12-2021

Component Damage Source Identification for Critical Infrastructure Systems

Nathan Davis
University of Arkansas, Fayetteville

Citation

Davis, N. (2021). Component Damage Source Identification for Critical Infrastructure Systems. *Graduate Theses and Dissertations* Retrieved from https://scholarworks.uark.edu/etd/4265

Component Damage Source Identification for Critical Infrastructure Systems

A thesis submitted in partial fulfillment
of the requirements of the degree of
Master of Science in Computer Science

by

Nathan Davis
University of Arkansas
Bachelor of Science in Computer Science, 2020

December 2021
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.

Brajendra Panda, Ph.D.

Dale Thompson, Ph.D.

John Gauch, Ph.D.

Abstract

Cyber-Physical Systems (CPS) are becoming increasingly prevalent for both Critical Infrastructure and the Industry 4.0 initiative. Bad values within components of the software portion of CPS, or the computer systems, have the potential to cause major damage if left unchecked, and so detection and locating of where these occur is vital. We further define features of these computer systems and create a use-based system topology. We then introduce a function to monitor system integrity and the presence of bad values as well as an algorithm to locate them. We then show an improved version, taking advantage of several system properties to increase efficiency. We additionally delve into the use of digital twins for simulating potential bad values faster-than-real-time. Finally, we show evidence of our non-digital twin model's effectiveness through simulation.

Acknowledgements

It is my honor to express my deepest and profound gratitude to my advisor, Dr. Brajendra Panda. His consistent advice and dependable guidance have allowed me to reach further academically than I believed, both in my undergraduate and graduate career. I would also like to express my gratitude to Dr. John Gauch and Dr. Dale Thompson for being committee members and sparking my passion for computer science and security, respectively. Finally, I would like to thank all the friends I made here at the University of Arkansas for giving me support and a sense of community, allowing me to always strive for the highest.

Table of Contents

Introduction

Cyber-Physical Systems (CPS) are often large and complex, with complicated relationships within their numerous components. These systems include the incredibly vital power grids and other Critical Infrastructure used throughout the country as well as in various manufacturing processes with the advent of Industry 4.0. Within these “cyber” systems, or computer systems, maintaining functionality is of utmost importance, as any downtime in a system could result in numerous other issues ranging from user dissatisfaction to the collapse of services relying on the system. Component failure within a computer system is an expected reality, and while stopping the source of those faults completely may a viable approach, mitigating the effects is often more achievable and is what we hope to accomplish in this work. Furthermore, while faults or errors within components may be detectable through various techniques, there could also be values that, while functional, are incorrect, which can lead to further inaccuracies and problems through the system. These values are called *bad values* and can be caused either from faults within a component or from external malicious attack.

We introduce a novel framework to detect bad values occurring in stateless components within a generalized computer system and further to accurately identify which components are producing bad values. We do this by constructing and updating a functional topology of a computer system then performing regular checks to determine that system’s overall integrity. Then, if we find the system integrity to be failing, searching through our generated topology to locate the faulty components. We then expand on this methodology by leveraging certain properties of CPS to improve the efficiency of our algorithm. Finally, we introduce the idea of Digital Twins into our framework, which we use to implement a faster-than-real-time verification system into the CPS to detect bad values.

We continue this section with works related to this research. Section 2 presents our framework and defines our methodology. Section 3 covers the approach for monitoring the system integrity and locating bad values within. Section 4 shows the inclusion of Digital Twins into our model and the benefits included with applying them to our framework. Section 5 presents our simulation for our model and analysis of the results. Finally, we conclude in Section 6 with our findings.

1.1 Related Works

Bad value detection in CPS builds from fault and bad data detection in sensor networks [3-5], which CPS are extensions of. Some examples of bad value detection in CPS focus on mathematical models or system-wide approaches [26-30], though scalability is a challenge for efficient detection. Bad data detection also has extensive research for power grids[6-12] where one focus is on state estimation to predict potential faults. Security in CPS has been studied extensively, such as the possibility of attacks from internal sources [38], such as workers or faults with individual components, or external sources like malicious actors. Since CPS have applications in Critical Infrastructure (CI), the effects of an attack could have far-reaching consequences [1, 2, 25] which must be addressed to mitigate potential damage as much as possible. Digital twins are a part of the Industry 4.0 research area, having been created in 2003 [13] with extensive research being performed on them, focusing on their ability to provide accurate and flexible real-time simulation data [14-16, 18] about manufacturing and other industrial processes. There is research into communication between digital twins, called experimental digital twins (EDT), by Schluse et al. [20] which can be valuable for complex simulations and monitoring. Industry 4.0 is a term for a new wave of manufacturing methodologies and ideas, continuing from the Internet of Things

(IoT) and working to create “smart factories” [31]. CPS are seen as a natural fit for the increased flexibility and decentralization desired for Industry 4.0 [21, 24, 36-37] and research has been conducted combining CPS and Digital Twins together to provide more comprehensive analysis, as well [17, 19, 33].

Framework Definitions

2.1 Components

The computer systems within a CPS are comprised of distinct components that perform a specified function and pass that information on to other components. Components, for the purposes of our work, are considered stateless black boxes with a qualified range of inputs and outputs. Components will have weights assigned to them within our model, described in Section 4. Components may also have weights or stored values as a part of the computer system, but we choose not to store this information as it simplifies our model. Our model will still work if a computer system includes stateful components, though we do not explore this any further in this work. A component's workflow is to accept inputs from other components or external information, perform some function on those inputs, then provide outputs to other components. We expect both the input and output to be qualified, or within the appropriate ranges specified for each component. Thus, a *bad value* can be defined as a value that is outside of these qualified ranges. A component has no limit on the number of inputs and outputs it can possess. We denote the set of all components in a system as C, and a specific component as c_i. Formally,

$$c_i \in C$$

2.2 Component Categories

To provided additional information, we categorize components in terms of importance to the system's core functionality. Components are either Critical, where their operation is directly needed to maintain system functionality, or Non-Critical, where the component is not required for system functionality. These categories are used within our model to focus our efforts, as a bad value produced by a Non-Critical component, while troublesome, will not cause any severe damage like if it were a Critical component. The set of all Critical components is denoted by *R* and the set of all Non-Critical components *N*. A Critical component is denoted by r_i and a Non-Critical component by n_i. A component is either Critical or Non-Critical, but not both. Formally,

$$r_i \in R \;\; and \;\; n_i \in N$$
$$R \;\subseteq\; V \;\; and \;\; N \subset V$$
$$R \;\cup\; N \;\equiv\; V \;\; and \;\; R \cap N \;\equiv\; \{\}$$

By definition, at least one Critical component is required in a system, though Non-Critical components are not necessary in a system. Since our goal is to maximize the uptime of the computer system, and because Non-Critical components, by definition, do not affect the primary function or uptime of the system, they do not need to be included in any of our detection or search methods. They are still included in our model, and they can be checked for bad values just like Critical components, but their function is not within the scope of our work here.

2.3 Component Relationships

Components are connected to one another within a system, accepting inputs from and providing output to other components. These links are the core of a system, and it is here that an issue in one component can affect other components, leading to reduced system integrity or worse, total system failure. We denote the set of all links within a system as *E*. we define the relationship between components as follows: If c_1 outputs to c_2, we say c_2 is a *receiver* to c_1, and likewise c_1 is a *source* of c_2. We refrain from labeling this relationship as the more typical “child/parent” to allow the case where two components both output and receive from one another. Figure 1 shows a simple

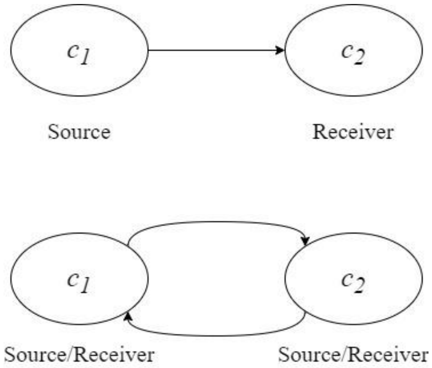


Figure 2: *Component Relationships*

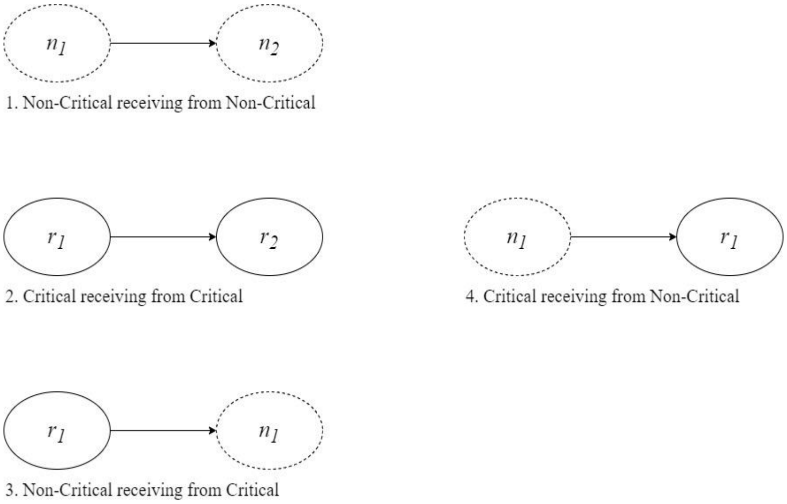

n_1
n_2
1. Non-Critical receiving from Non-Critical
r_1
r_2
2. Critical receiving from Critical
n_1
r_1
4. Critical receiving from Non-Critical
r_1
n_1
3. Non-Critical receiving from Critical

Figure 1: *Acceptable/Unacceptable Relationships*
Examples 1,2, and 3 are acceptable, while 4 is unacceptable

diagram of two components, where c_1 is the source to c_2 and c_2 is the receiver to c_1. Figure 1 also shows a case where both c_1 and c_2 are source and receiver to each other, which is allowed.

Expanding the source/receiver relationship to include Critical and non-Critical components leads to an additional limitation. While Critical components are free to be source and/or receiver to either Critical or Non-Critical components, and Non-Critical components can be a receiver to either type of component, they can only be a source to other Non-Critical components and are *not* allowed to be a source to Critical components. Therefore, only Critical components can be sources to other Critical components. This limitation reinforces the distinction between Critical and Non-Critical components that the latter have no effect on the system's critical functionality. Showing this from another perspective, any component that outputs to a Critical component must itself be a Critical component.

2.4 System View

The set of all components and the links between those components make up a system, denoted as *S*. For our purposes, a system is a directed graph, with the set of components, *C*, being the vertices, and the edges between components being the dependencies explained in Section 2.3, denoted *E*. Formally,

$$S = \{C, E\}$$

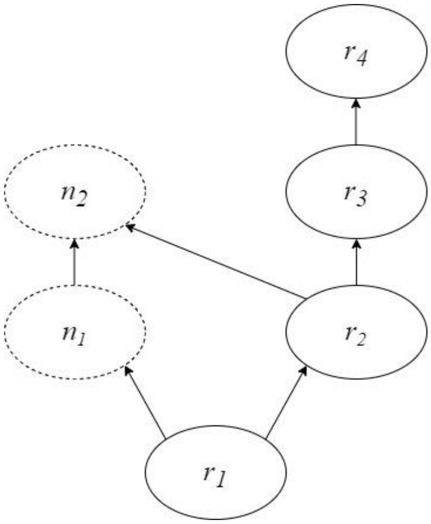


Figure 3: *Example System View*

Figure 3 shows a simple system view, consisting of six components, four Critical and two Non-Critical. Note that components may have multiple inputs and multiple outputs. Components with solid outlines are Critical components, while the dotted outlines are Non-Critical components. Also note that Critical components can output to Non-Critical components but not vice-versa.

2.5 External Components

For a computer system within a CPS, it is inherently not isolated. That is, not only is information flowing within the computer system, but there will also be information flowing into and out of the computer system. We are primarily interested in information entering the system, as that can alter and potentially disrupt the components within the computer system.

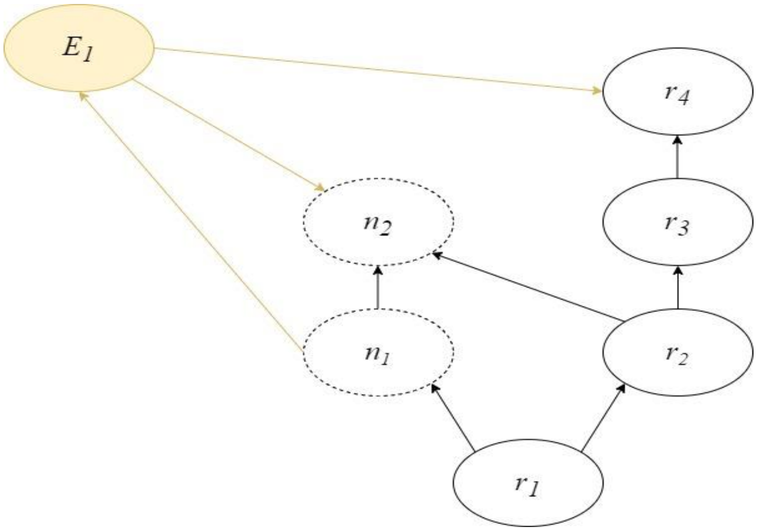


Figure 4: *External Component*

While we are not able to directly monitor these external sources, called "External" component from here, the information they send is recorded. These External components represent objects that are not local to the system, so they do not directly contribute to system integrity and do not warrant additional monitoring in our model beyond the information they send to other, non-External, components. External components are allowed to be the source or receiver to any component within the system. Figure 4 shows an example case where a system contains an External component. Note that while the External component contributes to the system, they are kept from influencing the functions defined further in.

2.6 Storing System Data

We assume the computer system will have a log of all interactions within and between components, which we believe is not a large assumption to make with modern computer systems. These log files, which ideally mark what data was transferred, timestamps of that transfer, and the components data travelled between, are integral to our model, since it will be used to create a model of the system and to aid in the creation of the digital twins for each component.

2.7 Creating Modified System Structure

Since all data essential to our model is stored within the log files, we can reconstruct the system's structure. That is, we can create a model of the computer system's component and dependencies between them. There is, however, one large difference between the system's structure and our structure created from the logs, from now on referred to as the *log-based system*. The actual computer system specifies components and dependencies between them, but not all components within a system will be in use. Some might be simply for emergency cases or part of

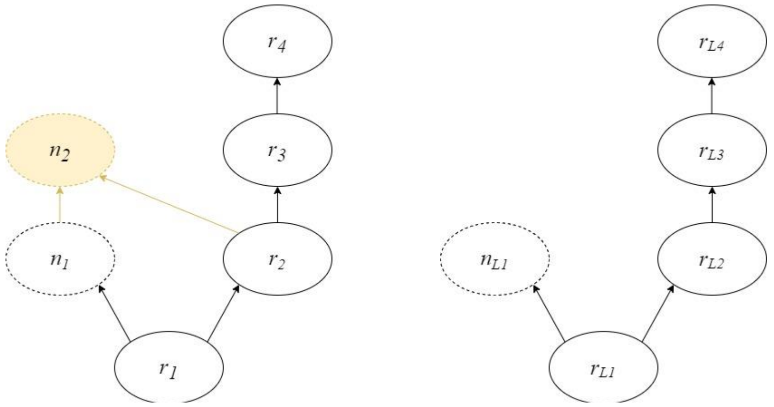


Figure 5: *Actual System vs. Log-Based System*
Highlighted components and links exist in the actual system but are ununused and so are absent from the log-based system

deprecated functions, but it is reasonable to assume that not all components or dependencies are utilized in a system at once.

The log files store data of interactions that have happened in the past, while the actual system defines interactions that are possible to occur in the future. Thus, the log-based system only contains components and dependencies that have already occurred. Figure 5 is an example of a log-based system and how it differs from the actual system. This simplifies our model, as we only need to check for components that have been used, as an unused component is effectively the same as not having a component there at all. If such a component would be used eventually, it would be trivial to add this component into the log-based model.1 Formally, this log-based system, S_L, is defined as

$$S_L = \{C_L, E_L\}$$
$$C_L \subseteq C\ , E_L \subseteq E$$
$$thus, S_L \subseteq S$$

Where C_L and E_L are the components and corresponding dependencies present in S_L. Because C_L and E_L are subsets of C and E respectively, S_L is therefore a subset of S.

2.8 System Tasks

One consideration we had to account for was components that are present in the actual computer system but are unused, which we discuss in Section 2.7. Another consideration we must make is once a system is large enough, it becomes increasingly likely that not all components will be related to one another. That is, there will emerge multiple disjoint subsets of components. The presence of these disjoint subsets requires us to accommodate for components that have no influence on each other.

We believe a reasonable classification of these disjoint subsets to be tasks within the system, or *system tasks*. A *task* is a function performed by the system, requiring one or more components. A component is likely to appear in more than one task within a system, and it this re-occurrence can be leveraged to better determine which component is producing a bad value, which is explained more in Section 4.3. To this end, we not only track which components are in use by the system, shown in Figure 3, but also which tasks a component is included in. An example of a system containing multiple tasks can be seen in Figure 6, as seen from the view of the system.

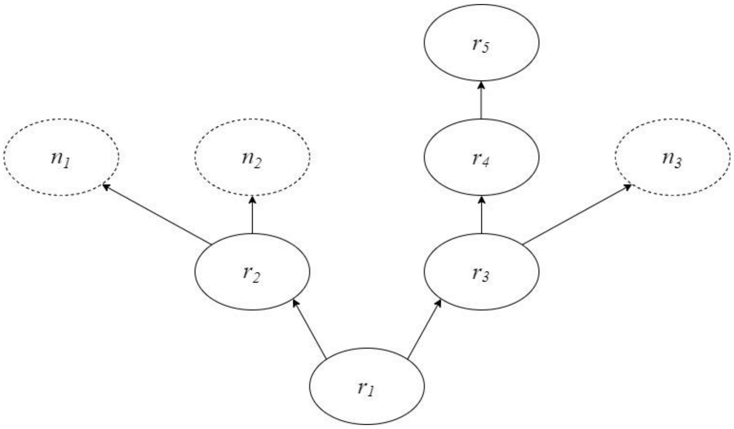


Figure 6: *System Containing Two Tasks*

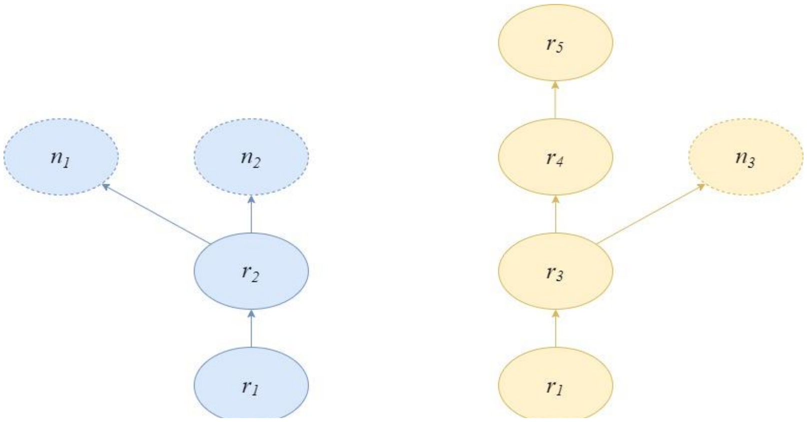


Figure 7: *Task View of Figure 6*

Figure 7 shows the same system but separates the two tasks. Note that component *r_1* is present in both tasks, since it may provide different values, or those values may be differently weighted, to each task.

Detection & Location Techniques

3.1 Detection Function

Our first technique to detect bad values within a computer system is a system-wide Detection function, or DF. The DF provides a weighted average of each Critical component within the system to produce an Integrity Value, or IV. The weights for each component are explained in more detail in Section 3.2. The resulting IV represents the computer system's overall integrity, and if the IV is below a specified threshold, the signifies the system operating in potentially unsafe conditions and should be checked for bad values. The IV is then normalized to a range between 0 – 100. The exact value of the IV threshold is to be determined on an individual basis, as some systems may require complete integrity of its components while others are robust enough to maintain safe conditions with a small loss of integrity.

The DF typically only includes Critical components as these are the components determined to be vital to the computer system's function and uptime, Non-Critical components would have no bearing on the critical functions of a system. The DF can be performed on Non-Critical components to ensure they are also performing under safe conditions, and this is shown further in this work, however they are supplemental to the main purpose of the DF, the Critical components. Below is the DF function, where *C* is the set of components, either Critical or, optionally, Non-Critical, *w* is the multiplication of weights for a given component, *v* being a component within *C*, and *IV* being the Integrity Value,

$$DF(C) = \frac{w_1v_1 + w_2v_2 + w_3v_3 + \cdots + w_iv_i}{\#\ of\ components\ in\ C} = IV, \qquad 0 \le IV \le 100$$

The DF is run on a fixed time interval, that can be set by the local system administrator, so that up-to-date information about the system is available. If the DF is performed against both

Critical and Non-Critical components, which again only Critical components are necessary, we differentiate the IV of each type of component into *RIV* for Critical components and *NIV* for Non-Critical components. The RIV and NIV can then be combined to produce a System Integrity Value, *SIV,* which gives information about all components within a system.

$$\frac{RIV + NIV}{2} = SIV$$

3.2 Detection Function Weights

The DF utilizes three weights, which are Sensitivity, *S*, Importance, *I*, and Probability of producing a bad value, *P*, all of which are assigned individually to each component within a system. We combine these weights into one variable, *w*, for each component, with w_{i} assigned to c_i.

$$w_{DF,i} = S_i \times I_i \times P_i$$

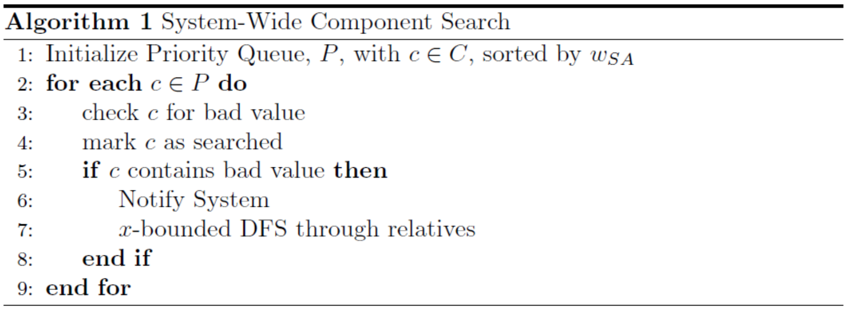
Sensitivity is a measure of the correlation between a component's input and output. That is, the higher *S* is, the more a component's output changes from the same change in input. For example, if two components receive the same inputs, the component with higher Sensitivity will have larger variation in its output compared to a component with a lower Sensitivity. We track this value in a component because if a bad value is sent into a less sensitive component, the magnitude of the effect on the component or system will be reduced. Likewise, if a component has a high Sensitivity, even a small deviation resulting from a bad value could have major consequences.

Importance is a numerical representation of a component's worth to the system and determines which components are the most necessary to the system's overall functioning. In effect, it quantifies the relationship between Critical and Non-Critical components. Below a specified threshold, all components will be Non-Critical because they are less important to the system's core functionality. Similarly, all components above that threshold will be Critical. This numerical value provides more information than the simple categorization of Critical versus Non-Critical, as some Critical components may be more vital to a system than another and so require more careful attention to bad values.

Finally, the last weight for the detection functions is Probability, *P*, which measures the likelihood of a given component producing a bad value, either because of some innate quality of the component or its likelihood to be targeted for malicious actors. These three weights combined provide a more accurate measurement of each component's integrity, and therefore the integrity of the entire system, which is used when we search through the computer system for a specific component at fault.

3.3 System Search

If the IV, or SIV if checking both Critical and Non-Critical components, is below the specified threshold, then we search the system. First, all the relevant components are placed into a priority queue. Because our focus is on Critical components, the rest of this section will assume only Critical components are being searched, though this can be expanded to include Non-Critical components, like the Detection Function, and is noted in the pseudocode of the algorithm. The order of components within the priority queue is based on a set of three weights, *I*, Importance, and *P*, Probability, the same weights from the Detection Function, and a new weight, *T*, Time since last used. These weights will be explained more in the following section.



Algorithm 1 System-Wide Component Search

```
Initialize Priority Queue, $P$, with $c \in C$, sorted by $w_{SA}$
for each $c \in P$ do
    check $c$ for bad value
    mark $c$ as searched
    if $c$ contains bad value then
        Notify System
        $x$-bounded DFS through relatives
    end if
end for
```

Once the components are in place within the priority queue, components are checked for bad values. While the specifics are left to the individual system, once a component is checked, it is marked as such. This continues until a bad value is located. When that occurs, the component is marked as containing a bad value for it to be removed, the component reset, or in some way remedied determined by the individual system.

There exists the possibility, however, that the bad value did not originate from the component that was just found. It is possible that the component's source was modified maliciously and propagated a bad value forward. To remedy this, once a bad value is located, a depth-first search is performed on the sources of the located component to find any components that may have passed the bad value. Theoretically, this depth-first search could continue until every component related to a source is checked, which is vastly inefficient and unlikely to return a related bad value. As such, we propose limiting the search to a certain depth, that can be set according to a system's needs.

While a component could receive a bad value from its source, the reverse is also true and must be checked. In addition to searching the sources of the located component, that component's receivers need to be checked to ensure the bad value has not spread further through the system. This should also be limited to a certain depth, as with the sources.

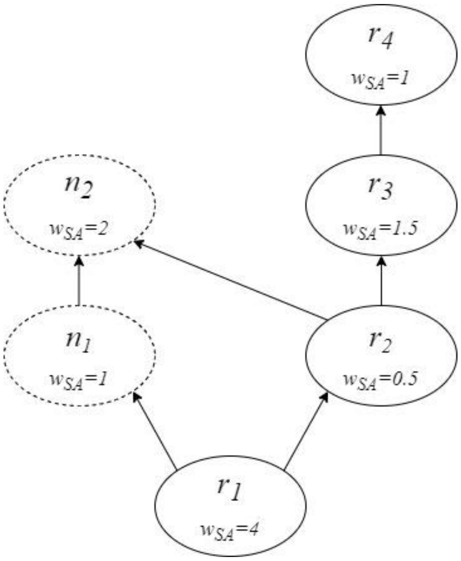


Figure 8: *System w/ Search Algorithm Weights*

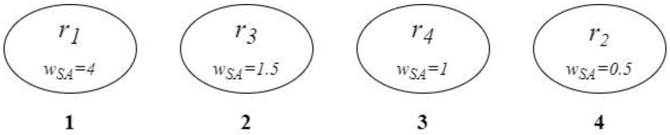


Figure 9: *Priority Queue*

Figure 8 begins an example of performing a search through a computer system. Figure 8 shows a simple system structure, with the relevant weights added to each component. The weights in this example are chosen arbitrarily, though the weights in practice will be calculated from observable or assigned parameters. This example only searches through Critical components, for simplicity, though an actual search could search through Non-Critical components as well, which is why the weights are added to the Non-Critical components even though they are not used. The Critical components are subsequently sorted by their weights and then inserted into a priority queue, as shown in Figure 9. The first component in the priority queue, r_1, is taken from the priority queue and checked for bad values as shown in Figure 10, though none are found. The search continues with the next component in the priority queue, r_3, which does return a bad value, shown in Figure 11. Because a bad value was discovered in r_3, Figure 12 shows the sources and receivers of r_3 being searched. This is typically bounded to a certain depth, but because r_1, an indirect source of r_3, has already been checked and cleared for bad values, the only remaining relatives to check are r_4 and r_2, though no bad values are found in either. This leads to Figure 13 showing that all the relevant components have been searched with only r_3 containing a bad value.

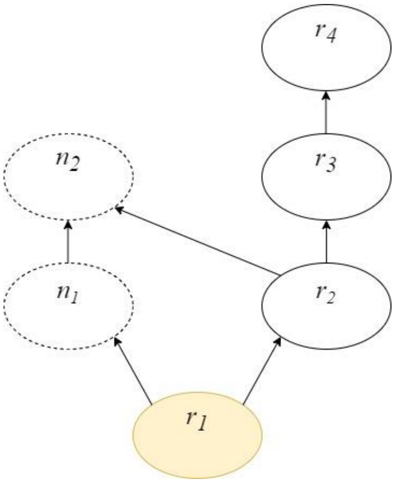


Figure 10: *1st Search Step*
No bad values found

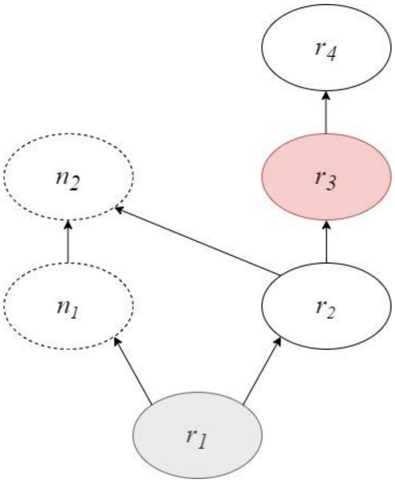


Figure 11: *2nd Search Step*
Bad value found

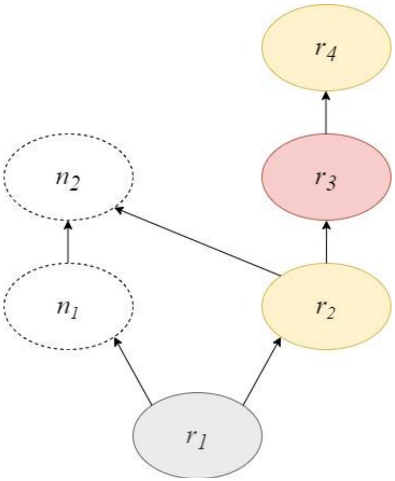


Figure 12: *3rd Search Step*
No bad values found among relatives

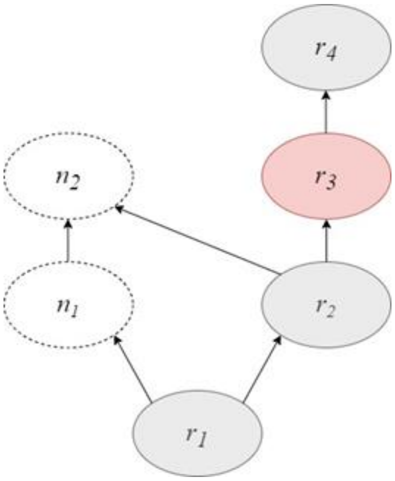


Figure 13: *Final Search State*
1 bad value discovered total

3.4 Search Algorithm Weights

These weights are used to determine which components should be searched first, achieved by placing those components sooner in the priority queue. We use three weights for this purpose, Importance, I, and Probability of bad value, P, the same weights as used for the Detection Function earlier, and another weight, T, Time since component was last used. These weights are derived similarly to the weights used in the Detection Function, from analysis of a particular computer system, with the value of I and P being the same for the search as they are for the Detection Function. If a component has not been used since T, then any bad value produced by that component is from that time, and if T is large, that corresponds to a lower likelihood that this component is currently affecting system integrity negatively. That is, if T is large, then it's corresponding component will be further down in the priority queue. The weights are combined to produce a single weight, for simplicity.

$$w_{SA,i} = I_i \times P_i \times \frac{1}{T}$$

3.5 Task-Based Detection

The detection function presented in Section 3.1 provides a value representing the system-wide integrity of Critical components, or optionally Non-Critical components. This information allows us to determine if bad values are present, or a large enough problem to warrant locating components. However, it is only able to determine if the integrity of the system is failing, but nothing else, such as which components are at fault, pushing that to a system-wide search which is already not time efficient in large enough systems. Ideally, there would be some way to narrow the potential search space from the entire system. Furthermore, it is unable to account for systems containing multiple disjoint subsets of components as it calculates the value of all Critical/Non-Critical components together, assuming all components influence every other component. This would lead to a case where a bad value could be present in a single subset of components, bringing down the IV, or SIV, though because the DF provides no information on which subset contains the bad values, time is wasted searching components that should not be considered at all.

To solve this issue, we leverage the grouping of components by task, as described in Section 2.8. As stated previously, it is reasonable to assume that a component will be included in multiple tasks within a system and, further, will have different weights for each task. By modifying the detection function from Section 3.1 to calculate an IV for all components within a *task* rather than the entire system, we get multiple benefits. Firstly, the size of a task will necessarily be smaller than the size of the entire system, already providing more information for where the affected component is located, allowing us to narrow our search space. This is because tasks are subsets of a computer system, meaning at worse, a computer system will contain a task that is equal in size or, from another perspective, the computer system consists of a single task. This is highly unlikely, especially as a system grows and the time difference becomes more noticeable, and so that case would in fact gain very little from this updated implementation and would perform similarly using the standard DF. Secondly, if a component is present in multiple tasks, which is more likely for larger system, the outputs, or IVs, of multiple DF's can be cross-referenced to look for common components between the tasks. This is the best case scenario, as it allows us to further pinpoint in which component a bad value is occurring. Both the smaller size of tasks and potential for cross-referencing reduce the time spent searching the system for bad values and make calculating IVs by task much more effective and valuable searching through an entire system.

The Task-based Detection Function is presented below:

$$TDF(C) = \frac{w_1p_1v_1 + w_2p_2v_2 + w_3p_3v_3 + \cdots + w_ip_iv_i}{\#\ of\ components\ in\ C} = IV, \qquad p = 1\ if\ present, else\ p = 0$$

There is major difference between this new function and the original detection function, *p*, which is specific to each component. This new value represents if a given component is present within the task the TDF is currently performing on. That is, if a component is present within a given task, *p* is 1. Otherwise, *p* is 0. This generalizes the function, only requiring us to keep track of which components are in which task, something that is trivial to do given our model, instead of needing to modify the function for each task.

The search algorithm changes very little to accommodate task-based detection. Instead of the entire system being searched at a time, only tasks that indicate bad values will be searched. The result will look very similar to the original search example from the previous section, but whereas the previous example represented an entire system structure, in this modified version, that would be a single task within a system. As many tasks are separate from one another, it can be seen as many smaller searches occurring throughout the system simultaneously, though the result remains the same.

Digital Twins

4.1 Definition

All the methods presented in Section 3 share one common weakness; they only detect if a bad value is already present within the system. This means in the time between a component producing a bad value and an integrity check is performed, damage could already be spreading throughout the system and potentially cause catastrophic damage before it is ever noticed. This is not to say the previous methods are not valuable, as they still provide valuable information in detecting bad values, however with our main goal to maximize the uptime in a computer system, it is better to prevent bad values from ever occurring in the first place, eliminating the possibility of a bad value cascading through the system.

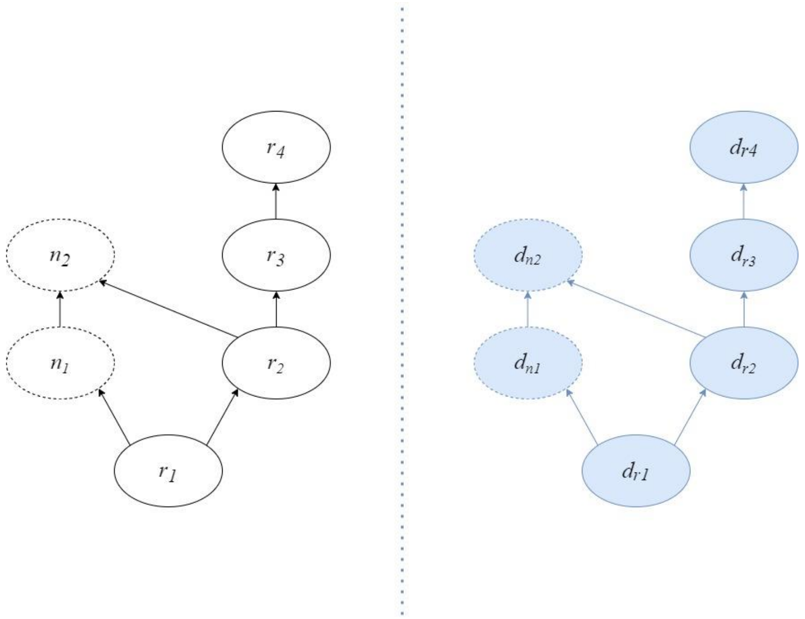


Figure 14*: System and Respective Digital Twins*
Each component has a digital twin, which essentially recreates the structure

To prevent such an occurrence from happening within a component, we would need some way to test the values going into a component first. We can accomplish this with digital twins. A *digital twin* is a simulation of an actual object synchronized to match any changes occurring to the original. That is, if a component *c* has a digital twin, denoted d_c, and the same change was applied to both *c* and d_c, both would remain identical after the change, such is the synchronicity between *c* and d_c. Figure 14 gives a basic view of implementing digital twins in a system, where the right layout is the actual system and the left being the digital twins, the “digital” system. Digital twins are often used to test changes to systems before they implemented in a physical system, and some components within a computer system are physical, though it is possible that some components only exist within the system, that is they virtual. This does not create a problem for using digital twins, as a digital twin can still simulate a virtual component and its function. If a component had a digital twin, each input could be applied to the digital twin first to see if the actual component would remain within the qualified ranges. With this, we can prevent a bad value from passing into the component, keeping the system from accepting bad input and potentially cause unsafe operating conditions.

4.1 Digital Twin Pre-Change Verification

We implement this idea of applying values to a digital twin before the actual component through what we call *digital twin pre-change verification*. With digital twin pre-change verification, anytime a component receives input, that input will be simulated in a digital twin for that component. If the resulting output in the digital twin is within the qualified range, the input can be passed to the component. However, if the input results in a value outside of the qualified range, the input can be rejected, or an alert be raised. Normal system operation should not create bad values, so if a digital twin does detect one, the malicious value must come from one or more of the input components. It is possible that an acceptable value in one component, when sent to another component as input, can result in a bad value in the receiving component. However, it is possible that the component at fault is indirectly related to the component at fault, potentially being separated by one or two intermediary nodes. Because of this, digital twins can prevent bad values from occurring, though they are not enough on their own to determine the source of the bad value, either an accidental or potentially malicious modification to a component that only produces a bad value after one or more components. Digital twins can, however, narrow down the list of possible components to check, as it must be a component related to the input components.

Digital twin pre-change verification provides us the most effective solution within our model, allowing us to detect potential bad values before they can disrupt the computer system. Simply using the simple detection function and search algorithm does not accommodate for computer systems with components that are unrelated to one another, as is the case with tasks allowing for disjoint subsets of components within a system. In addition, the search algorithm has a very large search space, essentially the entire computer system, which is infeasible for any complex computer system, and may waste time searching components unrelated to the bad value, again because it does not include the concept of system tasks and instead assumes that all components are related. Task-based detection allows us to narrow the search space down to a single task at worst or, if multiple tasks are flagged and share common components, a single component at best. This certainly helps us detect bad value within components, however our goal is to keep the system operational and maximize system uptime. If we simply detect a bad value that is already present within the system, there is a window between the bad value appearing in the system and the system check being performed when major damage could occur to the computer system. Utilizing digital twins allow us to detect bad values before there is a chance for damage to occur in a system while also keeping a narrow search space for locating the bad value.

Simulation

To test that our model is beneficial, we simulate a computer system and our model searching for bad values, specifically to show differences between our initial detection function and our task-based detection function. We have not simulated digital twins in a system, though it is reasonable to see the benefits of implementing digital twins over the two detection function models, preventing bad values instead of searching for them after-the-fact.

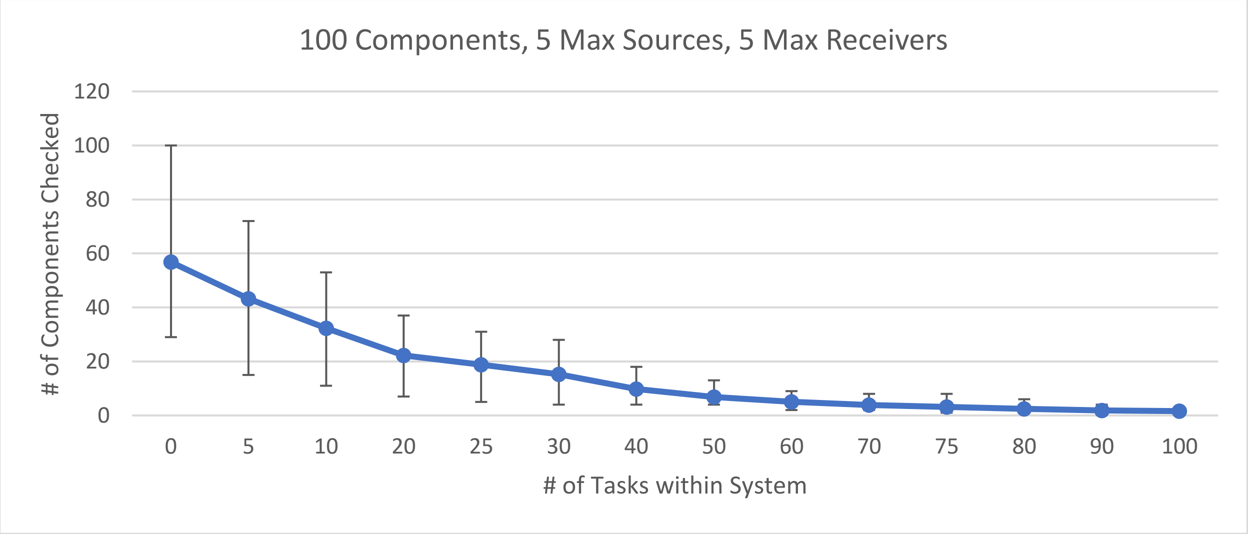
Our simulation builds a log-based computer system task-wise from the bottom-up, creating relationships between components, randomly assigning sources and receivers. In our simulation, a task contains a random number of components within the system and components have different relationships for each task. Each component within the simulation has randomly assigned weights for the detection functions and searches, as well. Within our simulation, we can control the number of components within the system, the maximum number of sources and receivers a single component can have, the threshold for Critical components, or what value of *I*, Importance, above which a component is considered Critical. We also control the percentage of components active within the system, considering our earlier assertion that not all components within a system are used. Finally, we control the number of tasks within the system, again with random components in each task. In real systems, some components would have more connections and be present in more tasks than other components. While this differs from our simulation, it is still effective, and assigning components randomly prevents bias from entering our data through task selection.

Once the system is created, a bad value is placed in a single random component within, and a search of the entire system is performed. This is analogous to the initial detection function implementation, which requires the entire system to be searched. The number of components searched is then recorded to be shown in our results. After this search and while maintaining the same system structure, we perform detection functions for a single task within the system. Since these tasks are randomly created, the bad value may or may not appear within a task. Once it is, a search is performed against that task, again tracking the number of components searched until the bad value is located. A key benefit of the task-based detection was looking for common components between tasks that signaled a bad value, and so in our simulation, as tasks are added to the system if a new task contains a bad value, only components shared between the tasks are searched. This continues until the maximum number of tasks is reached or if the only component searched is the one containing the bad value, with the total number of components searched recorded for each number of tasks within the system. These tests were performed with varying numbers of components in the system and maximum numbers of sources and receivers. To ensure that the number of components set was accurate, all components were considered Critical and were active within the system. Our results follow in the next section.

Results

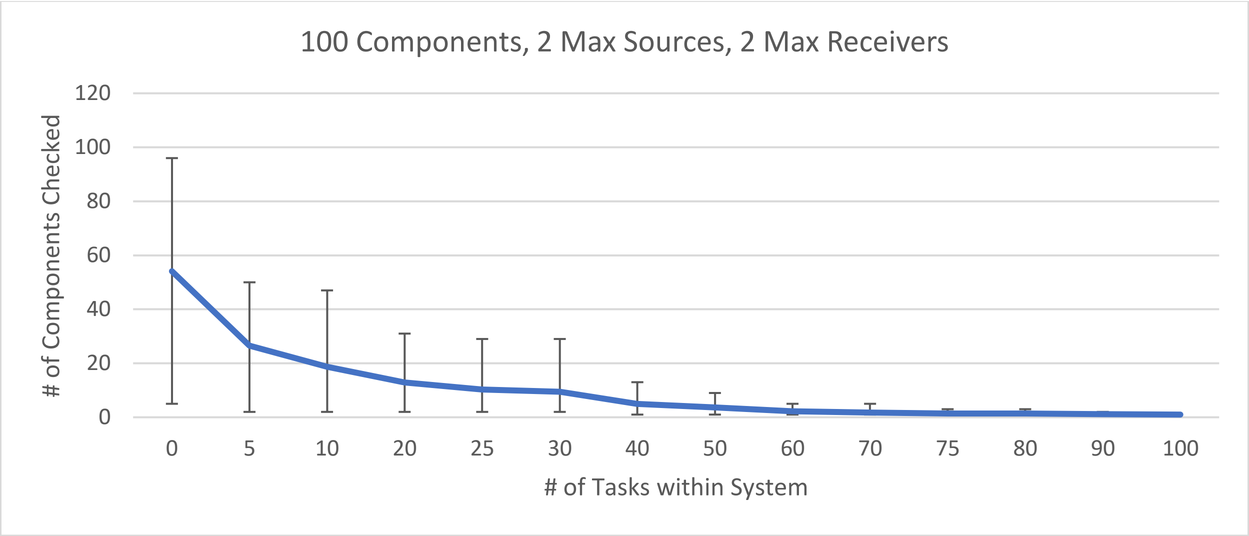
The results of our simulation testing are displayed by the number of components checked against the number of tasks present in the system. For example, the “0” mark represents checking the entire system, as if there are no tasks in a system it must all be checked. In the “5” mark, that represents the system having five tasks in total, but not necessarily five tasks containing the bad value. For each set of parameters tested, a total of ten runs were performed, the average of those ten runs are plotted on the line, while the bars above each point show the maximum and minimum values across the ten runs. The two parameters tested were the number of components within a system and the maximum number of sources and receivers to a component. The first parameter’s purpose is to determine if our model holds for larger systems, and how the two methods compare. The second parameter is included to discover any possible correlation between the efficiency of the search and the number of connections between components.

6.1 Data Set #1



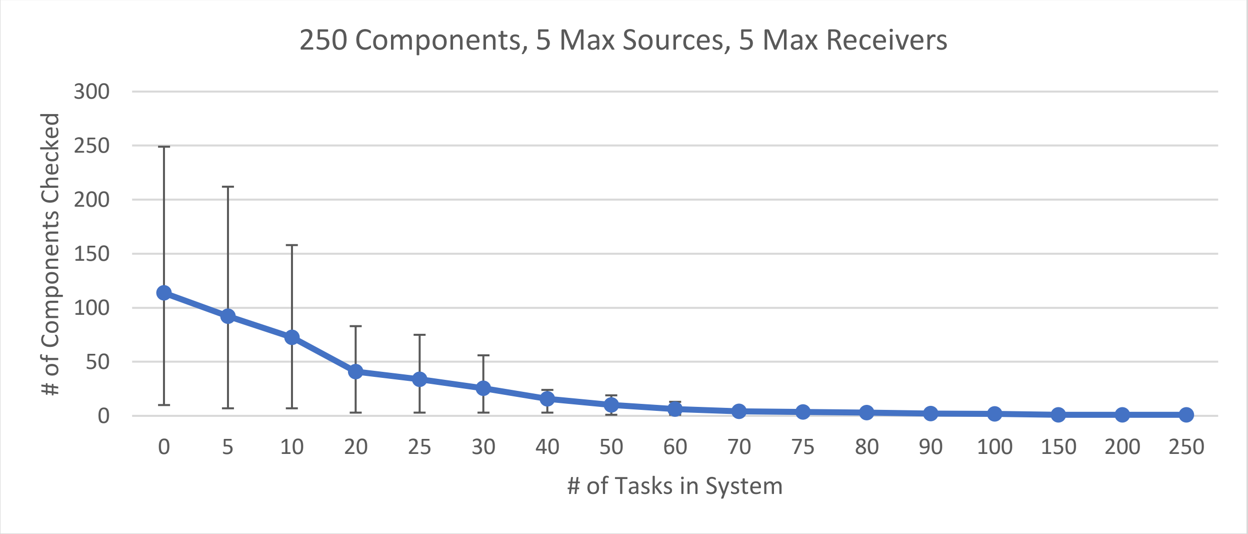
The first data set was performed on systems comprised of 100 components limited to five sources and five receivers at maximum. When searching through the entire system, it took on average just under 60 components to find the bad value though in one case it had to search every component, an unlikely but possible case. As tasks were added to the system, the search space narrows until around 90 total tasks when the bad value is the only component checked.

6.2 Data Set #2



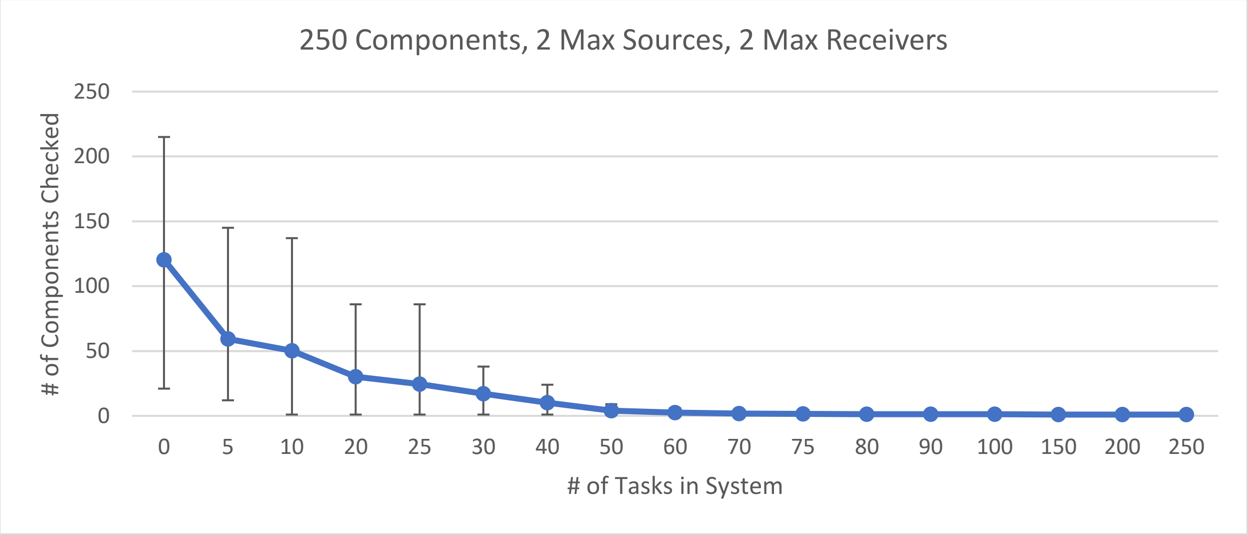
The second test contained a similar 100 components though they were limited to only two sources or receivers at maximum. Like the first set, the full system search was highly variable, though the search tended to a single component again. This was reached much faster, however, at around 70 total tasks.

6.3 Data Set #3



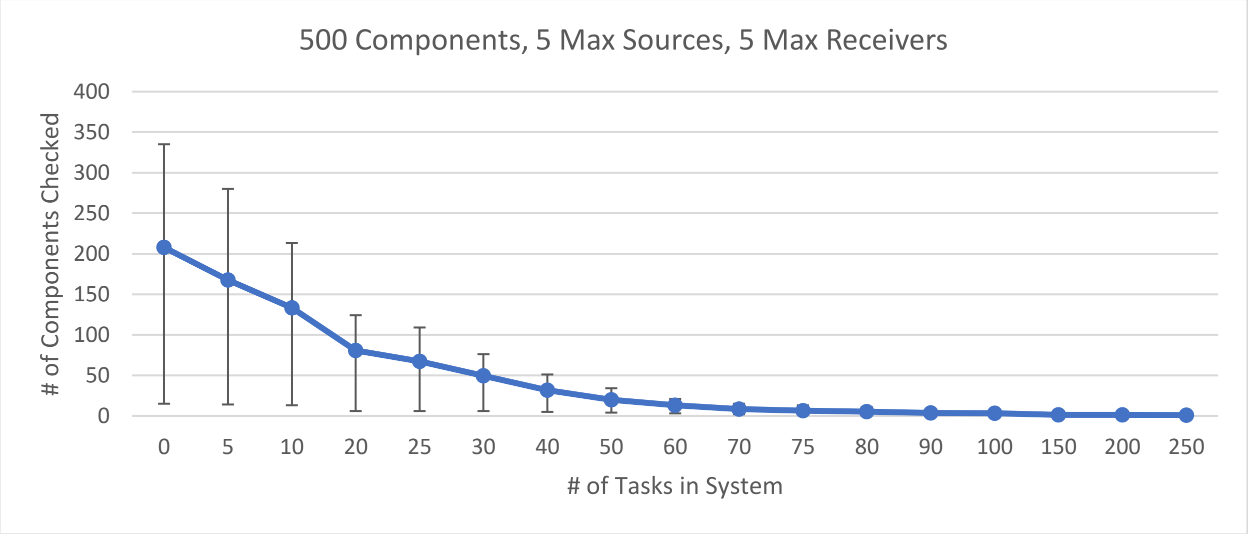
This test set increased the components in the system to 250. The maximum number of tasks was also increased to 250 in case the number of tasks needed to reach one component also increased. This showed to be true, as it took until almost 150 total tasks to reach a single component, though from 50 total task and on had on average 10 component searched, a significant increase from the full system search and smaller numbers of total tasks.

6.4 Data Set #4



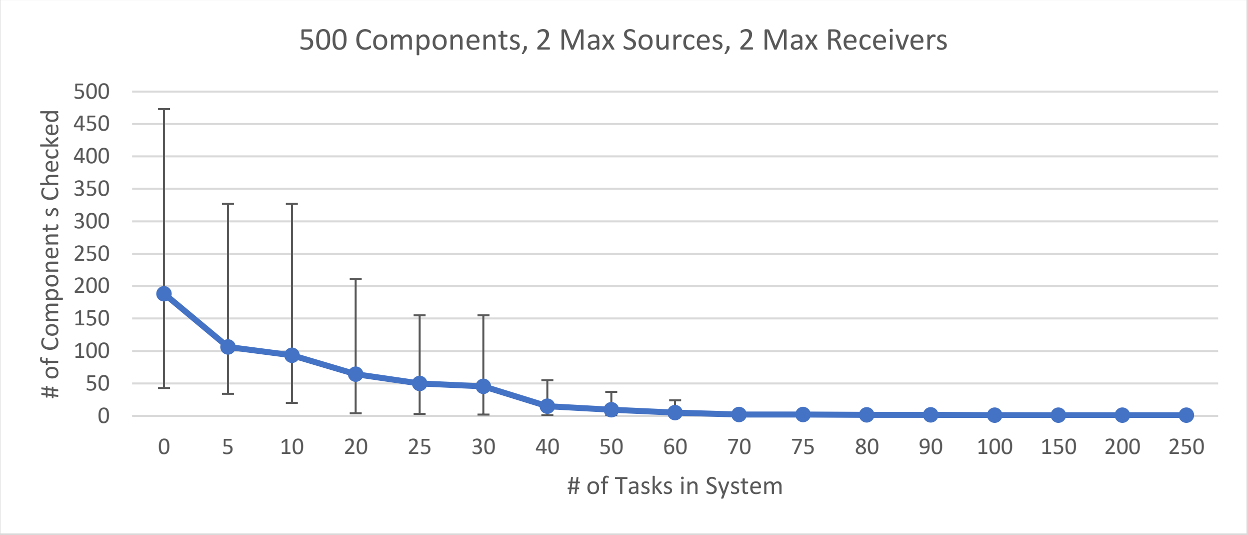
The fourth test was performed with 250 components in the system and 2 sources and receivers per component at maximum. The numbers of components checked again reaches one, though it is reached slightly quicker than the previous test at just over 100 total tasks. Also, slightly sooner than the last test, 40 total tasks marked the point where the average number of components reached below 10.

6.5 Data Set #5



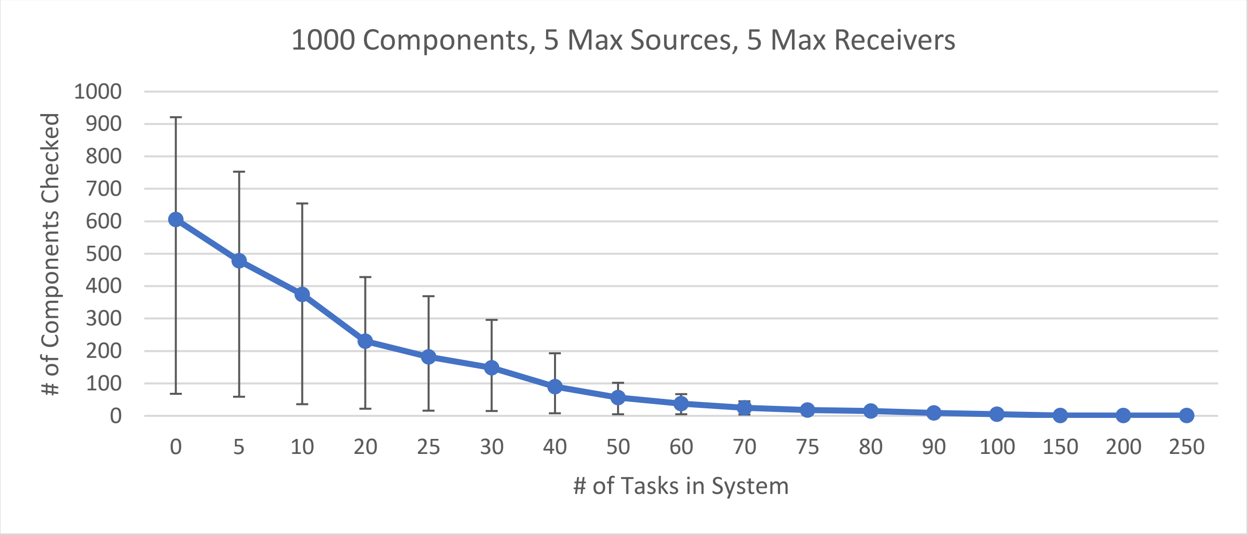
Increasing the system to 500 components does not result in much change from the previous sets. With the full system search and searches with few total tasks, the number of components checked varies wildly, but still exhibits a clear trend to checking only the correct component.

6.6 Data Set #6



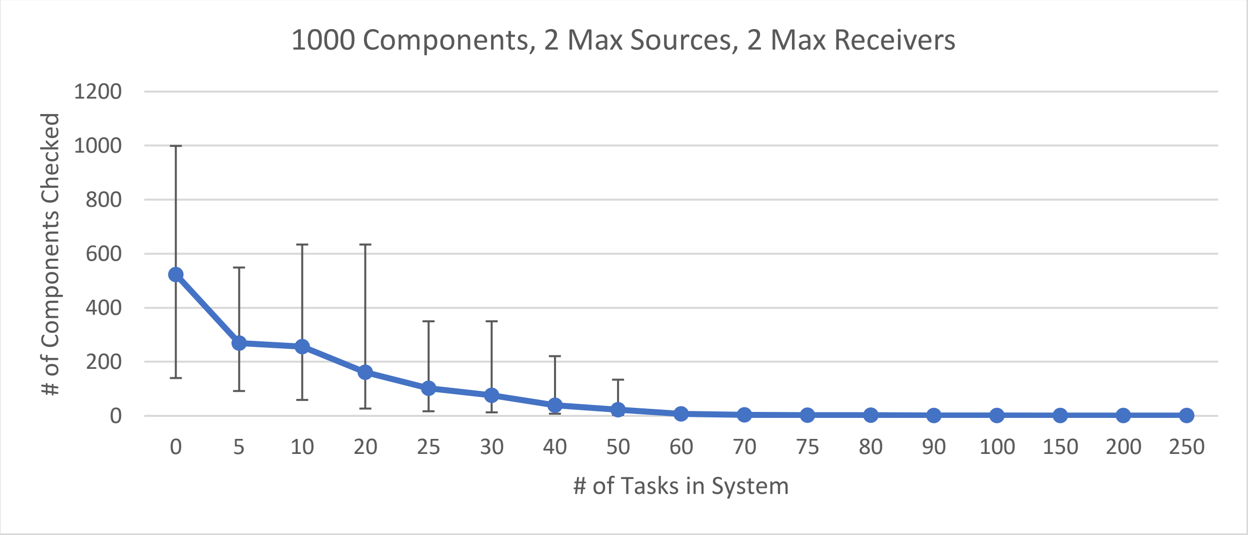
With 500 components and two sources and two receivers at maximum, the average reaches below two components searched on average at 70 total tasks, and finally reaching only a single component searched at around 150 total tasks. Due to the randomness with our weights and tasks, this set averages lower than expected in the first few marks.

6.7 Data Set #7



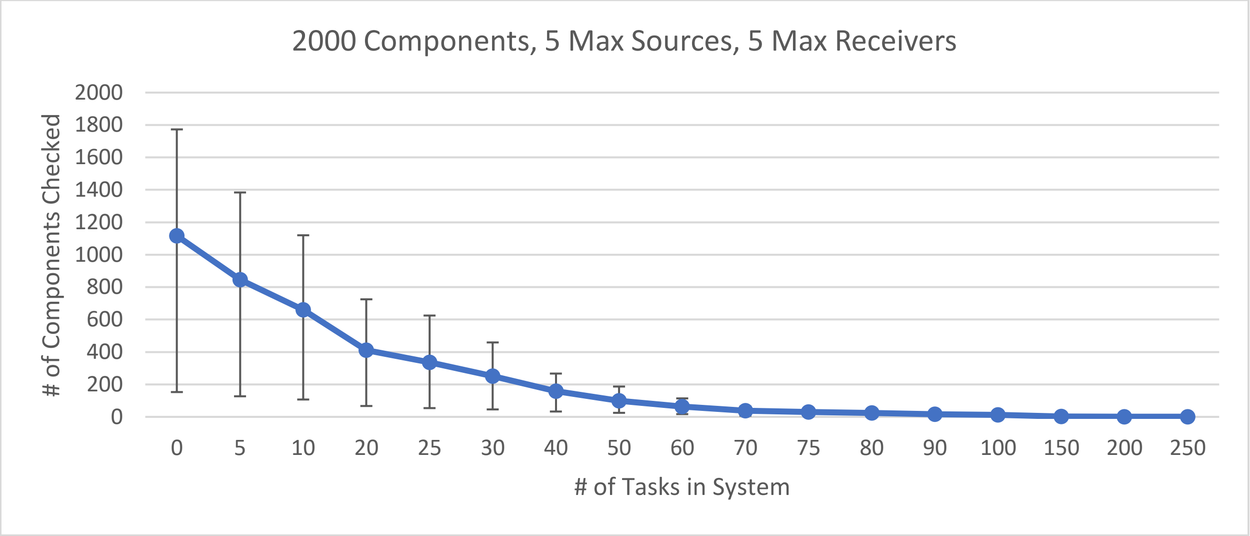
Increasing the number of components to 1,000 leads to an expected increase in the number of tasks needed to reach a single component searched on average, taking until almost 200 total tasks. Likewise, it takes until 90 total tasks to reach under 10 components searched on average. Overall, the decrease of components checked as the number of tasks increase becomes more noticeable.

6.8 Data Set #8



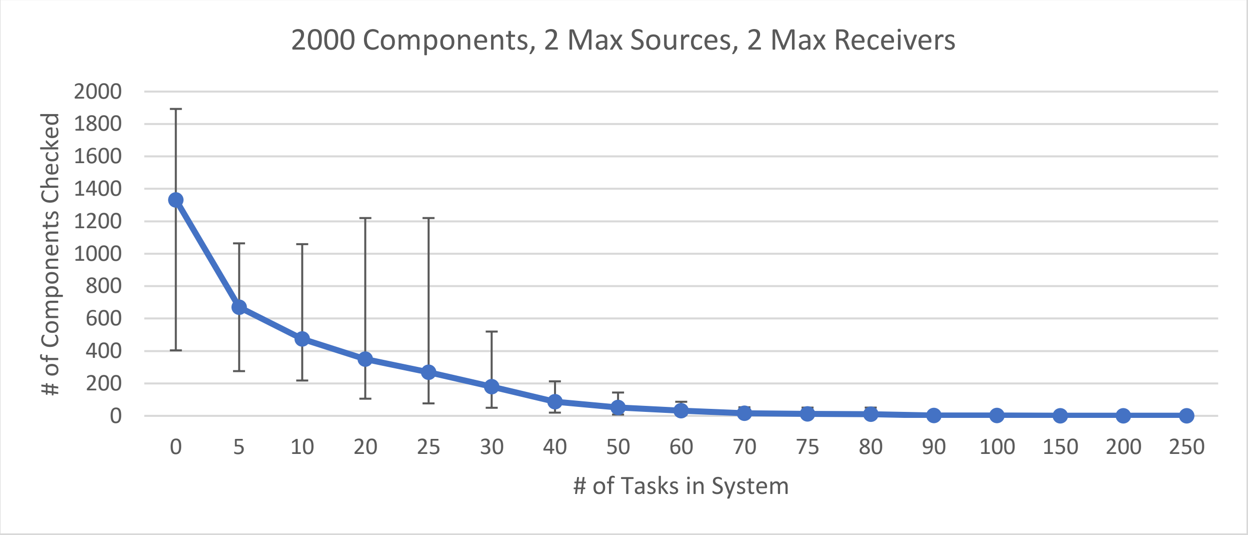
Lowering the maximum number of connections per component greatly reduces the number of components searched, reaching below 10 components searched on average at just 60 total tasks. Note that the maximum for 5 total tasks is lower than for 10 and 20 tasks. This is a result of the random nature of task selection in our simulation, where for certain runs the bad value was not found within the first five tasks. With a larger system and fewer connections between components, this is to be expected.

6.9 Data Set #9



With 2,000 components, the searches take longer to reach a minimum, only reaching below 10 components around 100 total tasks and reaching a single component at over 150 tasks. Relative to the size of the system, this is quite small and a large improvement over the full system search.

6.10 Data Set #10



With the reduced number of component connections, the average reaches below 10 components at 80 total tasks and reaches one at just over 100 total tasks, which is even faster than with 5 maximum sources and receivers. Once again, with 5 and 10 total tasks, the maximum is lower than for 20 and 25 total tasks for the same reason as in the sixth dataset, that because of the large number of components and small number of connections between them, the bad value was not present in the first 10 tasks for the largest run.

6.11 Result Analysis

Overall, the data shows that an increase in the total number of tasks results in a reduction in the number of components that must be checked to locate a bad value, even as the size of the system grows. This fully supports our assertion that while the original detection function can successfully locate a bad value, utilizing the shared components between tasks results in a smaller search space and thus requires fewer components to be checked. Our results also reveal a correlation that we did not expect from our model, that the fewer number of connections between components influence the number of components searched. For the datasets with 2 maximum sources and receivers per component, they reached lower numbers of components searched faster than the 5 maximum datasets in all our tests. This makes sense, as with potentially fewer components in a task, there are less components that are present in multiple tasks. This means the number of components shared between tasks that detected a bad value will naturally be smaller, as well.

Conclusion

This paper explores modeling the structure of computer systems in Cyber-Physical systems to detect and locate bad values to maintain computer system integrity and functionality. We present a new scalable framework that is capable of modeling various kinds of computer systems. We can then monitor a system's integrity through its components, split between core functionality in Critical components and non-essential Non-Critical components, and apply this knowledge to locating the bad values. We also explore optimizations of this framework, taking advantage of the grouping of components into tasks to improve the efficiency of our approach. We finally discuss the inclusion of Digital Twins and a pre-change verification technique to pro-actively detect bad values before they propagate through a system, which further improves the effectiveness of our approach.

Future work in this area includes performing additional simulations for the task-based detection function, case study testing on the present framework and testing of the digital twin pre-change verification methodology. Additional work could also be done in expanding the definitions of our system model, allowing for the work to apply to more complex situations where CPS are implemented.

References

[1] Ding, Jianguo & Atif, Y. & Andler, Sten & Lindström, Birgitta & Jeusfeld, Manfred. (2017). CPS-based Threat Modeling for Critical Infrastructure Protection. ACM SIGMETRICS Performance Evaluation Review. 45. 129-132. 10.1145/3152042.3152080.

[2] ŘEHÁK, David, Jiří MARKUCI, Martin HROMADA a Karla BARČOVÁ. Quantitative evaluation of the synergistic effects of failures in a critical infrastructure system. International Journal of Critical Infrastructure Protection [online]. 2016, 14, 3-17 [cit. 2021-04-22]. ISSN 2212-2087. Dostupné z: doi:10.1016/j.ijcip.2016.06.002

[3] Guo, Shuo & Zhang, Heng & Zhong, Ziguo & Chen, Jiming & Cao, Qing & He, Tian. (2014). Detecting Faulty Nodes with Data Errors for Wireless Sensor Networks. ACM Transactions on Sensor Networks. 10. 1-27. 10.1145/2594773.

[4] Krunic, Veljko & Trumpler, Eric & Han, Richard. (2007). NodeMD: Diagnosing node-level faults in remote wireless sensor systems. MobiSys'07: Proceedings of the 5th International Conference on Mobile Systems, Applications and Services. 43-56. 10.1145/1247660.1247669.

[5] Zhang, Tianyu & Zhao, Qian & Nakamoto, Yukikazu. (2017). Faulty Sensor Data Detection in Wireless Sensor Networks Using Logistical Regression. 13-18. 10.1109/ICDCSW.2017.37.

[6] F. Aeiad, W. Gao and J. Momoh, "Bad data detection for smart grid state estimation," 2016 North American Power Symposium (NAPS), 2016, pp. 1-6, doi: 10.1109/NAPS.2016.7747983.

[7] Y. Wu, Y. Xiao, F. Hohn, L. Nordström, J. Wang and W. Zhao, "Bad Data Detection Using Linear WLS and Sampled Values in Digital Substations," in IEEE Transactions on Power Delivery, vol. 33, no. 1, pp. 150-157, Feb. 2018, doi: 10.1109/TPWRD.2017.2669110.

[8] Angelos, Eduardo Werley S. and E. Asada. “Bad data processing for real-time equivalent networks.” 2015 IEEE Power & Energy Society General Meeting (2015): 1-5.

[9] H. M. Merrill and F. C. Schweppe, "Bad Data Suppression in Power System Static State Estimation," in IEEE Transactions on Power Apparatus and Systems, vol. PAS-90, no. 6, pp. 2718-2725, Nov. 1971, doi: 10.1109/TPAS.1971.292925.

[10] B. M. Zhang and K. L. Lo, "A recursive measurement error estimation identification method for bad data analysis in power system state estimation," in IEEE Transactions on Power Systems, vol. 6, no. 1, pp. 191-198, Feb. 1991, doi: 10.1109/59.131062.

[11] li, Qiao & Weng, Yang & Negi, Rohit & Ilic, Marija. (2015). Convexification of bad data and topology error detection and identification problems in AC electric power systems. IET Generation, Transmission & Distribution. 9. 10.1049/iet-gtd.2015.0191.

[12] T. V. Cutsem, M. Ribbens-Pavella and L. Mili, "Bad Data Identification Methods In Power System State Estimation-A Comparative Study," in IEEE Transactions on Power Apparatus and Systems, vol. PAS-104, no. 11, pp. 3037-3049, Nov. 1985, doi: 10.1109/TPAS.1985.318945.

[13] Grieves, Michael. (2015). Digital Twin: Manufacturing Excellence through Virtual Factory Replication.

[14] Sebastian Haag, Reiner Anderl, “Digital twin – Proof of concept”, Manufacturing Letters, Volume 15, Part B, 2018, Pages 64-66, ISSN 2213-8463, Jan. 2018

[15] F. Tao, H. Zhang, A. Liu and A. Y. C. Nee, "Digital Twin in Industry: State-of-the-Art," in IEEE Transactions on Industrial Informatics, vol. 15, no. 4, pp. 2405-2415, April 2019, doi: 10.1109/TII.2018.2873186.

[16] A. Rasheed, O. San and T. Kvamsdal, "Digital Twin: Values, Challenges and Enablers From a Modeling Perspective," in IEEE Access, vol. 8, pp. 21980-22012, 2020, doi: 10.1109/ACCESS.2020.2970143.

[17] Elisa Negri, Luca Fumagalli, Marco Macchi, A Review of the Roles of Digital Twin in CPS-based Production Systems, Procedia Manufacturing, Volume 11, 2017, Pages 939-948, ISSN 2351-9789, https://doi.org/10.1016/j.promfg.2017.07.198.

[18] Boschert S, Rosen R. Digital Twin – The Simulation Aspect. In: Hehenberger P, Bradley D, editors. Mechatronic Futures. Cham: Springer International Publishing; 2016.

[19] Thomas H.-J. Uhlemann, Christian Lehmann, Rolf Steinhilper, The Digital Twin: Realizing the Cyber-Physical Production System for Industry 4.0, Procedia CIRP, Volume 61, 2017, Pages 335-340, ISSN 2212-8271, https://doi.org/10.1016/j.procir.2016.11.152.

[20] M. Schluse, M. Priggemeyer, L. Atorf and J. Rossmann, "Experimentable Digital Twins—Streamlining Simulation-Based Systems Engineering for Industry 4.0," in IEEE Transactions on Industrial Informatics, vol. 14, no. 4, pp. 1722-1731, April 2018, doi: 10.1109/TII.2018.2804917.

[21] Rüßmann, Michael et al. “Industry 4 . 0 : The Future of Productivity and Growth in Manufacturing Industries April 09.” (2016).

[22] C. Brosinsky, D. Westermann and R. Krebs, "Recent and prospective developments in power system control centers: Adapting the digital twin technology for application in power system control centers," 2018 IEEE International Energy Conference (ENERGYCON), 2018, pp. 1-6, doi: 10.1109/ENERGYCON.2018.8398846.

[23] A. Joseph, M. Cvetkovic, and P. Palensky, ``Predictive mitigation of short term voltage instability using a faster than real-time digital replica," in Proc. IEEE PES Innov. Smart Grid Technol. Conf. Eur. (ISGT-Eur.), Oct. 2018, pp. 1-6.

[24] N. Jazdi, "Cyber physical systems in the context of Industry 4.0," 2014 IEEE International Conference on Automation, Quality and Testing, Robotics, 2014, pp. 1-4, doi: 10.1109/AQTR.2014.6857843.

[25] Zug, Sebastian & Brade, Tino & Kaiser, Jörg & Potluri, Sasanka. (2012). An Approach Supporting Fault-Propagation Analysis for Smart Sensor Systems. 7613. 162-173. 10.1007/978-3-642-33675-1_14.

[26] C. Alippi, S. Ntalampiras and M. Roveri, "Model-Free Fault Detection and Isolation in Large-Scale Cyber-Physical Systems," in IEEE Transactions on Emerging Topics in Computational Intelligence, vol. 1, no. 1, pp. 61-71, Feb. 2017, doi: 10.1109/TETCI.2016.2641452.

[27] Srivastava, N., & Srivastava, J. (2010). A hybrid-logic approach towards fault detection in complex cyber-physical systems. Annual Conference of the PHM Society, 2(1). https://doi.org/10.36001/phmconf.2010.v2i1.1888

[28] Barry Dowdeswell, Roopak Sinha, Stephen G. MacDonell, Finding faults: A scoping study of fault diagnostics for Industrial Cyber–Physical Systems, Journal of Systems and Software, Volume 168, 2020, 110638, ISSN 0164-1212, https://doi.org/10.1016/j.jss.2020.110638.

[29] F. Pasqualetti, F. Dörfler and F. Bullo, "Attack Detection and Identification in Cyber-Physical Systems," in IEEE Transactions on Automatic Control, vol. 58, no. 11, pp. 2715-2729, Nov. 2013, doi: 10.1109/TAC.2013.2266831.

[30] John D. McGregor, David P. Gluch, and Peter H. Feiler. 2017. Analysis and Design of Safety-critical, Cyber-Physical Systems. Ada Lett. 36, 2 (December 2016), 31–38. DOI:https://doi.org/10.1145/3092893.3092899

[31] Lasi, Heiner, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. "Industry 4.0." Business & information systems engineering 6, no. 4 (2014): 239-242.

[32] Jay Lee, Behrad Bagheri, Hung-An Kao, A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems, Manufacturing Letters, Volume 3, 2015, Pages 18-23, ISSN 2213-8463, https://doi.org/10.1016/j.mfglet.2014.12.001.

[33] Fei Tao, Qinglin Qi, Lihui Wang, A.Y.C. Nee, Digital Twins and Cyber–Physical Systems toward Smart Manufacturing and Industry 4.0: Correlation and Comparison, Engineering, Volume 5, Issue 4, 2019, Pages 653-661, ISSN 2095-8099, https://doi.org/10.1016/j.eng.2019.01.014.

[34] J. Shi, J. Wan, H. Yan and H. Suo, "A survey of Cyber-Physical Systems," 2011 International Conference on Wireless Communications and Signal Processing (WCSP), 2011, pp. 1-6, doi: 10.1109/WCSP.2011.6096958.

[35] E. A. Lee, "Cyber Physical Systems: Design Challenges," 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), 2008, pp. 363-369, doi: 10.1109/ISORC.2008.25.

[36] L. Monostori, B. Kádár, T. Bauernhansl, S. Kondoh, S. Kumara, G. Reinhart, O. Sauer, G. Schuh, W. Sihn, K. Ueda, Cyber-physical systems in manufacturing, CIRP Annals, Volume 65, Issue 2, 2016, Pages 621-641, ISSN 0007-8506, https://doi.org/10.1016/j.cirp.2016.06.005.

[37] Lihui Wang, Martin Törngren, Mauro Onori, Current status and advancement of cyber-physical systems in manufacturing, Journal of Manufacturing Systems, Volume 37, Part 2, 2015, Pages 517-527, ISSN 0278-6125, https://doi.org/10.1016/j.jmsy.2015.04.008.

[38] Cardenas, Alvaro, Saurabh Amin, Bruno Sinopoli, Annarita Giani, Adrian Perrig, and Shankar Sastry. "Challenges for securing cyber physical systems." In Workshop on future directions in cyber-physical systems security, vol. 5, no. 1. 2009.

[39] Cardenas, Alvaro & Amin, Saurabh & Sastry, Shankar. (2008). Research Challenges for the Security of Control Systems.