Simulation and Analysis of Tracing Damage Paths and Repairing Objects in Critical Infrastructure Systems

Justin Burns, Brajendra Panda, and Thanh Bui
Computer Science and Computer Engineering Department
University of Arkansas
Fayetteville, AR 72701 USA
email: {jdb083, bpanda, tbui}@uark.edu

***Abstract*—Critical infrastructure systems have recently become more vulnerable to attacks on their data systems through internet connectivity. If an attacker is successful in breaching a system's defenses, it is imperative that operations are restored to the system as quickly as possible. This research focuses on damage assessment and recovery following an attack. We review work done in both database protection and critical infrastructure protection and establish our own definitions of how damage affects the relationships between data and software. Then, we propose a model using a graph construction to show the cascading effects within a system after an attack. We also present an algorithm that uses our graph to compute an optimal recovery plan that prioritizes the most important damaged components first so that the vital modules of the system become functional as soon as possible. This allows for the most critical operations of a system to resume while recovery for less important components is still being performed. Lastly, we show results from simulations using our algorithm on data graphs with various parameters.**

Keywords-critical infrastructure; damage assessment; recovery.

I. INTRODUCTION

Critical infrastructure systems are those that are considered extremely critical to the functioning of a government or a country. As described in [2], critical infrastructures are like the vital organs of a body that need to perform their own roles for the human body to function efficiently and painlessly. The US Department of Homeland Security [3] declares that such systems are "so vital to the United States that their incapacity or destruction would have a debilitating impact on our physical or economic security or public health or safety." Therefore, the protection and smooth functioning of our nation's critical infrastructures are indispensable and cannot be ignored.

These systems are becoming prime targets of attackers – primarily state actors and organized crime – and a major attack on one can cripple the economy of the victim nation. These systems are also more likely to be connected to the internet now to provide benefits like cost reduction (where large systems can be remotely managed over the public network), increased capability (by providing sufficient computing resources for infrastructure hardware with less capability power), and improved efficiency and transaction speed. This connectivity unfortunately makes it easier for attackers to hack into these systems. Consider the New York Times report about the attack on Colonial Pipeline [4]. While the details of the attack are not yet disclosed, a group of cybercriminals was able to compromise data systems using the internet, which resulted in Colonial Pipeline shutting down their pipeline. This outage affected mass transit and other industries across the entire U.S. East Coast and exposed a lack of preparation for such a crisis. This illustrates how an external system can have a relationship with a critical infrastructure system and how such relationships can be exploited to carry out an attack.

It is clear from past incidents and recent reports [5]-[8], (to cite just a few) that attacks on critical infrastructures are occurring frequently, which indicates that prevention mechanisms are not enough to stop them. Thus, it is of utmost importance to aggressively prepare for post-attack activities, which include damage assessment and recovery mechanisms that are critical to making the affected systems available at full functioning mode as soon as possible. This research aims at meeting this important goal.

We propose a framework that models damage spread within a set of data objects based on object dependencies and prioritizes making repairs to the most critical objects first. The framework is based on some of the models explored in critical infrastructure protection and uses a version of previously proposed repair methods that is modified to focus on meeting specific goals when determining the order in which repairs are made.

The rest of the paper is an extension of the work in [1] and is organized as follows. Section 2 offers some work performed in this area. Section 3 defines the problem that we aim to build our model for. In section 4, the types of relationships that exist in data systems are explained. We provide details on our model in section 5, which includes three subsections to explain our definitions, model description, and algorithm. We then show experiments using our model and present the results in section 6. Section 7 concludes our work.

II. RELATED WORKS

This paper aims to examine methods and frameworks used for database and critical infrastructure protection and apply them towards protecting a set of data objects. This section describes some of the publications that are relevant to our proposed framework. Various types of critical infrastructure objects are described in [17]. The focus of our work falls under network and network nodes, which are defined as a "structure with one dominating dimension", and

junctions within the network, respectively. Furthermore, [18] establishes the concept of resilience, a cyclical process by which a system undergoes recovery, adaptation, and prevention between attacks. Turning this concept into a concrete value that can be used in risk assessment has been a point of interest in protecting systems [20]. Efforts have also been made to quantify vulnerability as a measurement that can be used to determine how frequently or severely a system can be at risk [19]. One of the major works on damage assessment and recovery within a database uses data dependency to find data affected by an attack to optimize recovery [9]. While this method relies on the direct relationships between data items, an alternate model to recover data from an attack instead uses the transaction log for assessment [10].

Kotzanikolaou et al. describe a model in [11] that assists in risk assessment for possible scenarios that can result in cascading failures within a CI system. For critical infrastructures with data-rich operations, the use of Cyber-Physical Systems can cause new vulnerabilities as described in [12]. Their model analyzes threats that can appear due to these vulnerabilities and analyzes the potential cascading damage they can cause. System dynamics modeling can also be used to analyze disruptive events to characterize such disruptions to critical infrastructure by risk assessment and various impact factors as shown in [13].

Rehak et al. [14] model an infrastructure system as elements and linkages with different types of relationships establishing dependencies and interdependencies. They note that these elements can have varying criticality, causing some elements to cascade more damage into the system than others in the event of a failure. This work is important because by establishing criticality, they quantify damage within a system. We use this concept of criticality later in this paper to direct the optimal repair path of data objects.

We also consider models that assist with recovery during an attack. In [15], an algorithm is proposed to restore damaged element paths by recursively breaking down demand flows into simpler problems. They use a centrality metric to rank damaged nodes and determine which ones should be repaired first and expand on the use of centrality to make repair decisions in further work [16]. We use the concept of centrality to rank data objects in a case where two or more are equally critical. In our algorithm, we also utilize their method of simplifying damage paths to find the fastest route to restoring intermediate data objects. However, the novelty of our approach is twofold: we must repair all components within the system because data objects cannot have computations rerouted, unlike the network components in the work we have reviewed, and we aim to restore the most important components first so that their functions can be restored while repairs to the system are still ongoing.

III. Problem Definitions

On the occasion when an adversary information attack succeeds, the victim must have the capability to degrade gracefully and recover damaged data and/or services in real-time if it is to survive. It is necessary to immediately carry out damage assessment and recovery process in order to bring the systems to working states. Otherwise, the damage would spread to other unaffected systems that are interconnected. This happens when a valid user or an unaffected system module reads a damaged object during its computation and updates another object based on the compromised value, causing the latter damaged as well. As time goes on, more and more objects become affected in this manner causing the spread of damage to fan-out through the system quickly.

State-of-the-art assessment and recovery currently exist in types of critical infrastructure other than data systems. A review of threats that influence how critical infrastructures are protected is done in [22]. Among other threats, important ones that affect this research are cyber-attacks and cascading effects. A data system is especially vulnerable to the cascading effects of a cyber-attack, because of the nature of constant updates being made between different systems using data. In [21], a dynamic inoperability input-output model is introduced to determine how damage in one part of a system, such as power plants, water collectors, and transmitters can ultimately cascade down to end-users of the system. This method, as well as other similar methods in [23] and [24], primarily focus on physical threats to critical infrastructure. This research focuses on applying the same kind of resilience to data operations of critical infrastructures to protect them from digital threats.

For damage assessment and recovery purposes information about all processes that have been executed must be stored in the log (more on this presented later). This will help in determining the relationships among the processes, thus helping in establishing the damage trail. Moreover, during recovery, the operations of processes that have spread the damage have to be undone and then redone in order to produce the correct states of affected objects. The problems with existing systems are: (1) They do not store process execution information in the log, and they purge the log periodically, (2) their recovery mechanisms are not designed to undo the effects of executed processes, (3) the size of the log, as it must not be purged, will make it almost impossible to continue the recovery process in real-time, and (4) during the damage assessment and recovery process, the system remains unavailable to users. This delay induces a denial-of-service attack, which is highly undesirable in time-critical applications that the critical infrastructures are designed to provide. Due to the massive amount of data in the log that needs to be processed, the problem becomes even worse.

The goal of this research is to develop fast, accurate, and efficient damage assessment and recovery techniques so that critical information systems not only survive the attacks gracefully but will continue to operate providing as many vital services and functions as possible even before the system is fully recovered. In the next section, we explain how the relationships between different parts of critical systems can affect the spread of damage.

IV. Types of Damage Dependencies

Attacks can affect not only data systems, but also software that produces data. This happens when the software is maliciously changed to perform unintended actions. In this section, we discuss possible attack scenarios involving damaged data objects and software and how they can cascade into other systems.

A. Data to Data Damage

Data objects are frequently dependent on other data objects for computational updates. This makes their systems vulnerable to malicious attacks and cascading damage. Once a data object uses a damaged object to make a change, it becomes damaged too. This can happen to as many objects as the initially attacked objects have that are dependent on it. Figure 1 shows an example of cascading damage between data objects. If node C is targeted for an attack, then node E will become damaged due to its dependency on C. Then, node F and node G are also damaged because they are dependent on E.

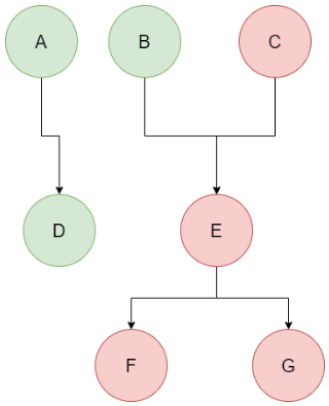

Figure 1: Cascading Damage from a Data Object

B. Software to Software Damage

If damaged software is used to influence computing done by another software, then the output of that software is also considered damaged. An example of a dependency between software is the use of libraries for applications. A damaged procedure within a library will result in damaged output for any software that calls that procedure. One difficulty with recovering from cascading damage from software to software is that it may be difficult to identify which cross-software calls used damaged procedures for computation. Consider a program *P* that uses two library functions *A* and *B*. *A* is an undamaged function and provides an undamaged output, while *B* is damaged and results in a damaged output. Even though we know the library is damaged, it is difficult to find which functions within the library are specifically damaged due to the library already being compiled into binary format. Therefore, we do not know if *A* or *B* is damaged, and after the library has been recovered, we must execute *P* entirely to ensure that we get the correct output, which causes recovery time to increase. After recovery, the resulting output from *A* is unchanged, while the output from *B* is corrected to the desired result. An example of software-to-software damage is shown in Figure 2. Even though the data is correct, a maliciously changed library can result in the software changing the output data.

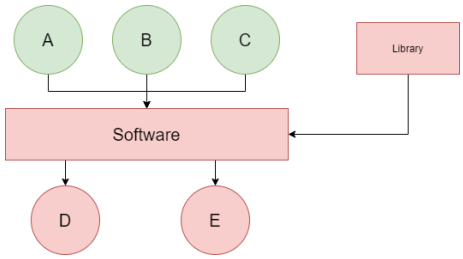

Figure 2: Cascading Damage from Software

C. Software to Data Damage

Any data changed by damaged software is also considered damaged. This is the simplest scenario involving damaged software because it is functionally the same as cascading damage between two data objects. After the software is repaired, any data that was damaged by the software must be computed again to complete recovery.

D. Data to Software Damage

Software that uses damaged data will not become damaged as a result. However, its output data will be considered damaged. This is because once a program is compiled, it cannot be changed by its input data. Therefore, if a software uses incorrect data to produce a damaged output, it is considered data-to-data damage instead. In Figure 3, a damaged node A is used as input for computation in a software. While the software itself is not damaged, its output nodes D and E become damaged.

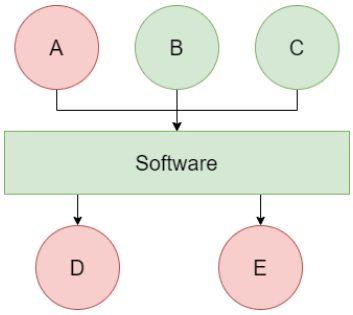

Figure 3: Cascading Damage from Data through Software

In the next section, we explain how our model can identify and recover damage optimally in detail. While it is

important to consider the possibility of damaged software affecting a system, our model focuses on the spread of damage within data objects. This is because damage can spread much faster through data objects than in software, and usually does so more frequently. Furthermore, the software can also be quickly fixed, while data objects are more numerous and may need many computations to get the desired output after repairs.

V. THE MODEL

In this section, we describe our model in detail. The first subsection defines important graphs and metrics that we use for our model. In the next subsection, we describe how the model is built and is used to determine an optimal recovery plan. Finally, we describe the algorithm we use to implement our model.

A. Definitions

We first define the concept of information flow in a system. This also defines dependencies among various objects in the system and is used in our graph-based model.

Definition 1: Given two objects O_i and O_j in a system, if the value of O_j is calculated using the value of O_i, we say that there is information flow from O_i to O_j. Thus, O_j is said to be dependent on O_i and is denoted as $O_i \rightarrow O_j$.

The above definition helps in determining the spread of damage in the system. That is, if an object is damaged, then all its dependent objects will be considered damaged. During recovery, the parent (pre-cursor) object must be recovered before any of its dependent objects can be recovered.

Next, we define a graph containing the set of objects and all possible paths among them. We call it Possible Paths graph and it spans the entire system of objects and all dependency paths among them. An example of this graph is shown in Figure 4(a).

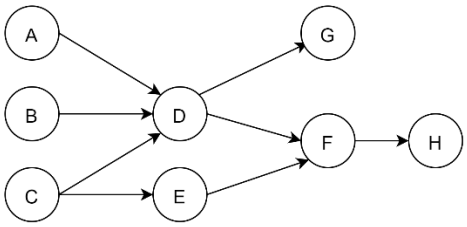

Figure 4a: The Possible Paths Graph (PPG)

Definition 2: Consider a system containing the set of objects O. The ***Possible Paths Graph*** (PPG) is built by having a node N_i for each object in O. There exists an edge E_{ij} from N_i to N_j in the PPG if there is a possibility that information may flow from N_i to N_j, that is, N_j may be modified based on the value of N_i.

The purpose of building a PPG is that it will help during the damage assessment preparation phase. By assuming the point of attack one can identify the set of items that may be affected consequently. Thus, security officers can be prepared for different types of eventualities.

The second set of objects contains the actual paths that were used to make changes in the system within a specified period, which for the purposes of the third graph that will be defined, is usually the time passed since an object has been damaged. This set is represented by the Active Paths Graph (APG), and all objects and dependencies in this set exist in the PPG. This graph will help in determining the damage flow in case of an attack. Given an initial attack point (an object), one can determine which objects in the system may be affected by the attack and which ones will not be. Therefore, the ability of the system to carry out its intended functions can be calculated. That is, during the recovery process, the set of damaged objects will be made unavailable while the rest can be made accessible. Knowing which objects will remain unaffected, one will be able to identify what services the system will be able to offer while the recovery continues.

Definition 3: The ***Active Paths Graph*** (APG) contains nodes N and edges E such that for every $N_i \in N$ and every $E_{ij} \in E$, both N_i and E_{ij} are also present in PPG, and E_{ij} illustrates an actual information flow; that is N_j was updated based on the value of N_i.

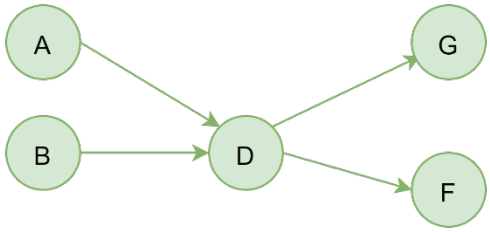

Figure 4b: The Active Paths Graph (APG)

Figure 4(b) provides an example of an Active Paths Graph and as can be seen, it is a sub-graph of Figure 4(a). As discussed before, once an initial attack point is determined, the APG will help in accurately determining the damage flow and the set of objects affected by the attack. As discussed before, as time goes on, more and more objects will be affected as new objects will be updated based on the value of an affected object. Thus, to stop the spread of damage, all affected objects must be quickly identified and taken offline as soon as possible. This can be achieved by doing a flow assessment using the APG. This leads to the concept of actual damage spread path showing exactly which objects were affected by an attack. If a system is damaged, we represent the spread of damage as the third set of objects, the Damage Spread Graph (DSG). The set of objects and dependencies in this graph must exist within the APG, as damage spread occurs when objects make changes based on their dependencies. Like how the APG is a subsection of the PPG, the DSG is a subsection of the APG. Figure 4(c) is an example of what a damage path may look like. It is important to note that over time, a damaged object will always cascade

its damage down to dependent nodes included in the APG. Definition 4 formally defines the DSG.

Definition 4: A ***Damage Spread Graph*** (DSG) contains nodes N and edges E such that for every $N_i \in N$ and every $E_{ij} \in E$, both N_i and E_{ij} are also present in APG and every node in N is damaged through an attack on the system. Moreover, an edge E_{ij} depicts that N_i was damaged first and then N_j was damaged through the flow of information from N_i to N_j.

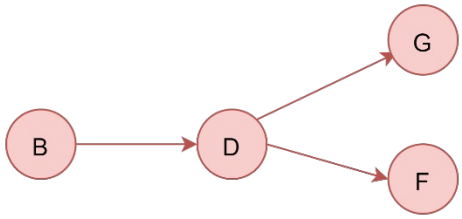

Figure 4c: The Damage Spread Graph (DSG)

Note that the edges between two objects may be bidirectional or recursive. For example, if an object O_j can have a dependency on object O_j and vice versa, then there will be a bidirectional edge between O_j and O_j. Similarly, if an object can be dependent on itself, it will result in a recursive graph. To clarify, let us consider an object "salary". When an employee receives an increment that is based on a percentage of the current salary of the employee, it causes the new salary to be dependent on the old salary and is depicted by using an edge from salary to salary itself. However, it must be noted that, for simplicity, we use neither bidirectional nor recursive edges in APG or DSG. Rather, when an object is modified, we note that as a new version of the object, thus creating a new node for the object with the version number.

To minimize the time needed to restore the most important objects within a system of object dependencies, we also define criteria used to determine the order in which repairs are made:

Definition 5: The ***criticality*** of a node N is its predetermined level of importance to the system's functions. This must be predetermined for the flexibility of the model to fit various systems and align the model with the goals of each specific system. For example, one system may need to prioritize certain components that other systems do not. The criticality of a component can be measured by various characteristics such as the intensity or scope of an impact caused by its failure as described in [14].

We assign a positive whole number to each node N to represent criticality. A lower assigned value indicates higher criticality. For example, a node N_i with a criticality of 2 would be considered more important than a node N_j with a criticality of 4. It is important to note that criticality values are not unique, meaning multiple nodes can have the same criticality value. When that happens, we use the following metric in the next definition to serve as a first "tiebreaker".

Definition 6: Objects that have more damaged dependencies take longer to repair. Therefore, the ***repair time*** of a node N is defined as how many inward-flowing edges E^i it is receiving damage from.

When two or more objects are assigned the same importance, we choose to first repair the one that has a lower repair time. For example, consider two nodes N_i and N_j that are equally critical. If N_i needs 5 other nodes repaired to repair it, and N_j needs 3 other nodes to repair it, then we will repair N_j first, because its operation can be restored more quickly than that of N_i.

Definition 7: The ***centrality*** of a node N is the number of outward-flowing edges E^o it has.

We use the above metric to decide the next object to repair when two or more are equal in both criticality and repair time. An object with a higher number of E^o will have higher centrality. Figure 5a and Figure 5b show two subsections of a DSG that highlights centrality. As shown in Figure 5a, N_4 has three nodes that are dependent on it: N_1, N_2, and N_3, while as Figure 5b depicts, N_6 only has a single node N_5 dependent on it. Assume that the repair algorithm has repaired the parent node(s) of N_4 and that of N_6. To clarify the situation, N_4 and N_6 need not have the same parents; it is just that both are in line to be repaired next. In this scenario, repairing N_4 before N_6 reduces the repair time for the three dependent nodes of N_4 instead of only one of N_6, which can make future repairs be performed faster. Therefore, N_4 is considered to have a higher centrality than N_6.

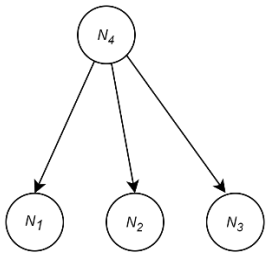

Figure 5a: A parent node with high centrality

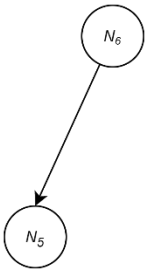

Figure 5b: A parent node with low centrality

Definition 8: The relationships between all nodes n in a data system can be expressed by an ***adjacency matrix*** A, where each element represents an edge $e_{i,j}$ between nodes n_i and n_j such that the parent node is given by the element's row and the child node is given by the element's column. The value of each element is binary – if an edge going from one node to

another exists, then the value of its respective element is 1, otherwise, it is 0. Therefore, for each element $a_{i,j}$ in A:

$$a_{i,j} = \begin{cases} 1 \text{ if } (i,j) \in N_i \\ 0 \text{ if } (i,j) \notin N_i \end{cases}. \quad (1)$$

Figure 6 depicts the PPG shown in Figure 4a. as an adjacency matrix. Each cell with a value of 1 is an existing edge on the graph. For example, node D has 3 parent nodes and 2 child nodes. The edges coming from nodes A, B, and C are highlighted on D's row. Likewise, its outgoing edges toward nodes F and G are highlighted on D's column. All node relationships are translated this way, so as there are 8 edges in the graph, there is also 8 highlighted elements in Figure 6.

Since an adjacency matrix can be used to represent the relationships between data nodes in a uniform manner, it is used for our implementation of a damage assessment algorithm, which will be covered in the next two subsections.

	A	B	C	D	E	F	G	H
A	0	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0
D	1	1	1	0	0	0	0	0
E	0	0	1	0	0	0	0	0
F	0	0	0	1	1	0	0	0
G	0	0	0	1	0	0	0	0
H	0	0	0	0	0	1	0	0

Figure 6: An adjacency matrix of Figure 4a.

B. Model Description

The model uses the three graphs defined in the previous section to construct a representation of a given system and its sustained damage from the time of the initial attack. The PPG is a preprocessed map of all components and dependency paths within a system. We assume that we know how much time has passed since the initial attack and build the APG by including components and dependency paths that were used in a transaction log in that period. By knowing the component where the initial attack occurred, we build the DSG by tracing the damage through the transaction log. For damage to spread from one component to the next, it must follow two criteria: 1) there is a damage node N_i that has an edge E_{ij} flowing from it to node N_j and 2) E_{ij} is used for a transaction while N_i is damaged. For the DSG to exist, the initial attack must occur within the APG, otherwise there is no cascading damage.

The goal of the model is to find the optimal sequence of repairs to restore the most important operations of a system as quickly as possible. We use the metrics defined in the previous section to decide which components should be repaired first. The first metric is criticality – the most critical components must be restored first to resume important operations. However, these components may also be dependent on other components that are damaged. These components must be repaired first before the base component can be repaired. At this point, the same problem is applied to the dependency components, and the most critical one is chosen first. If there is a tie, then components with a lower repair time are picked first. For example, a component that has two damaged parent components will be prioritized over a component with three or more damaged parent components if both components are equally critical.

To clarify, let us consider the graph presented in Figure 7. As shown in the figure, nodes N_1, N_2, and N_3 are dependent on N_4. Assume that the damage assessment method identified N_4 as damaged; thus, nodes N_1, N_2, and N_3 are also identified as damaged. During the recovery process, N_4 was recovered before the other three nodes. However, since it has three dependents all of which are damaged, the question is, which one should be repaired first. As our goal is to have the vital functions of the system to be made available before the other operations, our algorithm would choose the node among N_1, N_2, and N_3 having the most criticality.

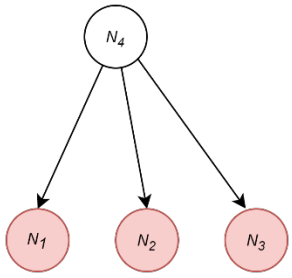

Figure 7: Recovery sequence decision

Repair time is not affected by how the parent components are ordered. For example, consider two scenarios with nodes N_1, N_2, N_3, and N_4. In the first scenario, node N_1 is dependent on nodes N_2, N_3, and N_4. In the second scenario, node N_1 is dependent on node N_2, node N_2 is dependent on node N_3, and node $N3$ is dependent on node N_4. In both scenarios, nodes N_4, N_3, and N_2 must all be repaired before node N_1 can be repaired, so N_1 will always have the same repair time. In short, repair time can simply be considered as the number of upstream components. Similarly, the third metric, centrality, can be considered as the number of downstream components. This is the third metric used to determine repair order in case multiple components are equal in criticality and repair time. This metric is not prioritized over the first two because it does not directly contribute toward the stated goals of our model, but it can optimize future repairs by lowering the repair time of more components than repairing other components would.

C. The Algorithm

First, we discuss the primary objective of our work. Let us consider the notations used in Table I.

TABLE I. NOTATIONS

Notations	Descriptions
$P = (V, E)$	Possible Path Graph
$A = (V_A, E_A)$	Active Path Graph ($V_A \subseteq V, E_A \subseteq E$)
$D = (V_D, E_D)$	Damage Spread Graph ($V_D \subseteq V_A, E_D \subseteq E_A$)
$D = (V_C, E_C)$	Critical Node Graph ($V_C \subseteq V_D, E_C \subseteq E_D$)
δ_{ij}	Decision to fix edge i to j
δ_i	Decision to fix node i
t_i	Time to fix node i
c_i	Centrality of node i
P_{ij}	Dependency indicator of node i and j

Our objective is to find $\min \sum_{i \in V_D} t_i \delta_i$ subject to

$$\delta_i \sum_{j \in V_C} P_{ij} \leq \sum_{j \in V_C} P_{ij} \delta_j \quad \forall i, j \in V_C \qquad (2)$$

$$\delta_i c_i \geq \sum_{(i,j) \in E_C} \delta_{ij} \quad \forall i \in V_C \qquad (3)$$

$$P_{ij} \in \{0,1\} \quad \forall i, j \in V_C \qquad (4)$$

$$\delta_i, \delta_{ij} \in \{0,1\} \quad \forall i \in V_C, (i,j) \in E_C \qquad (5)$$

That is, the goal is to minimize the time required to fix all critical nodes subjected to conditional constraints of the system. To make sure that each preceding nodes of i are fixed before node i being processed, condition (2) is used. For example, if there is a node j connecting to i but in a prequel order, the sum product of all nodes j status and dependency indicator P_{ij} should be greater or equal than the product of sum of all dependency indicator P_{ij} with node i. To make sure that there would not be more out-going flows than the given capability of node i, equation (3) is imposed to make sure the total out-going edge would not surpass the centrality of node i. Conditions (4) and (5) were built to impose the binary attribute of the dependency indicator P_{ij}, the decision whether to fix node i or edge from node i to node j.

The algorithms provided in this section use the model described in the previous section to compute the optimal order of repairs to restore the most important functions of a system first. When an attack occurs, we expect an Intrusion Detection System (IDS) to identify the attack and provide the initial point of damage. The working principles of IDSs are not within the scope of this work and so, not described here.

Algorithm 1 creates the PPG from the system's data. The PPG must record all possible dependency paths, including those that may not have been used by the system yet. Therefore, it is necessary to consult the system's designers so the algorithm can be provided full information on the system's data objects to fully construct the PPG. As stated in Definition 8, the PPG, APG, and DSG are all represented by a binary adjacency matrix, where child nodes and parent nodes are represented by the columns and rows, respectively. Each element in one such matrix is the edge between the given parent and child node, with a value of 1 indicating that the edge does exist in the graph and a value of 0 indicating that it does not exist.

Algorithm 1: Initializing the Possible Paths Graph
Result: Adjacency matrix M^{PPG} representing the PPG
1 for each object O_x and O_y in the system
1.1 if edge E_{xy}can exist
1.2 $M^{\text{PPG}}(x, y) = 1$
2 Return M^{PPG}

After receiving notification from an IDS, a precise damage assessment is performed. If the damage assessment process is unable to make accurate assessment, i.e., in case a damaged node is not correctly identified, it and its dependent nodes, which are also damaged, will remain unrecovered. This will result in valid users or procedures reading them and spreading damage by updating other objects, as discussed earlier. For a detailed discussion on damage assessment, one may review [9] and [10], which were developed particularly for database systems. However, the methods are still applicable to critical infrastructure systems. Below we provide a basic mechanism to carry out the assessment.

Damage assessment begins with the APG, which shows the actual dependency relationships among the objects in the system (Note that the APG can be built as transactions are executed and dependencies are established among various nodes of the PPG). Given the initial attack point, the corresponding node is then marked as damaged. This is the starting node of the DSG. Then by scanning the log from the corresponding location of the attack point, transactions that read the marked node are identified. Any objects written by those transactions are then marked as damaged in the APG. This process continues until the end of the log. Finally, all unmarked nodes and the edges showing their dependencies are removed. The resulting graph is the completed DSG.

The APG is constructed in Algorithm 2. This algorithm reads the transaction log and creates a node each time a new data object is mentioned in the log. The object's dependencies are depicted as the node's edges in the APG. When an existing data object is updated, it becomes a new object in the transaction log. As described earlier, this is done to prevent recursive dependencies in the APG.

Once damage assessment is carried out, recovery procedure must begin immediately in order to make the system operational quickly. We use Algorithm 3.1 as the main procedure to initialize an object set for repairs. The algorithm starts by initializing the set of damaged objects *O*. Each node *N* within *O* consists of a system component and its relationships with other nodes in *O*. As mentioned previously under Definition 4, some system components may have recursive or bidirectional dependencies between each other. Therefore, system components can have repeat nodes within *O* to represent their different versions. Each node is assigned values for criticality, repair time, and centrality. Using those metrics, the algorithm determines an initial target node N_0 based on criticality. If there are two or more nodes with the highest criticality, then the node with the lower repair time is selected. In the event of another tie, the node with higher centrality is selected. Further ties are broken by random

selection. N_0, along with O and the repair queue Q, are used to make the first call to the recursive function Algorithm 3.2 at step 4.5. Algorithm 3.1 proceeds until O is completely empty, and then the repair queue is finalized, and Q is printed.

As previously discussed, a node must have its parent nodes repaired before it can be considered eligible for repairs. Algorithm 3.2 ensures that nodes are scheduled for repairs in the proper order while still adhering to the rules set for determining priority. It does this by using a while loop to check the currently selected node N for repair eligibility. If N is eligible for repairs, then it is removed from O and Q is updated, then returned. If N is not eligible, then O', a subsection of O made up of all dependency paths above the currently selected node is created and used to find the next highest priority node N' within O'. Algorithm 3.2 is recursively called using N' and O', which can either result in the node's repair or another node being selected for repair again. The recursive nature of this algorithm ensures that each time a decision needs to be made on which node needs to be repaired next, it will prioritize criticality and efficiency among all the nodes that can be repaired at any given step. In this way, the bulk of the work done by the algorithm is choosing the next object for repair within each iteration. Each function call will result in one object being repaired and $n-1$ additional function calls, where n is the number of nodes within the set of nodes being passed. Since repaired objects need to be removed from the DSG, function calls will need to update and return the global DSG and Q.

Algorithm 2: Initializing the Actual Paths Graph
Result: Adjacency matrix M^{APG} representing the APG
Parameters: transaction log T, containing the set of edges E that were used to make updates
1 For each edge E_{xy} in T
  1.1 $M^{\mathrm{APG}}(x, y) = 1$
2 Return M^{APG}

Algorithm 3.1: Initialization for object set repair
Result: Queue of objects ordered by repair priority
1 Initialize set of damaged objects O
2 Preprocess object priority using criticality, repair time, and centrality
3 Initialize repair queue Q
4 while *O has damaged nodes remaining*
  4.1 Select the highest critical node(s) N within O
  4.2 if *Two or more nodes are tied for highest criticality*
    4.2.1 Select the node(s) N with the lowest repair time R within O
  4.3 if *Two or more nodes are tied for lowest repair time*
    4.3.1 Select the node(s) N with the highest centrality within O
  4.4 if *Two or mode nodes are tied for highest centrality*
    4.4.1 Select a single node at random from those still tied
  4.5 Update repair queue(N_0, O, Q) → Q
5 Print Q

Algorithm 3.2: Recursive repair function
Result: Schedules a node N for repairs and returns the updated repair queue Q
1 Update repair queue(*Selected node N, object set O, repair queue Q)*:
2 while *Current object has unrepaired dependencies*:
  2.1 Create subset of damaged nodes O' of all nodes N' and edges E' that N is dependent on
  2.2 Select the highest critical node(s) N' within O'
  2.3 if *Two or more nodes are tied for highest criticality*
    2.3.1 Select the node(s) N' with the lowest repair time R within O
  2.4 if *Two or more nodes are tied for lowest repair time*
    2.4.1 Select the node(s) N' with the highest centrality within O
  2.5 if *Two or mode nodes are tied for highest centrality*
    2.5.1 Select a single node at random from those still tied
  2.6 Update repair queue(N'_0, O', Q) → Q
  2.7 Remove the most recent object in repair queue from O
3 Repair N
4 Add N to Q
5 Return Q

The algorithm produces a list of system nodes in the order in which they should be repaired. Recovery procedure then continues to the next step to begin repairs on the system. It is important to note that while repairs are simulated by the algorithm, the process for repairing the actual components of the system is not within the scope of this work.

VI. Experiments and Analysis

In this section we present the results from simulations done using our model. We first describe the setup used for our experiment, then present the results from each simulation that we ran and explain the implications of the results.

A. Experiment Description

We evaluated the efficiency of our method by using it to find the number of nodes needed to repair every critical node in a DSG simulation with various parameters. The simulation constructed a DSG with randomly assigned relationships between nodes. The parameters used to build each DSG were total nodes, percentage of critical nodes, maximum number of parent nodes per node, and maximum number of children nodes per node. After the DSG was built, the set of critical nodes was randomly picked from the entire DSG for our method to compute the repairs required to bring all critical nodes back into a good state. For each simulation, 25 different sets of critical nodes were tested, and the average number of required repairs was reported by the simulation. We ran four experiments to test the changes to the number of

required repairs caused by changing each parameter. For every experiment, we set the control variables to be 10,000 total nodes, 5% critical nodes, and a maximum of 6 parent nodes and 6 child nodes per node. We compared our method to modified versions of two common traversal algorithms: breadth-first search (BFS) and depth-first search (DFS). The BFS algorithm repaired nodes starting with all the root nodes, then all the children of the root nodes, and then the next set of child nodes until it reached the bottom. On the other hand, DFS repairs a single parent node, then repairs one of its child nodes, and repeats that process until it reaches a node with no children. When that happens or if it has already repaired all child nodes for a given node, it goes back up to the previous node and repairs another child node. The exception to this process is if a node has other parent nodes that have not been repaired yet. The algorithm will break the traditional BFS or DFS order to ensure that the node is eligible for repair. We include the results for the BFS and DFS repair algorithms with our own method.

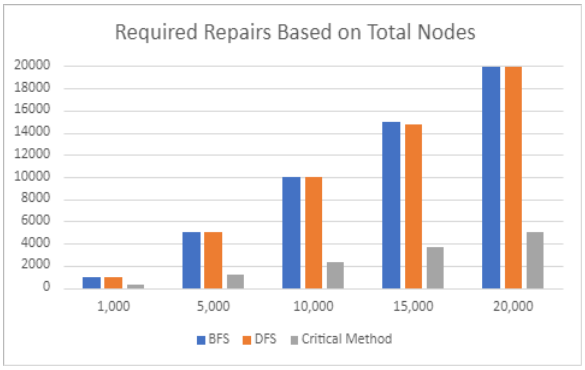

Figure 8a: Required repairs based on Total Nodes

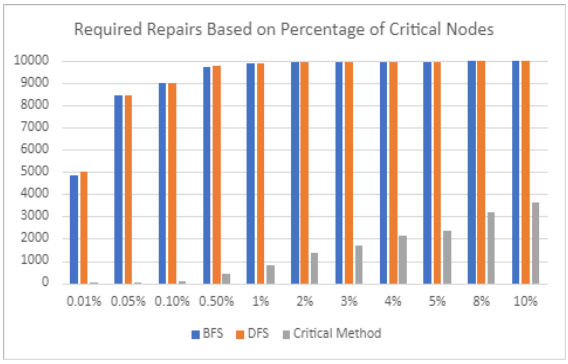

Figure 8b: Required repairs based on Critical Node Percentage

B. Results and Analysis

We present two graphs for each round of simulations in Figures 8 and 9. The former figure is a bar graph highlighting the different in required repairs between our algorithm and BFS and DFS traversals. Figure 9 shows the changes in the percentage of nodes repaired between each parameter of a simulation. Our results for required repairs based on total nodes is shown in Figure 8. It shows that the required repairs maintain a constant ratio between them and the total nodes with the given control variables, as each experiment resulted in a little less than 25% of the total nodes needing to be repaired. Figure 8b shows the results for required repairs needed for different percentages of critical nodes. We found that while the number of required repairs increases as the percentage of critical nodes increases, less additional repairs are needed when the percentage is higher. Therefore, a high number of critical nodes require less repairs per critical node than a lower number of critical nodes. Figures 8c and 8d show the change in required repairs for different maximums of children and parent nodes per node respectively. We found that with a low number of maximum children per node, more repairs are required. This is because the system of nodes becomes narrower in shape. In contrast, when there is a higher maximum of parent nodes per node, the required repairs see an increase. This indicates that a critical node is more likely to have additional parent nodes, which would require more repairs than a critical node with fewer parent nodes. Across all graphs, we can see that targeting specific nodes for repair drastically shortens the time needed to recover critical functions, as the average required repairs for the BFS and DFS algorithm in most simulations was slightly less than the total number of nodes in the simulation.

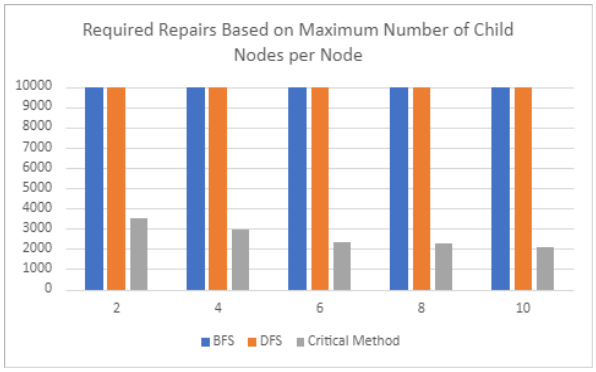

Figure 8c: Required repairs based on Maximum Number of Child Nodes

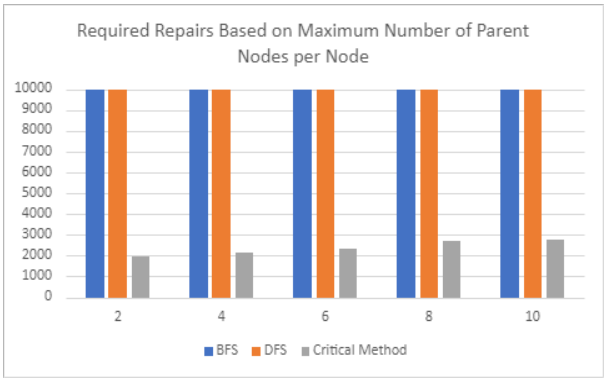

Figure 8d: Required repairs based on Maximum Number of Parent Nodes

C. On BFS and DFS algorithms

Before running our simulations, we expected our algorithm to outperform BFS and DFS searches due to it specifically targeting critical nodes for repair first. However, these two algorithms ended up traversing nearly the entire system before repairing all critical nodes. The best way to explain this is to consider the order by which each algorithm repairs nodes as a static ordered list. Our algorithm sorts this list after nodes are assigned criticality, so the final critical node that needs to be repaired ends up being close to the front of this list. On the other hand, BFS and DFS do not consider criticality when sorting this list, so each node's criticality on their lists after sorting is effectively random. What determines how many nodes need to be repaired is ultimately the final critical node in the list, so the odds of this node not being in the final thousand or even hundred nodes are quite low. Therefore, BFS and DFS will need to repair almost all the nodes except for when the percentage of critical nodes is lower. When that happens, it is more likely that the final critical node will be closer to the middle of the repair order instead.

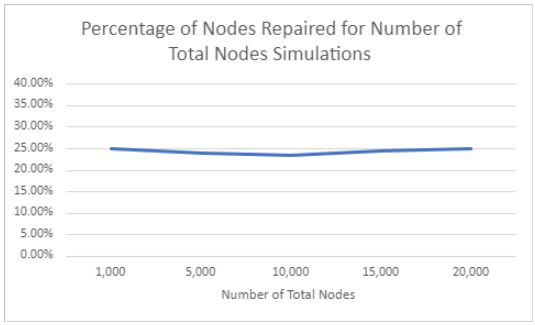

Figure 9a: Changes in required repairs based on maximum number of parent nodes

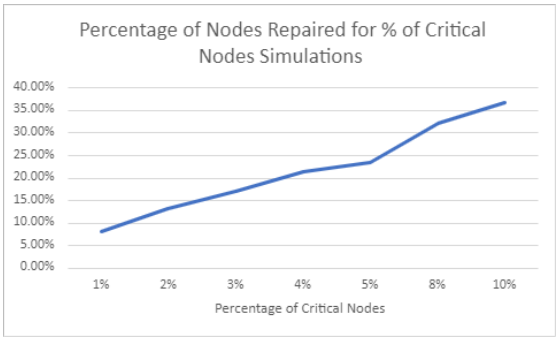

Figure 9b: Changes in required repairs based on the percentage of critical nodes

VII. Conclusion

In this research, we have presented a method to repair data objects that prioritizes quick recovery for the most important components of a system. This allows for the partial restoration of functions during the recovery process with an emphasis on restoring service to the most necessary functions. This was first done by building out three graphs to represent the entire system, what changes the system made after an attack, and the cascading damage as a result of those changes. Next, we developed an algorithm to optimally schedule repairs by using those graphs to find damage paths that affect the most critical nodes of a system and calculate the fastest repair order to fully restore those nodes. Lastly, we presented results from a simulation of our algorithm and the effects on changing the parameters of the damage paths. Our work is most applicable to protecting critical infrastructure systems where services need to be restored as quickly as possible to avoid economic or societal disruptions.

Further work includes considering the frequency at which an object is used to update its dependencies. Objects that are updated at a higher frequency would be prioritized as more important. Additionally, a method to select the order of repairs for non-critical objects after all critical objects have been repaired is also needed.

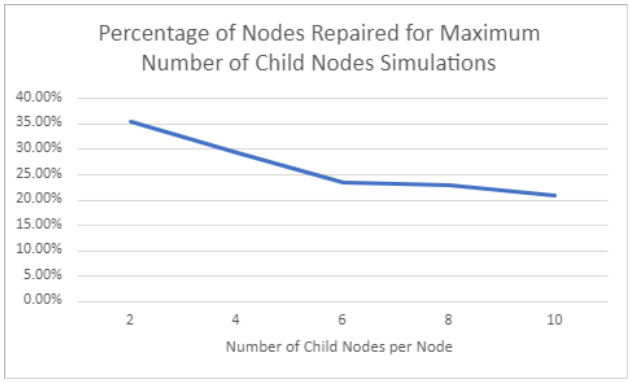

Figure 9c: Changes in required repairs based on the maximum number of child nodes per node

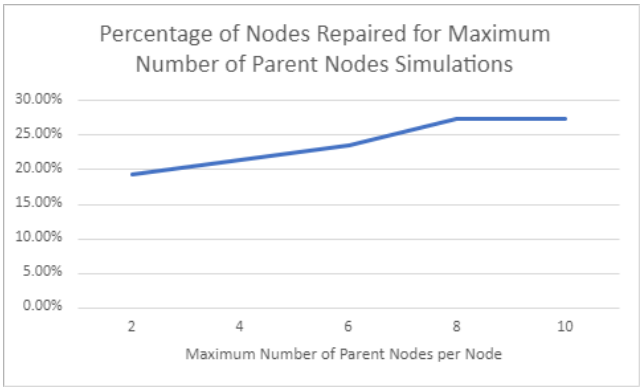

Figure 9d: Changes in required repairs based on the maximum number of parent nodes per node

Acknowledgement

This work has been supported in part by grant H98230-20-1-0419 issued by the National Security Agency as part of the National Centers of Academic Excellence in Cybersecurity's mission to expand cybersecurity research and education for the Nation.

REFERENCES

[1] J. Burns, B. Panda, T. Bui, "Modeling Damage Paths and Repairing Objects in Critical Infrastructure Systems", The Fifteenth International Conference on Emerging Security Information, Systems and Technologies SECUREWARE 2021, Athens, Greece, pp. 88-93, Nov. 14-18, 2021, ISBN 978-1-61208-919-5

[2] E. Viganò, M. Loi. and E. Yaghmaei, "Cybersecurity of Critical Infrastructure", In Christen M., Gordijn B., Loi M. (eds), The Ethics of Cybersecurity, The International Library of Ethics, Law and Technology, vol. 21, SpringerLink.

[3] *Critical Infrastructure Security*: https://www.dhs.gov/topic/critical-infrastructure-security. [retrieved: October 2021]

[4] D. E. Sanger and N. Perlroth, (2021, May 14). "Pipeline Attack Yields Urgent Lessons about U.S. Cybersecurity", https://www.nytimes.com/2021/05/14/us/politics/pipeline-hack.html. [retrieved: October 2021]

[5] A. Anastasios, "Is the Electric Grid Ready to Respond to Increased Cyber Threats?", https://www.tripwire.com/state-of-security/ics-security/electric-grid-ready-increased-cyber-threats/. [retrieved: October 2021]

[6] B. Barrett, "An Unprecedented Cyberattack Hit US Power Utilities", https://www.wired.com/story/power-grid-cyberattack-facebook-phone-numbers-security-news/. [retrieved: October 2021]

[7] K. O'Flaherty, "U.S. Government Issues Powerful Cyberattack Warning as Gas Pipeline Forced into Two Day Shut Down" https://www.forbes.com/sites/kateoflahertyuk/2020/02/19/us-government-issues-powerful-cyberattack-warning-as-gas-pipeline-forced-into-two-day-shut-down/#5f3061645a95. [retrieved: October 2021]

[8] M. Lewis, "Cyberattack Forces Gas Pipeline Shutdown", https://www.jdsupra.com/legalnews/cyberattack-forces-gas-pipeline-shutdown-76217/ [retrieved: October 2021]

[9] B. Panda and J. Giordano, "Reconstructing the Database after Electronic Attacks. In: Jajodia S. (eds) Database Security XII. IFIP — The International Federation for Information Processing, vol. 14. Springer, Boston, MA, 1999.

[10] S. Patnaik and B. Panda, "Transaction-Relationship Oriented Log Division for Data Recovery from Information Attacks" Journal of Database Management, vol. 14. No. 2. pp. 27-41, 2003.

[11] P. Kotzanikolaou, M. Theoharidou, and D. Gritzalis, "Cascading Effects of Common-Cause Failures in Critical Infrastructures". In: J. Butts and S. Shenoi (eds) Critical Infrastructure Protection VII. IFIP Advances in Information and Communication Technology, vol. 417, 2013. Springer, Berlin, Heidelberg.

[12] J. Ding, Y. Atif, S. Andler, B. Lindström, and M. Jeusfeld, "CPS-based Threat Modeling for Critical Infrastructure Protection". ACM SIGMETRICS Performance Evaluation Review. 45. pp. 129-132, 2017. 10.1145/3152042.3152080.

[13] E. Canzani, H. Kaufmann, and U. Lechner, "Characterising Disruptive Events to Model Cascade Failures in Critical Infrastructures", The Fourth International Symposium for ICS & SCADA Cyber Security Research, Belfast, Northern Ireland, 2016.

[14] D. Rehak, J. Markuci, M. Hromada, and K. Barcova, "Quantitative Evaluation of the Synergistic Effects of Failures in a Critical Infrastructure System", International Journal of Critical Infrastructure Protection, vol. 14, pp. 3-17, 2016. ISSN 1874-5482

[15] N. Bartolini, S. Ciavarella, T. F. La Porta, and S. Silvestri, "Network Recovery after Massive Failures," 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 97-108, 2016.

[16] S. Ciavarella, N. Bartolini, H. Khamfroush, and T. Porta, (2017). "Progressive Damage Assessment and Network Recovery after Massive Failures," IEEE INFOCOM 2017 – IEEE Conference on Computer Communications, pp. 1-9, 2017.

[17] J. Štoller and P. Dvořák, "Basic Types of Critical Infrastructure Objects," 2019 International Conference on Military Technologies (ICMT), pp. 1-7, 2019. doi: 10.1109/MILTECHS.2019.8870031.

[18] D. Rehak, P. Senovsky, and M. Hromada, "Analysis of Critical infrastructure Network. Z. Chan, m. Dehmer, F. Emmertsteib, and Y. Shi, *Modern and interdisciplinary* problems in network science: a translation research perspective. 1. Boca Raton, FL, USA: CRC Press, pp. 143-171, 2018. ISBN 978-0-8153-7658-3.

[19] M. Luskova, M. Titko, and B. Leitner, "Multilevel Approach to Measuring Societal Vulnerability due to Failure of Critical Land Transport Infrastructure", The World Multi-Conference on Systematics, Cybernetics and Informatics, Miami, FL, July 2016.

[20] B. Genge, P. Haller, I. Kiss, "A Framework for Designing Resilient Distributed Intrusion Detection Systems for Critical Infrastructures", International Journal of Critical Infrastructure Protection, vol. 15, pp. 3-11, 2016. ISSN 1874-5482, https://doi.org/10.1016/j.ijcip.2016.06.003.

[21] X. He, E. Cha, "Modeling the Damage and Recovery of Interdependent Critical Infrastructure Systems from Natural Hazards", Reliability Engineering & System Safety, vol. 177, pp. 162-175, 2018. ISSN 0951-8320, https://doi.org/10.1016/j.ress.2018.04.029.

[22] R. Osei-Kyei, V. Tam, M. Ma, F. Mashiri, "Critical Review of the Threats Affecting the Building of Critical Infrastructure Resilience", International Journal of Disaster Risk Reduction, vol. 60, 2021, ISSN 2212-4209, https://doi.org/10.1016/j.ijdrr.2021.102316.

[23] Y. Fang and G. Sansavini, "Optimum Post-disruption Restoration under Uncertainty for Enhancing Critical Infrastructure Resilience", Reliability Engineering & System Safety, vol. 185, pp. 1-11, 2019. ISSN 0951-8320, https://doi.org/10.1016/j.ress.2018.12.002.

[24] M. Xu, M. Ouyang, Z. Mao, X. Xu, "Improving Repair Sequence Scheduling Methods for Post-disaster Critical Infrastructure Systems", Computer Aided Civil and Infrastructure Engineering, vil. 34, pp. 506–522, 2019. https://doi.org/10.1111/mice.12435