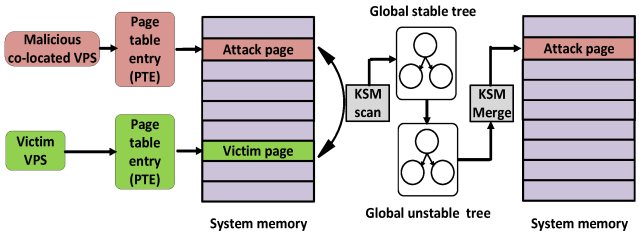

Figure 12: Merging *attack page* with the *victim page*.

12 MEMORYDEDUPLICATION+ROWHAMMER

12.1 Process of merging the duplicated .bss section

The process of merging the duplicated .bss section is shown in Fig. 12. As discussed earlier, the .bss section of the target control DLL is page aligned and is mapped to a page in the physical memory. Let us denote this page as the *victim page*. Similarly, the duplicated .bss section of the target control DLL file is also mapped to a different page in the memory. Let us denote this page as the *attack page*. The *attack page* and *victim page* both have same contents. The only difference between them is that the *attack page* is provided by the attacker, whereas the *victim page* is coming from the victim VPS.

The daemon `ksmd` of the KVM checks the contents of the *attack page* and the *victim page* in the registered address space. Either the *attack page* or the *victim page* is available to the daemon `ksmd` depending upon their order of arrival in the memory. If the *victim page* arrives first, the daemon `ksmd` marks this page as a candidate page to be merged. At first, this candidate page is searched in the stable tree using `memcmp()`. As this candidate page is not still available in the stable tree, it is then searched in the unstable tree by recalculating the checksum over the candidate page. If the checksum has not been changed, the daemon `ksmd` searches the unstable tree for this candidate page (`unstable_tree_search()`). In this case, as the occurrence of the candidate page (i.e., *victim page*) is first in the unstable tree, this candidate page cannot be found in the unstable tree. As a consequence, a new node is created in the unstable tree for this candidate page (i.e., *victim page*). In the next step, when the *attack page* arrives in the memory, the daemon `ksmd` marks this page again as the candidate page and searches this page in the unstable tree. As the content of the candidate page (i.e., *attack page*) is same as the *victim page*, this candidate page (i.e., *attack page*) will be merged with the similar node (i.e., *victim page*), which is created in the prior step, in the unstable tree. Then this node of the unstable tree will be merged into the stable tree. If a new candidate page arrives in the memory, this process iterates again.

12.2 Rowhammering on the merged .bss section

In Section 6, we discuss that how the target *victim page* is merged with the *attack page* using the memory deduplication technique. Note that the attacker cannot simply write to his attack page (i.e., deduplicated page) to change any data, as simply writing to the deduplicated page by the attacker triggers a CoW (Section 11.6) event to isolate the attack page from the victim page, and the main goal of the KSM may become invalid. That is the reason why the attacker needs something else to corrupt the deduplicated page without triggering the CoW event. Thanks to the Rowhammer bug present in DRAM, Rowhammer can be used to flip bits directly on the DRAM without triggering any CoW event.

Rowhammer [45] is a widespread vulnerability in recent DRAM devices in which repeatedly accessing a row of DRAM can cause bit flips in adjacent rows. To reliably craft our Rowhammer exploit on the deduplicated page, we have to overcome many challenges. The detail of these challenges is explained as follows.

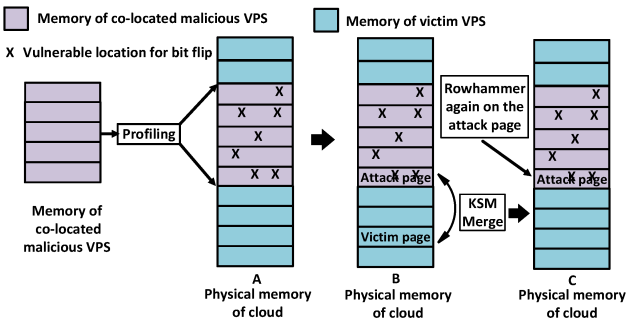

Figure 13: (A) Profiling the memory of the cloud. (B) Placing *attack page* in the vulnerable location. (C) After KSM merging, *victim page* is backed by the *attack page*.

12.3 Profiling the vulnerable locations of physical memory

A property of the Rowhammer is that the Rowhammer induced bit-flips tend to be repeatable. A memory location, where a bit flip occurs for the first time, there is a high chance that bit-flips will be reproducible in that location again. Therefore, it is possible to estimate whether a memory location of a DRAM tends to flip. This knowledge of exploitable bit locations is critical for the attacker to successfully exploit the Rowhammer bug from the co-located malicious VPS.

Therefore, the first step to initiate the Rowhammer attack is to find the *aggressor/victim* addresses in the physical memory of the running system. We name this step as profiling (Fig. 13(A)). The *aggressor* addresses are the memory locations within the process's virtual address space that are hammered, and the *victim* addresses are the memory locations where the bit flips occur. For a successful Rowhammer bit flip, the aggressor rows and the victim rows should be located in different rows but within the same bank of the DRAM

chip. If the aggressor rows and the victim rows are located in the different banks of the DRAM chip, the Rowhammer exploit may only read/write from those bank's *row-buffers* without activating aggressor rows repeatedly. This may not cause any bit-flip in the physical location of the DRAM chip. Therefore, before starting the profiling step, the attacker must ensure that aggressor rows satisfy the *"different rows, same bank"* requirement for the Rowhammer.

12.4 Refining the profiling step

To ensure different rows but same bank location of the aggressor rows, there are different methods. One method is to use physical addresses of the DRAM rows using an absolute physical address or relative physical address information. The absolute physical address information may not be available by the malicious VPS of the attacker. The relative physical address information can be achieved by using *large pages* [55] in Windows VPS. To use the large page support in Windows, the large page option should be activated first in the victim VPS, but it may not be explicitly turned on in the victim VPS. Therefore, double-sided Rowhammering is not a suitable way for the profiling step in the context of ICSs [66]. Another method is to use random address selection. This is a simpler approach, and the attacker does not need to know the absolute physical address or relative physical address of DRAM. To keep the attack model simpler and easily exploitable, *BayesImposter* uses this random address selection approach for profiling the bit-flippable memory locations of the physical memory. This approach also falls in the category of single-sided Rowhammering.

In the random address selection approach, the attacker allocated a large block of memory of 2 GiB using a large array filled with doubles. A value of $1.7976931348623157 \times 10^{308}$ is stored as double that gives 1 in memory locations. Next, the attacker randomly picks virtual aggressor addresses from each page of this large memory block and reads 2×10^6 times from each random aggressor address of that page. Then the attacker moves to the next page and repeats the same steps. As the attacker can know the number of banks of the running system from his VPS, he can calculate his chance of hammering addresses in the same bank. For example, in our experimental setup, the machine has 2 Dual Inline Memory Modules (DIMMs) and 8 banks per DIMM. Therefore, the machine has 16 banks, and the attacker has 1/16 chance to hit aggressor rows in the same bank. This 1/16 chance is high for the attacker. Moreover, the attacker hammers 4 aggressor rows in the same iteration that increases the chance of having successful Rowhammering.

After finishing hammering the entire block of memory, the attacker checks the array for possible bit flips. If any bit-flip occurs on any page, the attacker records that page and the offset. In this way, the attacker profiles the memory for vulnerable page/location, where a bit flip is more probable. After profiling, the attacker has aggressor/victim addresses in hand.

The next step is to place the target *victim page* (i.e., page aligned .bss section of the target control DLL) in one of these vulnerable pages. This memory placement must be done for a successful bit-flip in the target *victim page*. This process is discussed next.

12.5 Placing the target victim page in the vulnerable location

As the attacker has aggressor/vulnerable addresses from the profiling step, the attacker places the *attack page* in the vulnerable addresses first (Fig. 13(B)). When the target victim VPS starts, the target victim page is merged with the attacker's provided *attack page* using the memory deduplication process (Section 6). Therefore, after merging with the *attack page*, as the *attack page* is used to back the memory of the *victim page*, then, in effect, the attacker controls the physical memory location of the *victim page*. As the *attack page* is placed in the vulnerable addresses for possible bit-flip, then, in effect, the target *victim page* is also placed in the same vulnerable location for possible bit-flip ((Fig. 13(C)).

12.6 Rowhammering on the aggressor rows

From the profiling step, the attacker knows the aggressor rows for the vulnerable memory locations. After placing the *attack page* in one of the vulnerable locations, the attacker hammers again on the aggressor rows corresponding to that vulnerable location ((Fig. 13(C)). This results in bit-flips in the *attack page* that in effect changes the value of the control programming and supervisory control related variables in the .bss section of the target control DLL.