SAND2020-1750C

SCIBORG: Secure Configurations for the IOT Based on Optimization and Reasoning on Graphs

Hamed Soroush*, Massimiliano Albanese†, Milad A. Mehrabadi*, Ibifubara Iganibo†, Marc Mosko*, Jason H. Gao‡, David J. Fritz‡ Shantanu Rane*, Eric Bier*,
*Palo Alto Research Center, Palo Alto, CA 94304, USA, Email: {hsoroush,srane,bier,mmosko}@parc.com
†George Mason University, Fairfax, VA 22030, USA, Email: {malbanes,iiganibo}@gmu.edu
‡Sandia National Laboratories, XXXXX, USA, Email: {jhgao,djfritz}@sandia.gov

***Abstract*—Addressing security misconfiguration in complex distributed systems, such as networked Industrial Control Systems (ICS) and Internet of Things (IoT) is challenging. Owners and operators must go beyond tuning parameters of individual components and consider the security implications of configuration changes on entire systems. Given the growing scale of cyber systems, this task must be highly automated. Prior work on configuration errors largely ignores the security impact of configurations of connected components. To address this gap, we present SCIBORG, a framework that improves the security posture of distributed systems by examining the impact of configuration changes across interdependent components through a graph-based model of the system and its vulnerabilities. It defines a Constraint Satisfaction Problem from the graph-based model and uses an SMT solver to find optimal configuration parameter values that minimize the impact of attacks while preserving system functionality. The framework also provides supporting evidence for the proposed configuration changes. We evaluate SCIBORG on an IoT testbed.**

I. Introduction

As cyber systems become increasingly complex and connected, configuration analytics begins to play an increasingly critical role in their correct and secure operation. Attackers typically rely on unpatched vulnerabilities and configuration errors to gain unauthorized access to system resources. Misconfiguration can occur at any level of a system's software architecture, and correctly configuring systems becomes increasingly complex when multiple interconnected systems are involved. *Security Misconfiguration* was listed by OWASP amongst the ten most critical web application security risks in 2017 [12]. Current configuration security approaches focus on tuning the configuration of individual components while lacking a principled approach to managing the complex relationships between the configuration parameters of the components of a complex system. Additionally, current configurations are mostly static, governed by slow and deliberative processes, with significant involvement from human analysts.

In this paper, we first corroborate the significance of security misconfiguration vulnerabilities by analyzing data from the National Vulnerability Database (NVD)[1] and Shodan[2]. We then present the design, implementation, and evaluation of SCIBORG, a system that addresses the mentioned limitations. Our key contributions can be summarized as follows. First, we model a composed system using a multi-layer graph comprising a dependency sub-graph that captures the functional relationships among system components, a configuration subgraph that accounts for relationships among configuration parameters within and across components, and an attack subgraph modeling the system's vulnerabilities and their use in multi-step attacks. Second, we characterize the potential impact of multi-step attacks enabled by configuration settings. Prior work on minimizing a system's attach surface does not capture the intricate relationships between configuration parameters, attack paths available to an adversary, and functional dependencies among system components. Thus, it generally fails to reduce the risk associated with residual vulnerabilities. Third, we develop algorithms and software tools to jointly analyze the subgraphs of the multi-layer graph in order to reason about the impact of a candidate configuration set on the security and functionality of the composed system. We use a Satisfiability Modulo Theory (SMT) solver to express the complex relationships among the configuration parameters as constraints in a security optimization problem.

[1]https://nvd.nist.gov
[2]https://www.shodan.io/

We have implemented SCIBORG as a scalable pipeline that (i) ingests system requirements, configuration files, software documentation and various types of configuration vulnerabilities, (ii) builds a queryable, graph-based representation of the relationships between configuration vulnerabilities and attack scenarios, configuration parameters and system components, (iii) provides an API to perform a quantitative, comparative analysis of the security impact of configuration settings, (iv) automatically constructs a constraint satisfaction problem based on the model and utilizes Z3 SMT solver to solve for optimal parameter values, and (v) provides human-readable evidence about optimality of the selected configuration.

The remainder of the paper is organized as follows. Section II presents background information to motivate our work. Then, Section III presents the proposed model in detail, and Section VI reports results from our evaluation. Finally, Section VII discusses related work, and Section VIII gives some concluding remarks and identifies future research directions.

II. Motivation

A significant fraction of the downtime of critical infrastructure has been attributed to misconfigurations. Configuration-related vulnerabilities can cause adverse outcomes that impact security and functionality, including data breaches, denial of service, system downtime, and inefficient operation. However, many configuration-related vulnerabilities are not reported to

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

vulnerability databases, such as NVD, because they are considered user errors rather than software issues. Consequently, estimating the prevalence and significance of such vulnerabilities is challenging. Nevertheless, one can get lower bounds on these metrics by analyzing those vulnerabilities that get reported and by using services such as Shodan that index information about the configuration of Internet-connected devices.

Below, we present a longitudinal analysis of configuration vulnerabilities reported to NVD as well as an analysis of devices on Shodan that suffer from security-related vulnerabilities. This analysis allows us to conclude that configuration vulnerabilities linger in Industrial Control and IoT systems for an unacceptable amount of time compared to non-configuration related vulnerabilities, in spite of their generally higher impact.

A. Dataset I: Configuration-Related Vulnerabilities in NVD

We gathered all Common Vulnerability and Exposure (CVE) entries[3] from NVD reported between 2010 and 2018. A typical NVD entry has one or more Common Weakness Enumeration Specification (CWE) labels[4] indicating the type of vulnerability. We identified several such categories, listed in Table I, as configuration-related vulnerabilities in our *Dataset I*. After removing entries with no CWE label, this dataset contains 67,742 vulnerabilities. Fig. 1 shows how the number of reported vulnerabilities has changed over the analysis period along with the fraction of configuration-related CVEs for each year. As a longitudinal study, Fig. 2 shows the evolution of the impact score derived from the Common Vulnerability Scoring System[5] (CVSS) version 3.0 for config and non-config vulnerabilities over the analysis period. Config vulnerabilities have generally higher impact than non-config ones. The impact score of recent configuration vulnerabilities has lower variance, indicating higher confidence in their impact.

TABLE I. SOFTWARE CONFIGURATION VULNERABILITY CATEGORIES

CWE ID	Name	NVD Short Description
CWE-16	Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
CWE-255	Credential Management	Weaknesses in this category are related to the management of credentials.
CWE-264	Permissions, Privileges, and Access Controls	Weaknesses in this category are related to the management of permissions, privileges, and other security features that are used to perform access control.
CWE-275	Permission Issues	Weaknesses in this category are related to improper assignment or handling of permissions.
CWE-284	Improper Access Control	The software does not restrict or incorrectly restricts access to a resource from an unauthorized actor.
CWE-285	Improper Authorization	The software does not perform or incorrectly performs an authorization check when an actor attempts to access a resource or perform an action.
CWE-552	Files or Directories Accessible to External Parties	Files or directories are accessible in the environment that should not be.
CWE-665	Improper Initialization	The software does not initialize or incorrectly initializes a resource, which might leave the resource in an unexpected state when it is accessed or used.
CWE-769	Uncontrolled File Descriptor Consumption	The software can be influenced by an attacker to open more files than are supported by the system.

[3]http://cve.mitre.org/

[4]https://cwe.mitre.org/

[5]https://nvd.nist.gov/vuln-metrics/cvss

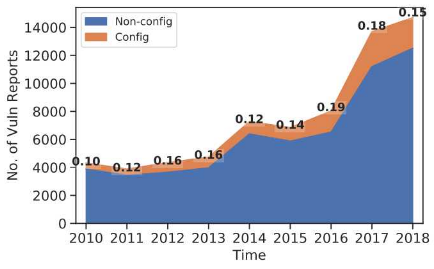


Fig. 1. Number of config versus non-config vulnerability reports over time.

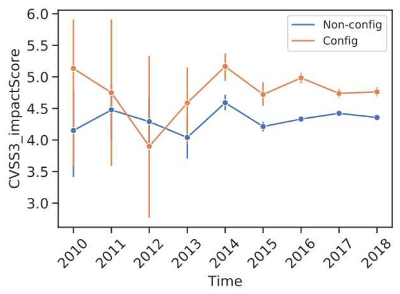


Fig. 2. CVSS 3.0 impact score for config and non-config vulnerabilities.

Fig. 3 shows the complementary CDF of different scores for config and non-config vulnerabilities, for both CVSS 2.0 and CVSS 3.0. In particular, we analyze the distribution of the *Impact Score*, *Exploitability Score*, and *Severity Score*, which is a combination of the Impact and Exploitability scores. As Fig. 3(a) shows, config-related vulnerabilities have higher impact than non-config-related vulnerabilities for both CVSS 2.0 and CVSS 3.0 The exploitability score of config-related vulnerabilities is lower compared to non-config vulnerabilities, as shown in Fig. 3(b). However, as shown in Fig. 3(c), higher impact of config-related vulnerabilities prevails over the exploitability and makes for an overall higher severity score.

B. Dataset II: Config-Related Vulnerabilities in Shodan

Shodan is a popular search engine for characterizing Internet facing IoT devices and services. It utilizes custom crawlers that scan the Internet regularly and store information about hosts, such as potential device tag name(s), product name, IP address, vulnerabilities, crawler type, and a timestamp of the scan. Shodan makes this longitudinal information available through a graphical user interface and an API.

We established our second dataset, *Dataset II*, by obtaining all data from Shodan until August 2019, focusing our analysis on ICS and IoT devices in the United States. For each such device, we acquire detailed historical information (e.g., vulnerability/infection information over time or whether they've been tagged as honeypots in the past) by querying Shodan using its API. We remove IP addresses that had at least one report of being tagged as a honeypot in the past (88 addresses for ICS and 0 for IoT devices) or that have no vulnerability information available. Our *Dataset II* includes 3,143 and 1,839 distinct IP addresses with vulnerability information for ICS and IoT devices, respectively. These data include standard vulnerabilities from NVD and Microsoft security bulletins[6]. We identify the type of each vulnerability by looking up its

[6]https://docs.microsoft.com/en-us/security-updates/

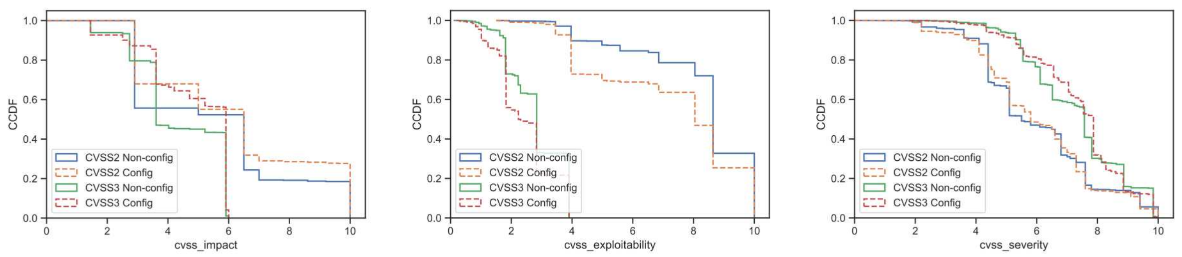


Fig. 3. Complementary cumulative distribution function (CCDF) of config versus non-config (a) impact score, (b) exploitability score, and (c) severity score of CVE entries in NVD dataset from 2010 to 2018 for CVSS versions 2.0 and 3.0.

CWE label from NVD and label configuration-related and non-configuration related vulnerabilities according to Table I. To identify the type of each vulnerability, we focus on NVD-based vulnerabilities and extract CWE labels from NVD data to be consistent with Section II-A.

Fig. 4 shows an analysis of the lingering time (in days) of vulnerabilities in IoT and ICS systems. Config vulnerabilities last longer in both ICS and IoT systems. While almost 18% of non-config vulnerabilities for IoT systems linger for more than 300 days (out of the 16-month period of available historical data), about 28% of config-related vulnerabilities last more than 300 days. The percentages are higher for ICS systems: 30% and 40% for non-config and config vulnerabilities, respectively. These results indicate that, despite their high impact, configuration vulnerabilities linger for an unacceptable amount of time in ICS and IoT systems, emphasizing the need for solutions that discover and remediate them.

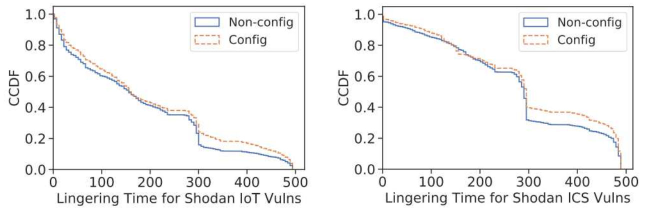


Fig. 4. Lingering time of configuration and non-configuration vulnerabilities in (a) IoT devices and (b) ICS devices, as reported by Shodan.

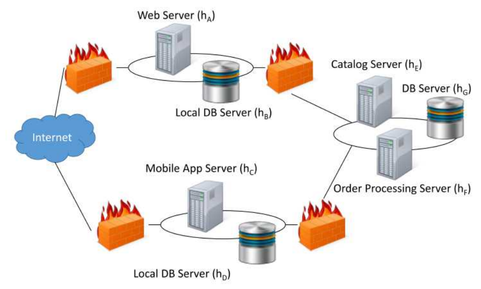


Fig. 5. Network diagram of a notional e-commerce system.

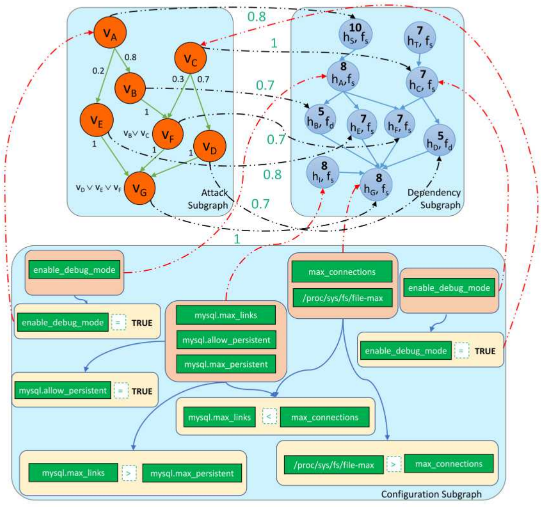


Fig. 6. A graph corresponding to the system of Fig. 5.

III. Multi-Layer Graph Model

SCIBORG's approach is based on modeling the distributed system (or *composed system*) as a three-layer directed graph efficiently encoding the information needed to reason upon the optimality of system configurations. The three layers are (i) a dependency subgraph; (ii) a configuration subgraph; and (iii) a vulnerability subgraph. Directed edges between the three subgraphs define the functional composition and the attack surface for a configuration set. For illustrative purposes, a three-layer graph corresponding to the notional distributed system of Fig. 5 is depicted in Fig. 6. The SCIBORG implementation and evaluation of an actual IoT system is discussed in Section VI.

A. Dependency Subgraph

Configuration changes in one component can have a dramatic impact on the security and functionality of other components. Globally optimal security decisions – e.g., deciding which vulnerabilities to make unreachable through configuration changes – need dependency information. To this aim, we explicitly models dependencies. While finding all dependencies is a difficult problem beyond the scope of SCIBORG, we derive a useful set of dependencies by analyzing standard operating procedures or using approaches such as [4], [11].

A node in the dependency subgraph represents a system component, and a directed edge represents a dependency between two components. Depending on the level of granularity of the model, a component may be a host or an individual service running on a host. When dependencies are captured at the lowest possible level of granularity, the dependency graph is expected to be acyclic. Current approaches to dependency discovery may generate graphs with cycles, but such cycles are often an indication that the system has not been analyzed at a sufficient level of granularity, and can be *broken* by breaking macro-components into sub-components. Literature on call graphs shows how we can identify dependencies at the level of individual procedure and function calls and construct acyclic graphs modeling such dependencies [13].

To capture a wide range of possible relationships between a system's components, we model each dependency as a function from a family $\mathcal{F}$ of functions of the form $f : [0,1]^n \rightarrow [0,1]$, with $f(0,\dots,0) = 0$ and $f(1,\dots,1) = 1$. Each component has a value (or utility) for the organization and its *dependency function* defines its ability to deliver its expected value, based on the status of the components it depends on: the arguments of this function are the percentage residual values of such components and are in turn computed through each component's respective dependency function. A dependency function returns 1 when the component can to deliver 100% of its value, and 0 when the component has been completely compromised and cannot deliver any value. We identify three major categories of dependency relationships, namely (i) *redundancy* (f_r), wherein a component depends on a redundant pool of resources; (ii) *strict dependence* (f_s), wherein a component strictly depends on a pool of other components, such that, if one fails, the dependent component no longer delivers value; and (iii) *graceful degradation* (f_d), wherein a component depends on a pool of other components such that, if one fails, the system continues to work with degraded performance. Such classification is not intended to be exhaustive, and other dependency relationships can be introduced by defining the corresponding dependency functions, as shown below for the three categories listed above.

$$f_r(l_1,\dots,l_n) = \begin{cases} 1, \text{ if } \exists i \in [1,n] \text{ s.t. } l_i = 1 \\ 0, \text{ otherwise} \end{cases} \tag{1}$$

$$f_d(l_1,\dots,l_n) = \frac{1}{n}\sum_{i=1}^{n} l_i \tag{2}$$

$$f_s(l_1,\dots,l_n) = \begin{cases} 1, \text{ if } \forall i \in [1,n], l_i = 1 \\ 0, \text{ otherwise} \end{cases} \tag{3}$$

Fig. 6 shows the dependency subgraph for our notional system. An edge from h_A to h_B denotes that h_A depends on h_B. Each node is labeled with the type of dependency and a number representing the value of the corresponding component. Such values can be assigned by a domain expert or automatically derived by computing graph-theoretic centrality metrics [7], which indicate how important (or central) each node is for the operation of a system or mission. In the security field, ad-hoc centrality measures are used for botnet detection and mitigation [17].

B. Configuration Subgraph

The configuration subgraph models relationships between configuration parameters, both within a component and across different components of the composed system. There are two classes of vertices in the configuration subgraph, namely, *Class 1 vertices*, which represent per-component configuration parameters, and *Class 2 vertices*, which capture constraints on one or more configuration parameters. Edges from one or more Class 1 vertices to a Class 2 vertex identify the parameters involved in a constraint. Some of these constraints are specified in the component's or composed system's documentation. More importantly, some of the relationships between configuration parameters might enable or disable preconditions for vulnerabilities in one or more components. SCIBORG captures this information by means of directed edges from Class 2 vertices of the configuration subgraph to relevant nodes in the vulnerability subgraph. The constraints associated with a given system configuration induce a specific vulnerability subgraph for the composed system. For instance, in Fig. 5, the constraint `enable_debug_mttode = TRUE`, which must be satisfied when the system is in debug mode, creates the preconditions to exploit vulnerability v_a.

The degree to which configuration parameter dependencies, within and across components, can be captured depends to a large degree on the complexity of the components themselves and the completeness of their documentation, including the set of standard operating procedures adopted by an organization. Additionally, SCIBORG extracts configuration information – in a variety of formats, including XML and JSON – from specification documents for commercial off-the-shelf (COTS) components. Finally, for each component, SCIBORG also ingests vulnerability information from NVD.

C. Vulnerability Subgraph

Vulnerability subgraphs, also known as attack graphs, are powerful conceptual tools to represent knowledge about vulnerabilities and their dependencies. To assess the impact of configuration changes on a system's attack surface, we will employ vulnerability subgraphs as formalized in [1], which in turn relies on the compact representation of attack graphs that was proposed in [3]. Such representation keeps one vertex for each exploit or security condition, leading to an acyclic attack graph of polynomial size in the total number of vulnerabilities and security conditions. A vulnerability subgraph for our notional system is depicted in Fig. 6: vertices represent known vulnerabilities, and an edge from vulnerability v_A to vulnerability v_B indicates that exploiting v_A creates the preconditions for exploiting v_B. These subgraphs can be generated by combining information from network scanners (e.g., Nessus[7]) and vulnerability databases (e.g., CVE, NVD), as shown in [3], [6], [15].

SCIBORG's approach differs from the traditional idea of minimizing the attack surface by minimizing, for instance, the *number* of exploitable resources available to the adversary. Instead, we analyze the *paths* that an adversary can traverse in a multi-step attack that seeks to achieve a well-defined goal (e.g., compromising a series of devices that lead up to a database and then exfiltrating sensitive information from that database), and evaluate the impact resulting from such attacks. The edges in the vulnerability subgraph of Fig. 6 are labeled with probability values, which can be used to infer the most likely paths that an attacker might take in a multi-step attack. Determining the probability values is an open research problem, though useful heuristics exist [2], [1]. For instance, the likelihood that an attacker will exploit a given vulnerability can be derived from (i) the skill level of the attacker relative to the complexity of the exploit [8]; (ii) the resources and time available to the adversary; (iii) other metrics defined in CVSS. The rationale is that vulnerabilities that require more resources, time, and skill are less likely to be exploited. For example, CVSS defines the Access Complexity (AC) of a vulnerability as a measure of the intricacy of the attack required to exploit that vulnerability once an attacker has gained access to the target system. These probabilities are used to determine the security impact of a given configuration.

[7]http://www.tenable.com/products/nessus

D. Edges across Subgraphs

In addition to edges within subgraphs, our model includes directed edges across the three subgraphs as described below.

Dependency Subgraph → Configuration Subgraph. A directed edge from a component in the dependency graph to a Class 1 vertex in the configuration graph represents the list of configuration parameters associated with that component. There are no edges between the dependency subgraph and Class 2 vertices in the configuration subgraph.

Configuration Subgraph → Vulnerability Subgraph. A directed edge between a Class 2 node in the configuration subgraph to a vertex in the vulnerability subgraph implies that the constraint expressed in the Class 2 vertex represents a precondition for exploiting that vulnerability.

Vulnerability Subgraph → Dependency Subgraph. An edge from a vulnerability in the vulnerability subgraph to a component in the dependency subgraph represents the *exposure factor* of the component to the exploitation of that vulnerability. The exposure factor, which ranges from 0 to 1, is interpreted as a percentage. In classical risk analysis terminology, the Single Loss Expectancy (SLE), the loss due to a single incident, is defined as the product of the Asset Value (AV) and the Exposure Factor (EF): $SLE = AV \times EF$.

IV. Technical Approach

In this section, we first illustrate how to assess the impact of multi-step attacks, and then present our graph-based approach to optimizing configurations using constraint satisfaction.

A. Impact Calculation for Multi-Step Attacks

We can compute the impact on a distributed system of multi-step attacks that are enabled under a given system configuration. Suppose that an attacker exploits vulnerability v_C in Fig. 6. This makes component h_C completely unavailable, as its exposure factor with respect to v_C is 1. As h_T strictly depends on h_C, h_T also becomes unavailable, leading to a marginal impact of $7+7=14$. Based on these observations, we define the *impact function* for a single attack step as

$$\text{impact}(v_j) = \sum_{h \in H} (s_{j-1}(h) - s_j(h)) \cdot u(h), \tag{4}$$

where $s_{j-1}(h)$ and $s_j(h)$ respectively denote the relative residual utility of component h before and after exploitation of v_j in an attack path $P = (v_1, \ldots, v_n)$, and $u(h)$ is the original utility of h. For a given attack step v_j, this impact function adds up the marginal losses for all the components affected (either directly or indirectly) by the exploitation of v_j. Therefore, the impact of exploiting v_j depends on what other vulnerabilities were exploited in previous attack steps and how they impacted the system. In fact, in a multi-step attack, the utility of each component may further decrease after each attack step. In practice, $s(h)$ can be defined as follows:

$$s_i(h) = \begin{cases} 1, \text{ if } i = 0 \\ f_h\left(s_i(h_1), \ldots, s_i(h_n)\right), \text{ otherwise} \end{cases} \tag{5}$$

where f_h is the dependency function associated with component h and $h_1, \ldots, h_n$ are the components that h depends on.

Our graph model provides non-obvious insights about security optimization. For instance, after exploiting v_C, the attacker may take one of two steps, exploiting either v_D with probability 0.7 or v_F with probability 0.3. Intuition suggests that, as the attacker is more likely to exploit v_D, that vulnerability should be preferentially patched or addressed before v_F. However, this approach turns out to be inefficient, as we now explain. The additional impact of exploiting v_D would be $0.7 \cdot 5 = 3.5$, as h_C and h_T are already unavailable because of the previous exploit. In comparison, the additional impact of exploiting v_F would be $0.7 \cdot 7+8+10 = 22.9$, as compromising h_F also makes h_A and h_S unavailable. This suggests that, even though the attacker is more likely to exploit v_D, the security benefit of addressing v_F is greater. Quantitatively, the impact of an adversary sequentially exploiting $v_1, \ldots, v_n$ in a path $P = (v_1, \ldots, v_n)$ in the vulnerability subgraph is:

$$\text{impact}(P) = \sum_{j=1}^{n} \sum_{h \in H} (s_{j-1}(h) - s_j(h)) \cdot u(h) \tag{6}$$

In this analysis, it is critical to compare attack paths and prioritize countermeasures. Our goal is to identify configuration changes that minimize the system's attack surface, by blocking high-impact attack paths. To achieve this goal, we define attack surface metrics that consider the likelihood and potential impact of each attack path, rather than simply counting the vulnerable entry points. A simple yet effective metric is:

$$\text{attack\_surface}(S) = \sum_{i=1}^{m} \text{impact}(P_i) \cdot \Pr(P_i) \tag{7}$$

where $P_1, \ldots, P_m$ are known attack paths, and impact(P_i) and Pr(P_i) are the impact and likelihood of P_i respectively.

Our impact calculation can be extended to assess the impact of multiple attacks executed concurrently. The worst case is one where, at each step, the attacker exploits, with probability 1, all vulnerabilities for which preconditions are satisfied. If $\langle V_1, \ldots, V_m \rangle$ is a topological sort of all the nodes in the attack graph, then the attack surface metric can be defined as

$$\text{attack\_surface}(S) = \sum_{j=1}^{m} \sum_{h \in H} (s_{j-1}(h) - s_j(h)) \cdot u(h) \tag{8}$$

In other words, Eq. 8 defines the attack surface as the potential impact of a multi-step attack in which all attack paths are pursued concurrently. Although unrealistic in practice, this scenario provides an upper bound on a system's susceptibility to attacks. A more realistic worst-case scenario would consider the relative complexity of exploiting different vulnerabilities, providing a trade-off between the two scenarios of Eqs. 7 and 8. However, intuition suggests that minimizing the attack surface as defined by Eq. 7 would (at least sub-optimally) minimize any other reasonable attack surface metrics.

B. Config Security as a Constraint Satisfaction Problem

SCIBORG aims to find configurations that minimize security impact while satisfying configuration constraints and preserving the functionality of the distributed system. These secure configurations are computed as follows: Without loss of generality, denote the ith configuration parameter as f_i and

TABLE II. INFORMATION INGESTED BY SCIBORG.

Data Items	Source	Used By
Configuration Parameters Meta Data (type, default value, required?, text description)	Specification sheets on manufacturer websites in machine-readable data formats including HTML, CSS, JSON, XML, or in natural language.	Modeling Framework (Configuration Subgraph)
Configuration Parameters Values	Configuration Files	Modeling Framework (Configuration Subgraph)
Available constraints on configuration parameters to ensure legitimacy of parameter values	Standard operating procedure and/or component documentation, in machine readable format, natural language, or from user input	Modeling Framework (Configuration Subgraph)
Available constraints on configuration parameters to ensure a functional system	Standard operating procedure for the distributed system, provided both in machine readable formats and natural language.	Modeling Framework (Configuration Subgraph)
Functional dependencies between system components	Entity in charge of the design and commissioning of the system.	Modeling Framework (Dependency Subgraph)
Known vulnerabilities in system components	National Vulnerability Database (NVD) bug reports	Modeling Framework (Vulnerability Subgraph)
Security best practices and bad practices	Domain experts in IoT security and represented in machine readable data formats or in natural language.	Modeling Framework (Vulnerability Subgraph)
Prioritization of security versus functionality	System administrators and operators.	Reasoning Framework

the entire configuration by $F = (f_1, f_2, \cdots, f_k)$. At a high level, we solve this constraint satisfaction problem (CSP):

Find configuration $F^* = (f_1^*, f_2^*, \ldots, f_k^*)$ such that:

1) Configuration subgraph constraints are satisfied
2) Dependency subgraph constraints are satisfied
3) $F^* = \arg\min_{F} \sum_{P \in A(F)} \text{impact}(P)$

where $P = (v_1, \ldots, v_n)$ is any path in the vulnerability subgraph $A(F)$ induced by the configuration F.

In SCIBORG, F^* is obtained using a Satisfiability Modulo Theory (SMT) solver. We will describe later how dependency subgraph constraints and configuration subgraph constraints are derived and provided as inputs to the solver. The solver also takes as input the initial system configuration F, which we can assume to correspond to parameter settings that put the system in a working state, although not optimal with respect to security or functionality. Our goal is to find a configuration that improves the security and/or functionality. During the course of solving the CSP, SCIBORG can encounter combinations of constraints that cannot be simultaneously satisfied. Some of the constraints therefore must be carefully relaxed. SCIBORG performs this relaxation step-by-step according to a predefined policy that balances security over functionality. For the example in Fig. 6, the solver determines that debug_mode must be set to false for both h_A and h_C.

V. SCIBORG DESIGN AND IMPLEMENTATION

Having described our technical approach, we now detail SCIBORG's implementation, which includes (a) ingesting information about the configuration, functionality and potential vulnerabilities of a distributed system; (b) using ingested information to construct the multi-layer graph model described in Section IV; (c) reasoning about the security and functionality of possible configurations using a theorem prover; and (d) generating evidence that certain configurations improve security-functionality tradeoffs, and guidance for constructing such configurations. SCIBORG's architecture is shown in Fig. 7.

A. Data Ingestion Framework

To construct a graph-based model, SCIBORG ingests the items listed in Table II from several information sources. Depending on the type of information, system component, and manufacturer or vendor, these items are available in different data formats, including XML, HTML/CSS, JSON, and natural language. Consequently, data ingestion is semi-automatic, with customized parsers for some components. For flexibility, SCIBORG allows advanced users to visually create ingestion data flows and comes equipped with ingestion mechanisms for

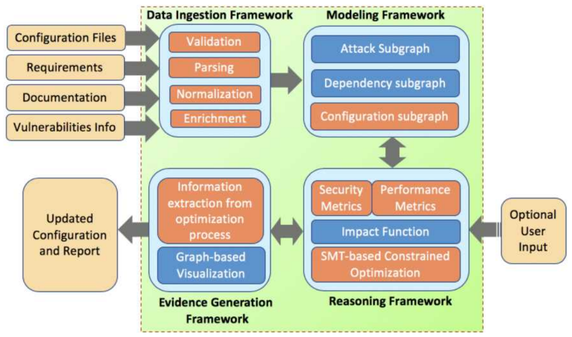


Fig. 7. Overall Architecture of Sciborg

components of interest (e.g., PFSense Firewall). These data flows are primarily implemented in Apache NiFi[8].

Ingesting configuration information. SCIBORG's data flows extract configuration information, including the names of parameters, data types, default values, current values if available, range of possible values, and free-form text descriptions.

Ingesting vulnerability information. SCIBORG distinguishes three types of vulnerabilities: *Type-1*, software vulnerabilities reported in vulnerability databases and identified by vulnerability scanners; *Type-2*, per-component bad security practices, currently specified by user input; and *Type-3*, not-best security practices per component, also currently specified by user input. For Type-1 vulnerabilities, we ingest relevant information from NVD, including CVE ID, various CVSS v2 and v3 scores, access complexity, CWE category, and natural language description. Additionally, we ingest information about the privileges that an attacker will gain by exploiting the vulnerability. This information, combined with access complexity, allows us to construct attack graphs in the downstream modeling framework. SCIBORG provides a pluggable interface that allows users, to define Type-2 and Type-3 vulnerabilities on a per-component basis. Examples of Type-2 and Type-3 vulnerabilities are provided in Tables III and IV.

Ingesting dependency information: Information about dependencies between components is extracted from two different sources in SCIBORG: (i) direct user input, similar to ingestion of Type-2 and Type-3 vulnerabilities, and (ii) third-party tools such as NSDMiner [11] for discovering service dependencies through traffic observation and call graph analysis. This information is used to construct the dependency subgraph in the downstream modeling framework.

Ingesting functionality requirements. SCIBORG distinguishes two classes of functionality requirements. The first

[8]https://nifi.apache.org/

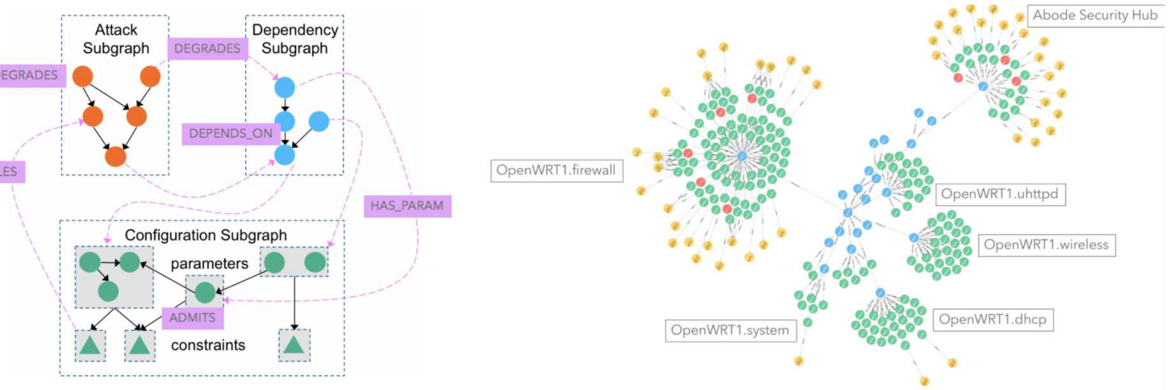


Fig. 8. (a) Semantics of the relationship among subgraphs, (b) partial view of the graphical model of the IoT testbed.

class is *parameter range constraints* specifying legitimate ranges of values that can be assigned to parameters. Such ranges are found by the configuration parameter information extraction described above. The second class is *functionality and performance requirements*, ingested from user input through an interface. SCIBORG models such requirements as constraints in the configuration space and allows users to specify them using ingested parameter names as variables. These constraints are specified in an SMTLIB-2.0[9] compliant manner for efficient downstream reasoning by the Reasoning Framework.

B. Modeling Framework

The SCIBORG Modeling Framework stores relationships between system components, configuration parameters, configuration predicates, and vulnerabilities in a queryable, graph-based form. It also provides an API to quantitatively evaluate the security of different system configurations using topological vulnerability analysis. The modeling framework is built on top of Neo4j[10] and converts all ingested information into a graphical format. The APIs providing security evaluation and configuration impact analysis are implemented as a Neo4j plugin that can (i) analyze attack scenarios (i.e., finite sequences of vulnerabilities that can be exploited by an attacker), (ii) compute various attack surface metrics, and (iii) asses the security impact of configuration changes. Fig. 8(a) illustrates the semantics of the relationships between the various subgraphs of our model. Fig. 8(b) shows part of the graph corresponding to our testbed.

C. Reasoning Framework

The SCIBORG Reasoning Framework computes a new configuration for the target system, across all components, that minimizes security risk while preserving functionality. It is written in Java with Microsoft Z3[11] as its solving engine. To reason in the configuration space, SCIBORG constructs a Constraint Satisfaction Problem (CSP) by querying the modeling framework described in Section V-B. The variables in the CSP correspond to unique names of nodes of the model representing configuration parameters. In addition, the CSP includes the following types of constraints derived from the modeling framework: (1) *CurrentConfig* constraints, i.e., predicates representing assignment of current values to system parameters, (2) *Functional* constraints that include predicates consisting of functionality requirements as discussed above, (3) *Security* constraints of two kinds: (a) *negation* of predicates that represent preconditions for Type-2 vulnerabilities (i.e., bad security practices) and (b) predicates that represent preconditions for enabling best security practices (i.e., prevent Type-3 vulnerabilities). Once the CSP problem is formulated, it is fed into Z3 to find a solution with values for each parameter.

SCIBORG assumes that the initial system configuration has been at least partially tested for functional and non-functional requirements, representing a reasonable starting point from which to find optimal configurations; hence it uses *CurrentConfig* constraints. In cases where the current configuration is sub-optimal or violates security or functionality constraints, SCIBORG makes necessary adjustments in the CSP, based on the desired reasoning strategy, as described below. The formulated CSP may not be satisfiable. In many cases, however, solvers can return the *unsatisfiable core* consisting of a set of clauses whose conjunction is still unsatisfiable. If the CSP is not satisfiable, we utilize unsatisfiable core information along with constraint type and constraint impact information (by querying the modeling framework) to form a new CSP, by dropping certain clauses from the unsatisfiable core of the previous CSP per SCIBORG's constraint relaxation strategy. This operation is done for a number n of rounds until the CSP is satisfied, the number of trials exceeds n, or the solver fails to produce both the unsatisfiable core and a solution.

Constraint Relaxation Strategy. Our Reasoning Framework can be configured to use one of three strategies in the reasoning process: (1) Prioritize Functionality, (2) Prioritize Security, and (3) No Priority. These strategies differ in the way constraint relaxation occurs in the event of unsatisfiability of a CSP formulated in a previous reasoning round. When prioritizing functionality, SCIBORG forms the new CSP by first removing constraints of type *CurrentConfig* that appear in the unsatisfiable core of the previous CSP. If the problem is still not satisfiable, it removes constraints of type *Security* with the smallest security impact. When prioritizing security, it forms the new CSP by first removing constraints of type *CurrentConfig* that appear in the unsatisfiable core of the previous CSP. If the problem is still not satisfiable, SCIBORG removes *Functional* constraints. Note that the recommended configuration found under this mode may violate functional requirements and therefore should not be used for deployment. However, it is useful in analysis and to further understand the

[9]http://smtlib.cs.uiowa.edu
[10]https://neo4j.com
[11]https://github.com/Z3Prover/z3/wiki

system requirements and their trade-offs with security. When operating in *No Priority* mode, SCIBORG removes constraints of type *CurrentConfig* that appear in the unsatisfiable core of the previous CSP. If the problem is still unsatisfiable, SCIBORG just reports the unsatisfiable core and exits.

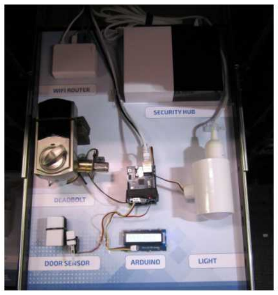
D. Evidence Generation Framework

The Evidence Generation Framework provides graph-based visualizations and human-readable text describing the optimality of the computed configuration. It collects reasoning artifacts, including unsatisfiable cores associated with each reasoning rounds, dropped clauses and their impact, description of vulnerabilities that have been addressed or are outstanding, and renders them in different formats (e.g., PDF).

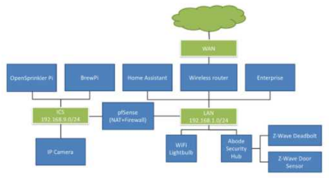
VI. SCIBORG Evaluation

A. SCIBORG Evaluation Testbed

To evaluate SCIBORG, we built a physical testbed, which includes IoT devices, a mock office Information Technology (IT) environment with virtualized servers and PCs, test harness software to sense and actuate IT and IoT components, and logging facilities. The test harness software runs predefined scenarios that actuate experimental-plane components and read their state to determine if the experimental system is still operating as required after configuration changes. The IoT components form two sets, a Consumer-IoT set, and an Industrial Control System-IoT (ICS-IoT) set. The IoT devices were selected to: (i) cover a breadth of configuration space elements, from minimally configurable (e.g., Wi-Fi light bulb) to extensively configurable (e.g., Wi-Fi router); (ii) range from highly dependent on an internet connection and cloud services (e.g., Abode Security Hub) to fully self-contained (e.g., BrewPi); (iii) span different physical connectivity methods (Wi-Fi, Ethernet, Z-Wave); (iv) include both direct physical effects (e.g., light, temperature, motion) and networked effects (e.g., network segmentation, internet connectivity); (v) consist of commercially-available and/or open-source off-the-shelf systems, for reproducibility. The office IT environment comprises computer systems and servers typically found in corporate IT networks, some of which may interact with IoT devices and be critical to their operation. They were chosen to represent a realistic IoT environment that interacts with and depends on traditional IT systems. The components are instrumented with out-of-band sensors to provide ground truth as to their current state (light on/off, door locked/unlocked, etc.). Thus, the actual state of the system can be ascertained throughout experiments, regardless of the reported state on the experiment plane (due to misconfiguration, attacks, etc.). Fig. 9(a) shows the instrumented Consumer-IoT components with sensors and actuators attached. The topology of the testbed is shown in Fig. 9(b). To use the testbed, the experimenter configures the devices according to an experiment plan and launches a *scenario* using the web interface. A scenario drives the test harness software through a series of steps and validates that the IoT devices and emulated IT systems are operating as expected. This allows us to test whether SCIBORG's recommended configuration changes break functional requirements.



(a)



(b)

Fig. 9. (a) IoT testbed and (b) testbed network topology.

TABLE III. Type-2 Vulnerabilities

Type-2 Vulnerability	**Testbed Component**
Enabling packet forwarding by default	OpenWRT Firewall
Allowing all protocols from WAN to LAN	OpenWRT Firewall
Enabling packet forwarding by default	OpenWRT Firewall
Disabling Tamper Siren	Security Hub
Setting alarm duration to 0	Security Hub
Silencing all gateway sounds	Security Hub
Using default or no password	Open Sprinkler
Bypassing password check	Open Sprinkler
Allowing App Installation from Unknown Sources	Tablet
Enabling DDNS for a normal IoT device	IP Camera
Disabling watermark	IP Camera
Having short watermark characters	IP Camera
Disabling Login Failure Monitoring	IP Camera
Disabling Network Disconnect Monitoring	IP Camera
Disabling IP Conflict Monitoring	IP Camera
Disabling anti-lockout rule	PFSense Firewall

B. Evaluating SCIBORG on Testbed

As discussed before, SCIBORG ingests information about testbed components, their configuration parameters, and system requirements from various sources. We ingest vulnerability information by running openVAS scanner from different network segments of the testbed and cross referencing the found components and vulnerabilities with NVD using our own tools. In addition, we ingest per-component bad security practices and best security practices from user input.

Overall, SCIBORG reasoned on 5,460 vulnerabilities, 1,188 configuration parameters, and 43 components (including subcomponents). On average, it took SCIBORG 3 minutes and 14 seconds to populate a graphical model based on ingested data using a MacBook Pro laptop and an additional 14 minutes and 4 seconds to compute an improved configuration by reasoning on it. SCIBORG executed 433 rounds of reasoning to come up with the new configuration for the entire system under its *Prioritize Functionality* reasoning strategy. In every unsatisfied reasoning round, SCIBORG reformulated the corresponding constraint satisfaction problem by evaluating the impact of the constraints of the unsatisfiable core for that round as described in Section V. The SCIBORG evidence generation framework summarizes the constraints that cause unsatisfiability as well as their impact, giving the user full visibility into the compositional security analysis.

As expected, the most challenging part of using SCIBORG is the ingestion process. We make this task easier by providing (1) customized ingestion tools for common components (e.g., pfSense Firewall), and (2) a reusable library of Apache NiFi data ingestion flow templates. While this is helpful, we plan to apply machine learning and language processing techniques to further automate ingestion.

TABLE IV. TYPE-3 VULNERABILITIES

Type-3 Vulnerability	Testbed Component
Having (a potentially unnecessary) firewall rule enabled (in a default deny setting)	PFSense Firewall
Disabling protection against DNS Rebinding attacks	PFSense Firewall
Disabling logging for successful log-in attempts	PFSense Firewall
Allowing login by-pass	PFSense Firewall
Disabling protection against HTTP REFERER attacks	PFSense Firewall
Not using HTTPS	PFSense Firewall
Disabling Logging	Open Sprinkler
Having High Login Failure Threshold	IP Camera
Disabling Illegal Access Email Alerts	IP Camera
Having Long Illegal Access Alarm RelayOut Delay	IP Camera
Disabling Illegal Access Alarm RelayOut	IP Camera
Disabling remotely locating the device	Tablet
Using no API password	Home Assistant

VII. RELATED WORK

The state of the art in configuration security focuses narrowly on the configuration parameters of individual system components and so lacks a principled approach to coping with the complex relationships among the configuration parameters in a complex composed system. As a consequence, most of the existing approaches for solving configuration errors cannot tackle errors involving cross-component dependencies [18], let alone address the security implications of such dependencies. Cross-component errors are common [10] and may result in service disruptions that are costly to identify and address. This issue becomes more critical for complex systems where independent teams develop each component. Malicious actors are likely to use such configuration dependencies, alongside system vulnerabilities, to create context-aware Advanced Persistent Threats (APTs). To address this issue, we model a composed system as a multi-graph that captures interdependencies and provides insight into security optimization. This model finds the optimal configuration that adequately reduces the attack surface while ensuring that configuration and functionality constraints are satisfied.

Assessing the impact of attacks on a system requires defining its attack surface [16]. An attack metric should accurately consider all attack paths by conducting an in-depth analysis of each path's entry and exit points, implicit and explicit interdependencies, and vulnerabilities [1]. The approach in [9] enumerates reachable elements, but, while showing that systems with fewer vulnerabilities are more secure, it does not account for the cascading effects of these elements on the system if compromised. Manadhata and Wing developed an entry/exit-point framework, which considers the methods, channels, and data items of a system, also known as resources. Intuitively, their work implies that a larger attack surface makes a system less secure. However, a larger attack surface is not necessarily more vulnerable than a smaller one [5]. Past literature narrowly measures attack surface either through an attacker-centric approach [14] or a system-centric approach [9]. However, using these limited approaches produces an incomplete view of the overall system security posture. To address these limitations, we assess a system's attack surface based on an in-depth analysis of the impact of each multi-step attack, considering the complex interdependencies among system components, vulnerabilities, and configuration parameters.

VIII. CONCLUSIONS AND FUTURE DIRECTIONS

To our knowledge, SCIBORG is the first system to address security misconfigurations in networked distributed systems. It builds a graph-based model that captures relationships among system vulnerabilities, configuration parameters, and system components by ingesting information from multiple sources (e.g., documentation, configuration files, vulnerability databases). Using this model, SCIBORG formulates a constraint satisfaction problem and solves it to improve the configuration.

Future plans include further automating the ingestion process and improving SMT solver performance. The automation effort will involve integrating with configuration management systems, investigating annotation of configuration files, and applying machine learning and natural language processing to extract additional relationships among system components, configuration parameters, and vulnerabilities.

REFERENCES

[1] M. Albanese and S. Jajodia. A graphical model to assess the impact of multi-step attacks. *Journal of Defense Modeling and Simulation*, 15(1):79–93, January 2018.

[2] M. Albanese, A. Pugliese, and V. Subrahmanian. Fast activity detection: Indexing for temporal stochastic automaton-based activity models. *IEEE TKDE*, 25(2):360–73, February 2013.

[3] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. In *Proc. of ACM CCS 2002*, pages 217–224, Washington, DC, USA, November 2002.

[4] P. Bahl, P. Barham, R. Black, R. Chandra, M. Goldszmidt, R. Isaacs, S. Kandula, L. Li, J. MacCormick, D. Maltz, R. Mortier, M. Wawrzoniak, and M. Zhang. Discovering dependencies for network management. In *Proc. of ACM HotNets-V*, Irvine, CA, USA, November 2006. ACM.

[5] S. M. Bellovin. Attack surfaces. *IEEE Security & Privacy*, 14(3):88–88, May 2016.

[6] S. Jajodia, S. Noel, and B. O'Berry. *Managing Cyber Threats: Issues, Approaches, and Challenges*, volume 5 of *Massive Computing*, chapter Topological Analysis of Network Attack Vulnerability. Springer, 2005.

[7] N. Kourtellis, G. De Francisci Morales, and F. Bonchi. Scalable online betweenness centrality in evolving graphs. *IEEE Transactions on Knowledge and Data Engineering*, 27(9):2494–2506, April 2015.

[8] D. J. Leversage and E. J. Byres. Estimating a system's mean time-to-compromise. *IEEE Security & Privacy*, 6(1):52–60, Jan./Feb. 2008.

[9] P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, May 2011.

[10] Matt-Welsh. What I wish systems researchers would work on. *Volatile and Decentralized*, May 2013.

[11] A. Natrajan, P. Ning, Y. Liu, S. Jajodia, and S. E. Hutchinson. NSDMiner: Automated discovery of network service dependencies. In *Proc. of IEEE INFOCOM 2012*, Orlando, FL, USA, 2012. IEEE.

[12] OWASP. Owasp top 10 - 2017: The ten most critical web application security risks. Technical report, The OWASP Foundation, 2017.

[13] B. G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979.

[14] B. Schneier. *Secrets and Lies: Digital Security in a Networked World*, chapter Attack Trees, pages 318–333. John Wiley & Sons, 2015.

[15] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Proc. of IEEE S&P 2002*, pages 273–284, Berkeley, CA, USA, May 2002.

[16] C. Theisena, N. Munaiahb, M. Al-Zyoudc, J. C. Carver, A. Meneely, and L. Williams. Attack surface definitions: A systematic literature review. *Information and Software Tech.*, 104:94–103, Dec. 2018.

[17] S. Venkatesan, M. Albanese, and S. Jajodia. Disrupting stealthy botnets through strategic placement of detectors. In *Proc. of IEEE CNS 2015*, pages 55–63, Florence, Italy, September 2015. IEEE.

[18] T. Xu and Y. Zhou. Systems approaches to tackling configuration errors: A survey. *ACM Computing Surveys*, 47(4), July 2015.