Automatic Generation and Analysis of NIDS Attacks

Shai Rubin, Somesh Jha, and Barton P. Miller
University of Wisconsin, Madison
Computer Sciences Department
{shai,jha,bart}@cs.wisc.edu

Abstract

A common way to elude a signature-based NIDS is to transform an attack instance that the NIDS recognizes into another instance that it misses. For example, to avoid matching the attack payload to a NIDS signature, attackers split the payload into several TCP packets or hide it between benign messages. We observe that different attack instances can be derived from each other using simple transformations. We model these transformations as inference rules in a natural-deduction system. Starting from an exemplary attack instance, we use an inference engine to automatically generate all possible instances derived by a set of rules. The result is a simple yet powerful tool capable of both generating attack instances for NIDS testing and determining whether a given sequence of packets is an attack.

In several testing phases using different sets of rules, our tool exposed serious vulnerabilities in Snort—a widely deployed NIDS. Attackers acquainted with these vulnerabilities would have been able to construct instances that elude Snort for any TCP-based attack, any Web-CGI attack, and any attack whose signature is a certain type of regular expression.

1. Introduction

The goal of a Network Intrusion Detection System (NIDS) is to alert a system administrator each time an intruder tries to penetrate the network. A *signature-based* NIDS defines penetration via malicious signatures: if an ongoing activity matches a signature, an alarm is raised [25, 33]. Such systems are widely deployed [36, 46] because they are simple to use and provide concrete information about the events that have occurred. A weakness of a signature-based NIDS is its inability to recognize an attack that is different from the attack signature it uses.

An attacker wishing to stealthily penetrate a network monitored by a signature-based NIDS can exploit this weakness in two ways. First, they can use an attack whose signature is not known to the NIDS. However, such attacks are usually difficult to find. Second, they can use a known attack, but try to elude the NIDS by finding an instance of the attack that the NIDS does not detect. For example, to elude a NIDS that does not perform TCP reassembly, the attacker can fragment the attack signature into several TCP packets [12, 30, 41]. Or, to elude a NIDS that uses only printable characters in its signatures, an attacker can change the signature of an HTTP attack by substituting equivalent hexadecimal ASCII values for the characters in a URL [10]. If an attacker can find an instance of a known attack that eludes the NIDS, then the NIDS is useless.

We study the ability of attackers to find attack instances that elude a NIDS and the ability of a NIDS to detect such instances. To be more concrete, we translate these abilities into the following two problems:

1. The *black hat* problem: given a NIDS and an instance of an attack $\mathcal{A}$, find an instance of $\mathcal{A}$ that evades the NIDS.
2. The *white hat* problem: given an instance of an attack $\mathcal{A}$ and a sequence of packets s, determine whether s is an instance of $\mathcal{A}$.

We propose a novel approach to rigorously tackle both problems. We observe that variants of the same attack can be methodically computed, or derived, from each other. We express the attacker's knowledge in a set of inference, or transformation, rules; each rule represents an atomic transformation the attacker can use to hide the attack signature. Then, starting from a known attack instance, we use an inference engine [42] to successively apply the rules and automatically compute all attack instances based on any combination of the rules. Finally, to deal with the black hat problem, we feed the instances into the given NIDS until we find one that is undetected. To deal with the white hat problem, we check whether the given instance matches one of the instances generated.

In particular, this paper makes the following contributions.

AGENT, a NIDS testing and attack analysis tool. Based on the notion of attack derivation, we implemented AGENT: *A*ttack *Ge*neration for *N*IDS *T*esting tool. AGENT addresses both the black and white hat problems and has the following advantages:

1. Unlike other tools [22, 31, 39, 41], AGENT is sound, generating instances of real attacks only. Therefore, when a NIDS misses an AGENT-generated attack, the NIDS is vulnerable.
2. AGENT is exhaustive, capable of generating all attack instances from a known instance using a set of rules. Hence, even if a single instance evades a NIDS, AGENT can find it. Similarly, AGENT can show that a NIDS correctly identifies all possible attack instances derived by a given set of transformations. Our results show that AGENT effectively finds NIDS vulnerabilities even without generating all instances.
3. Given a sequence of packets, AGENT can provide a proof, a sequence of transformations used for obfuscation, that the sequence is a real attack. Developers can use AGENT to analyze attacks and to identify the exact transformation that their NIDS fails to handle.

An attack derivation model for computing attack instances. We formalized AGENT's inference engine as a natural deduction system [28] and developed a formal model for computing attack instances. Using this model we formalize the black and white hat problems. The model has the following advantages:

1. The model is sound: it computes only real attacks. The model is exhaustive: given a set of transformations and an attack instance, the model computes all attack instances that are derivable from this instance using the transformations.
2. The model is self explanatory; it provides proof that a network event is or is not a mutated attack. If an event is a mutated attack, the proof is a sequence of transformations that links the event to a known attack instance; if the event is benign, the model asserts that, with respect to the rules used, such proof does not exist.
3. The model (and AGENT) appears to be insensitive to the derivation starting point. While different starting points may derive different attack instances, our experience shows that for the rules and attacks we consider in this paper, any starting point derives the same instances.

We believe that our model has other applications besides testing. To our knowledge, no formal method to determine whether or not a TCP sequence implements a given attack has been previously developed. For an attack $\mathcal{A}$, our model formally defines the notion of "a TCP sequence implementing $\mathcal{A}$": our model computes all TCP sequences implementing $\mathcal{A}$ by deriving them from an initial (known) instance of $\mathcal{A}$, using transformations that preserve $\mathcal{A}$'s semantics. We believe that such a formal model is essential for examining the capabilities of a single NIDS and for comparing capabilities of different NIDSs.

Improving a widely deployed NIDS. Using AGENT, we found several vulnerabilities in Snort v2.0.1 [33]. We exposed vulnerabilities in the TCP engine of Snort, the way Snort handles HTTP requests, and its pattern-matching algorithm. An attacker acquainted with these vulnerabilities could have caused Snort to miss any TCP-based attack, any HTTP scripting attack, and many attacks that require wildcard characters in their signatures, such as "`foo*bar`". We reported (and suggested fixes to) these vulnerabilities to the Snort development team. Some were immediately fixed, others will be fixed in the near future.

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 illustrates how variants of a real attack can be derived from each other and presents AGENT's architecture. Section 4 discusses AGENT's implementation and Section 5 formalizes the notion of derivation using natural deduction. Section 6 presents the vulnerabilities we found in Snort and demonstrates AGENT's capability to analyze attack instances.

2. Related Work

We review related work in the areas of attack transformation, NIDS evaluation, counter evasion techniques, protocol verification, and deductive databases.

Attack transformation methods. Ptacek and Newsham [30] used semantics preserving IP and TCP transformations to elude every NIDS they tested (they also implemented a tool for packet manipulation [29]). Similarly, Handley and Paxson [12, 25] discussed evasion techniques based on inherent ambiguities of the TCP/IP protocol. These researchers were the first to systematically address NIDS evasion techniques, but unlike our work, their research provides neither a formal model to combine transformations nor a tool to automatically do so. Nevertheless, various tools that mutate attacks originated from this work (Table 1). Next, we discuss those tools and how they differ from AGENT.

Attack transformation tools. Two attack-transformation tools that combine multiple transformations are Fragroute, which mutates TCP-based attacks [41], and Whisker, which mutates HTTP attacks [31]. They randomly combine transformations specified by the user, and have been used for NIDS testing [45]. They have two limitations that we address in our work. First, they do not always preserve the attack semantics. Second, they do not systematically search the space of attack instances, so they may miss instances that elude the NIDS.

Snot [39], Stick [11], and Mucus [22] are a family of *packet-mutation* tools. Snot and Stick synthesize a raw network packet (e.g., a TCP packet) such that it matches a Snort signature. They use this mutated packet to perform a DoS attack on Snort. Mucus, on the other hand, is a NIDS testing tool. It synthesizes many packets that match a given Snort signature by writing random data in packet fields not required by the signature; for testing, it then feeds the pack-

Name	Generation Capabilities	Sound	Transformations Implemented	Handles white hat problem?
Fragroute [41]	Full-session TCP-based attacks	N	Subset of the transformations from [30].	N
Whisker [31]	Full-session HTTP attacks	N	Full-session TCP and HTTP transformations.	N
Mucus [22]	Single TCP/UDP packets	N	Single packet TCP/UDP transformations.	N
THOR [18]	Full-session attacks	Y	IP fragmentation.	N
AGENT	Full-session attacks	Y	Full-session TCP, Finger, HTTP, and FTP.	Y

Table 1. Major tools that use attack variation as a testing methodology.

ets to another NIDS. Since these tools generate raw packets, they are useful for testing either the NIDS pattern-matching mechanism or the NIDS ability to detect UDP or IP based attacks. However, for TCP-based attacks, when a NIDS does not produce an alert for a TCP packet generated by Mucus, for example, it is hard to tell whether the NIDS has discarded the packet because it is not part of a TCP session or has scanned the packet but missed the signature due to a vulnerability in its pattern matching algorithm. In comparison, when a NIDS misses an AGENT-generated attack, it is necessarily the case that the NIDS is vulnerable. We believe that AGENT and Mucus complement each other: we intend to explore ways to incorporate randomness into AGENT.

The work closest to ours is THOR [18]. THOR launches mutated attacks to analyze the detection capabilities of a set of IDSs. THOR mutates attacks without altering their semantics; it is based on the concept of activity variations [2], that are analogous to our semantics-preserving transformations. At the conceptual level, we extend their work by developing a computational model to combine transformations in a systematic way. In practice, we address issues that their work does not address, such as handling inference among transformations or bounding the number of generated attacks. In particular, THOR's implementation includes only a single transformation (IP fragmentation) while AGENT's implementation includes both TCP and application-level transformations.

As far as we can tell, none of the tools mentioned above have the ability to find all the vulnerabilities that AGENT discovered. Tools that do not generate full TCP sessions cannot expose vulnerabilities, like the Evasive RST (Table 3), in a NIDS mechanism that tracks the state of a TCP session. Tools that do not use both transport and application-level transformations cannot expose vulnerabilities, like the FTP Padding (Table 3), in the interaction between a NIDS mechanism that handles TCP packets and the one that performs pattern matching. Concurrently to our work, Vigna et al. [48] developed a tool that does combine transformations. They also found serious vulnerabilities in both Snort and in ISS's RealSecure [13]. Although they use a different set of transformations, we believe that AGENT could have found the vulnerabilities they found and vice versa. Unlike this and our earlier work [34], Vigna et al. neither investigate a theoretical model to systematically apply the transformations nor address the white hat problem.

Other applications of attack transformations. Dacier et al. [6] noticed that different IDSs handle different sets of transformation methods. They developed a tool that evaluates the potential of a set of IDSs to handle a large set of transformation methods. They manually identified the methods each IDS is supposed to handle (e.g., using the IDS documentation), and used Prolog rules to formulate this knowledge. Then, they automatically found the set of transformations that the set of IDSs should be able to handle. In contrast, we actually test a NIDS by attacking it rather than by analyzing its potential capabilities.

Wagner and Soto [49] developed a formal model based on language theory to find attack instances that elude a host-based IDS. They added system calls to a known attack such that the attack semantics are preserved but the HIDS is no longer able to detect it. Tan et al. [43, 44] provide evidence that this theoretical model can be used in practice. We strongly support the use of formal models in intrusion detection and, to the best of our knowledge, believe that we are the first to provide a formal model for NIDS evasion.

One type of transformation that is beyond the scope of this paper is code obfuscation [8]. Recent research suggests that it is possible to identify the transformations used to obfuscate viruses [4]. In the future, we intend to explore ways to integrate obfuscation transformations into our attack derivation model.

NIDS evaluation. Lippman et al. [16, 17] present a comprehensive effort to evaluate IDS capabilities (with a critique by McHugh [19]). They compared the capabilities of several IDSs to detect a set of different attacks. In contrast, we test a single NIDS for its ability to detect many instances of the same attack.

Lee et al. [15] study the ability of a NIDS to handle packet loss due to a resource attack on a NIDS. In general, packet loss is a semantics preserving transformation in which the NIDS misses a packet that the host accepts. Although we do not investigate it in this paper, there is no fundamental limitation on the use of AGENT for generating instances with missing packets.

Resisting evasion attacks. Handley et al. [12] and

Shankar et al. [37] present techniques that remove TCP and IP ambiguities from network connections. These techniques can be used to prevent the Evasive RST vulnerability we found in Snort (Table 3). To the best of our knowledge, these methods are not widely deployed. Finally, Kruegel et al. [14] have designed a NIDS that is highly robust against resource attacks; we believe that AGENT, when implemented with packet loss transformations, can be used for testing their system.

Deductive databases and security protocol verification. For protocol verification, deductive systems have been used to model the capability of the participants and the adversary in a security protocol (e.g., [20, 24]). These techniques are related to our approach because we also use deductive systems to model the power of the adversary. To improve AGENT performance, we intend to explore techniques available for logic programs like state-space reduction [38] as well as efficient bottom-up [32] and top-down evaluation [9].

3. Approach Overview

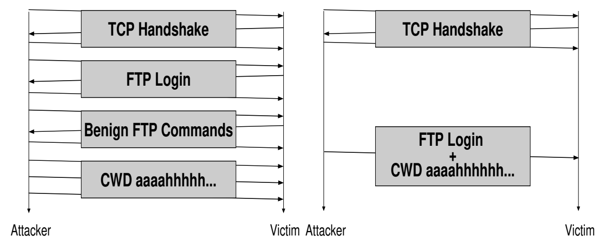
We illustrate the main idea behind our work: instances of the same attack can be systematically computed from each other. We start with examples of two attack instances of a known FTP vulnerability. We illustrate that the two instances are variants of each other: one instance can be derived from the other by repeatedly applying single step transformations. While the example we present is simple, it is based on a real vulnerability we discovered in Snort (Section 6.1). Last, we present AGENT's architecture that is based on the idea of attack derivation.

Our example vulnerability is a published buffer overflow in a commonly used FTP server (*BlackMoon* FTP server for Windows, CAN-2002-0126 in [21]); exploiting the overflow may crash the server or present root privileges. The exploit causes the overflow by providing an overly-long argument for the FTP CWD (change directory) command. We call this attack *ftp-cwd*.

The first instance of *ftp-cwd* we present is similar to instances that can be found on many hacker sites (e.g., [1]). Since this is a typical instance, we call it *ftp-cwd*$_{typ}$ (Fig. 1a). It contains four phases, each containing several TCP packets: (i) TCP handshake, (ii) FTP login, achieved by anonymous login, (iii) benign phase in which the attacker browses the server using benign FTP commands, and (iv) attack phase in which the attacker launches the attack by sending a long CWD command.

To illustrate the derivation of one *ftp-cwd* instance from another, we present a much shorter instance of *ftp-cwd* (Fig. 1b). We called it the *root* of *ftp-cwd* (denoted *ftp-cwd*$_{root}$) because, with respect to our rules, it derives all other *ftp-cwd* instances.

The *ftp-cwd*$_{root}$ instance contains only the necessary data for a successful *ftp-cwd* attack; furthermore, this data is condensed into a single TCP packet (except the TCP-handshake packets). Putting the FTP messages required for *ftp-cwd* (i.e., USER, PASS and CWD) in a single packet is possible because (i) no server response is necessary to carry out the attack, and (ii) FTP is an application-level protocol that is (and should be) indifferent to the number of TCP packets used to deliver its messages. While the attacks considered in this paper can be implemented using a single packet, our computational model and AGENT can handle multi-packet attacks.



(a) The *ftp-cwd*$_{typ}$ instance **(b)** The *ftp-cwd*$_{root}$ instance

Figure 1. Two *ftp-cwd* variants.

While the *ftp-cwd*$_{typ}$ and the *ftp-cwd*$_{root}$ might look different, from the attacker point of view they are the same, if an attacker can exploit the victim with *ftp-cwd*$_{typ}$, they can also exploit the victim with *ftp-cwd*$_{root}$. Intuitively speaking, one can infer *ftp-cwd*$_{typ}$ from *ftp-cwd*$_{root}$, and vice versa, as follows.

From here on, s_i denotes a TCP sequence and a_i denotes a TCP sequence that implements an attack $\mathcal{A}$ (also called an instance of $\mathcal{A}$). Consider the following two *transformation* rules:

1. r_1 (*TCP-fragmentation*): if a_1 is an instance of an attack $\mathcal{A}$, and a_2 is obtained from a_1 by copying a_1's packets and then fragmenting a single packet into two fragments, then a_2 is an instance of $\mathcal{A}$.
2. r_2 (*FTP-padding*): if a_1 is an instance of an FTP attack $\mathcal{A}$ that consists of at least one malicious FTP command after login (e.g., like the CWD command in the *ftp-cwd* attack), and a_2 is obtained from a_1 by inserting a benign FTP command between the login and the malicious command (but not the "QUIT" command), then a_2 is an instance of $\mathcal{A}$.

We call these rules *semantics preserving* because they do not alter the semantics of a_1. According to the TCP specification [26], it is legal to fragment TCP packets as desired. To the best of our knowledge, every FTP attack can be inflated, or padded, using benign FTP commands. If there does exist an FTP attack that cannot be padded or its packets cannot be fragmented, then the rules are changed to only allow legal modifications.

If *ftp-cwd*$_{root}$ is an instance of the *ftp-cwd* attack, then by using r_1 and r_2 it is possible to derive the conclusion that the *ftp-cwd*$_{typ}$ is also an instance of *ftp-cwd*. We apply r_1 on *ftp-cwd*$_{root}$ to fragment the single attack packet into the FTP login packets and the malicious CWD packet of *ftp-cwd*$_{typ}$. Then, we apply r_2 and add benign FTP commands to the attack. Using natural deduction terminology, we say that the *ftp-cwd*$_{typ}$ is *derived* from *ftp-cwd*$_{root}$ using the rules r_1 and r_2. More formally, we write: *ftp-cwd*$_{root} \vdash_{\{r_1,r_2\}}$ *ftp-cwd*$_{typ}$.

Three observations should be noted from the process illustrated above:

1. r_1 and r_2 define a closure over a subset of *ftp-cwd* instances. r_1 and r_2 can be used to derive not only the *ftp-cwd*$_{typ}$ instance, but also other instances of *ftp-cwd*. These rules derive every instance of *ftp-cwd* that contains several benign FTP commands and is fragmented into several TCP packets. This observation motivates us to automate the derivation process because this enables identification of every *ftp-cwd* instance from the above category and generation of such instances for NIDS testing purposes.
2. There is no interference between r_1 and r_2. To derive *ftp-cwd*$_{typ}$, it is possible either to first fragment the attack and then to add FTP commands or first add the commands and then fragment the attack. This observation simplifies both the practical implementation of AGENT (Section 4) and the theoretical aspects of our formal derivation model (Section 5).
3. The inference process can be bi-directional. Consider the shrinking rules: $\overleftarrow{r_1}$ as de-fragmentation and $\overleftarrow{r_2}$ as removal of padding. It is easy to see how *ftp-cwd*$_{root}$ can be derived from *ftp-cwd*$_{typ}$. This bi-directional property suggests that a derivation process can start from any attack instance, so finding instances that elude a NIDS is not overly sensitive to the derivation starting point. We use this observation in the next section when we define the starting points for our attack derivation model.

To translate the above observations into practice, we built AGENT, a tool that derives attack instances from a representative instance of the attack using a set of transformation rules. AGENT can be used to address both the black and white hat problems (Section 6). AGENT is comprised of the following components (Fig. 2):

1. **Closure Generation**. Given a set of rules and an attack instance, AGENT computes the root of the instance (e.g., *ftp-cwd*$_{root}$ from Fig. 1b), and generates all instances derived from this root. Each instance is represented as a list of TCP packets and contains both attack and response packets. The instances are stored in a file which is passed to the next stage.

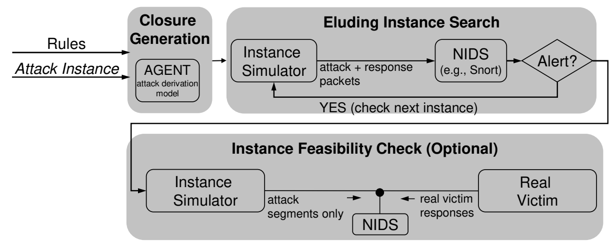


Figure 2. AGENT Architecture.

2. **Eluding-Instance Search**. This stage finds an attack instance that eludes Snort. To perform this search, we implement an *instance simulator* that successively writes the instances to the network. On the simulator's machine we installed Snort, which reads from the network. This search stops when an undetected instance is found or when all instances have been checked.

   We implemented the simulator using C libraries that construct raw TCP packets [35, 47]. The simulator plays complete TCP sessions, including TCP handshake, attacker and victim's packets, and termination procedures. It simulates an average of 350 instances per second on a Pentium III, 850MHz.
3. **Instance Feasibility Check**. Here we used two machines connected by a LAN to separate the attacker from the victim. In this stage, we used the instance simulator to send the attacker's packets only, the victim responses were generated by a real application. Strictly speaking, this stage is unnecessary because AGENT is sound. However, we include it to validate our own methodology and to illustrate that the attacks we found work in the wild.

4. AGENT Implementation

We discuss the core components of AGENT. First, we describe the implementation of AGENT's inference engine. Then, we present the transformation rules we used and how we constructed the attack instance used to start the derivation process. Last, we discuss how we customized the inference-engine to address the black and white hat problems.

4.1. Inference Engine

We implemented AGENT's inference engine in Prolog [42]. Starting from a set of known facts, a Prolog program applies a set of inference rules to successively generate new facts. Our implementation represents the root instance as a Prolog fact and the inference rules as Prolog rules. A Prolog rule specifies a set of conditions whose conjunction must hold to derive a conclusion. For example, a rule may specify the conditions that must hold to conclude that two TCP packets are fragments of another packet.

We represent an attack as a sequence of TCP packets that contains both *attack packets* (sent by the attacker) and *response packets* from the victim. We choose a TCP sequence to represent an attack because the majority of known attacks use TCP; for example, 88% of Snort rules target TCP communication. Since this representation exposes both TCP parameters and application data, it is easy to manipulate attacks using TCP transformations, application-level transformations, or both. However, TCP representation is not essential to AGENT; it is easy to define transformation rules using other representations, such as UDP or IP packets.

4.2. Inference Rules

AGENT's inference rules take a TCP sequence that implements an attack $\mathcal{A}$ and returns a different sequence that also implements $\mathcal{A}$. The inference rules are, by definition, *sound*: they always produce a sequence that implements $\mathcal{A}$. Soundness ensures that AGENT only derives instances that are real attacks.

From our experience, defining sound rules is not a difficult task. However, one should consider three issues when defining rules.

1. **The attack application-level protocol.** Obviously, FTP rules (e.g., r_2 in Section 3) are not sound with respect to HTTP attacks. Hence, a rule set is specific to a given application-level protocol.
2. **The operating system on the victim's host.** Since protocol semantics may depend on a particular implementation [37], a rule may not preserve $\mathcal{A}$'s semantics on all operating systems. Hence, we customize the rule set for a particular operating system. While this sounds complex, in practice, it is an easy process: differences between operating systems usually affect only a small number of inference rules in the network or transport levels [37].
3. **Interference among rules.** A rule that preserves $\mathcal{A}$'s semantics when applied alone may not preserve the semantics when applied with other rules. Like others [18], we could not find an application-level transformation that interferes with a transport level one; we encountered interference only at the TCP level, between the TCP-permutation rule (permutes the packets of the sequence) and the TCP-retransmission rule (adds retransmitted packets to the sequence). This interference occurs because we define that a retransmitted packet appears after the original packet, so we cannot permute those two packets. We address this issue in Section 4.4.

For each transformation we define two rules: *expanding* and *shrinking* rules. A shrinking rule reverses the effect of its corresponding expanding rule, and vice versa. For example, adding a benign FTP command expands an FTP attack, while removing such commands shrinks it. For transformations that do not change the attack length (e.g., sending TCP packets out-of-order) we artificially assign the expanding rule (e.g., TCP-permutation) and the shrinking one (e.g., TCP-sorting). Since overly shrinking an attack may change its semantics, applying a shrinking rule may require checking preconditions, which is not necessary for an expanding rule. For instance, a rule that removes FTP commands must ensure not to remove the malicious command.

Building a set of inference rules is a one time effort. Since a single set usually can be shared by many attacks with the same application-level protocol, the amortized cost of this effort is low. However, there are cases in which a specific rule is unsound with respect to a specific attack. For example, the HTTP-space-padding rule (Table 2) may nullify a buffer overflow attack that requires an HTTP request with a particular length. While we envision a large set of rules distributed with AGENT, one should consider the soundness of each rule with respect to the attack under consideration. Our experience indicates that this can be done in a straightforward manner.

Table 2 summarizes the TCP and application-level rules that are currently supported by AGENT (detailed rule descriptions are available in an earlier report [34]). At the TCP level we cover transformations such as fragmentation, out-of-order transmission (i.e., permutation), retransmission, and header change (e.g., PUSH flag change). We developed transformations for three application-level protocols: FTP [27], HTTP [10], and `finger` [50]. We chose FTP and HTTP because they are common and used `finger` during AGENT development because of its simplicity. Nevertheless, using `finger` we exposed the Evasive-RST vulnerability in Snort (Section 6.1). To ensure soundness, we simulated the attack instances that evaded Snort on real servers (BSD `finger v0.17` and Apache HTTP server v1.3.28) and verified that they work.

Our rules are also unique because they add a victim's TCP acknowledgments as part of the transformations. Since a NIDS that performs a stateful TCP inspection (e.g., [3, 14]) uses acknowledgments to update its internal TCP state, different orderings of attack packets and acknowledgments induce different TCP states. Hence, even with a limited ability to influence the ordering between attack packets and victim's acknowledgments, an attacker can create an ordering that induces a TCP state in which a NIDS misses an attack (e.g., Snort Evasive-RST vulnerability, Section 6.1). The technical details about the way we added acknowledgments appear in an earlier report [34]. We note, however, that the issue of victim's responses (both at the TCP and application levels) as part of a transformation is yet to be fully investigated.

4.3. Derivation starting point

The derivation process starts from an attack instance called $root$. For an attack instance a_i we compute the root of a_i by successively applying our shrinking rules to a_i until no more rules can be applied. In general, two different in-

	Name	Description
TCP	Fragmentation (r_1)	Fragments an attack packet into two packets. Adds victim acknowledgment after each new packet.
TCP	Permutation (r_2)	Permutes packets in a TCP stream. To be sound, the permutation preserves the original order between attack packets and victim responses. Interference with TCP-retransmission is handled Section 4.4.
TCP	Retransmission (r_3)	A family of rules that add a retransmitted attack packet to the original stream. We focused on retransmission of evasive RST packets: a packet is retransmitted with the RST flag set.
TCP	Header Change (r_4)	A family of rules that change the header of an attack packet. (was not used in practice in this paper).
Application	finger padding (r_5)	Adds spaces before the username.
Application	FTP Padding (r_6)	Adds benign FTP commands before a malicious command. For example, the rule adds "CWD /tmp\n" and "LIST", but not "QUIT" because it does not preserve semantics.
Application	HTTP space padding (r_7)	Insert spaces after an HTTP method: for example, changes an HTTP request from "`GET_<URL>`" into "`GET___<URL>`"
Application	HTTP Multiple Requests (r_8)	Adds benign HTTP requests before a malicious request (e.g., "`GET ...   CMD.EXE`"). Such a benign commands may be "`GET ... /INDEX.HTML`", but without a "`Connection: close`" option because this changes the attack semantics.
Application	HTTP URL Encode (r_9)	Substitute printable characters in a URL with their equivalent ASCII values (was not used in practice in this paper).

Table 2. AGENT's Inference rules (only expanding rules are shown).

stances of the same attack may produce two different roots. Furthermore, the root of an instance may not be unique; it may depend on the application order of the shrinking rules. However, for the attacks and rules we consider in this paper, we found a unique root that is common to all attack instances. We further discuss the formal requirements from shrinking rules and the definition of roots in Section 5.

4.4. Customizing the Inference Engine

Black Hat Customizations. Although AGENT enables derivation of an infinite number of instances, in practice the number of instances we can feed into a NIDS is finite. Hence, it is necessary to select a finite subset of instances that has a high probability of finding vulnerabilities. One common strategy to construct this subset is a testing technique called equivalence partitioning [23]. In this technique, one splits the test cases into classes such that each class represents cases that exercise different features of the software under testing. Since attack transformations represent features a NIDS should handle, we define a class by the combination of transformations used to derive its instances.

We use three techniques to implement our equivalence partitioning strategy. First, we apply the inference rules in a depth-first order: we first apply application level rules because they are independent of transport rules, then we fragment each instance, permute the instances we get, and add retransmitted packets (Fig. 3). This order ensures that all instances in a given derivation path belong to different classes; since we apply TCP-retransmission after TCP-permutation, this order also resolves the interference between those two rules (Section 4.2). Second, we prune some of the derivation paths to prevent generation of a large number of instances from the same class. For example, we prune instances that only change the way an attack is fragmented; we do not fragment packets shorter than 5 bytes. Third, we split the testing process into phases that use different transformations, so instances from different phases belong to different classes. Even though this strategy impairs AGENT's exhaustiveness, our experimental results show that it is an effective way to find NIDS vulnerabilities (Section 6.1).

White Hat Customizations. In the white hat settings our goal is to find a derivation path from the root to a given instance (if one exists). To do so, we can start from the root and derive instances until we hit the given instance. Unfortunately, this approach is ineffective. First, it is difficult to determine when to stop searching. Second, if instance is far from the root (e.g., the instances in Section 6.2), such a search is infeasible. Hence, in the white hat settings, AGENT performs a bottom-up search: it uses only shrinking rules and shrinks the given instance until either it hits a known root or no further shrinking is possible. In the latter case, we need to manually determine whether we found a new root or the instance does not implement the attack.

5. A Formal Model for Attack Derivation

AGENT's inference engine can be generalized into a computational model for deriving attack instances. Our computational model is based on natural deduction [28] and can be used to formally define the black and white hat problems. As we discuss below, we believe that the model has applications beyond NIDS testing.

A natural deduction system is a pair $\langle F, \Phi \rangle$ where F is a set of facts, and Φ is a set of inference rules. Such a sys-

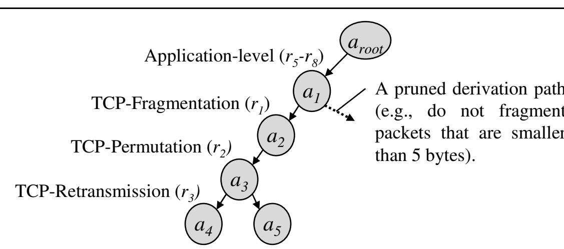


Figure 3. Application order of inference rules. Each node represents a generated instance. In the black hat setting, only expanding rules are used (Table 2).

tem enables derivation of new facts by applying the inference rules on already known facts. We say that Φ and a fact f_1 *derives* a new fact f_n, denoted $f_1 \vdash_\Phi f_n$, if there is a derivation sequence $\langle f_1, \ldots, f_n \rangle$ such that $f_1 \in F$ and every f_{i+1} is a result of applying a derivation rule $r \in \Phi$ on f_i. We say that a derivation sequence $\langle f_1, \ldots, f_n \rangle$ *terminates in* f_n if no rule can be applied to f_n.

To ensure a finite number of derivation starting points we make three assumptions on the rules in Φ: each rule has an expanding and shrinking version (Section 4.2), a shrinking version of a rule does not increase the length (in bytes) of an instance, and a derivation sequence containing only shrinking rules has no cycles. Note that these assumptions hold for the rules we describe in this paper. While we believe that these assumptions hold in general, the thorough investigation of their validity is left for future work.

Given a set of attack instances, $\{a_1 \ldots a_n\}$, we define for each a_i its derivation starting points, or roots. Intuitively, a root is the most compact and simple representation of a_i. More formally, a root of a_i is an instance that terminates a derivation sequence containing only the shrinking rules in Φ and that starts from a_i. The three assumptions described in the previous paragraph ensure that any sequence that starts from a_i terminates and that the set of roots for each a_i is finite. We denote the set of roots for a_i as *roots*$_\Phi(a_i)$ and for a set of instances as *roots*$_\Phi(a_1, \ldots, a_n)$

Below we formally define a derivation model for an attack as well as the black and white hat problems. For readability, the definitions are based on a single attack instance, but they can be trivially extended to a set of instances.

Definition 1 (Derivation Model of an Attack) *Let a_i be an instance of an attack $\mathcal{A}$ and Φ be a set of sound inference rules with respect to $\mathcal{A}$.*

- *A **derivation model of** $\mathcal{A}$ is a natural deduction system $\langle roots_\Phi(a_i), \Phi \rangle$.*
- *The **closure** of a derivation model, denoted $Cl_\Phi(roots_\Phi(a_i))$, is the set of all TCP sequences that are derived from $roots_\Phi(a_i)$ using Φ's rules.*

Definition 2 (NIDS View) *Let N be a NIDS. **N's view with respect to an attack** $\mathcal{A}$, denoted $S_N^{\mathcal{A}}$, is the set of TCP sequences that N recognizes as $\mathcal{A}$.*

Definition 3 (Black Hat Problem) *Let $\langle roots_\Phi(a_i), \Phi \rangle$ be an attack derivation model of $\mathcal{A}$, and N be a NIDS. Let $S_N^{\mathcal{A}}$ be the view of N with respect to $\mathcal{A}$. The black hat problem is to find a TCP sequence that is derivable from $roots_\Phi(a_i)$, but is not in $S_N^{\mathcal{A}}$. More formally, find $s \in Cl_\Phi(roots_\Phi(a_i)) \backslash S_N^{\mathcal{A}}$.*

Definition 4 (White Hat Problem) *Let $\langle roots_\Phi(a_i), \Phi \rangle$ be an attack derivation model of $\mathcal{A}$ and s be a TCP sequence. The white hat problem is to determine whether $s \in Cl_\Phi(roots_\Phi(a_i))$.*

Properties of the Attack Derivation Model. For an attack $\mathcal{A}$, we envision a derivation model that, with respect to a set of rules, is *sound*: derives only TCP sequences that implement $\mathcal{A}$; *complete*: can derive any TCP sequence that implements $\mathcal{A}$; and *decidable*: given a TCP sequence, there is an algorithm that determines whether or not the sequence is derived from the root.

For the black hat problem, soundness means that any instance we discover that evades the NIDS implies vulnerability in the NIDS; completeness means that eventually the model will generate all instances that evades the NIDS. In the white hat case, soundness means the lack of false positives, and completeness means the lack of false negatives. Decidability is important because without it, we cannot solve the white hat problem.

Our derivation model is sound because, by definition, we require that rules, and their combinations, are sound. With respect to the inferences rules we explore in this paper, our model is also decidable (Section 4.4). This does not mean that it is decidable in general. For example, when we consider code obfuscation rules [4], it is likely that the model becomes undecidable.

To prove completeness we would need to show that our derivation model computes all TCP sequences that adhere to a formal definition of the notion "a TCP sequence that implements $\mathcal{A}$". Since, to our knowledge, such a formal definition does not yet exist, proving completeness is not possible. However, a derivation model can be used to inductively define "implementing $\mathcal{A}$": a TCP sequence implements $\mathcal{A}$ if and only if it can be derived from another TCP sequence that implements $\mathcal{A}$. First, an expert determines the induction base, a TCP sequence implementing $\mathcal{A}$ and transformation rules that preserve $\mathcal{A}$'s semantics, then our model defines the set of all TCP sequences that implement $\mathcal{A}$. We believe that such formalism for the notion "implementing $\mathcal{A}$" is crucial for studying and understating NIDS capabilities. When another formal model for defining attack instances is developed, it will be possible to either use our model as a formal reference or compare our model to the new one.

6. Using AGENT in Practice

We present the results of using AGENT for NIDS testing and attack analysis.

6.1. NIDS Testing Using AGENT

We used AGENT to generate instances of known attacks and to feed them into Snort (v2.0.1). When Snort missed an instance, we stopped and investigated Snort code to find out the reason. We generated instances of three known attacks: (i) *finger-root*, used to gain root sensitive information from a victim (CVE-1999-0612), (ii) *perl-in-cgi*, used to execute arbitrary commands on a Web server (CAN-1999-0509), and (iii) *ftp-cwd*, a buffer overflow used to gain root access to an FTP server (CAN-2002-0126).

Name	Description	**Implications:** it is possible to eludes Snort for any
Evasive RST	Snort accepts an out-of-window TCP RST packet, and stops tracking a live TCP connection.	TCP-based attack. Fixed in Snort v2.0.2.
Flushing[1]	Snort misses a signature that is fragmented over several TCP packets.	attack whose signature can be inflated by an application-level rule.
HTTP space padding	Exploits Snort's default configuration together with its nature to report only a single alert per TCP packet. Snort misses the attack or generates a general alert instead of the *perl-in-cgi* alert.	Web-CGI attack. With a default configuration, Snort misses the attack; with a user-defined configuration, Snort generates a general HTTP alert rather than the specific alert for the attack.
HTTP multiple requests[1]	Snort analyzes only a single HTTP request per TCP packet.	Web-CGI attack. Fixed in Snort v2.1.0.
FTP Padding	Snort does not recognize a certain type of regular expressions.	attack with a signature such as "`foo*bar`". Fixed in Snort v2.0.6.

[1]Concurrently to our work, Sommer and Paxson also reported this vulnerability [40].

Table 3. Snort bugs found by AGENT. See earlier report for details [34].

For each attack we tested, AGENT found instances that eluded Snort. These instances exposed vulnerabilities in different sections of the Snort code: the TCP engine, the HTTP decoder, and the pattern matching mechanism. Some of the vulnerabilities have been fixed and others will be fixed in the upcoming releases of Snort. Table 3 presents a summary of vulnerabilities our testing effort exposed.

We chose Snort as a target NIDS for several reasons. First, Snort comes with more than 1500 signatures, so it was easy to find the signatures for our chosen attacks. Second, Snort is considered a state-of-the-art NIDS. Snort performance is comparable to that of a commercial NIDS [7, 46, 48], and it seems to be aware of many evasion techniques that were reported in the past [5, 12, 30]. Therefore, Snort uses techniques such as IP and TCP reassembly, HTTP encoding, TTL checks, and balanced data structures. Third, since it is maintained regularly, we assumed that it would be non-trivial to find instances that elude it.

Testing Effort Summary. Based on our equivalence-partitioning testing strategy (Section 4.4), we performed a total of seven testing phases that yielded five vulnerabilities. We started with a simple attack and with a rule set that derived a small number of instances. To cover more classes, in each phase we either added rules to AGENT or changed the attack.

Table 4 presents a summary of our seven test phases. In the first two phases, we used *finger-root* with only transport rules. In the second phase, AGENT exposed the Evasive-RST vulnerability. We continued to use the *finger-root* attack, but added the *finger-padding* rule. Using this rule alone did not yield new vulnerabilities (Phase 3), but combining it with transport rules exposed the Flushing vulnerability (Phase 4). We continued with *perl-in-cgi* and each HTTP rule we used exposed a vulnerability in Snort's HTTP decode engine (Phases 5 and 6). Finally, we tested Snort with instances of the *ftp-cwd* attack and discovered the FTP padding vulnerability (Phase 7).

We believe that AGENT's success in finding vulnerabilities arises from a combination of features. Since AGENT is sound, it generates only instances that are real attacks. Hence, we tested millions of instances and did not waste time determining whether or not a given TCP sequence is a real attack. Since AGENT generates all instances from a given set of rules or generates a subset of the instances based on feature testing (Section 4.4) , it finds vulnerabilities that are exposed only by a small number of instances (e.g., Phase 4). Last, AGENT exposes vulnerabilities that requires a combination of transport and application level transformations (e.g., Phase 7).

Phases 1 and 3 illustrate the implications of AGENT exhaustiveness. In these phases, AGENT generates all instances from a set of rules (i.e., no pruning). In Phase 1, Snort identifies all instances, so we can assert that Snort correctly reassembles TCP streams with six characters or less. After Phase 3 we can assert that Snort's pattern matching algorithm correctly ignores spaces before a signature. These are simple claims, but they provide important information about Snort reliability. To the best of our knowledge, such reliability claims are unknown in the context of network intrusion detection. We hope that in the future AGENT will be able to assert more complex properties.

AGENT also has a few limitations. In Phase 4, AGENT found the first instance that eluded Snort only after generating more than a million instances. If we uniformly sam-

Phase	Attack Used	Φ: Rules Used (Table 2)	Generated Instances	% of eluding instances	Vulnerability Discovered (Table 3)
1	*finger-root*	$\{r_1, r_2\}$	1,631	0	None
2	*finger-root*	$\{r_1, r_2, r_3\}$	3,628,960	33	Evasive RST
3	*finger-root*	$\{r_5\}$	25	0	None
4	*finger-root*	$\{r_1, r_2, r_5\}$	6,820,346	0.15	Flushing
5	*perl-in-cgi*	$\{r_7\}$	677,960	> 99	HTTP Space Padding
6	*perl-in-cgi*	$\{r_8\}$	100	99	HTTP Multiple Requests
7	*ftp-cwd*	$\{r_1, r_6\}$	178,585	23	FTP Padding

Table 4. Testing effort summary.

ple the instances derived in this phase, we should have found such an instance after about 10,000 samples (0.15% of 6,820,346). This observation suggests a way to improve AGENT efficiency by using random sampling methods (e.g., [22]).

6.2. Attack Analysis Using AGENT

To report a bug in a NIDS, one usually includes a trace of packets (e.g., captured by tcpdump [47]) that demonstrates an attack the NIDS misses. The first task of the NIDS development team is to verify that this trace is a real attack.

We demonstrated that AGENT can automate this task. We created 10 *ftp-cwd attack* instances by randomly applying the FTP-padding, TCP-fragmentation, TCP-permutation, and TCP-retransmission rules (Table 2) on the $\textit{ftp-cwd}_{root}$ instance (Fig. 1b). While the $\textit{ftp-cwd}_{root}$ instance contains 100 bytes in a single TCP packet (except the handshake packets), our instance contains an average of 3270 bytes in 693 packets. Then, we duplicated the instances and created 10 *benign* ones. In a benign instance we changed one byte to nullify the attack (e.g., we changed a sequence number of a packet, so the packets cannot be reassembled). We obtained 20 instances of which only 10 of them are real attacks.

We used AGENT as a white hat tool (Section 4.4) to distinguish between attack and benign instances. For each attack instance, AGENT found a derivation sequence to the root in about 0.5 seconds with an average length of 509 rules. For every benign instance, AGENT exhaustively searched for a derivation sequence but could not find one, on average, such a search took 70 seconds. Essentially, AGENT proved that, with respect to the rules we consider in this paper, a benign instance is not a real attack.

This experiment illustrates AGENT's ability to analyze complex attack instances. Manually distinguishing these instances would have required a tremendous effort. One might be able to automate such a process using a NIDS cross-testing technique [22], but such a technique cannot provide a sequence of transformations used to obfuscate an attack or assert that such a sequence does not exist.

7. Future Work

We are expanding our knowledge-base of rules. We are exploring other link, transport, and payload level rules, to model attackers' knowledge. In particular, we intend to investigate code obfuscation rules that enable attackers to change binary code of network exploits. We also intend to explore techniques for proving that the rules cover all possible ways to modify an attack.

We intend to explore other usages of our attack derivation model. For example, the model can be used to partition attack instances into families according to the transformations used to create them, or characterize and compare NIDS based on the mutations they handle.

Acknowledgments

We thank Marc Dacier, Jon Giffin, Dennis Gopan, Brian Fields, Vinod Ganapathy, and the anonymous reviewers for useful comments on an earlier draft of this paper.

References

[1] Digital information society. Available at www.phreak.org.

[2] D. Alessandri, editor. *Towards a Taxonomy of Intrusion Detection Systems and Attacks*. IBM Zurich Research Laboratory, September 2001. Deliverable D3, Project MAFTIA IST-1999-11583, Available at www.maftia.org.

[3] A. R. Baker, Barnyard, B. Caswell, M. Poor, R. Alder, J. Babbin, J. Beale, A. Doxtater, J. C. Foster, T. Kohlenberg, and M. Rash. ***Snort 2.1 Intrusion Detection***. Syngress, 2 edition, May 2004.

[4] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *USENIX Security Symposium*, Washington, DC, August 2003.

[5] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security Symposium*, Washington, DC, August 2003.

[6] M. Dacier, editor. *Design of an Intrusion-Tolerant Intrusion Detection System*. IBM Zurich Research Laboratory, August 2002. Deliverable D10, Project MAFTIA IST-1999-11583, Available at www.maftia.org.

[7] H. Debar and B. Morin. Evaluation of the diagnostic capabilities of commercial intrusion detection systems. In *International Symposium on Recent Advances in Intrusion Detection*, Zurich, Switzerland, October 2002.

[8] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. S. Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack Online Magazine*, **61**, August 2003.

[9] S. W. Dietrich. Extension tables: Memo relations in logic programming. In *The Fifth International Conference and Symposium on Logic Programming*, San Francisco, CA, August 1987.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *RFC 2616 - Hypertext Transfer Protocol*. The Internet Engineering Task Force, June 1999.

[11] C. Giovanni. Fun with packets: Designing a stick, March 2001. Endeavor Systems, INC.

[12] M. Handley and V. Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *USENIX Security Symposium*, Washington, DC, August 2001.

[13] Internet Security Systems. RealSecure Network 10/100. Available at http://www.iss.net/.

[14] C. Kruegel, F. Valeur, G. Vigna, and R. A. Kemmerer. Stateful intrusion detection for high-speed networks. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2002.

[15] W. Lee, J. B. D. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and Y. Zhang. Performance adaptation in real-time intrusion detection systems. In *International Symposium on Recent Advances in Intrusion Detection*, Zurich, Switzerland, October 2002.

[16] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. Analysis and results of the 1999 DARPA off-line intrusion detection evaluation. In *International Symposium on Recent Advances in Intrusion Detection*, Toulouse, France, October 2000.

[17] R. P. Lippmann, D. J. Fried, I. Graf, J. W. Haines, K. R. Kendall, D. McClung, D. Weber, S. E. Webster, D. Wyschogrod, R. K. Cunningham, and M. A. Zissman. Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation. In *DARPA Information Survivability Conference and Exposition*, Hilton Head, SC, January 2000.

[18] R. Marti. THOR: A tool to test intrusion detection systems by variations of attacks. Master's thesis, Swiss Federal Institute of Technology, March 2002.

[19] J. McHugh. Testing intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory. *ACM Transactions on Information and System Security*, **3**(4), November 2000.

[20] C. Meadows. A model of computation for the NRL protocol analyzer. In *IEEE Computer Security Foundations Workshop (CSFW)*, Franconia, NH, June 1994.

[21] MITRE Corporation. CVE: Common Vulnerabilities and Exposures. Available at `www.cve.mitre.org`.

[22] D. Mutz, G. Vigna, and R. A. Kemmerer. An experience developing an IDS stimulator for the black-box testing of network intrusion detection systems. In *Annual Computer Security Applications Conference*, Las Vegas, NV, December 2003.

[23] G. J. Myers. ***The Art of Software Testing***. John Wiley & Sons, 1 edition, 1979.

[24] L. Paulson. Mechanized proofs of security protocols: Needham-Schroeder with public keys. Technical Report 413, University of Cambridge Computer Laboratory, 1997.

[25] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, **31**(23/24), December 1999.

[26] J. Postel. *RFC 793 - Transmission Control Protocol*. The Internet Engineering Task Force, Sept. 1981.

[27] J. Postel and J. Reynolds. *RFC 959 - File Transfer Protocol*. The Internet Engineering Task Force, 1985.

[28] D. Prawitz. ***Natural Deduction: a Proof-Theoretical Study***. Almquist and Wiskell, 1965.

[29] T. H. Ptacek and T. N. Newsham. Custom attack simulation language (CASL). Available at `www.sockpuppet.org/tqbf/casl.html`.

[30] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical Report T2R-0Y6, Secure Networks, Inc., Calgary, Alberta, Canada, 1998.

[31] Rain Forest Puppy. A look at whisker's anti-IDS tactics – just how bad can we ruin a good thing?, December 1999. Available at `www.wiretrip.net/rfp/txt/whiskerids.html`.

[32] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Efficient bottom-up evaluation of logic programs. In ***Computer Systems and Software Engineering: State-Of-The-Art***. Kluwer Academic Publishers, June 1992.

[33] M. Roesch. Snort: the Open Source Network Intrusion Detection System. Available at `www.snort.org`.

[34] S. Rubin, S. Jha, and B. P. Miller. Attack generation for NIDS testing using natural deduction. Technical Report 1496, University of Wisconsin, Madison, January 2004.

[35] M. D. Schiffman. Libnet: A C library for portable packet creation and injection. Available at `www.packetfactory.net/libnet`.

[36] Security Administrator Newsletter. Instant poll: Do you use Snort to implement an intrusion detection system (IDS) on your network?, October 2002. Available at `http://www.winnetmag.com/Poll/`.

[37] U. Shankar and V. Paxson. Active mapping: Resisting NIDS evasion without altering traffic. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2003.

[38] V. Shmatikov and U. Stern. Efficient finite-state analysis for large security protocols. In *IEEE Computer Security Foundations Workshop (CSFW)*, Rockport, MA, June 1998.

[39] Sniphs. Snot, January 2003. Available at `www.stolenshoes.net/sniph/index.html`.

[40] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *ACM Conference on Computer and Communications Security*, Washington, DC, October 2003.

[41] D. Song. Fragroute: a TCP/IP fragmenter, April 2002. Available at `www.monkey.org/∼dugsong/fragroute`.

[42] L. Sterling and E. Shapiro. ***The Art of Prolog***. MIT Press, 1994.

[43] K. M. C. Tan, K. S. Killourhy, and R. A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *International Symposium on Recent Advances in Intrusion Detection*, Zurich, Switzerland, October 2002.

[44] K. M. C. Tan, J. McHugh, and K. S. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *The 5th International Workshop on Information Hiding*, Noordwijkerhout, Netherlands, Oct. 2002.

[45] The NSS Group. Intrusion detection systems (IDS) group test (Edition 2), December 2001. Available at `www.nss.co.uk/ids/edition2/index.htm`.

[46] The NSS Group. Intrusion detection systems (IDS) group test (Edition 4), 2003. Available at `www.nss.co.uk/ids/edition4/index.htm`.

[47] The Tcpdump Group. TCPDUMP/LIBPCAP. Available at `www.tcpdump.org`.

[48] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *ACM Conference on Computer and Communications Security*, Washington, DC, October 2004.

[49] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM Conference on Computer and Communications Security*, Washington, DC, November 2002.

[50] D. Zimmerman. *RFC 1288 - The Finger User Information Protocol*. The Internet Engineering Task Force, December 1991.