Automatic Generation of Security Argument Graphs

Nils Ole Tippenhauer*†, William G. Temple†, An Hoa Vu†, Binbin Chen†,
David M. Nicol‡, Zbigniew Kalbarczyk‡, and William H. Sanders‡

*Singapore University of Technology and Design, Singapore
E-mail: nils_tippenhauer@sutd.edu.sg

†Advanced Digital Sciences Center, Illinois at Singapore Pte. Ltd., Singapore
E-mails: {william.t, anhoa.vu, binbin.chen}@adsc.com.sg

‡University of Illinois at Urbana-Champaign, IL, USA
E-mails: {dmnicol, kalbarcz, whs}@illinois.edu

***Abstract*—Graph-based assessment formalisms have proven to be useful in the safety, dependability, and security communities to help stakeholders manage risk and maintain appropriate documentation throughout the system lifecycle. In this paper, we propose a set of methods to automatically construct *security argument graphs*, a graphical formalism that integrates various security-related information to argue about the security level of a system. Our approach is to generate the graph in a progressive manner by exploiting logical relationships among pieces of diverse input information. Using those emergent *argument patterns* as a starting point, we define a set of *extension templates* that can be applied iteratively to grow a security argument graph. Using a scenario from the electric power sector, we demonstrate the graph generation process and highlight its application for system security evaluation in our prototype software tool, CyberSAGE.**

***Keywords*-Security argument graph; security assessment; automatic graph generation; argument patterns; extension templates**

I. Introduction

Critical public infrastructure systems, such as those found in the electric power and water sectors, must operate safely and reliably for decades. During their operating lifetimes, these systems are often modified to face evolving operating conditions and requirements. For example, infrastructure systems have adopted greater communication and control capabilities in recent years. However, while these advanced features enable greater system visibility and more efficient control strategies, they also open new avenues for malicious cyber attacks on the system [1]–[3].

To understand evolving system requirements, operational contexts, and/or security threats, practitioners often employ graph-based reasoning techniques. Such approaches include safety cases [4]–[6], fault tree analysis [7]–[9], and attack trees/graphs [10]–[14]. Historically, development and maintenance of those graphical approaches required significant human effort. Recently, several efforts have begun in the safety and reliability communities to automate those processes [5], [6], [8], [9]. However, in the security domain, automation has been largely restricted to specific applications, such as construction of attack graphs [12]. The various challenges about security assessment were discussed in, e.g., [15], [16].

Our approach for conducting holistic security assessment is to develop *security argument graphs*, a graphical formalism that integrates diverse inputs—including workflow information for processes executed in the system, physical network topology, and attacker models—to argue about the level of security for the target system. In our earlier work [17], we presented an integrative security assessment framework that reasons about security by progressively combining heterogeneous types of information to construct such holistic security argument graphs. Section II will recap the proposed framework and describe the overall structure of the generated security argument graphs.

Such holistic security argument graphs are beneficial in multiple ways: they make explicit the underlying interdependecies of different pieces of security-related information; also, the graph structure can be used to combine various numerical evidence to yield holistic quantitative security metrics.

In this paper, we provide a rigorous set of methods for constructing the holistic argument graphs introduced in [17]. Our approach is to leverage recurring *argument patterns* that emerge naturally from the need to integrate heterogeneous information about a system and possible attacks. We formalize these argument patterns using *extension templates* that can be iteratively and automatically applied to grow our argument graphs. Using our Cyber Security Argument Graph Evaluation (CyberSAGE) tool [18], which we are actively developing, we demonstrate the automated graph generation for an example electric power grid system [19].

The remainder of this paper is structured as follows: Section II reviews our assessment framework and the structure of the resulting security argument graphs. In Section III, we present the argument patterns that can be used to generate such argument graphs. In Section IV, we formalize the argument patterns as extension templates. In Section V, we show how to use these extension templates to automate the argument graph construction process. A practical example of the graph construction is presented in Section VI, with a use case

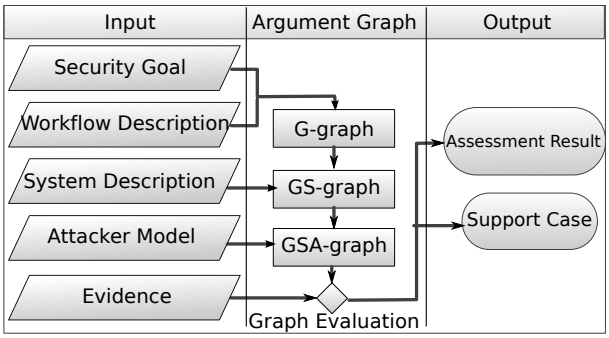


Fig. 1: The security assessment framework proposed in [17].

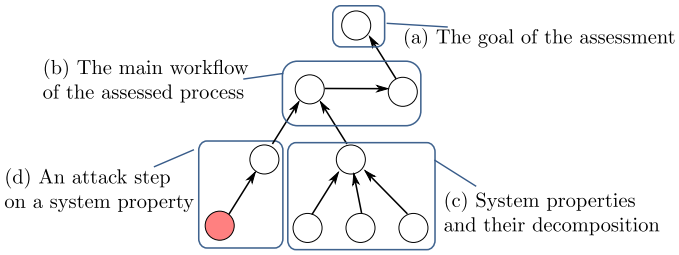


Fig. 2: Basic components of the argument graph: (a) the workflow steps of the assessed process, (b) a system and device property and its decomposition, and (c) a potential attack step and its immediate connection to system property.

from the electric power sector. We discuss related work in Section VII, and conclude in Section VIII.

II. Security Argument Graphs

Many safety [4], reliability [7], and security [20] assessment methodologies rely on graphical structures to organize and present information. In a security context, we use the term *security argument graph* to denote a graphical formalism that integrates diverse pieces of security-related inputs to argue about the security of the target system [15], [17]. We envision that those argument graphs will help provide a precise underpinning for threat modeling and quantitative evaluation of system-level security metrics.

A particular challenge in constructing security argument graphs is to deal with the large amount of heterogeneous information inputs that need to be incorporated, which may include security requirements, business processes, system architecture, physical device specifications, known vulnerabilities, and attacker models. In our earlier work [17], we presented a high-level workflow-oriented security assessment framework, which organizes the diverse set of security-related information inputs. The overall assessment process, which is illustrated in Fig. 1, relies on a unique argument graph structure that progressively incorporates Goal, System, and Attacker (GSA) information.

As a brief overview, the process starts with a precisely defined security goal, which may relate to properties such as Confidentiality, Integrity, or Availability. The analyst then identifies system processes that are relevant to the security goal, and represents them as workflow diagrams. Those workflow diagrams provide a sequence of actions and their respective actors. Those two inputs are combined to form a simple argument graph, called a Goal (G) graph, that captures information about actors and interactions that may affect the security goal. When detailed system information (e.g., actor to device mapping, network topology, or device configuration) is available, we use the G-graph to integrate that information and generate a more detailed security argument graph: the Goal, System (GS) graph. Finally, we incorporate information about possible attacker actions and capabilities into the GS-graph to generate the Goal, System, Attacker (GSA) graph.

Our GSA graph structure, which is represented in Fig. 2, is system-focused (like fault trees [7]), but allows for the modeling of attacks (like attack graphs [11]). The graph contains vertices of different types, such as system (attacker) actions and system (attacker) properties. This graph structure has no explicit vertices to denote aggregation semantics (e.g., OR relations or AND relations), as each vertex contains information that defines the aggregation of its incoming neighbors. Thus, the graph has only a single type of dependency relationship among the vertices.

III. Argument Patterns in Security Assessment

While applying our structured approach to generate security argument graphs (e.g., in the context of smart grid infrastructure), we observed several recurring patterns for integrating various information inputs. We call these *argument patterns*. Intuitively, argument patterns capture direct logical relationships among different pieces of information. Before formalizing these patterns (manually) into precise and reusable *extension templates* (see Section IV-A) aimed at automating the graph generation process, in this section, we first present the intuitions behind several useful argument patterns we identified from assessments in the smart grid domain. We shall see that the patterns to be presented are fairly generic, so they are applicable to other domains as well. In addition, other domains might also contribute patterns that we have not yet encountered.

In general, two classes of patterns occur in our arguments: *intra-type* patterns, and *inter-type* patterns. Recall that there are different types of vertices in a security argument graph, each type corresponding to certain class of security-related information. As the name suggests, intra-type patterns introduce and connect vertices of the same type in a security argument graph, and inter-type patterns introduce and connect vertices of different types. The following describes the two pattern classes in more detail and provides five example patterns we identified. We number the patterns according to the order in which they typically appear in our argument graphs in the generation process.

Inter-Type patterns. Inter-type operations connect vertices of different types. For example, at a high level, a security goal is directly defined in the context of one or more workflows to

which the goal is related. We characterize such a relationship by the following argument pattern:

Pattern P_1: Security goals or requirements directly depend on processes that occur in the system.

As another example, consider the workflow steps in our argument graph. Each workflow step is performed by one or multiple actors with certain properties. This association of workflow steps with actors is our argument pattern:

Pattern P_3: To successfully complete actions in the workflow, their respective actors need to be available.

In addition, these abstract actors have to be mapped to concrete devices in the system to allow a more detailed decomposition. That is our argument pattern:

Pattern P_4: Devices in the system adopt one or more workflow actor roles to provide functionality.

Intra-Type patterns. Workflows in the system have a number of actions that have to be executed in sequence. Our graph generation starts with the final step of a workflow, and then adds its prerequisite step (i.e., the generation works backwards in time). That is one of our argument patterns:

Pattern P_2: The successful completion of workflow actions may depend on the other preceding or concurrent actions.

For example, assume that the final step in a workflow depends on the receiving of a message. The intra-type pattern can be used to identify the workflow step that is related to the transmission of that message. Another important intra-type pattern relates to specific devices (within a system) that may be involved in a workflow. The refinement of device properties is our argument pattern:

Pattern P_5: Device properties depend on sub-properties and their composition semantics.

For example, the availability of a device depends on the availability of its software and hardware, so this property can be decomposed.

In general, our argument patterns capture individual direct logical relationships that make up the entire argument. Such a focus on simple patterns of abstract relationships allows us to apply the patterns with only local knowledge of the argument graph. As each argument pattern captures a generic relationship, we can construct our argument graph using a small number of patterns. The patterns presented in this paper are used to support an availability assessment use case (see Section VI).

In the next section, we show how to formalize those argument patterns by constructing *extension templates*. The resulting set of extension templates allow us to automatically generate an argument graph based on several classes of inputs which drive the security assessment.

IV. Graph Generation Based on Argument Patterns

So far, we have introduced the concept of argument patterns that emerge during the construction of argument graphs; we also provided several informal examples to illustrate our intuition. We now present a formalism that rigorously defines these patterns and the manner in which they can be applied. In the following, we start with a formal definition of the security argument graphs. We then formalize how to progressively generate such graphs through the application of *local extensions*. We define how a *local extension* is generated through the instantiation of an *extension template*, which formalizes a corresponding argument pattern. With all these building blocks in place, we then present our overall process for generating the graph.

A. Graph Structure and Local Extensions

We first define the structure of our graph and how a graph can be generated by the application of local extensions.

Graph Definition. For the following discussion, a graph ω_i is defined to be a triple

$$\omega_i := \langle V_i, E_i, l_i \rangle ,$$

where V_i is a finite set of vertices, E_i is a finite set of directed edges, and l_i is the labeling function. Each vertex is itself a static tuple that contains the type of the vertex, and some of type-specific additional data. The type and data are set when the vertex is created, and cannot be changed afterward. Two vertices are considered identical whenever all their static data are identical. An edge $e = \langle v_s, v_t \rangle$ is represented by a tuple that contains references to its source vertex v_s and target vertex v_t. The labeling function, $l_i(v)$, returns the mutable attribute(s) of a vertex $v \in V_i$. For example, a variable attribute could be the probability that this vertex's property is true (to be determined in the graph evaluation later). Note that $l_i(v)$ does not need to be a single numerical value: it can encapsulate a list of different types of information, such as an expression relating the attributes of its incoming neighbors to its own attribute.

Local extension. We progressively generate the security argument graph through the application of *local extensions*. A local extension r is defined as a tuple of its matching vertex v_r and the resulting graph ω_r:

$$r := \langle v_r, \omega_r \rangle$$

The resulting graph ω_r is a *star graph*: it contains one central vertex v and at least one leaf vertex. Each leaf vertex has one outgoing edge towards v. Other than those edges, there is no other edge in ω_r.

We use $\omega_a \overset{r}{\Longrightarrow} \omega_b$ to denote the application of a local extension r to a graph ω_a, which generates a new graph ω_b (see Fig. 3). Here, we assume that r is applicable, i.e., the matched vertex v_r is indeed a vertex of ω_a. Given that, the local extension is applied as follows:

$$\omega_a \overset{r}{\Longrightarrow} \omega_b = \langle V_a \cup V_r, E_a \cup E_r, l_b \rangle$$

where

$$l_b(v) = \begin{cases} l_r(v) & \text{if } v \in (V_r \setminus V_a) \cup \{v_r\} \\ l_a(v) & \text{otherwise.} \end{cases}$$

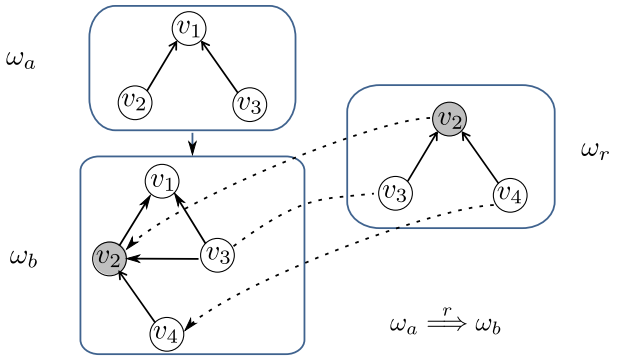


Fig. 3: Local extensions: Transformation of graph ω_a using local extension r into graph ω_b (i.e. $\omega_a \overset{r}{\Longrightarrow} \omega_b$). The local extension r matches v_2, and inserts ω_r in its place. Slashed arrows denote logical connections between graphs. For illustration purposes, the variable attribute of a vertex is its color.

The additional vertices from the star graph ω_r and the associated edges are added to the original graph. Note that the additional vertices in ω_r may or may not be present in the original graph ω_a. For each vertex that is already present in the original graph ω_a, except for v, the old labeling function is preserved; otherwise, the labeling function is taken from the star graph.

B. Extension Templates and Graph Generation

Having introduced local extensions, we now consider their generation. We introduce the notion of an *extension template,*

$$\gamma := \langle m_\gamma, f_\gamma \rangle ,$$

where m_γ is a *matching function* and f_γ is an *extension generation function*. Specifically, $m_\gamma(v, \Sigma)$ shows γ's matching score for vertex v, where Σ is the *environment*: a placeholder for additional information. Recall that the attribute of a vertex $l(v)$ is a tuple that contains its type and type-specific additional information, both of which can be used to determine the numerical value of $m_\gamma(v, \Sigma)$. A value of $m_\gamma(v, \Sigma) = 0$ means that extension template γ is not applicable to the generation of an extension for v. If multiple extension templates are applicable (i.e., several extension templates can be applied to the same vertex), then m_γ could be implemented as chosen from a range, to indicate precedence. Otherwise, m_γ can be implemented as a Boolean function indicating whether γ is applicable.

If an extension template γ is applicable to a vertex v, the local extension generation function $f_\gamma(v, \Sigma)$ uses the information from vertex v and environment Σ to generate a local extension that can be used to expand v. In that case, Σ is a placeholder for various pieces of information that are relevant to the transformation of the graph, such as the workflow information, the actor-to-component mapping, or the network topology graphs. We use Γ to denote the set of extension templates.

Algorithm 1 Graph generation loop

```
1:  function GenerateGraph(ω_a, Γ, Σ)
2:    U ← Vertices(ω_a);
3:    while U ≠ ∅ do
4:      v ← GetOneElement(U);
5:      Γ_v ← GetMatchingTemplates(v, Γ);
6:      ∀γ ∈ Γ_v :
7:        r ← f_γ(v, Σ);
8:        ω_a ⟸ ω_a;   (r)
9:        U ← U ∪ (V_r \ V_a);
10:     U ← U \ {v};
11:   end while
12:   return ω_a;
13: end function
```

Algorithm 1: Graph generation loop

ID	Comment
T_1	Template to connect goal node to assessed workflow(s)
T_2	Template to look up required previous steps in a workflow
T_3	Template to create requirements for the actor of a workflow step
T_4	Template to create device requirements for an actor
T_5	Template to decompose requirements for devices
T_6	Template to identify potential attacks on leaf properties
T_7	Template to create requirements for an attack step

TABLE I: A sample set of extension templates.

Graph Generation Algorithm. We define *graph generation* as the repeated application of a set of extension templates to a graph. More formally, it is the application of Γ to a graph ω_a, using the environment Σ. The underlying algorithm is summarized as pseudocode in Algorithm 1 below.

In the pseudocode, Vertices(ω_a) returns all vertices of the graph, while GetMatchingTemplates(v, Γ) returns a set Γ_v of all templates applicable to v, and GetOneElement(U) simply picks an arbitrary element of the set U. Note that when no template is applicable, $\Gamma_v = \emptyset$, whereas when several templates are applicable, the one with the highest matching score m_γ is chosen. In Table I, we list some of those extension templates and describe their functions. The template numbering corresponds to the pattern numbering in Section III; the set has been expanded to include two additional templates, which relate to attacker modeling. Here, we concentrate on extension templates related to availability of a process, matching our case study in Section VI. We are currently working on additional extension templates to model the transmission path of messages, human-machine interactions, and more complex workflow mechanisms.

V. Using Extension Templates to Generate Argument Graphs

Our security assessment process [17], shown in Fig. 1, contains three types of security argument graph: the G-graph, GS-graph, and GSA-graph. All of those graphs are generated through application of Algorithm 1; the differences arise from the specific inputs provided and the extension templates used

for graph generation. The meta-process for constructing the argument graphs is:

$$\begin{aligned} \omega_g &\leftarrow \text{GENERATEGRAPH}(\omega_0, \Gamma_g, \Sigma_g) \\ \omega_{gs} &\leftarrow \text{GENERATEGRAPH}(\omega_g, \Gamma_s, \Sigma_s) \\ \omega_{gsa} &\leftarrow \text{GENERATEGRAPH}(\omega_{gs}, \Gamma_a, \Sigma_a) \end{aligned}$$

In the following three subsections, we describe individual steps, their inputs, and the associated templates in greater detail. Finally, in Section V-D, we focus on a single template to illustrate the level of specification required, and the importance of templates in automating the graph generation process.

A. Graph Generation Using Workflow Input

To generate the first stage of our argument graph, the G-graph, Algorithm 1 is called as follows: $\text{GENERATEGRAPH}(\omega_0, \Gamma_g, \Sigma_g)$. That will generate the G-graph, which includes the workflow input in Σ_g, and is based on an initial base-graph ω_0 and a set of extension templates Γ_g. The initial graph will contain only a vertex representing the goal of the assessment. Γ_g contains T_1, T_2, and T_3, the formal extension templates for patterns P_1, P_2, and P_3 as defined in Section III.

Workflow information in Σ_g. The workflow input describes *how the system provides a functionality*, and identifies necessary actors as well as the information they exchange. Our internal structure for this input is a workflow graph: $\omega_w = \langle V_w, E_w, l_w \rangle$. For $v \in V_w$, $v = \langle type, attributesTuple \rangle$. Workflow vertices have only one type of vertex: $\text{TYPE}(v) \in \{\text{WorkflowStep}\}$. A vertex's attributesTuple field depends on its type. For simplicity, in the following we consider only sequential workflows, with workflow step vertices. The attribute tuple of each workflow step vertex has two static attributes: `actor` and `action`. Thus, the attributeTuple is $\langle actor, action \rangle$. The labeling function l_w is used to store additional workflow information, e.g., on branching, merging, and conditions in the workflow.

Argument graph vertex types and templates Γ_g. We construct our argument graph iteratively, starting with ω_0, which is a directed graph with the assessment goal as its only vertex. The assessment goal determines which properties of the system we are interested in, for example, the availability of the components involved in a workflow. The goal vertex is directly created from user input, without any need for an extension template. Based on that goal vertex, extension template T_1 is used to connect assessed workflows that are related to the assessment goal (more detail on P_1 implemented by T_1 in Section III). For each connected workflow, the relevant final workflow step vertex is connected first. Then, the final workflow step vertex is connected to the required previous steps in the workflow using T_2 (implementing P_2), and those steps are expanded as well. All vertices generated by T_1 and T_2 will be of type "ActionAvailability." In addition, required properties for each actor, such as actor and communication link availability, are added using extension template T_3 (implementing P_3). That will add vertices of type "ActorAvailability" and "MessageAvailability." A vertex with `type` "ActionAvailability" has static attributes `action` and `actor`. For example, in the context of the smart grid, the `action` can be "MeterReading," and the `actor` can be "Utility". A vertex with `type` "ActorAvailability", has static attribute `actor`. For example, the `actor` can be "Utility." We call the resulting graph at this stage the *G-graph*, as it models the immediate requirements related to the goal of the assessment.

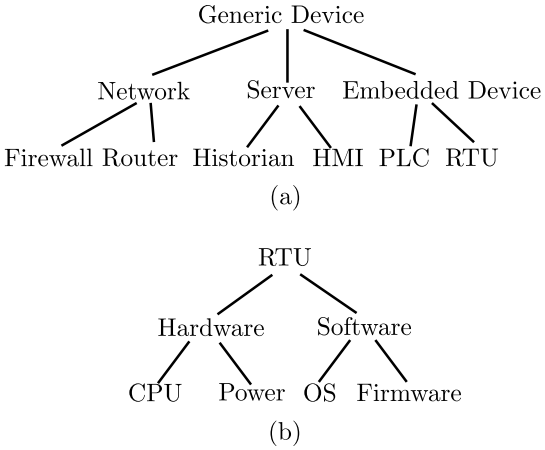


Fig. 4: Examples of a device type hierarchy (a) and composition tree (b). The device types are based on the power grid substation example, with an HMI (Human-Machine Interface), PLC (Programmable Logic Controller), and RTU (Remote Terminal Unit).

B. Graph Generation Using System Input

As the next step, the argument graph ω_G is expanded using Algorithm 1, this time with different operands: $\text{GENERATEGRAPH}(\omega_g, \Gamma_s, \Sigma_s)$. Here, the operands are the system information Σ_s and a set of extension templates Γ_s. Here, Γ_g contains T_4 and T_5, the formal extension templates for patterns P_4 and P_5.

System information inputs. The system description contains multiple types of inputs, including the following.

- An actor-to-component mapping: This is similar to the deployment diagram in UML. It maps the actor from the workflow to a `componentType`.
- A network topology graph $\omega_n = \langle V_n, E_n, l_n \rangle$, where each vertex $v \in V_n$ is a physical device in the system, and each edge $e \in E_n$ is a link (which can be a single-hop physical communication link, a network path, or physical). The attribute function $l_n(v)$ describes various attributes of a device v, including its type, physical location, and access privileges, among others. For a link e, $l_n(e)$ describes its attributes, like type, capacity, and delay.
- A `componentType` hierarchy diagram as depicted in Fig. 4a.
- The device composition information as depicted in Fig. 4b.

Vertex types and templates for GS-graph generation. Based on those inputs, two extension templates can be applied to the argument graph: T_4 and T_5. T_4 makes it possible to connect the abstract actor in the workflow with the concrete device that executes this action, thus introducing "ComponentAvailability" vertices to the graph. A vertex of that type has 2 static attributes: `component` and `componentType`. The `componentType` is taken from a hierarchy of device types with increasing specificity towards the graph's leaves. The `component` is taken from a tree that describes the subcomponents for each `componentType`. If no subcomponent tree is available for a given `componentType`, the subcomponent tree of a parent `componentType` will be used instead. We describe T_5 in more detail in Section V-D.

C. Graph Generation Using Attacker Input

As the last step, the argument graph ω_{gs} is expanded using Algorithm 1, this time with different operands: GENERATEGRAPH$(\omega_{gs}, \Gamma_a, \Sigma_a)$. Here, the operands are the system information Σ_a and a set of extension templates Γ_a. The two patterns for the extension templates T_6 and T_7 in Γ_a were not introduced earlier. They are similar to P_2 and P_3, but relating to the attacker (instead of the system).

Attacker information input. The input placeholder Σ_a here contains the attacker model, which contains a set of attacker properties and a set of attack sequences Ω_r (a set of star graphs). Each star graph ω_r contains a potential attack step and its immediate prerequisites. The vertices in ω_r are of `type` "AttackStep" and "AttackerProperty." Attacker property information relates to methods, knowledge, and physical access of the attacker, e.g., access to a company's compound or server room. Our security argument graph connects any possible single attack step of the attacker to the GS-graph, and we do not model the exact goal of the attacker.

Vertex types and templates for GSA-graph generation. In the GSA-graph, new vertices of `type` "AttackStep" and "AttackerProperty" are introduced (matching the nodes from the attacker model). Currently, only single step attacks and their requirements can be modeled, however we are developing additional vertex types to incorporate multi-step attacks.

D. Automatic Generation using Templates

In the previous sections, we have introduced several extension templates that we distilled from common argument patterns that we observed. As part of our effort toward automating the security assessment of complex systems, we are compiling an extension template library. All of the extension templates in Table I are defined in pseudocode. In this paper, we present only the core set of templates we are currently working on: additional extension templates are omitted from this paper because of space limitations.

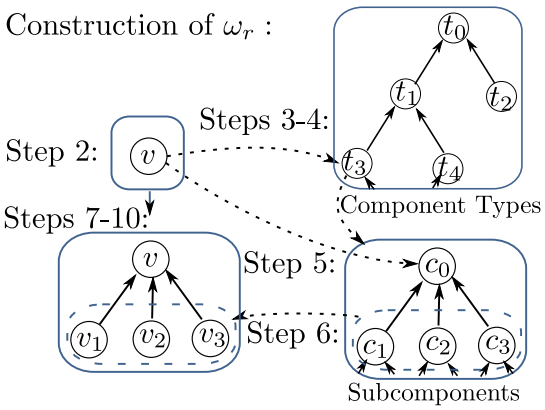
We use the extension template T_5 (*device decomposing requirements*) to illustrate the underlying extension process. Extension template T_5 relies on component type and device composition hierarchies, which are specified as an input to the security assessment (part of Σ_{gs}). Fig. 4 presents examples of the composition hierarchies, which were discussed in Section V-B. Template T_5 uses the supporting hierarchies to expand a single graph vertex v. We show the extension generation function in Fig. 5a and represent the process graphically in Fig. 5b. In particular, the local extension for a component property is created by finding the best matching `componentType` from the `componentType` tree. That best `componentType` is then used to find the matching property composition tree, and to look up the next decomposition for the current property. All potential decomposition nodes are added to a star graph ω_r, and returned together with the node v as a local extension.

That precise description of the extension template application process and required input information allows the extension templates to be readily implemented in a software tool. In the next section, we present an example security assessment, and show that the process can be automated using a supporting software tool that is currently under development.

Algorithm 2 Template T_5

```
1:  function T5(v, Σgs)
2:    ωr ← NewGraph({v}, ∅, 0);
3:    ot ← GetComponentType(v);
4:    ωc ← GetCompositionInfo(ot, Σgs);
5:    oc ← GetComponent(v);
6:    C ← GetSubComponents(oc, ot, ωc);
7:    ∀c ∈ C:
8:      vc ← NewVertex(
                "ComponentAvailability", c, ot);
9:      Vr ← Vr ∪ {vc};
10:     Er ← Er ∪ {⟨vc, v⟩};
11:   lr(v) ← lc(oc);
12:   return ⟨v, ωr⟩;
13: end function
```

(a)



(b)

Fig. 5: Details on extension template T_5: (a) Pseudocode of the extension function $T_5(v, \Sigma_a)$. (b) Visualization of the process.

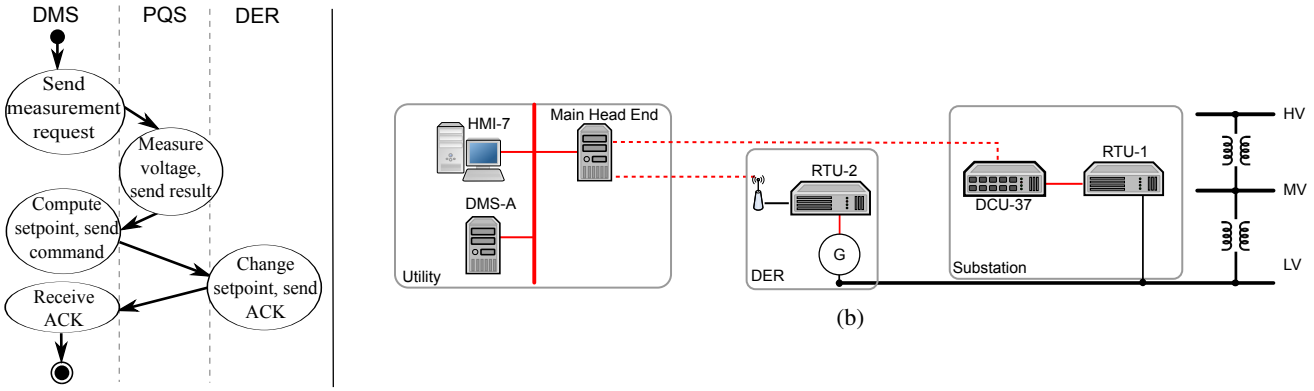


Fig. 6: Smart grid substation use case: (a) Voltage measurement and control workflow; (b) System topology.

VI. Application: An Electric Power Grid Use Case

In this section, we apply the algorithms and templates presented earlier to an illustrative use case from the electric power sector. We start by manually deriving a security argument graph to explain important details. We then discuss graph generation in CyberSAGE [18], a security assessment tool that is currently under development. Using CyberSAGE, we can generate the argument graph automatically, based on a library of extension templates and a set of inputs.

A. Assessment Input

We consider an example power system use case adapted from [19]. This scenario is a typical supervisory control and data acquisition (SCADA) operation, connecting a utility company's network with intelligent field devices that manipulate physical power grid parameters. In this example, a central distribution management system (DMS) monitors the voltage at a specific point (i.e., bus) in the low voltage (LV) power distribution network as reported by the power quality sensor (PQS) there, and uses that information to trigger a control action in another device located in the network. The control device in this example is a distributed energy resource (DER), e.g., an electric vehicle or small wind turbine. An example workflow for the reactive power control process is shown in Fig. 6a.

Our security goal in this example is to ensure the availability of the measurement and control process. A simplified system diagram is shown in Fig. 6b, which includes both communication links (red) and physical power network connectivity (black). Wide-area communication between different physical locations is indicated by dashed edges. In the system shown in Fig. 6b, RTU-1 is the device measuring the bus voltage, while RTU-2 is responsible for controlling the DER's reactive power output. Both devices are of `componentType` RTU. A server device (`componentType` "server") in the corporate network, DMS-A, is responsible for distribution management functionality.

The attacker model in this example includes attacks (i.e., "AttackStep" vertices) targeting:

- Device power supply (Physical Tampering)
- Device operating system (Exploit Vulnerability)
- Device network connectivity (Denial of Service)

The "AttackerProperty" information that enables these attack steps is specific to the class of device being targeted.

B. Security Argument Graphs

Knowledge of the assessment goal, the workflow (Fig. 6a), and the system (Fig. 6b) inputs, the actor `componentType` mapping, and an attacker model allows us to apply the extension templates defined in Table I to construct a security argument graph according to our framework [17]. The graph generation process is depicted visually in Fig. 7. First, the base graph (representing the goal) is extended using workflow input. Extension templates T_1 to T_3 are used during this stage to identify dependencies between actions and actors. Next, templates T_4 and T_5 are used to enrich the argument graph with system-specific information (GS-graph). The fully developed GSA-graph, formed by applying T_6 and T_7, is shown at the bottom of Fig. 7.

This complete security argument graph (GSA-graph) organizes security-related information that originated from disparate sources and formats. The human-readable structure is intended to help system designers and other stakeholders understand dependencies and possible security threats in a complex system. In particular, it is clear from Fig. 7 that the distribution management system (DMS) is critical to the voltage control process—it is involved in 3 separate actions, occuring at different times. However, since the DMS is located in the utility's back-office network it may pose less of a security risk than the DER, which is a field device. While this analysis is qualitative, the GSA-graph can be used for quantitative security assessment as well. In [17] we outline an approach for quantitative evaluation over the argument graph. That quantitative evaluation has been refined and is implemented in our software tool, which is discussed below.

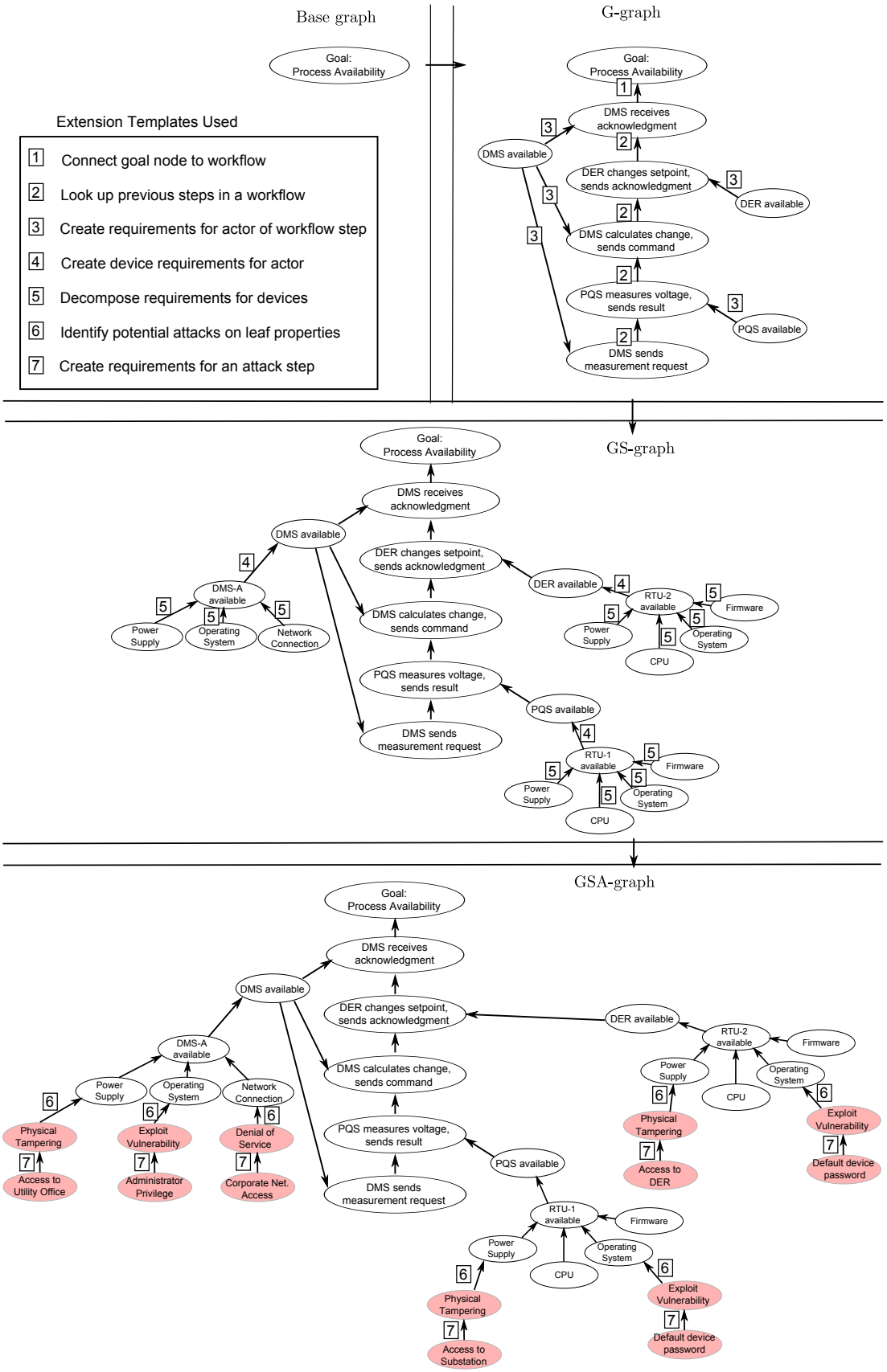


Fig. 7: Manually derived security argument graphs for the distribution automation use case. Each edge is annotated to show the extension template applied during graph generation.

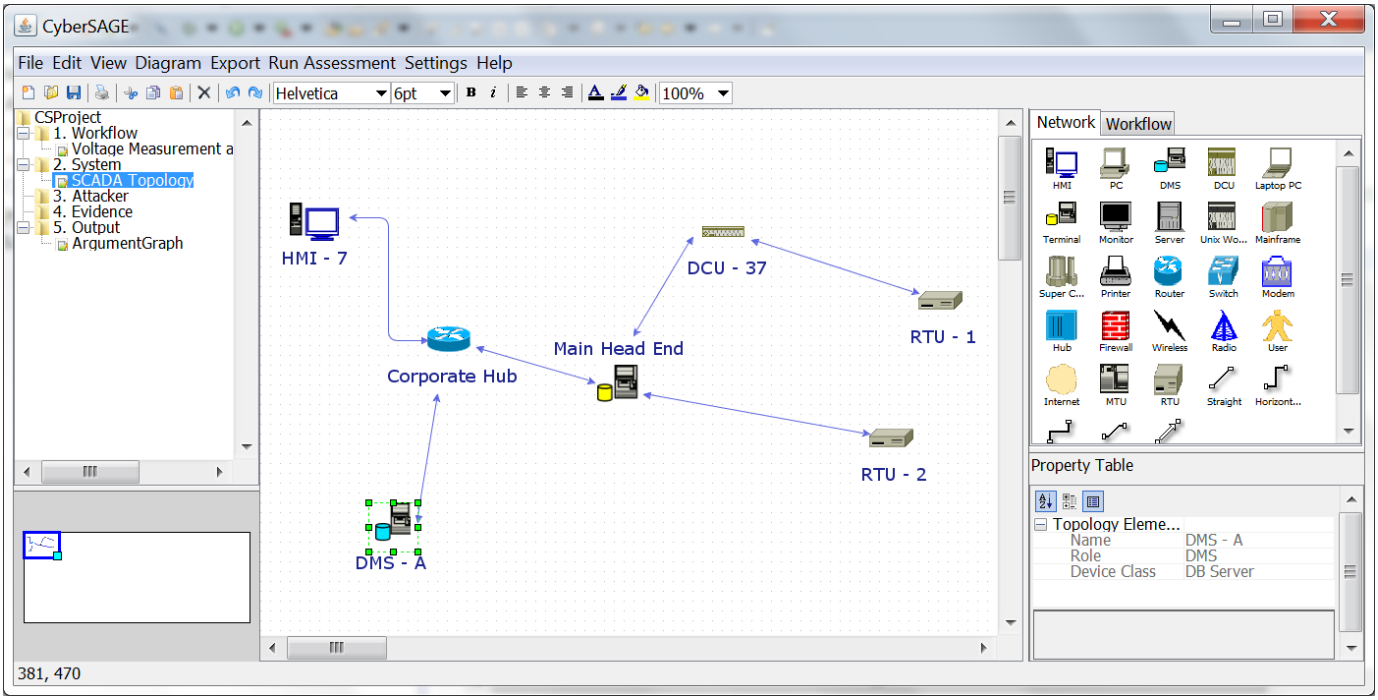


Fig. 8: CyberSAGE prototype: a snapshot of the software with system input configuration (topology, device information, mapping of actors to devices).

C. Automation in the CyberSAGE Tool

We implemented the proposed template-based graph generation using our prototype assessment tool, which we call CyberSAGE (Cyber Security Argument Graph Evaluation) [18]. Based on the inputs described in Section VI, CyberSAGE is able to automatically generate the argument graph.

Currently, CyberSAGE can process workflow and system inputs that are in XML format. The CyberSAGE tool provides built-in functionality to edit these inputs; it can also directly import XML files that are output by other applications. For example, CyberSAGE can import the system topology input directly from the output files of the CSET tool [21]. Once CyberSAGE has imported those files, it builds its internal data structures and visualizes their contents in a user-friendly manner. Fig. 8 shows a screen-shot of the tool, which is displaying the system topology for the electric power grid use case introduced earlier in this section.

The remaining inputs, namely the attacker models and extension templates, are currently static and provided by CyberSAGE. With the extension templates implemented, CyberSAGE automatically builds the argument graph by following Algorithm 1. In this use case, it takes the tool less than 1 second to produce the final argument graph, which contains around 50 vertices. The tool also provides some degree of customization, such as allowing the user to enable or disable a particular subset of extension templates when building the security argument graph.

VII. Related Work

We see parallels between our work on security argument graph generation and work from the safety and reliability communities, as well as related efforts within the security domain. In this section, we discuss related efforts in graph-based modeling and highlight unique features of our framework.

Safety case generation. A safety case uses certain argument strategies to organize a body of evidence so as to provide a compelling case for supporting certain safety claims (goals) [4]. Safety cases are typically constructed manually. Recent efforts (e.g., [5], [6]) have begun to introduce formal semantics to help automate the safety case generation process. Compared with those recent efforts, our proposed approach focuses on argument patterns that incorporate various security-related evidence, including security goals and attacker models. We also formalize the template in a local way, which simplifies the definition and instantiation of the template while still allowing progressive generation of the argument graph.

Fault tree generation. Fault tree analysis is a classic deductive method used to determine what combinations of basic component failures can lead to a system-level fault event [7]. While fault trees are usually constructed manually in practice, there has been a steady stream of efforts to automate the fault tree generation process. For example, Pai et al. [8] propose a method to transform a UML system model to dynamic fault trees. Joshi et al. [22] propose a method to automatically generate a dynamic fault tree from an Architectural Analysis and Design Language (AADL) model. Recently, Xiang et

al. [9] propose an automatic synthesis method to generate a static fault tree from a system model specified with SysML.

Compared with that line of work, our proposed approach not only considers various security-specific information, but also intends to cover a broader scope of security-related claims and heterogeneous pieces of evidence.

Attack trees and other security assessment techniques. Attack trees and their variations (e.g., attack graphs [11], [12], ADVISE [13], and attack-defense trees [14]) have been shown to be useful for security assessment. Inspired by the fault tree formalism, an attack tree graphically represents how a potential threat can be realized through various possible combinations of attacks. Attack trees are usually constructed manually. For the specific domain of network security, multiple efforts (e.g., [11], [12]) have tried to automate the generation of attack graphs, which are meant to model how an attacker can use staged attacks to compromise certain assets in a network. ADVISE [13] automates the search of attack strategies. Those efforts differ from ours in that they do not provide a framework that can automatically integrate heterogeneous pieces of information (e.g., relating to security goals, workflows, system information, and the attacker) to produce a holistic security argument.

VIII. Conclusion

In this work, we consider graph-based security assessment for critical infrastructure and other complex systems. Such assessments can be used to identify potential attacks and to compare the security properties of different designs. Because of the size of real-world systems, the assessment process should ideally be tool-supported and automated to a large extent. In our earlier work [17], we proposed a holistic security assessment framework that combines a collection of heterogeneous input information into a *security argument graph*. Building on that work, in this paper we identify a collection of *argument patterns* that emerge from relationships between security-related information inputs. We formalize these argument patterns as *extension templates*, and provide a method for automated graph generation using these extension templates. We describe how we implemented the proposed method in our security assessment tool, CyberSAGE [18], and demonstrate the automated generation process for an example use case from the power sector.

Acknowledgments

This work is supported by Singapore's Agency for Science, Technology, and Research (A*STAR) under the Human Sixth Sense Programme (HSSP). We thank Jenny Applequist for her help with editing. We thank Sumeet Jauhar and Xinshu Dong for helpful discussions.

References

[1] S. Amin, X. Litrico, S. Sastry, and A. M. Bayen, "Cyber security of water SCADA systems - part I: Analysis and experimentation of stealthy deception attacks," *IEEE Trans. Contr. Sys. Techn.*, vol. 21, no. 5, pp. 1963–1970, 2013.

[2] R. Anderson and S. Fuloria, "Who controls the off switch?" in *Proc. of the IEEE Conference on Smart Grid Communications (SmartGridComm)*, 2010.

[3] W. G. Temple, B. Chen, and N. O. Tippenhauer, "Delay makes a difference: Smart grid resilience under remote meter disconnect attack," in *Proc. of the IEEE Conference on Smart Grid Communications (SmartGridComm)*, 2013.

[4] T. Kelly, "Arguing safety: A systematic approach to managing safety cases," Ph.D. dissertation, University of York, UK, 1998.

[5] E. Denney, G. Pai, and J. Pohl, "Heterogeneous aviation safety cases: Integrating the formal and the non-formal," in *Proc. of the Conference on Engineering of Complex Computer Systems (ICECCS)*, 2012.

[6] E. Denney and G. Pai, "A formal basis for safety case patterns," in *Proc. of the Conference on Computer Safety, Reliability and Security (SAFECOMP)*, 2013.

[7] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, *Fault Tree Handbook*. U.S. Nuclear Reg. Comm., 1981.

[8] G. J. Pai and J. B. Dugan, "Automatic synthesis of dynamic fault trees from UML system models," in *Proc. of the Symposium on Software Reliability Engineering (ISSRE)*, 2002.

[9] J. Xiang, K. Yanoo, Y. Maeno, and K. Tadano, "Automatic synthesis of static fault trees from system models," in *Proc. of the Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, 2011.

[10] B. Schneier, "Attack trees: Modeling security threats," *Dr. Dobb's Journal*, 1999.

[11] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. Wing, "Automated generation and analysis of attack graphs," in *Proc. of the IEEE Symposium on Security and Privacy*, 2002.

[12] X. Ou, W. Boyer, and M. McQueen, "A scalable approach to attack graph generation," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2006.

[13] E. LeMay, M. Ford, K. Keefe, W. H. Sanders, and C. Muehrke, "Model-based security metrics using ADversary VIew Security Evaluation (ADVISE)," in *Proc. of the Conference on Quantitative Evaluation of SysTems (QEST)*, 2011.

[14] B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer, "Foundations of attack-defense trees," in *Proc. of the conference on Formal Aspects of Security and Trust (FAST)*, 2011, pp. 80–95.

[15] D. M. Nicol, W. H. Sanders, and K. S. Trivedi, "Model-based evaluation: from dependability to security," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 48–65, 2004.

[16] V. Verendel, "Quantified security is a weak hypothesis," in *Proc. of the New Security Paradigms Workshop (NSPW)*, 2009.

[17] B. Chen, Z. Kalbarczyk, D. M. Nicol, W. H. Sanders, R. Tan, W. G. Temple, N. O. Tippenhauer, A. H. Vu, and D. K. Yau, "Go with the flow: Toward workflow-oriented security assessment," in *Proc. of the New Security Paradigms Workshop (NSPW)*, 2013.

[18] A. H. Vu, N. O. Tippenhauer, B. Chen, D. M. Nicol, and Z. Kalbarczyk, "CyberSAGE: A tool for automatic security assessment of cyber-physical systems," in *Proc. of the Conference on Quantitative Evaluation of SysTems (QEST)*, 2014.

[19] CEN-CENELEC-ETSI Smart Grid Coordination Group, "Smart grid reference architecture," http://ec.europa.eu/energy/gas_electricity/smartgrids/doc/xpert_group1_reference_architecture.pdf, November 2012.

[20] B. Kordy, L. Pietre-Cambacedes, and P. Schweitzer, "DAG-based attack and defense modeling: Don't miss the forest for the attack trees," *CoRR*, vol. abs/1303.7397, 2013.

[21] ICS-CERT, "CSET: The cyber security evaluation tool," ics-cert.us-cert.gov/satool.html, last accessed on April 11, 2013.

[22] A. Joshi, S. Vestal, and P. Binns, "Automatic generation of static fault trees from AADL models," in *Proc. of DSN Workshop on Architecting Dependable Systems*, 2007.