Employing Attack Graphs for Intrusion Detection

Frank Capobianco*, Rahul George*, Kaiming Huang*, Trent Jaeger*, Srikanth Krishnamurthy†, Zhiyun Qian†, Mathias Payer‡, Paul Yu◇

*Penn State University
†UC Riverside
‡Ecole Polytechnique Fédérale de Lausanne
◇U.S. Army Combat Capabilities Development Command Army Research Laboratory

ABSTRACT

Intrusion detection systems are a commonly deployed defense that examines network traffic, host operations, or both to detect attacks. However, more attacks bypass IDS defenses each year, and with the sophistication of attacks increasing as well, we must examine new perspectives for intrusion detection. Current intrusion detection systems focus on known attacks and/or vulnerabilities, limiting their ability to identify new attacks, and lack the visibility into all system components necessary to confirm attacks accurately, particularly programs. To change the landscape of intrusion detection, we propose that future IDSs track how attacks evolve across system layers by adapting the concept of *attack graphs*. Attack graphs were proposed to study how multi-stage attacks could be launched by exploiting known vulnerabilities. Instead of constructing attacks reactively, we propose to apply attack graphs *proactively* to detect sequences of events that fulfill the requirements for vulnerability exploitation. Using this insight, we examine how to generate modular attack graphs automatically that relate adversary accessibility for each component, called its *attack surface*, to flaws that provide adversaries with permissions that create threats, called *attack states*, and exploit operations from those threats, called *attack actions*. We evaluate the proposed approach by applying it to two case studies: (1) attacks on file retrieval, such as TOCTTOU attacks, and (2) attacks propagated among processes, such as attacks on Shellshock vulnerabilities. In these case studies, we demonstrate how to leverage existing tools to compute attack graphs automatically and assess the effectiveness of these tools for building complete attack graphs. While we identify some research areas, we also find several reasons why attack graphs can provide a valuable foundation for improving future intrusion detection systems.

KEYWORDS

Intrusion detection; attack graphs; attack surface

ACM Reference Format:
Frank Capobianco*, Rahul George*, Kaiming Huang*, Trent Jaeger*, Srikanth Krishnamurthy†, Zhiyun Qian†, Mathias Payer‡, Paul Yu◇. 2019. Employing Attack Graphs for Intrusion Detection. In *New Security Paradigms Workshop (NSPW '19), September 23–26, 2019, San Carlos, Costa Rica.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3368860.3368862

NSPW '19, September 23–26, 2019, San Carlos, Costa Rica

ACM ISBN 978-1-4503-7647-1/19/09…$15.00
https://doi.org/10.1145/3368860.3368862

1 INTRODUCTION

Network and host intrusion detection systems (NIDS/HIDS) are now commonly deployed to detect malicious activities, even restricting system operations in some cases (as intrusion prevention systems, IPSs). NIDS deployments, such as Bro [66], Snort [73] and Suricata [63], can collect packets at one or more network monitoring locations to detect network-borne attacks. On the other hand, current HIDS solutions, such as OSSEC [19] and Samhain [91], examine system operation logs for behaviors often associated with attacks, such as root logins and sensitive resource modifications.

Nonetheless, even with IDSs deployed widely, the number of intrusions continue to increase each year [29]. In addition, the sophistication of attacks also continues to increase, raising questions about the efficacy of continuing solely with the current approaches to IDSs [46]. One acknowledged weakness of IDSs is that they focus on known attacks and attack behaviors, which limits the ability to detect new attacks and attacks similar to benign behaviors. Modern attacks, such as so-called advanced persistent threats (APTs), leverage operations similar to benign behaviors to evade intrusion detection and anti-virus systems, as Stuxnet demonstrated several years ago [26]. Another limitation that we highlight is that NIDS and HIDS each lack visibility into the programs where vulnerabilities are often exploited. As a result, NIDS and HIDS must infer attacks based on communications and system calls, respectively, lacking critical information about threats and their exploitation.

IDS vendors and researchers have explored a variety of ways to improve IDS effectiveness, but they do not completely overcome the limitations above. First, to improve visibility, systems sometimes combine network and host IDS into a so-called *hybrid IDS*, such as SolarWinds [79] and Sagan [71], that integrate network and system monitoring to correlate log entries from both layers to improve attack detection confidence. While correlating network and host operations can improve attack detection, hybrid IDSs still focus on known attack behaviors and lack visibility into programs. Second, to reduce dependence on known attacks, systems have been proposed to perform vulnerability-focused detection as well as exploit-focused detection [20, 27, 59]. While this approach reduces dependence on known attacks, these approaches currently examine known vulnerabilities, limiting the ability to detect new attacks. Third, many systems now employ some form of AI and machine learning to detect attack behaviors from datasets of known attacks and benign operations [12, 14, 35, 42, 49, 77]. However, classification methods may not identify the features necessary to detect attack

scenarios that are similar to benign operations [80], demonstrated by the recent research area of adversarial machine learning [16, 30, 65], where adversaries actively disrupt training and/or find samples that have malicious behavior but will be classified as benign.

To overcome these limitations in current intrusion detection systems, we argue that future IDSs need to: (1) identify threats and possible attacks in a principled manner, rather than depend on known attacks and/or vulnerabilities and (2) correlate actions across the network, host, and program layers. Fortunately, researchers have already proposed a model for tracking how attacks may be perpetrated system-wide, called *attack graphs* [23, 64, 67, 75, 78], but researchers have not explored the potential of attack graphs to track intrusions proactively to detect new attacks across systems.

We acknowledge that attack graphs were not originally developed for intrusion detection, but rather for intrusion analysis: to understand how known vulnerabilities could be or were exploited as multi-stage attacks across systems. While researchers have started to explore using attack graphs for intrusion detection [60, 89], these initial approaches have only leveraged known vulnerabilities for network-level detection, so they suffer from the same limitations as prior work. Instead, we propose to leverage the common definition for "vulnerability" to guide the application of attack graphs. A *vulnerability* is often said to be a software flaw accessible to an adversary who can exploit that flaw [92]. Using this insight, we propose to construct attack graphs for proactive intrusion detection, where *attack states* correspond to the permissions made available to adversaries by software flaws as possible preconditions to attacks, *attack actions* correspond to the possible operations adversaries may use to leverage those permissions for exploits, and *attack surfaces* [36] correspond to the possible sources of adversary input that provide accessibility to flaws.

To employ attack graphs for proactive intrusion detection, we have to address several key limitations. First, traditional attack graphs were often constructed manually from known vulnerabilities and attacks. However, the emergence of a variety of security configurations, such as firewall policies, access control policies, and program defenses, provide a bounty of information to automate attack graph generation. In this paper, we explore how to apply such information to compute attack graphs automatically.

Second, traditionally attack graphs have been employed for network-level intrusion analysis, focusing on hosts with known vulnerabilities and their network connectivity to propagate attacks. To extend intrusion across network, host, and program layers, we need a way to connect flows between layers to understand how to correlate events. In this paper, we propose to utilize the concept of attack surfaces, which identify the sources of adversary accessibility, to express the sources of input from other layers.

Third, while attack graphs can express attack states and actions about network, host, and program resources, representing all possible states and actions in a single graph will be impractical. Instead, we need a way to construct and reason about attack graphs modularly. We find that a benefit of making the system layering explicit is that attack surfaces can serve as boundaries between modules, enabling attack graphs to be computed per system component [1]

[1]We use the term "component" broadly to refer to all types of network, host, and program entities in need of intrusion analysis.

Importantly, attack surfaces specify each component's "API" for adversary-controlled input, which identifies the starting points for each component's attack graphs.

In this paper, we explore the challenges of building attack graphs for performing intrusion detection. We define an intrusion detection approach based on attack graphs and identify a set of research problems that must be solved to complete the approach. While some problems may be solved with methods available today, others will require non-trivial research efforts. We identify opportunities to solve these research problems based on current research, and highlight outstanding issues to examine further. We then perform case studies to examine how the proposed intrusion detection approach could apply to two specific cases: (1) attacks on file retrieval, such as TOCTTOU attacks [13, 48], and (2) attacks propagated among processes, such as on Shellshock vulnerabilities [51]. Importantly, both types of attacks span multiple layers of a system, demonstrating requirements for attack graphs across layers.

This paper makes the following contributions.

- We derive requirements for intrusion detection systems from examples of complex vulnerabilities that span network, host, and program layers.
- We define a model of intrusion detection based on the *attack graph concept*, where *attack states* represent preconditions that enable adversaries to gain permissions to threaten components, *attack actions* represent operations that leverage those permissions to propagate attacks, and *attack surfaces* identify entrypoints for adversary-controlled input.
- We outline a vision for intrusion detection based on the attack graph model and investigate key research questions for automating attack graph generation for intrusion detection.
- We explore the application of the proposed approach using two case studies that involve attacks that combine network, host, and program behaviors to demonstrate how specific tasks in computing attack graphs may be accomplished to enable intrusion detection.

2 MOTIVATION

To motivate the requirements of future intrusion detection systems, we examine two types of vulnerabilities: (1) those resulting from the Shellshock vulnerability discovery [51] and those related to attacks on file retrieval including TOCTTOU attacks [13, 48]. These types of vulnerabilities present challenges for intrusion detection because they involve multiple systems and software components (Shellshock) and exploit permissions normally available to the victim (file retrieval). We then examine the lessons these challenges impart on intrusion detection systems.

2.1 Shellshock Vulnerability

Shellshock refers to a group of vulnerabilities in the Bash shell program that were caused mainly due to how inputs from environment variables passed from web servers were parsed. The Bash shell was first released in 1989 and has been used for a multitude of purposes. Bash introduced the export shell function definition feature in version 1.03 in 1989. This feature itself has been employed in a variety of ways, including for web servers and embedded systems. The Shellshock vulnerability was discovered in 2014, and characterized

primarily in the context of web servers. However, modern embedded systems continue to use the export shell functionality to parse and execute inputs, and several of these systems were not patched, providing the main attack vector for exploiting Shellshock [50].

Bash supports exporting shell function definitions to Bash instances via environment variables. Web servers and other programs utilize this functionality to specify new functions for execution by Bash in processing web requests. The Shellshock vulnerability was a result of incorrect parsing of these function definitions. The initial problem was that Bash continued to parse input even after the function definition ends. Hence, anything after the function definition was parsed and executed as well. This vulnerability enabled adversaries to direct Bash to execute any functionality that the invoking principal (e.g., web server) could execute, exposing a large number of options for adversaries to propagate attacks.

Shellshock is exploitable because adversaries may make network requests that both update environment variables and trigger the execution of Bash shells. Web servers used such functionality commonly to run scripts to process web requests. Embedded systems that run Bash also often propagate network request input to Bash shells. With the proliferation of Internet of Things devices, opportunities for exploitation of embedded devices is greater since not all have been patched.

Discovery of the Shellshock vulnerability resulted in a series of patches to fix the vulnerability completely and to address other vulnerabilities that were discovered as a result of the study into the initial Shellshock vulnerability. The initial vulnerability (CVE-2014-6271) was caused by the parsing error described above, but the initial patch (Patch 25 for Bash 4.3-25) did not address related flaws in parsing (CVE-2014-6278 and CVE-2014-7169). In addition, several memory errors were discovered in Bash processing related to this parsing that were also discovered and patched (CVE-2014-6277, CVE-2014-7186, and CVE-2014-7187).

The patch history related to the vulnerabilities is interesting because the third patch in the sequence (Patch 27, Bash 4.3-27) aimed to address the parsing problems systematically by clearly demarcating the environment variables that may be used for function definitions and the bounds of function definitions for parsing. However, errors remained in parsing (CVE-2014-6278) that were not addressed correctly in this patch, so they were fixed in later patches (Patch 30, Bash 4.3-30).

2.2 File Retrieval Attacks

Programs may also be exploited when accessing files by adversaries who control either the input used to build file pathnames or the filesystem resources used to resolve those file pathnames to specific files. Most famous are so-called *time-of-check-to-time-of-use* (TOCCTOU) attacks that exploit races between file system operations of the adversary and victim [13, 48], but several types of attacks do not require race conditions, being more direct *confused deputy* attacks [32].

Researchers have reported that approximately 10% of reported CVEs are vulnerabilities of this type [88], and this problem continues to plague current systems, such as Android. As with Shellshock, these attacks may span network, host, and program layers, receiving untrusted network input and falling victim to host-program attack surfaces by exploiting program flaws. We examine attacks against the Apache web server program, leveraging some prior research in this analysis [86] that, like Shellshock, found exploitable flaws in web servers for file serving.

There are two distinct attack actions that may be leveraged independently or in concert to enable an adversary to gain unauthorized access to host resources.

First, an adversary could provide input used to build file pathnames that is not properly filtered by the program to prevent unauthorized access. For example, a programmer may not expect adversary input at a particular program attack surface associated with constructing file pathnames (i.e., an unexpected attack surface [86]), so an adversary can choose the file pathname. In addition, a programmer may recognize a threat from an attack surface, but fail to filter the input from this attack surface properly, permitting an adversary to gain access to unauthorized files.

Second, an adversary may control a program running on the host already and leverage that program's permissions to modify directories to direct pathname resolution to a file of the adversary's choosing [13, 48]. Given write access to a directory, an adversary can create and delete files in that directory. Simply by planting a file used by the victim, an adversary may provide malicious input to the victim if it trusts that file. In addition, an adversary may plant a symbolic link to redirect the pathname resolution to retrieve a file chosen by the adversary. Files accessed via symbolic links are authorized using the victim's identity, so if an adversary can trick a victim with the permissions necessary to access a file referenced by the planted symbolic link then the victim may mistakenly operate on a file unauthorized for the adversary on behalf of the adversary.

Both of these attacks are difficult to prevent. Incorrect filtering in programs remains an open problem. Researchers have explored program analysis techniques to detect missing filters [9], but even the task of identifying filtering code in programs is non-trivial [45], making attacks from network input possible. Researchers have also found that programmers often miss attack surfaces made available in deployments, causing vulnerabilities because no filtering is present at all [86]. On the other hand, attacks on pathname resolution have been known since the 1970s [48] and attacks due to race conditions, called time-of-check-to-time-of-use (TOCTTOU) vulnerabilities, were made precise in the early 1990s [13]. Thus, attacks on programs that exploit the access control configurations of hosts are also possible. Many solutions have been proposed to fix programs and provide protection mechanisms in operating systems, but researchers have demonstrated many attacks to circumvent such protections, leading Cai et al. [15] to identify that any effective defense requires knowledge of programmer intent, meaning that vulnerabilities often remain latent in programs because such intent is not explicit.

2.3 Lessons for Intrusion Detection

We find that the Shellshock and file retrieval attacks demonstrate the following lessons for the future of intrusion detection.

First, exploiting these vulnerabilities involves tracking data flows at the network, host, and program layers of the system. Both types of attacks may originate from network input, such as web server requests. Second, flaws may exploit host resources. Shellshock payloads are propagated from the web server to Bash shells through the setting of environment variables. On the other hand, file retrieval

attacks may be possible because another process on the host may attack the victim (e.g., web server). Third, ultimately, exploits are possible because of flaws in the Bash shell and victim programs, where the parsing errors, memory errors, and/or errors in checking the results of system calls may be exploited to further adversary control. Finally, exploits of Shellshock result in operations on host resources once again, such as executing other programs.

Next, exploitation of such vulnerabilities are difficult to prevent without concrete information about the vulnerabilities. From the perspective of a NIDS, only the web requests are seen, and the variants of requests that may occur make it difficult to identify known malicious requests that would include Shellshock or file retrieval payloads. For example, web requests were able to set environment variables for several years using input, such as `HTTP_ACCEPT='{ :;}; rm -rf'`, which would remove files. From the perspective of a HIDS, the web server will commonly pass information to Bash shells via environment variables and open files, and Bash shells commonly spawn other processes, making it difficult to identify malicious requests from a compromised Bash shell. A hybrid IDS could correlate risky web requests with host operations, but such IDSs currently require knowledge of a specific vulnerability or attack, enabling adversaries to circumvent checks before the specific attack was discovered.

In addition, the ability to detect either of these vulnerabilities would benefit significantly from visibility into program execution. Critical operations in exploiting either vulnerability occur inside the victim programs (e.g., web server and Bash shell programs). Recently, several developer organizations have decided to expend vast resources on software testing, such as fuzz testing, to detect flaws within programs more quickly. However, many flaws are still missed in testing or only unveiled in particular, unusual deployments, and the results of fuzz testing are not employed to guide IDS detection of such situations, leaving blind spots in testing as blind spots in detection. In addition, the relationship between flaws (e.g., crashes) and the threats (e.g., adversary access to new permissions) that such flaws enable is not yet well understood.

Both improving the effectiveness of detection and improving visibility of programs to detect intrusions will require more knowledge about how attacks may proceed and how programs link host and network actions, which in turn will require new automated methods to compute such knowledge. At present, knowledge of attack methods is either based on known attacks or attack behaviors (e.g., writing executable files), and IDS vendors aim to leverage such information for effective use in detection. Generating IDS data from known attacks can be automated, but methods to predict new attack cases and methods are limited to coarse attack strategies, such as *kill chains* [38].

Finally, expanding IDS to include program state will challenge the scalability of detection methods. Since this knowledge appears necessary to make significant improvements, we must explore methods to extend intrusion detection in a scalable manner. Current hybrid IDS techniques of combining system operations and network operations into a common format will be insufficient to serve as a basis for managing scalability.

In summary, we find that the following broad requirements apply to future intrusion detection systems.

- **Relate to Attack Principles**: Reason about components and their defenses in a principled manner to detect intrusions.
- **Increase Visibility**: Represent attack behaviors for network, host, and program layers.
- **Automate Intrusion Models**: Identify possible attack operations automatically.
- **Improve Scalability**: Perform detection over richer, more complete attack models efficiently.

In the rest of this paper, we will explore how the attack graph concept may be employed to achieve these requirements, and we will identify the challenges in applying attack graphs to achieve those requirements.

3 ATTACK GRAPHS

In this section, we review the evolution of attack graph approaches, including their motivations and research challenges, and then provide a definition for the attack graph model we will use in this paper. We will then review some of the challenges in applying attack graphs to IDS that have not been addressed by prior work.

3.1 Prior Attack Graphs Approaches

Work on representing attacks in terms of attack graphs emerged in the mid-1990s. The problem was to develop techniques to enable administrators to understand how vulnerabilities in their system could enable adversaries to escalate their permissions through a sequence of operations to complete an attack.

A wide variety of research has been undertaken on attack graphs. The origin of attack graphs appears to be a paper by Dacier and Deswarte [22] that models permission transformations in an access control system (the Typed Access Matrix) to detect violations of the *safety problem* [33]. While permission changes in an access control system for a variety of reasons, Dacier et al. [23] extended this approach to reason about the relationship among permission states created by vulnerabilities, which they still called a permission graph, but which started to capture the notions in attack graphs. In particular, Dacier et al. wanted to aid administrators in computing the permission escalation enabled through vulnerabilities, so the edges in their permission graph represented vulnerabilities and administrators could compute sequences of permission escalations possible given a sequence of vulnerability exploitations.

Phillips and Swiler developed an attack graph model for networks in 1998 [67]. This model incorporates key elements of the attack graph, relating network flows, possible sets of attacker permissions, and attack specifications, including post-conditions. This attack graph model connects network flows and the adversary permissions obtained to possible attacks. Interestingly, Phillips and Swiler advocated generating network flows from configurations directly, although adversary permissions and attack specifications were determined in an ad hoc way from known vulnerabilities.

Researchers then explored methods to compute attack scenarios from attack graphs. Sheyner et al. developed a model checking approach to compute attacks [78]. Sheyner's approach computes paths to attack goal states in the attack graph. Ammann made the critical insight that we can reason about adversaries as being monotonic (i.e., they do not release permissions once obtained) to reduce the number of meaningful attack paths to consider through

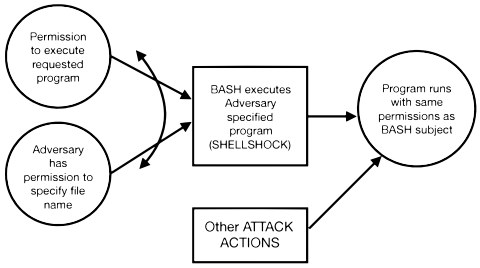

Figure 1: Attack graph example for Shellshock showing how *attack actions* (rectangles) depend on conjunction of *attack states* (circles) and *attack states* are created by any one of a disjunction of *attack actions*.

an attack graph [4]. Ou et al. [64] further optimized the representation to make the attack graph independent of hosts, only relating pre-conditions (i.e., permissions) and attacks (i.e., vulnerabilities to exploit), which they found is a special case of Schneier's attack trees [75]. We adopt a version of the attack graph model of Ou et al., which we define below.

Noel and Jajodia explored a variety of uses for attack graphs, including vulnerability analysis using attack graphs [41, 62], network risk analysis and hardening [2, 61], intrusion detection configuration and analysis [60, 89], and detection of new instances of known vulnerabilities [3]. Typically, these approaches leverage known vulnerabilities to identify hosts that may be exploitable. Graphs leverage network flows for permissions to access hosts with vulnerabilities and such vulnerabilities may be exploited if reachable. While we examine the connection between attack graphs and intrusion detection in this line of work further below, we would like to produce attack graphs without known vulnerabilities.

3.2 Attack Graph Model

In this paper, we adopt the attack graph definition of the Ou et al. [64] paper[2].

Definition 3.1. An *attack graph*, $G = (V, E)$, is a graph where: (1) the set of vertices, $V = S \cup A$, where S is the set of *attack states* (called *facts* in Ou et al. [64]) and A is the set of *attack actions* (called *derivations* in Ou et al. [64]) and (2) the set of edges, $E = R \cup P$, where $(u, v) \in R$ is a *precondition edge* when $u \in S$ and $v \in A$ and $(u, v) \in P$ is a *postcondition edge* when $u \in A$ and $v \in S$.

Since there are multiple ways of causing an attack state to become true, the set of attack actions directly preceding an attack state in an attack graph form a disjunction. On the other hand, in order for an attack action to be executed, all the necessary attack states must be achieved forming a conjunction.

Shown in Figure 1, consider that the attack action of exploiting Shellshock by an adversary choosing a program to execute requires that the Bash shell subject (likely the same as the web server subject) has the file system permissions to execute the requested program and that the adversary has the program permission to specify the file name argument of the execute function (e.g., `execve` or `system`). Thus, this attack action requires two distinct attack states, one in the program layer and one in the host layer, to both be satisfied.

On the other hand, once the program is executed it obtains all the permissions associated with that execution, based on the subject under which it is run. However, there are multiple ways of obtaining those permissions (i.e., through "Other ATTACK ACTIONS") , meaning that an adversary can obtain permissions from a disjunction of attack actions.

Ou et al. [64] represent attack states as facts and attack actions as rules (derivations) in Datalog. In that paper, facts referred to permissions available to adversaries and rules referred to methods of exploitation of known vulnerabilities given those permissions. We envision using a Datalog representation for our attack graphs.

3.3 Challenges Using Attack Graphs in IDSs

There are several challenges in applying attack graphs to intrusion detection in a manner that can satisfy the requirements listed in Section 2.3.

First, traditional attack graph configuration requires non-trivial manual effort. Phillips and Swiler [67] propose using network configurations to automatically generate network flows and Noel and Jajodia [41] leverage the Nessus vulnerability scanner to identify possible vulnerabilities to exploit automatically. However, many other aspects of attack graphs must be determined and configured manually. For example, identifying attack states (permissions) necessary to exploit a vulnerability were determined manually. In addition, the resultant attack states after exploiting a vulnerability were specified manually. Finally, exploiting a vulnerability is often more complex than simply having connectivity, so the details of attack actions themselves must be specified more precisely. Thus, automation remains a significant challenge.

Second, while recent IDSs explore utilizing known vulnerabilities along with known attacks to detect attacks, researchers commonly find use of known information insufficient to detect advanced attacks. Much research has focused on artificial intelligence and machine learning attacks to classify behaviors more accurately [12, 14, 35, 42, 49, 77]. However, a challenge is to identify the system features that identify attacks effectively without overfitting, and thus not improving significantly on detection using known attacks. One approach would be to leverage attack graphs to generate classifiers, but traditional attack graphs work from known vulnerabilities/exploits as well limiting their effectiveness to extend IDS knowledge. Using attack graphs for intrusion detection will require definitions of attack states and attack actions that reflect the principles necessary to detect attacks in modern systems.

Third, challenges arise in correlating intrusion detection and attack graphs. For example, a proposed intrusion detection approach [89] compares the knowledge encoded in an attack graph with the intrusion alerts seen so far to identify inconsistencies between the attack graph and the IDS alerts, where such inconsistencies may imply events missed by the IDS. However, such inconsistencies emerge partly because attack graphs and intrusion detection are configured independently. A question is whether the generation of attack graphs could be an input to configuring intrusion detection to avoid such inconsistencies.

[2]We change the terms in the Ou et al. definition for facts (to *attack states*) and derivations (to *attack actions*) to associate the attack graph with proactively tracking threats that may lead to exploitation.

Finally, while scalability has been examined by Ammann [4], Ou et al. [64], and Noel and Jajodia [61] among others, the study of scalability of attack graphs is still quite limited. In addition, intrusion detection aims to be performed at runtime, so we have a limited budget to devote to attack graph analysis (and upkeep). Attack graph approaches are often agnostic about how they may be modularized to enable better scalability. Prior research has often limited the application domain (e.g., just to network flows) to create tractable attack graphs, but we need approaches that encompass network, host, and program layers systematically.

Provenance vs. Attack Graphs. Given these challenges, an obvious question is whether the attack graph concept should even be considered for improving intrusion detection. Another technique that has historically been applied to forensic (offline) analysis of security is *provenance* [11, 44, 69]. Han et al. [31] explore the opportunities and challenges in utilizing provenance at runtime for intrusion detection. Provenance graphs record host operations (e.g., system calls) and their dependencies forming a more complete and accurate representation of system runtime behaviors. However, unlike attack graphs, provenance graphs are not directly related to attack states (threats and their permissions) and actions (potential exploits). Without this knowledge, intrusion detection may only improve slightly (e.g., by having a more complete picture of runtime operations). This knowledge could conceivably be added to provenance graphs, but one needs to encode such knowledge in some form. We explore encoding such knowledge in attack graphs, where provenance techniques could be configured based on attack graphs to enable detection at runtime or via forensic analysis.

Challenges Summary. Based on these findings, we propose to apply attack graphs to intrusion detection to leverage their ability to model attacks more accurately. However, to do so, we will examine how to overcome the challenges above, as summarized here.

- **Generate Attack Graphs Automatically**: Compute attack states and attack actions from available information (e.g., configurations).
- **Without Known Attacks/Vulnerabilities**: Detect attacks from the principles of vulnerability exploitation, rather than known attacks/vulnerabilities.
- **To Direct Intrusion Detection**: Determine intrusion detection operations from the attack graph.
- **Account for Scalability**: Budget resources for intrusion detection based on attack graphs.

We do not claim to know acceptable answers to all of the challenges at present, but in the next two sections we present a vision for intrusion detection using attack graphs and explore the problems, opportunities, and open issues for achieving that vision.

4 INTRUSION DETECTION VISION

Our vision for intrusion detection using attack graphs is based on the following insight: that attack graphs naturally correlate to the two aspects of misuse intrusion detection (1) claiming permissions that create threats and (2) using threatening permissions to perform operations that obtain access to additional permissions. Attack states imply preconditions that grant subjects with permissions that violate some security property, such as information flow, that enables a threat to the system. Pieters similarly proposes reasoning explicitly about *access claims*, where adversaries may claim use permissions to launch attacks [68], similar to attack states here. In addition, attack actions are operations adversaries can employ given those permissions to launch attacks. Thus, attack graphs represent sequences of threats and operations that enable possible misuses that may be composed by adversaries into full attacks.

This implies that we can leverage the ability to compute threats and exploits to construct attack graphs, which can then be employed to detect intrusions at runtime. There are several known techniques for computing flaws of various types that grant threatening permissions. In this paper, we will examine how benign components allow memory errors and information flow errors that may be exploited. We further note that researchers have explored computing flaws at the network, host, and program layers. Although less mature, researchers have also proposed methods to generate exploits automatically, for both memory errors and information flow errors. In the next section, we discuss research challenges in computing attack states and attack action and combining these two concepts to construct effective attack graphs.

To enable continuing research and development of methods to compute attack graphs, we propose that IDSs should be built as a framework that supplies a suite of methods for constructing attack graphs and applying them to intrusion detection to enable new methods to be deployed in a plug-and-play manner. That is, building and utilizing attack graphs involves a sequence of decisions, and researchers can plug solutions into the framework to improve the effectiveness of decisions made for attack graph generation and intrusion detection incrementally.

A critical challenge will be managing the scalability of using attack graphs. One key insight that we will aim to leverage is that by predicting the attack surfaces of each component we can compute attack graphs independently for each component, improving the scalability of (offline) attack graph computation and (online) use. An *attack surface* of a component [36] identifies the entrypoints that may receive input controlled by an adversary. Attack surfaces then describe all the sources of adversary-controlled inputs, so attack graphs must originate only at the attack surface entrypoints. Thus, attack graphs can be constructed locally from the attack surface entrypoints for each component. A challenge is that even attack graphs for each component may be sufficiently complex as to present scalability challenges, particularly to runtime monitoring. One possible side-benefit may be that once an attack graph is computed, defenders may proactively remove flaws to reduce attack graphs for their components. However, if component attack graphs are too large, techniques will be necessary to prioritize the parts of the attack graph to utilize, such as via machine learning.

To implement intrusion detection from our proposed attack graphs, the basic approach would be to use the attack graph to instrument system components to monitor each component for attack states being achieved and attack actions being performed. For each state, the IDS would record the associated permissions obtained. For each action, the IDS would record the successful completion of an operation associated with exploitation behaviors implied by the action. For both the permissions obtained and exploit behaviors completed, an IDS has options about whether to track worst-case, probabilistic, and/or best-case scenarios for each to produce *detection hypotheses* about adversary control. Detection hypotheses

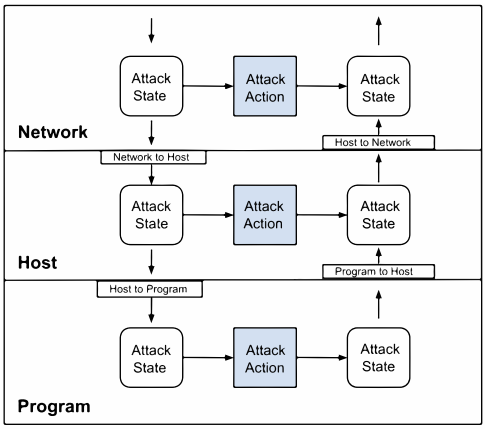

Figure 2: Attack graph vision for intrusion detection, where individual network, host, and program components have independent attack graphs relating attack states and attack actions generated based on that component's attack surface.

are similar to attack states in that they describe permissions under adversary control, but unlike attack states, which we envision will imply best-case permissions (for adversaries), hypotheses may be generated using other thresholds. At runtime, hypotheses relative to a single component could be computed using local attack graphs. Researchers have examined techniques to estimate attack probabilities for attack graphs by reasoning about them as Bayesian networks [53]. However, to use such a technique, a challenge is to relate attack states and attack actions to probabilities.

5 RESEARCH PROBLEMS

In this section, we examine the main research problems to be solved to generate attack graphs for intrusion detection. Figure 2 shows an attack graph example, showing the network, host, and program layers. This attack graph shows a simplified system with only one host and only one program. Each network, host, and program component has its own attack graph rooted at its attack surface. Note that the attack surfaces provide the link between layers. For example, the host-to-program attack surface represents how programs may use host resources or network resources through the host, such as sockets, to receive adversary-controlled input. Attack graphs show how each component uses its attack surface input to create threats that give the adversary permissions as attack states to launch attacks against the component via attack actions. Finally, attack graphs show that attack states may enable adversaries to have permissions to propagate attacks to attack surfaces of other layers as output, such as the program-to-host attack surface. Below, we examine the research problem in computing attack surfaces, attack states, and attack actions for network, host, and program components, and in using attack graphs for intrusion detection.

5.1 Computing Attack Surfaces

The first step is to identify the possible sources of attacks for each component, which we propose to do by computing each component's attack surface. Figure 2 shows the locations of attack surfaces, which connect the network, host, and program layers. Importantly, the host layer faces adversary-controlled input from both the network and program layers.

As entrypoints imply communication from outside the component, they naturally identify the possible inter-layer sources. If components only receive adversary-controlled inputs from attack surface entrypoints, then all adversary-controlled inputs can be identified from those entrypoint sources, independent of other data flows. We leverage this assumption for computing attack graphs below. In this section, we examine the research problems, opportunities, and issues in estimating component attack surfaces.

The Problem. Computing attack surfaces requires solving three main challenges: (1) identifying entrypoints of each component; (2) identifying adversaries of the component that may launch attacks; and (3) determining the entrypoints for each component that are accessible to any of that component's adversaries. First, what constitutes an entrypoint depends on the component. For programs, entrypoints typically correspond to system calls, although, to distinguish among system calls of the same type researchers designate the library calls that cause system calls (e.g., to `libc`) as entrypoints. For networks, untrusted remote hosts or networks are entrypoints [47, 87]. For hosts, resources updated from untrusted parties, such as network sockets and files of unknown provenance, form attack surfaces. Second, attacks originate from adversaries of each component, so we need a method to predict such adversaries. Third, we need to connect a component's adversaries to its entrypoints to determine which of the component's entrypoints may actually be accessible to adversaries.

Each of these three challenges may present ambiguities that prevent exact solutions. For example, library calls are a relatively reliable source of external input for programs (i.e., entrypoints), but programs may use specialized techniques, such as memory-mapped files, to create other entrypoints. Finding references to memory-mapped files in programs is intractable in general, as it requires solving the aliasing problem. Second, identifying adversaries depends on the threat model for the particular component, which may vary depending on the context. Third, identifying which entrypoints are accessible to adversaries depends on how components are used in practice. Such use will depend on the configuration of the component's runtime environment. As a result, we have to make choices about whether to overapproximate or underapproximate attack surfaces. Ideally for security, we would overapproximate attack surfaces for each component, so we would never miss an attack, but the complexity of systems means that we will likely generate many false positives. On the other hand, underapproximating attack surfaces means we are actually tracking true attack surfaces, although we have to develop strategies for handling new attack surfaces when they are discovered.

Opportunities. Researchers have explored a variety of methods to identify program attack surfaces, including measurement of the importance of data accessible from the entrypoint [47], the stack traces associated with crashes [84], resources accessible to untrusted subjects based on the access control policy [87], and where programmers filter inputs to limit program inputs [86]. These disparate techniques typically underapproximate the attack surface because they depend on runtime results, but the resultant attack

surfaces nonetheless have been found to correlate well with vulnerabilities [84, 87], even detecting new vulnerabilities. However, several of these methods do not offer techniques for identifying adversaries nor their accessibility to specific entrypoints, limiting the value of these techniques to flaw identification rather than intrusion detection.

Researchers often set threat models for their experiments, but there have not been many research efforts that propose methods for computing adversaries and the threats they present. One such method is the *Integrity Wall* approach [87]. This approach proposes a conservative threat model that aims to maximize the number of adversaries of a program. In this model, a program only trusts those programs that run under the same domain, have the permissions to modify the program's executable file, or can modify critical kernel data (e.g., physical memory). In general, subjects with such permissions can trivially compromise the program, so they must be trusted for the program to operate. Researchers applied this technique to SELinux systems, finding that even with this large set of adversaries per program: (1) program attack surfaces were only a small fraction of the number of entry points and (2) reported vulnerabilities could be associated with a large fraction of the identified attack surface. We examine the application of the Integrity Wall approach for computing adversaries in one of the case studies in Section 6.1.

Since it is difficult to predict the resources that may be accessed by individual library calls, if they are computed dynamically, it is often difficult to predict which library calls may access adversary-controlled resources, implying that they are part of the program's attack surface. Vijayakumar et al. [86] found that if a program lacks any filtering for an entrypoint, then the program assumes that no adversary-control input will be received. Thus, such an input can be blocked to prevent an *unexpected attack surface* that would be vulnerable. If the program filters input at that entrypoint, then that entrypoint may be a legitimate attack surface, and IDS will have to examine the impact of its use, lest risking a false positive. Thus, one can apply a method that detects the filtering to detect legitimate attack surfaces.

We also need methods to compute host and network attack surfaces. Fortunately, researchers have explored attack surfaces from networks to hosts [36, 47, 52], as this was the original point of the attack surface metric. These works enumerate the various entrypoints that make the system accessible from the network, as most network entrypoints enable untrusted access. In addition, researchers identified that host resources of unknown origin may be adversary-controlled sources [52], and hence part of the host attack surface as well.

Issues. An often overlooked issue is the computation of the program-host attack surface shown in Figure 2. The problem is that programs may propagate adversary-controlled input that they receive to system resources, which may propagate attacks to other processes. This is part of the cause of the Shellshock vulnerability, where a web server transfers adversary-controlled input to Bash through environment variables. While we can compute whether information written to a host resource is tainted by adversary input in a variety of ways, these methods have historically had severe limitations. Dynamic taint tracking is accurate, but expensive, whereas sound static analysis often greatly overapproximate tainted flows. However, recent work has shown that applying static analysis judiciously can reduce the overhead of dynamic taint analysis significantly [10] (below 10%).

However, dynamic analysis is inherently incomplete, so we may underapproximate attack surfaces. Thus, we need to address the challenge where new attack surfaces may be discovered at runtime. For the program-host attack surface, this creates the possibility of a mutual dependence between the host and program attack surfaces, where program flows may extend host attack surface which in turn likely extends one or more program attack surfaces, and so on. In general, program and host attack surfaces must be computed until a fixed point is reached. However, many programs do not expect untrusted input propagated by programs they trust (e.g., lack filtering), so these so-called *unexpected attack surfaces* [86] should halt propagation. The IDS must effectively choose between blocking input and tracking threats.

5.2 Computing Attack States

Given the attack surfaces, we aim to build attack graphs in two stages. In this first stage, we compute attack states from security property violations, because such violations often introduce new permissions to adversaries. The aim is to identify operations that may enable adversaries to leverage such permissions to launch attacks (i.e., attack actions) that are accessible to adversary-controlled input.

Security properties restrict an operation to only utilize permissions that comply with security requirements, so a violation indicates a flaw that may permit an adversary to exploit the component. For example, an adversary request may cause a program to access a database containing some information to which an adversary is not authorized or to utilize a key value that must be kept secret from adversaries. Such permissions are not allowed to the requestor, so there use by the database may present a problem, if exploitable, for the adversary to utilize those permissions in unforeseen ways.

The Problem. The problem of computing attack states from security properties involves the following challenges: (1) determining the security properties to employ; (2) identifying where components violate those security properties; and (3) estimating the permissions obtained by the security violation.

In contrast to attack surfaces and attack actions, we aim for attack states to overapproximate the threat posed to identify the possible permissions that may be obtained. Thus, we employ security properties to compute the maximal permissions that may be made available to adversaries to create threats to exploit the component.

Opportunities. We identify two broad types of properties we consider for identifying new attack states that may grant new permissions to adversaries at present. Other security properties may be considered without modifying the basic approach. The main requirement for a security property is that it be broadly applicable, so it can be applied automatically.

We observe that adversaries commonly obtain permissions by: (1) abusing weaknesses in access control enforcement and (2) stealing permissions by hijacking program executions. First, access control enforcement is often weak because subjects are granted permissions that may enable compromise. For example, we have long known

that secrecy and integrity of a computer system can be protected by enforcing an information flow policy [24, 55]. However, in practice information flow is often too restrictive to permit desirable, if risky, functionality, such as allowing a process holding secrets to reply to a process that is not authorized for such secrets. Such an operation is risky because it may enable the secret values to be leaked. Thus, information flow is a security property that identifies attack states where adversaries may gain unauthorized access.

Information flow violations may occur at the program, host, and network layers. Researchers have developed several methods for detecting information flow errors in program [24, 54–56] and in host [17, 18, 34, 40, 74, 95] as well as in host and network combined [82, 83].

In addition to information flow violations, a variety of attacks on programs involve hijacking program executions through exploitation of memory errors. In languages that are not type safe, memory references may not point to memory associated with the referenced resource leading to unauthorized reads and writes. Prior work classified pointers into safe, sequential, and wild categories [58], where safe pointers must satisfy type safety, but sequential and wild pointers may not. Bounds checking techniques have been developed to enforce buffer bounds [57] to ensure that sequential pointers comply with memory bounds restrictions. While the performance overhead for the initial bounds checking techniques was high, subsequent methods based on fat pointers [25] have much better performance and hardware support for fat pointers is being explored [90].

Issues. A challenge is to predict the permissions implied by a security property violation for each attack state, and manage the accumulation of those permissions across attack states. In traditional attack graphs [62, 64, 78], the attack states are specific to each vulnerability, so the permissions (or facts in Ou et al. [64]) implied are also ad hoc (i.e., determined manually). One alternative is to arrange permissions in a lattice, as proposed for decentralized information flow control [43, 96], to enable transitions to be more systematic as the adversary obtains more access. This would apply well for properties that are commonly represented as lattices, such as information flow. Other properties could be represented in separate lattices. However, security properties related to memory errors are typically not represented in such a form. However, current exploit generation techniques assume broad access to memory [39]. Thus, the management of permissions is an open research issue for memory security properties.

Support for checking restrictions on wild pointers currently remains an open research issue. Wild pointers may reference arbitrary memory -locations, such as pointers to a set of objects on the heap. Prior work identifying wild pointers [58] applied ad hoc checks to limit their access. While they estimated that wild pointers were relatively uncommon (around 1% of pointers), that still created significant manual effort. The development of sanitizers to check for memory errors of various kinds using fuzz testing indicates that precise checks for such memory errors can be done with compiler support, but sanitizer-based detection can have high runtime overhead, 3X or more [81]. While this overhead may be acceptable for fuzz testing, it will not be acceptable in normal runtime use, so monitoring for violations for wild pointers will require lighter-weight sanitizers that are still reasonably accurate.

5.3 Computing Attack Actions

To complete the attack graph, we must compute attack actions, identifying exploits that lead to new attack states. As attack states represent operations where adversaries obtain additional permissions due to security property violations, the question is how adversaries may take advantage of such permissions to gain more permissions.

Once a security property is violated, this opens opportunities for adversaries to launch attacks depending on the permissions available from the violation and the permissions they already possess. The goal in this step is to determine the possible attack actions leveraging those permissions that may enable the acquisition of additional permissions to further attacks.

The Problem. The problem of computing attack actions from threatening permissions is to identify the relevant operations that an adversary may want to employ to propagate an attack (e.g., use permissions to access unauthorized data and/or increase adversary permissions) and determine whether those operations are possible within the component defenses employed.

Similar to attack surfaces, we aim to compute attack actions that are likely to be possible to identify misuses of security property violations. Thus, we aim to identify malicious behaviors relative to the permissions gained and others that have been gained previously.

Opportunities. We find that this problem is similar to the problem of exploit generation. Exploit generation methods leverage known vulnerabilities for computing attacks. Early systems to produce exploits from memory errors automatically, such as AEG [6], also assumed that no defenses were in place. However, recent systems like Data-oriented Programming [37] and Block-oriented Programming [39] (BOP) assume that only attacks that comply with program control flows will be permitted by defenses. Most techniques still assume that the vulnerability provides the adversary with full permissions over memory access, referred to as an *arbitrary write primitive* (AWP), but recent work also explores determining the permissions available to launch exploits for use-after-free vulnerabilities [93]. These exploit generation tools then generate a single exploit path, if possible. Most tools search for a specific, known attack, although BOP enables specification of a set of attacks in the form of exploit programs that are mapped to the victim code.

While these approaches have previously been employed to compute exploits for known crashes and/or vulnerabilities, another insight is that they may be used to study the impact of security property violations in attack states. Fundamentally, BOP only depends on the assumption that certain permissions may be available to an adversary (e.g., arbitrary write primitive to write to any memory location) to implement a particular exploit program. Thus, one can leverage BOP to determine whether exploit programs corresponding to attack actions are possible given particular attack states by assuming the availability of permissions from the attack state. Currently, BOP assumes arbitrary read and write access, so we will need to explore the impact of more limited permissions.

We envision that a similar approach can be taken to assess information flow violations as well. When information flow violations occur, we assume that adversaries can access unauthorized data, so the question is what actions are available to the adversary using that data. We can also apply exploit generation techniques like BOP

to determine whether a particular attack is possible given access to that unauthorized data. Since BOP follows legal control flows and no further memory errors are necessary to generate such an exploit, a technique like BOP could also be adapted to exploit information flow errors. We explore using BOP for generating attack actions for both memory errors and information flow errors related to Shellshock case studies in Section 6.

Issues. One issue with using current exploit generation techniques is that they only find one exploit and declare victory. For building attack graphs, we want to have as comprehensive a view of the attacks given the available permissions as possible. However, as we see in the case study, non-trivial changes will need to be made to the exploit generation methods to build attack graphs.

Another issue is that exploit generation techniques focus just on the last mile of the attack, rather than all of the steps in the kill chain [38]. On the other hand, exploit generation could benefit from more focused techniques for individual steps of the kill chain as well.

Another issue is that exploit generation typically assumes one set of permissions for all cases. For example, BOPC assumes the presence of an arbitrary memory write primitive and control-flow integrity [1] defenses. However, a particular security violation may impart a more or less restricted set of permissions. Also, adversaries may accumulate permissions over time from reaching other attack states.

5.4 Intrusion Detection with Attack Graphs

In considering the potential for leveraging attack graphs to improve intrusion detection, we must examine the impact of the proposed approach in the context of the base-rate fallacy [7, 8]. The base-rate fallacy occurs because people often fail to account for the basic rate of incidence of an event in assessing the frequency of producing false positives. Thus, even if a detector is very accurate (e.g., has a 99% detection rate), the frequency of non-intrusion events evaluated may lead to a high false-positive rate in practice.

We envision that our proposed intrusion detection approach based on attack graphs mitigates the base-rate fallacy in two ways. First, all intrusion detections will be based on both an attack state and its subsequent attack action. While violating a security property of an attack state may be fairly common, and thus prone to false positives, we will never identify an intrusion from an attack state alone. We also require evidence of abuse of that attack state in the form of an attack action. Should all attack actions be examples of true misuses, the proposed approach should not incur false positives because attack actions are examples of true intrusive behavior. However, if we utilize techniques that do not identify only true misuses, some false positives are possible.

Second, in this proposed approach, intrusions will be detected based on a sequence of state-action pairs (i.e., corresponding to pairs of attack states and attack actions). The aim is that the probability of such a sequence of state-action pairs implying an intrusive action rather than a benign action should be sufficiently high to prevent non-intrusive actions from dominating the detection process. In terms of Bayes Theorem as described by Axelsson [7], the value $P(I)$ in equation 10 should be much greater than shown in the presented example because the likelihood of that trace of state-action pairs being emblematic of the basic rate of incidence should be quite low (i.e., $P(I) >> P(\not I)$ rather than the other way around).

One main challenge is to convert state-action events into estimates of intrusion. Prior research in attack graphs associates vulnerabilities and attacks by computing probabilities [53, 70, 94] or tracking correlations [72, 85]. To compute an overall attack probability from attack graphs, researchers encode the attack graphs as Bayesian Networks. A challenge then is to determine the individual probabilities, and researchers often employ available knowledge of vulnerabilities, such the Common Vulnerability Scoring System [21] (CVSS). However, such estimates are not available for the security violations and generated exploits that we employ in this attack graph model. An alternative is to reason about correlations between states and subsequent actions and/or among state-action pairs themselves. However, the correlations proposed thus far are heuristics that may themselves be error-prone. While events that are not commonly correlated may indicate anomalies, we are more interested in looking for sequences of state-action pairs that correlate as misuses (attacks).

Another challenge that we envision for the proposed approach is that it aims to detect intrusions in a stateful manner. To detect attacks in terms of sequences of state-action pairs, the IDS must collect sequences of state-action pairs and track trigger rules based on those sequences. Collecting such sequences at runtime could incur non-trivial overhead, so judging how to use the attack graphs wisely for monitoring is an open challenge. An opportunity is to integrate detection with the component processing to detect intrusions locally, perhaps at low cost. However, in general, intrusions may span multiple components, so efficient and effective ways to produce summaries for (more) centralized analysis are necessary. Research has explored summarization methods with low loss of information for network IDSs [5].

6 CASE STUDIES

In this section, we examine case studies for the two examples presented in Section 2, file retrieval and Shellshock attacks. We present the file retrieval case study first, as the Shellshock case study builds on that, particularly for computing attack surfaces.

6.1 Detecting Attacks on File Retrieval

We also examine the application of the proposed attack graph approach to detect attacks on file retrieval. Details on this vulnerability and its variants are provided in Section 2.2.

Computing Attack Surfaces. To compute attack surfaces for Apache, we collect its relevant entrypoints, identify its adversary subjects, and compute which relevant Apache entrypoints may use resources accessible to those adversaries. As is typical, any entrypoint that may receive adversary-controlled input by reading an adversary-controlled resource is part of Apache's attack surface.

To compute Apache's adversaries, researchers leveraged the Integrity Wall adversary model [87] that conservatively identifies adversaries (i.e., maximizes adversaries) by only trusting the subjects that Apache must necessarily trust. That is, each program only trusts subjects that could modify critical kernel resources or the code files run by Apache or its trusted subjects, based on the host's mandatory access control policy (e.g., SELinux [76]). All other subjects were considered to be adversaries of the program.

Given the set of program adversaries, computing the entrypoints in the attack surface was done through a runtime analysis using test suites provided with Linux packages. By running the test suites and collecting the file accesses at relevant entrypoints, the researchers could determine whether a program's entrypoint ever accessed a file that could be written by a subject considered to be an Apache adversary. This attack surface includes entrypoints that use directories in pathname resolution that may be modified by adversaries. Notably, even with this conservative model only five attack surface entrypoints were identified in the prior work (see Figure 10 in Vijayakumar et al. [86]).

This approach to computing attack surfaces did not include program flows that propagate adversary-controlled data to modify host resources (i.e., the program-to-host attack surface). For example, the researchers identified that log files should be considered as adversary-controlled [87], even though only the program itself could write to the log files. Instead of using the program data flows to detect this propagation of data, the researchers labeled these files manually. Research is needed to automate this task.

Computing Attack States. In this case, there are three security properties. In the first security property, the file name arguments used in library calls that perform name resolution (e.g., `open`) must not depend on adversary-controlled input. In the second security property, if the first security property is violated, the output of a name resolution (i.e., file referenced by the returned file descriptor) must be authorized to be accessed by Apache adversaries. In the third security property, if the first security property is not violated, then the output of a name resolution must be protected from access by Apache adversaries. Some name resolutions comply with all three security properties, e.g., if they use a hard-coded file name guaranteed to return a file protected from adversary access. However, several programs fail at least the first property and many others also fail the second or third property as well, which implies a successful attack [86, 88].

The researchers identified the library calls that may violate one or more of these security properties using a runtime analysis that collected all the information flows from adversary-controlled inputs to file pathname arguments in library calls that perform name resolution. They further identified whether some form of filtering restricted how the input could be used to build file pathnames on those information flows to differentiate whether the second or third security property was relevant.

Computing Attack Actions. Attack actions in this case are operations to modify file retrieval to gain access to unauthorized data or trick Apache into using an adversary-controlled file. Two broad classes of attack actions are possible: (1) supply input used to build file pathnames chosen by the adversary and (2) change the filesystem to redirect name resolution. A challenge is to find attack actions that would be successful given the adversary permissions to control file names and modify the filesystem namespace.

Researchers defined a runtime method to test programs for name resolution vulnerabilities using a kernel-based system that deterministically changes the filesystem for an attack whenever an adversary of the program under test has modify permissions to a directory used in name resolution [88]. In this case, the types of attack actions that are possible were identified a priori. However, rather than performing intrusion detection, this prior approach performed testing as if it were an adversary.

Instead, an IDS could apply these attack actions to detect when either the second or third security property is being violated at runtime. If so, then prior work raised an alert [86], although some false positives were reported. Thus, we may want to track the program further to determine if Apache would be compromised by processing that file. Other researchers have explored techniques to compute the response of the operating system necessary to direct programs to the chosen operations [28] (e.g., leak the file contents). We could leverage such a technique to compute subsequent attack actions following the malicious actions on name resolution.

6.2 Detecting Shellshock Attacks

Now we examine the application of the proposed attack graph approach to detect attacks on the Shellshock vulnerability by exploiting web requests. Details on this vulnerability and its variants are provided in Section 2.1.

Computing Attack Surfaces. Computing attack surfaces introduces challenges because the web server propagates adversary-controlled data to host resources, expanding the host attack surface for itself and the Bash shell.

Starting with the attack surface in the web server that receives web requests, as described for file retrieval above, we compute resources that may be tainted by attack surface input. First, we compute the exit points tainted by data flows from the web request entry point that is accessible to adversaries within the web server, including the exit points that set the environment variable. As we are looking to estimate an underapproximation of attack surfaces, we perform a static taint analysis without aliasing. This detects that the environment variables passed to Bash (a host resource) reaches a program-to-host attack surface. In general, any program that can be started by a web server can be victimized by these environment variables, expanding their attack surfaces. With traditional UNIX access controls, a web server is authorized to run many programs, but using MAC enforcement the programs that may be executed can be limited, thus limiting the impact of the new attack surface. However, environment variables are not specifically managed by current access control mechanisms, either DAC or MAC, so host data flow analyses [17, 18, 34, 40, 74, 95] will need to be expanded to represent such information.

In addition, the discovery of the new program-to-host attack surface may expand the attack surface of other programs, such as Bash in this case, which may cause yet further attack surface expansion. In general, one can envision this expansion continuing until a fixed point is reached. However, in the case of Shellshock, the expansion of the Bash attack surface enables an adversary to dictate program execution, which is not a permission that Bash intends. Thus, the attack surface should not be expanded for the execute call, since Bash assumes that the executor of the shell trusts the input provided to that shell. This is an example of an unexpected attack surface [86], which should terminate propagation. However, Shellshock patches introduced filtering for such input, basically certifying the use of environment variables as a legitimate attack surface.

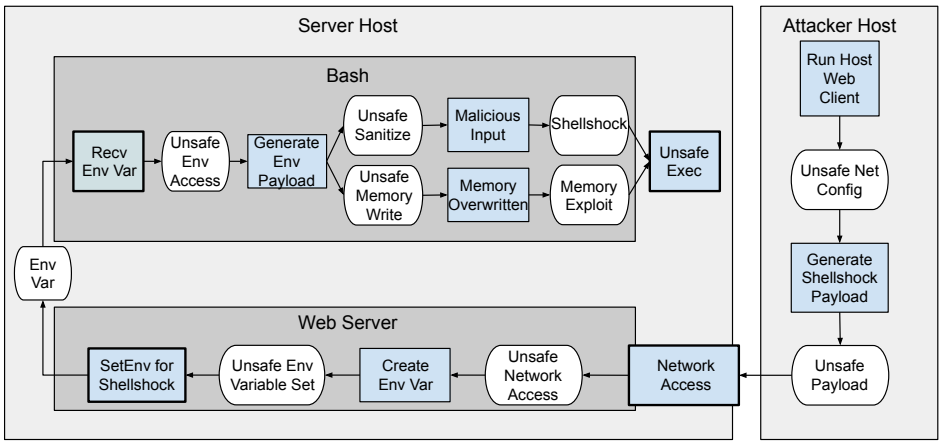

Figure 3: Full Shellshock Attack Graph: Attack states are in white and attack actions are in blue. This attack graph includes both the traditional Shellshock vulnerability as well as a memory error vulnerability discovered later.

Computing Attack States. With regard to Shellshock, multiple security property violations occur when unfiltered adversary-controlled input is passed to the program entry point in Bash. The original violation was that the adversary can control critical arguments to execute functions, which can be detected as an information flow violation. Using taint analysis, we can detect such errors. In addition, other Shellshock vulnerabilities enabled violation of memory errors. At present, detecting such errors accurately is difficult. However, identifying risky pointer operations, as proposed in Ccured [58] is practical. We plan to explore utilizing such a method.

The next step is to estimate the permissions made available to adversaries due to security property violations. For information flow, a question is how much control an adversary may obtain if they can provide part of the input for the file name submitted to the execute function. As in the previous case study, control of a file name used in a name resolution gives the adversary access to any file accessible to the principal, the web server.

While patching Bash, one of the early patches was incomplete, leaving a memory error that could be exploited by adversaries through the same mechanism as Shellshock. In this case, developers introduced new memory violations and for a brief period of time, adversaries could remotely execute arbitrary code and deny service before a subsequent patch was released. As discussed above in Section 5.3, current exploit generation tools assume that memory errors grant an arbitrary write primitive (AWP) to the adversary for attack actions [39]. However, in the future a more accurate estimate of memory accessibility will be necessary to prevent creating many unattainable attack actions.

Computing Attack Actions. We leveraged block-oriented programming (BOP) as implemented by the BOPC tool [39] in an effort to identify possible attack actions from vulnerabilities due to the receipt of adversary-controlled environment variables. The relevant part of the Bash program's call graph for the BOPC analysis is shown in Figure 4. Function calls before `parse_and_execute` are omitted since they are not related to the vulnerability. `parse_command` perform the parsing of commands including environment variables, and would return to `parse_and_execute` if parsing is not needed or finished. BOPC aims to find a single vulnerability, preferring the shortest path to success. We define an SPL program (BOPC's programming language) to run `execve` with a program of our choice in Figure 5.

The complete attack graph for Shellshock is shown in Figure 3. Note that we used BOPC to study from the "Generate Env Payload" step. The remaining attack graph was produced manually at present.

Since Bash is an interpreter, the most direct way to execute a program is to specify it in the script input. Since BOPC is looking for the shortest, context-sensitive attack path, it finds this path instead of the more complex path using the environment variables. We tried to start BOPC in other places in the call graph, even to exploit the memory errors instead, but received the same result. The reason is that BOPC will only compute the shortest-path solution to exploit the target binary. Since the function that processes environment variable input, `yyparse`, will return to `parse_and_execute` when it finishes, BOPC regards directly modifying the argument of `execve` as the optimal solution. The exploit path then is from `parse_and_execute` to `execve` circled in red.

A challenge from this analysis is that exploit generation tools are not yet designed to look for multiple attack actions, but rather to find one proof-of-concept exploit. We found that directing BOPC to look for multiple attack actions would require an extension to collect multiple edges between functional steps in its representation of the search (called a delta graph in the BOPC paper [39]).

7 DISCUSSION

We now review the intrusion detection requirements set forth in Section 2.3.

Relate to Attack Principles. The aim of this requirement is to connect the acquisition of threatening permissions (attack states) to adversary operations to leverage those permissions to perform attacks (attack actions) in a systematic way to detect intrusions.

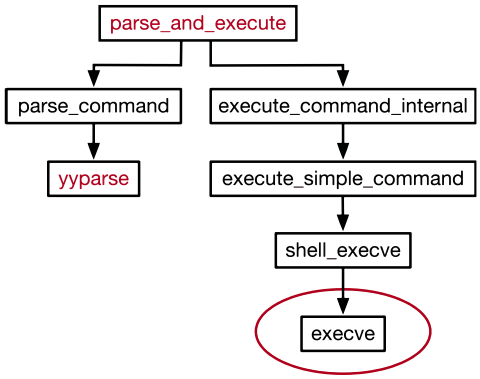

Figure 4: Call graph for Shellshock exploits in Bash

We propose to satisfy this requirement by defining attack graphs in terms of attack states and attack actions that directly correlate to these corresponding intrusion detection concepts. We propose using security property violations, which imply the acquisition of threatening permissions, for attack states and exploit operations, which imply exploitation of those permissions for attack actions. There are challenges in computing security properties and exploits, which require further research, but the basic approach relates attack graphs and intrusion detection.

Increase Visibility. The aim of this requirement is to compute security violations and exploit operations for network, host, and program layers. In this paper, we focused primarily on attack graphs for host and program layers, but prior work has demonstrated connections between host and network flows suitable for computing security property violations for the information flow. We have shown that the attack graphs can span and connect network, host, and program layers through attack surfaces and per-component attack graphs.

Automate Intrusion Models. The aim of this requirement is to compute attack states and attack actions in attack graphs automatically. We show that some types of security properties are general and methods exist to compute them. Exploit generation methods also exist for automating exploit computation, but more such methods will be necessary. Computing modular attack graphs require computing attack surfaces automatically.

Improve Scalability. The aim of this requirement is to enable scalable intrusion detection. The proposed approach in this paper utilizes attack surfaces to improve scalability for computing attack graphs (offline) and performing intrusion detection (online). First, computing attack surfaces enables us to compute attack graphs of components independently. However, the computation of attack surface itself may be complicated by the propagation of adversary-controlled data between programs, hosts, and the network. Next, although we proposed to compute attack graphs for components locally, which improves scalability, such attack graphs may still be complex. Researchers will need to explore techniques to generalize

```
1   void payload() {
2       string prog = "/bin/sh\0";
3       int64 *argv = {&prog, 0x0};
4
5       __r0 = &prog;
6       __r1 = &argv;
7       __r2 = 0;
8
9       execve(__r0, __r1, __r2);
10  }
```

Figure 5: Example SPL program for building attack actions for `execve`.

attack states and actions to simplify attack graphs. Finally, the resultant IDS system will leverage a trace of state-action events to detect intrusions. While we envision that some intrusions may be detected locally using the component's own attack graph, some correlation among the components may be necessary by a centralized IDS.

8 CONCLUSIONS

In this paper, we argue that current intrusion detection systems are designed to either defend against previously known attacks or have limited visibility of all layers of a system preventing them from defending against new attacks or multi-stage attacks that traverse a system's network, host, and program layers. We define a new approach to intrusion detection systems by leveraging attack graphs and showing that attack graphs naturally correspond to the intrusion detection steps of acquiring threatening permissions and leveraging them in operations to perform exploits. Then, we examined our approach in two case studies, to show how this approach applies to file retrieval attacks and the well-known Shellshock exploit and discuss limitations and additional challenges that need to be researched to deploy such an approach effectively.

9 ACKNOWLEDGMENTS

Thanks to our shepherd, Cormac Herley, and the anonymous reviewers. This research was sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA) and National Science Foundation grants CNS-1801534 and CNS-1801601. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

REFERENCES

[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity: Principles, Implementations and Applications. In *Proceedings of the 12^{th} ACM Conference on Computer and Communications Security (CCS 2005).*

[2] Massimiliano Albanese, Sushil Jajodia, and Steven Noel. 2012. Time-efficient and Cost-effective Network Hardening Using Attack Graphs. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).*

[3] Massimiliano Albanese, Sushil Jajodia, Anoop Singhal, and Lingyu Wang. 2013. An Efficient Approach to Assessing the Risk of Zero-Day Vulnerabilities. In *Proceedings of the International Conference on Security and Cryptography.*

[4] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. 2002. Scalable, Graph-based Network Vulnerability Analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002).*

[5] Azeem Aqil, Karim Khalil, Ahmed O.F. Atya, Evangelos E. Papalexakis, Srikanth V. Krishnamurthy, Trent Jaeger, K.K. Ramakrishnan, Paul Yu, and Ananthram Swami. 2017. Towards Network Intrusion Detection at ISP Scale. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT).*

[6] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. Automatic Exploit Generation. In *Proceedings of the 2011 Network and Distributed Systems Symposium.*

[7] Stefan Axelsson. 1999. The Base-rate Fallacy and Its Implications for the Difficulty of Intrusion Detection. In *Proceedings of the Second International Workshop on Recent Advances in Intrusion Detection.*

[8] Stefan Axelsson. 1999. The Base-rate Fallacy and Its Implications for the Difficulty of Intrusion Detection. In *Proceedings of the 6th ACM Conference on Computer and Communications Security.*

[9] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2008. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy.*

[10] Subarno Banerjee, David Devecsery, Peter M Chen, and Satish Narayanasamy. 2019. Iodine: Fast Dynamic Taint Tracking Using Rollback-free Optimistic Hybrid Analysis. In *2019 IEEE Symposium on Security and Privacy (SP).* IEEE.

[11] Adam Bates, Dave (Jing) Tian, Kevin R.B. Butler, and Thomas Moyer. 2015. Trustworthy Whole-system Provenance for the Linux Kernel. In *Proceedings of the 24th USENIX Security Symposium.*

[12] Anaël Beaugnon, Pierre Chifflier, and Francis Bach. 2017. ILAB: An Interactive Labelling Strategy for Intrusion Detection. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses.*

[13] Matt Bishop and Michael Digler. 1996. Checking for Race Conditions in File Accesses. *Computer Systems* 9, 2 (Spring 1996).

[14] Ahmet Salih Buyukkayhan, Alina Oprea, Zhou Li, and William K. Robertson. 2017. Lens on the Endpoint: Hunting for Malicious Software Through Endpoint Data Analysis. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses.*

[15] Xiang Cai, Yuwei Gui, and Rob Johnson. 2009. Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In *Proceedings of the 30th IEEE Symposium on Security and Privacy.*

[16] Nicholas Carlini and David A. Wagner. 2016. Towards Evaluating the Robustness of Neural Networks. *CoRR* abs/1608.04644 (2016). arXiv:1608.04644 http://arxiv.org/abs/1608.04644

[17] Haining Chen, Ninghui Li, William Enck, Yousra Aafer, and Xiangyu Zhang. 2017. Analysis of SEAndroid Policies: Combining MAC and DAC in Android. In *Proceedings of the 33rd Annual Computer Security Applications Conference.*

[18] Hong Chen, Ninghui Li, and Ziqing Mao. 2009. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *Proceedings of the 2009 Network and Distributed Systems Symposium.*

[19] Daniel B. Cid. 2008. *OSSEC.* http://www.ossec.net/

[20] Cisco. 2015. *Vulnerability-Focused Threat Detection: Protect Against the Unknown.* https://www.cisco.com/c/en/us/products/collateral/security/ips-4200-series-sensors/white_paper_c11-470178.html

[21] CVSS 2019. Common Vulnerability Scoring System SIG. https://www.first.org/cvss/.

[22] Marc Dacier and Yves Deswarte. 1994. Privilege Graph: An Extension to the Typed Access Matrix Model. In *Proceedings of the 1994 European Symposium on Research in Computer Security*, Dieter Gollmann (Ed.).

[23] Marc Dacier, Yves Deswarte, and Mohamed Kaâniche. 1996. Models and Tools for Quantitative Assessment of Operational Security. In *Information Systems Security: Facing the Information Society of the 21st Century.* Springer US, Boston, MA, 177–186.

[24] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (1976), 236–242.

[25] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *Proceedings of the 2017 Network and Distributed Systems Symposium.*

[26] Nicolas Falliere, Liam O Murchu, and Eric Chien. 2011. Symantec Security Response, W32.Stuxnet Dossier, Version 1.4. (Feb. 2011).

[27] Amer Farroukh, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2011. Towards Vulnerability-based Intrusion Detection with Event Processing. In *Proceedings of the 5th ACM International Conference on Distributed Event-based Systems.* ACM.

[28] Ioannis Gasparis, Zhiyun Qian, Chengyu Song, and Srikanth V. Krishnamurthy. 2017. Detecting Android Root Exploits by Learning from Root Providers. In *Proceedings of the 26th USENIX Security Symposium.*

[29] Pierce Gibbs. 2017. Intrusion Detection Evasion Techniques and Case Studies. STI Graduate Student Research in SANS. https://www.sans.org/reading-room/whitepapers/detection/paper/37527

[30] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015).*

[31] Xueyuan Han, Thomas Pasquier, and Margo Seltzer. 2018. Provenance-based Intrusion Detection: Opportunities and Challenges. In *Proceedings of the 10th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2018).*

[32] Norman Hardy. 1988. The Confused Deputy. *Operating Systems Review* 22, 4 (Oct. 1988), 36–38.

[33] Michael Harrison, Walter Ruzzo, and Jeffrey D. Ullman. 1976. Protection in Operating Systems. *Commun. ACM* 19 (Aug. 1976). Issue 8.

[34] Boniface Hicks, Sandra Rueda, Luke St. Clair, Trent Jaeger, and Patrick McDaniel. 2010. A Logical Specification and Analysis for SELinux MLS Policy. *ACM Transactions on Information and System Security (TISSEC)* 13 (July 2010).

[35] Hanan Hindy, David Brosset, Ethan Bayne, Amar Seeam, Christos Tachtatzis, Robert C. Atkinson, and Xavier J. A. Bellekens. 2018. A Taxonomy and Survey of Intrusion Detection System Design Techniques, Network Threats and Datasets. *CoRR* abs/1806.03517 (2018).

[36] Michael Howard, Jon Pincus, and Jeannette Wing. 2003. Measuring Relative Attack Surfaces. In *Proceedings of Workshop on Advanced Developments in Software and Systems Security.*

[37] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy.*

[38] Eric M Hutchins, Michael J Cloppert, and Rohan M Amin. 2011. Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains. *Leading Issues in Information Warfare & Security Research* 1, 1 (2011), 80.

[39] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS 2018).*

[40] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. 2003. Analyzing Integrity Protection in the SELinux Example Policy. In *Proceedings of the 12th USENIX Security Symposium.*

[41] Sushil Jajodia, Steven Noel, and Brian O'Berry. 2005. *Topological Analysis of Network Attack Vulnerability.* Springer US, 247–266.

[42] Amin Kharraz and Engin Kirda. 2017. Redemption: Real-Time Protection Against Ransomware at End-Hosts. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses.*

[43] Maxwell N. Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information Flow Control for Standard OS Abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007).*

[44] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of the 20th Network and Distributed System Security Symposium, NDSS 2013.*

[45] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: Specification Inference for Explicit Information Flow Problems. *SIGPLAN Notices* 44, 6 (June 2009), 75–86.

[46] Druce MacFarlane. 2018. Threat Detection: ItâĂŹs About 'TimeâĂŹ. https://www.csoonline.com/article/3304252/threat-detection-it-s-about-time.html

[47] Pratyusa K. Manadhata and Jeannette M. Wing. 2011. An Attack Surface Metric. *IEEE Transactions on Software Engineering* 37, 3 (2011), 371–386.

[48] William S. McPhee. 1974. Operating system integrity in OS/VS2. *IBM Systems Journal* 13 (September 1974), 230–252. Issue 3.

[49] Shagufta Mehnaz, Anand Mudgerikar, and Elisa Bertino. 2018. RWGuard: A Real-Time Detection System Against Cryptographic Ransomware. In *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions, and Defenses.*

[50] Trend Micro. 2014. *Summary of Shellshock-Related Stories and Materials.* https://blog.trendmicro.com/trendlabs-security-intelligence/summary-of-shellshock-related-stories-and-materials/

[51] MITRE. 2014. *Shellshock CVE 6271.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271

[52] Divya Muthukumaran, Sandra Rueda, Nirupama Talele, Hayawardh Vijayakumar, Jason Teutsch, Trent Jaeger, and Nigel Edwards. 2012. Transforming Commodity Security Policies to Enforce Clark-Wilson Integrity. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC 2012).*

[53] Luis MuÃśoz-GonzÃąlez, Daniele Sgandurra, Andrea Paudice, and Emil C. Lupu. 2016. Efficient Attack Graph Analysis through Approximate Inference. arXiv'19: https://arxiv.org/abs/1606.07025.

[54] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL 1999).*

[55] Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. In *Proceedings of the 16th ACM Symposium on Operating*

System Principles.
[56] Andrew C. Myers and Barbara Liskov. 1998. Complete, Safe Information Flow with Decentralized Labels. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*.
[57] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*.
[58] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the ACM Conference on the Principles of Programming Languages*.
[59] Palo Alto Networks. 2019. What Is An Intrusion Prevention System? https://www.paloaltonetworks.com/cyberpedia/what-is-an-intrusion-prevention-system-ips
[60] Steven Noel and Sushil Jajodia. 2008. Optimal IDS Sensor Placement and Alert Prioritization Using Attack Graphs. *Journal of Network and Systems Management* 16, 3 (Sep 2008), 259–275.
[61] Steven Noel, Sushil Jajodia, Brian O'Berry, and Michael Jacobs. 2003. Efficient Minimum-cost Network Hardening via Exploit Dependency Graphs. In *Proceedings of the* 19^{th} *Annual Computer Security Applications Conference*.
[62] Steven Noel, Eric Robertson, and Sushil Jajodia. 2004. Correlating Intrusion Events and Building Attack Scenarios through Attack Graph Distances. In *Proceedings of the* 20^{th} *Annual Computer Security Applications Conference*.
[63] Open Information Security Foundation (OISF). 2010. *Suricata*. https://suricata-ids.org/
[64] Xinming Ou, Wayne F. Boyer, and Miles A. McQueen. 2006. A scalable approach to attack graph generation. In *Proceedings of the* 13^{th} *ACM Conference on Computer and Communications Security*.
[65] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. 2016. The Limitations of Deep Learning in Adversarial Settings. In *Proceedings of the 2016 IEEE European Symposium on Security and Privacy*.
[66] Vern Paxson. 1999. Bro: A System for Detecting Network Intruders in Real-time. *Computer Networks* 31, 23-24 (Dec. 1999), 2435–2463.
[67] Cynthia Phillips and Laura Painton Swiler. 1998. A Graph-based System for Network-vulnerability Analysis. In *Proceedings of the 1998 New Security Paradigms Workshop (NSPW 1998)*.
[68] Wolter Pieters. 2019. Everything-as-a-Hack: Claims-Making for Access to Digital and Social Resources. In *Proceedings of the 2019 New Security Paradigms Workshop*. ACM.
[69] Devin J. Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. 2012. Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In *Proceedings of the* 28^{th} *Annual Computer Security Applications Conference (ACSAC '12)*.
[70] Nayot Poolsappasit, Rinku Dewri, and Indrajit Ray. 2012. Dynamic Security Risk Management using Bayesian Attack Graphs. *IEEE Transactions on Dependable and Secure Computing* 9, 1 (2012), 61âĂŞ74.
[71] Quadrantsec. 2015. *Sagan*. https://quadrantsec.com/
[72] Elias Raftopoulos and Xenofontas Dimitropoulos. 2011. Detecting, Validating and Characterizing Computer Infections in the Wild. In *Proceedings of the 2011 Internet Measurement Conference*.
[73] Martin Roesch. 1999. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the* 13^{th} *USENIX Conference on System Administration (LISA 1999)*.
[74] Sandra Rueda, David H. King, and Trent Jaeger. 2008. Verifying Compliance of Trusted Programs. In *Proceedings of the* 17^{th} *USENIX Security Symposium*.
[75] Bruce Schneier. 2000. *Secrets & Lies: Digital Security in a Networked World* (1st ed.). John Wiley & Sons, Inc., New York, NY, USA.
[76] SELinux 2017. SELinux Project Wiki. https://selinuxproject.org/page/Main_Page.
[77] Dave Shackleford. 2018. Intrusion Detection Evasion Techniques and Case Studies. STI Graduate Student Research in SANS. https://www.sans.org/reading-room/whitepapers/detection/paper/38350
[78] Oleg Sheyner, Joshua W. Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. 2002. Automated Generation and Analysis of Attack Graphs. In *2002 IEEE Symposium on Security and Privacy*.
[79] SolarWinds 2019. *SolarWinds*. https://www.solarwinds.com/
[80] Robin Sommer and Vern Paxson. 2010. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*.
[81] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2018. SoK: Sanitizing for Security. *CoRR* abs/1806.04355 (2018).
[82] Nirupama Talele, Jason Teutsch, Robert F. Erbacher, and Trent Jaeger. 2014. Monitor Placement for Large-Scale Systems. In *Proceedings of the* 19^{th} *ACM Symposium on Access Control Models and Technologies*.
[83] Nirupama Talele, Jason Teutsch, Trent Jaeger, and Robert F. Erbacher. 2013. Using Security Policies to Automate Placement of Network Intrusion Prevention. In *Proceedings of the* 5^{th} *International Symposium on Engineering Secure Software and Systems*.
[84] Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. 2015. Approximating Attack Surfaces with Stack Traces. In *Proceedings of the* 37^{th} *International Conference on Software Engineering - Volume 2 (ICSE 2015)*.
[85] Fredrik Valeur, Giovanni Vigna, Christopher Kruegel, and Richard A. Kemmerer. 2004. Comprehensive Approach to Intrusion Detection Alert Correlation. *IEEE Transactions on Dependable and Secure Computing* 1, 3 (2004), 146âĂŞ169.
[86] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. 2014. JIGSAW: Protecting Resource Access by Inferring Programmer Expectations. In *Proceedings of the 23rd USENIX Security Symposium*.
[87] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. 2012. Integrity Walls: Finding attack surfaces from mandatory access control policies. In *Proceeedings of the* 7^{th} *ACM Symposium on Information, Computer, and Communications Security (ASIACCS 2012)*.
[88] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. 2012. STING: Finding Name Resolution Vulnerabilities in Programs. In *Proceedings of the* 21^{st} *USENIX Security Symposium*.
[89] Lingyu Wang, Anyi Liu, and Sushil Jajodia. 2006. Using Attack Graphs for Correlating, Hypothesizing, and Predicting Intrusion Alerts. *Computer Communications* 29, 15 (Sept. 2006), 2917–2933.
[90] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*.
[91] Rainer Wichmann. 2006. *SAMHAIN*. https://la-samhna.de/samhain/
[92] Wikipedia contributors. 2019. Vulnerability (computing)— Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Vulnerability_(computing) [Online; accessed 17-November-2019].
[93] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *Proceedings of the* 27^{th} *USENIX Security Symposium*.
[94] Peng Xie, Jason H. Li, Xinming Ou, Peng Liu, and Renato Levy. 2010. Using Bayesian Networks for Cyber Security Analysis. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.
[95] Wenjuan Xu, Xinwen Zhang, and Gail-Joon Ahn. 2009. Towards System Integrity Protection with Graph-Based Policy Analysis. In *Proceedings of the* 23^{rd} *Annual IFIP WG 11.3 Working Conference Data and Applications Security*.
[96] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David MaziÃĺres. 2006. Making Information Flow Explicit in HiStar. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*.