Formalizing Attack Mutation for NIDS Testing

Shai Rubin
Somesh Jha
Barton Miller

Technical Report #1522

March 2005

Formalizing Attack Mutation for NIDS Testing

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

March 11, 2005

Abstract

Attack mutation is a common way to test a misuse Network Intrusion Detection System (NIDS). In this technique, a known instance of an attack is transformed by repeatedly applying attack transformations into many distinct instances. For example, we can generate many instances of an HTTP attack by splitting it into TCP segments in many different ways. The underlying intuition behind attack mutation is that many attack instances are derivable from a few simple exemplary instances.

We formally justify the intuition behind attack mutation. We prove that for many transformations, all mutations of an attack are derivable from each other. Furthermore, we show that all mutations can be derived from a few atoms which are the simplest versions of the attack.

Based on our findings, we developed two algorithms: testing and forensics. Given a set of transformations, our testing algorithm derives all attack mutations (up to a certain length) from an exemplary attack instance. Our forensics algorithm complements the testing one; it determines whether two mutations are derivable from each other. Our algorithms accommodate most of the known transformations, so the algorithms can be immediately integrated into existing NIDS testing tools.

1 Introduction

The goal of a Network Intrusion Detection System (NIDS) is to detect malicious activities, or attacks, on the network. A misuse NIDS defines an attack via an attack signature, typically, a regular expression that matches a pattern of the attack [17, 21]. Ideally, each time an ongoing activity matches an attack signature, the NIDS raises an alarm. Thousands of organizations depend upon such systems [9] because they are simple to understand, enable customization of the signature database, and provide concrete information about the events that have occurred.

Abstractly, a signature usually corresponds to a single attack, a sequence of events that exploits a given vulnerability. In practice, however, a signature should match many equivalent attack forms, or *attack instances*. For example, the same attack can be split into TCP or IP packets in many different ways. Therefore, the reliability of a NIDS ultimately depends on its ability to detect any instance of a given attack. Unfortunately, researchers (and attackers) have successfully evaded many NIDS by mutating an attack instance that the NIDS recognizes into an instance that it misses. For example, to evade a NIDS that only uses a signature of ASCII characters they used the *URL encoding* transformation that replaces the ASCII characters of a URL with their equivalent hexadecimal values [7, 29].

To increase NIDS reliability, we should test the NIDS against as many attack instances as possible. A common way to generate many instances of the same attack is to use an *attack mutation system* [12, 16, 22, 29]. In such a system, we express a mutation, such as URL encoding above, as a *transformation rule* that generates a new attack instance from a known one. Then, we compute the closure of a few simple exemplary instances; this closure is all instances that are derived from the exemplary instances by repeatedly applying

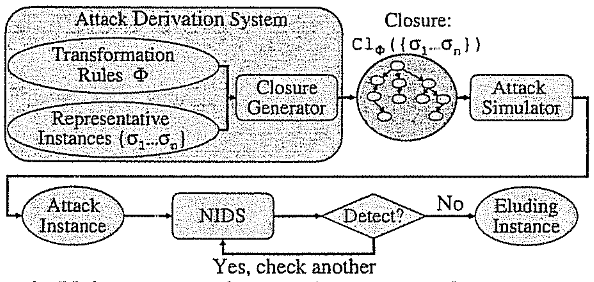
Attack Derivation System
Transformation Rules Φ
Representative Instances $\{\sigma_1 ... \sigma_n\}$
Closure Generator
Closure: $\mathtt{Cl}_\Phi(\{\sigma_1 ... \sigma_n\})$
Attack Simulator
Attack Instance
NIDS
Detect?
No
Eluding Instance
Yes, check another

Figure 1: Using an attack mutation system for NIDS testing.

the rules (Figure 1).

While attack mutation has successfully uncovered vulnerabilities in various NIDS, researchers have not yet formally investigated its underlying principle: the idea that attack instances are generally derivable from each other and, in particular, are derivable from a few simple instances.

To formalize this underlying principle of attack mutation, we define the two following problems:

1. **Testing problem.** Given a set of transformations rules Φ, attack instance σ, and integer k, generate the Φ_k*-closure* of σ: all instances that are shorter than k bytes derivable from σ using the rules in Φ.

2. **Forensics problem.** Given a set of transformations rules Φ and two attack instances, σ and τ, determine whether σ and τ are derivable from each other using the rules in Φ.

In this paper we address these problems. We prove that, given a set of transformations which satisfy the uniformity and reversibility properties we describe below, all attack instances are derivable from a few representative instances, called *atoms*. Furthermore, atoms split attack instances into equivalence classes: two instances are in the same class if and only if they are derivable from the same atom. Using atoms, we developed a two-phase testing algorithm. Given an attack instance, we first automatically compute its atom; then, we generate all instances that are derivable from this atom. Similarly, atoms facilitate an efficient forensics algorithm: to determine whether two instances are derivable from each other, we check whether they share the same atom.

The correctness of our algorithms in based on the notion of a *uniform and reversible attack deduction system*. Uniformity means that if an attack instance σ derives an instance τ, then there exists a derivation from σ to τ in which we first simplify σ as much as possible and then complicate the result until we reach τ. Reversibility means that any transformation in our system has corresponding inverse one. Together, these properties facilitate our algorithms: we derive τ from σ by first simplifying σ to its most concise representation, its atom, and then complicating the atom until we reach τ. We show that the majority of the transformations that current NIDS use are indeed uniform and reversible.

Our work has both theoretical and practical implications. From the theoretical point of view, our work provides a formal foundation for previous work on attack mutation [12, 16, 19, 22, 27, 29]. Furthermore, splitting attack instances into equivalence classes provides a consistent, complete, and unambiguous way to define an attack. For example, we can say that two TCP streams implement the same attack if and only if they can be derived from the attack atoms. To the best of our knowledge, such a concise definition for a network attack has not yet been developed.

Practically, our findings facilitate a systematic and maintainable NIDS testing process. First, we can split the testing process into distinct phases based on the attack equivalence classes. Second, when an unfamiliar attack instance surfaces, a NIDS developer can use our forensics algorithm to check whether the instance belongs to a known equivalence class. If it does not, the developer knows that this instance either forms a new class or was derived using a new transformation. In the latter case, we provide a methodology to check whether our algorithms can accommodate the new transformation. Last, we demonstrate that our algorithms can handle almost any set of transformations known today.

In summary, this paper makes the following contributions:

1. We define and prove the properties of a uniform and reversible attack deduction system. We describe the necessary steps one should take to prove uniformity, We show that in such a system, attack instances form equivalence classes, each represented by a simple instance.
2. We provide a testing algorithm and a linear-time forensics algorithm. We empirically evaluate the efficiency of our algorithms and their ability to handle attacks that contain thousands of bytes.
3. We present a practical instance of a uniform and reversible attack deduction system. We prove that the most common transformations used in known NIDS testing tools (e.g., [16, 22, 29] form a uniform and reversible system. We demonstrate that other transformations can be proved uniform as well.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 informally describes a uniform and reversible attack deduction system. Section 4 defines such a system and provides the algorithms for the testing and forensics problems. Section 5 proves that a common set of transformations form a uniform and reversible system. Section 6 provides empirical evaluation of our algorithms and Section 7 concludes the paper.

2 Related Work

We review related work in the areas of attack transformations, using attack mutation for NIDS testing, and using uniform proofs in logic programming.

Attack transformations. Fundamentally, network attacks can be modified, or transformed, at any level of the protocol stack. Ptacek and Newsham [18, 19] as well as Handley and Paxson [10, 17] were the first to introduce IP and TCP transformations (e.g., fragmentation, packet reordering).

Based on their work, tools that use attack transformations for NIDS testing, or evasion, have been developed. Fragroute, which transforms TCP-based attacks [25], and Whisker, which transforms HTTP attacks [20], randomly combine transformations specified by the user. Mucus [16] uses attack transformations to perform cross-testing of two NIDS: it builds packets that match a signature of the first NIDS, transforms them, and checks whether the other NIDS identifies the modified packets. Recently, Vigna et al. [29] developed a tool that applies application-level transformations (e.g., HTTP encoding, injection of Telnet escape characters) in addition to TCP/IP transformations. Other testing tools that are based on attack transformations are Snot [24], Stick [8], and Thor [1, 12]. To the best of our knowledge, all of these tools successfully found attack instances that evade the NIDS they had tested. As we discuss in Section 5, the majority of the transformations those tools use form a uniform deduction system, so the instances they generate can be concisely described using the notion of atoms we develop in this paper.

Attack deduction systems. Essentially, the testing tools mentioned above are based on the hidden assumption that attack instances can be derived from each other. In this paper, we formalize the properties of a testing tool, called AGENT [22], that is explicitly based on this assumption (Figure 1). In this paper we show that AGENT uses a uniform deduction system, we prove that AGENT generates a closure of a set of rules, and show how to use AGENT's deduction system to determine equivalent between two attack instance.

Dacier et al. [6] use attack mutation to evaluate the potential of a set of different IDSs to handle a large set of transformations. However, unlike our work here, they did not investigate the properties of their a deduction system.

Uniform proofs. Miller et al. [14] describe uniform proofs as proofs where right-introduction rules (which are analogous to shrinking rules) appear before left-introduction rules (which are similar to expanding rules). The main intuition behind introducing uniform proofs was to capture goal-directed search. Miller et al. also proved that in the framework of logic programming uniform proofs are complete, i.e., if a term is provable then it has an uniform proof. Uniform proofs have also been in explored in other contexts [11, 23]. Special structure of derivations has also been used in security-protocol verification [3, 4, 5, 13]. To our knowledge, the current paper is the only work that explores uniform derivations as the basis for generating attack mutations for NIDS testing.

3 Technical Overview

We use an exemplary attack to demonstrate the fundamental concepts behind our algorithms for solving the testing and forensics problems. These concepts are a partial order of attack instances, atoms, and a uniform attack deduction system.

The *perl-in-cgi* exploit (CAN-1999-0509 [15]): a Perl interpreter is installed in the `cgi-bin` directory on a Web server, allowing remote attackers to execute arbitrary commands.

Consider an instance of *perl-in-cgi*, denoted σ, that contains a single HTTP `GET` request: "`GET <web page>/sgi-bin/perl.exe`". Assume that σ uses a single TCP packet (not including the TCP handshake packets).

Consider the following transformation rules that attackers might use to transform σ:

1. r_1^+ (TCP-fragmentation): if τ is obtained from σ by copying σ's packets and then fragmenting a single packet into two packets, then τ is an instance of *perl-in-cgi.*

2. r_2^+ (HTTP encoding): if τ is obtained from σ by replacing an ASCII character in σ's URL with its hexadecimal value, then τ is an instance of *perl-in-cgi.*

3. r_3^+ (HTTP pipelining): if τ is obtained from σ by inserting a benign HTTP `GET` request (e.g., "`GET <web page>/index.html`") before the malicious `GET` request, then τ is an instance of *perl-in-cgi.*

Partial order of attack instances. It seems clear that r_1^+, r_2^+, and r_3^+ can be used to complicate σ: we can add arbitrary benign HTTP commands, obfuscate URLs, and fragment σ into small TCP packets. At the same time, the impact of the rules is *reversible*: we can undo r_1^+ by merging TCP packets, undo r_2^+ by normalizing URL to only use ASCII characters, and undo r_3^+ by removing benign HTTP requests.

Thus, a transformation has two forms: an *expanding* form that complicates an instance and a *shrinking* form that simplifies it. Given an arbitrary attack instance, an attack deduction system must use both expanding and shrinking transformations to generate a closure. We denote the shrinking, or reverse, versions of r_1^+, r_2^+, and r_3^+ as r_1^-, r_2^-, and r_3^-, respectively.

Expanding and shrinking transformations imply a partial order over the instances of *perl-in-cgi.* For example, the length (in bytes) of an instance can be used to rank the instance complexity: the longer the instance the higher its complexity. Note that r_1^+, r_2^+, and r_3^+ increase an instance complexity, while r_1^-, r_2^-, and r_3^- reduce it. (r_1^+ increase the complexity because each additional TCP packets requires an additional TCP header.)

Atoms. Intuitively, the instance σ is atomic. First, we cannot shrink σ any further because it uses a single TCP packet, does not include benign HTTP requests, and contains only ASCII characters. Second, σ is the simplest form of the attack, any byte in σ is required for a successful attack. Third, with respect to our rules, σ is the a building block of all other instances. Using expanding rules alone, σ derives any *perl-in-cgi* instance that is fragmented into several (non-overlapping) TCP packets, contains benign HTTP commands, and its URL uses either ASCII characters or their hexadecimal values.

A uniform attack deduction system. In a uniform attack deduction system, if an instance σ derives τ, then there exists a *uniform derivation* from σ to τ: a derivation in which all shrinking transformations precede all expanding ones. For example, it is easy to see that if we first expand an instance by fragmenting it (i.e., using r_1^+) and then replacing an hexadecimal value with an ASCII character (i.e., using r_2^-), then it is possible to first replace the character and then to fragment the instance.

Summary of observations. The concepts described above capture the intuition behind attack mutation systems. Shrinking and expanding transformations correspond to our intuition that we can simplify or complicate attack instances. Atoms correspond to our intuition that some attack instances cannot be simplified any further and these instances are the building blocks for other attack instances. Uniformity corresponds to our intuition to simplify an attack instance before we make it more complicated.

4 Modeling Attacker Transformations

We formalize the intuition behind attack mutation. First, we define a general attack deduction system and a test-generation algorithm that is complete and sound with respect to that system. We show that a complete and sound test-generation algorithm solves both the testing and forensics problems. However, we argue that such an algorithm is unlikely to be found for a general attack deduction system. So, we refine the general system into a uniform and reversible one. Then, based on the refined system, we present a sound and complete test-generation algorithm that solves the two problems.

4.1 Attack Derivation and a Testing Algorithm

In this section we model attacks as sequences over the alphabet Σ.

4.1.1 Basic Definitions.

Let Σ be an alphabet set, $\Sigma^\star$ be the set of sequences over Σ, and $\Sigma^k \subseteq \Sigma^\star$ be the set of sequences of length $\leq k$. An inference rule r has the following form:

$$\frac{\sigma, \mathit{pre}(\sigma)}{\sigma', \mathit{post}(\sigma, \sigma')}$$

In the expression given above, σ and σ' are sequences over Σ, and *pre* and *post* are predicates. The rule is interpreted as follows: if a sequence σ satisfies the predicate *pre*, then σ' is derivable from σ provided that $\mathit{post}(\sigma, \sigma')$ is true. If a sequence σ' can be derived from σ using a rule r, we write it as $\sigma \xrightarrow{r} \sigma'$.

A *deduction system* Φ is a collection of rules. We say that a sequence σ' is derivable from σ, denoted $\sigma \overset{\Phi}{\Rightarrow} \sigma'$, in the deduction system Φ if and only if there exists a sequence of rules $\langle r_1, \ldots, r_k \rangle$ in Φ, called a *derivation*, such that $\sigma \xrightarrow{r_1} \sigma_1 \xrightarrow{r_2} \cdots \xrightarrow{r_k} \sigma_k = \sigma'$. Given a sequence σ and a deduction system Φ, the *closure* of σ with respect to Φ, denoted $\mathit{Cl}_\Phi(\sigma)$, is the set of sequences that are derivable in Φ from σ; formally, $\mathit{Cl}_\Phi(\sigma) = \{\sigma' \mid \sigma \overset{\Phi}{\Rightarrow} \sigma'\}$. Given a finite set of sequence $S \subseteq \Sigma^\star$, its closure $\mathit{Cl}_\Phi(S)$ is given by $\bigcup_{\sigma \in S} \mathit{Cl}_\Phi(\sigma)$.

A *test-generation* algorithm, *TG*, is an algorithm that takes a finite set of sequences $S \subseteq \Sigma^\star$ and returns another set of sequences $\mathit{TG}(S)$ such that $S \subseteq \mathit{TG}(S)$. Intuitively, a test-generation algorithm takes a set of attack instances and returns a larger set of instances that are variations of the original instances.

Next, we define the soundness and completeness of a *TG* with respect to a deduction system Φ.

Definition 1 *Suppose we are given a deduction system Φ and a test-generation algorithm $\mathit{TG} : \Sigma^\star \longrightarrow \Sigma^\star$.*

- *TG is called* **sound with respect to** Φ, *denoted Φ-sound, if an only if for all $S \subseteq \Sigma^\star$, $\mathit{TG}(S) \subseteq \mathit{Cl}_\Phi(S)$. Intuitively, a Φ-sound test-generation algorithm only generates attack instances that are derivable from S in the deduction system Φ.*

- *TG is called* **complete with respect to** Φ, *denoted* Φ*-complete, if and only if for all* $S \subseteq \Sigma^\star$, $TG(S) \supseteq Cl_\Phi(S)$. *Intuitively, a* Φ*-complete test-generation algorithm covers all possible sequences derivable from* S *in the deduction system* Φ.
- *TG is called* **k-complete with respect to** Φ, *denoted* Φ^k*-complete, if and only if for all* $S \subseteq \Sigma^\star$, $TG(S) \cap \Sigma^k \supseteq Cl_\Phi(S) \cap \Sigma^k$. *A* Φ^k*-complete test-generation algorithm consists of all possible sequences of length* $\leq k$ *derivable from* S *in the deduction system* Φ.

If a test-generation algorithm *TG* is Φ-complete and Φ-sound, then for all $S \subseteq \Sigma^\star$, $TG(S) = Cl_\Phi(S)$. Similarly, if *TG* is Φ^k-complete and Φ-sound, then for all $S \subseteq \Sigma^\star$, $TG(S) \cap \Sigma^k = Cl_\Phi(S) \cap \Sigma^k$.

4.1.2 Addressing the Testing and Forensics Problems.

Based on definitions above, it is clear that a solution of our testing problem (see Introduction) is the closure $Cl_\Phi(\sigma) \cap \Sigma^k$. Hence, any *TG* algorithm that is Φ^k-complete and Φ-sound solves this problem.

The forensics problem can be formally stated as follows: given two sequences σ and σ' and deduction system Φ, is σ' derivable from σ in Φ, or $\sigma \overset{\Phi}{\Rightarrow} \sigma'$? In general, the forensics problem in undecidable. Sequences can be used to model configurations of a Turing machine, and rules can model the transition relation of that machine. Therefore, the halting problem of a given Turing machine is reducible to the forensics problem for the corresponding deduction system.

Note that any Φ^k-complete and Φ-sound *TG* can be used to solve the forensics problem:

> Let *TG* be a Φ^k-complete and Φ-sound test-generation algorithm. Let $k = |\sigma'|$. Use *TG* to compute $Cl_\Phi(\sigma) \cap \Sigma^k$. Check that $\sigma' \in Cl_\Phi(\sigma) \cap \Sigma^k$.

Thus, for a general deduction system, there is no Φ^k-complete and Φ-sound *TG*.

To better understand the difficulty in constructing a Φ^k-complete and Φ-sound *TG*, consider a standard work-list algorithm that builds a closure of σ by recursively deriving successors of σ. It is difficult to determine when to terminate such a derivation process. Suppose the algorithm derives an instance σ' such that *length*$(\sigma') > k$. Intuitively, since σ' is too long to be included in $Cl_\Phi(\sigma) \cap \Sigma^k$, we would be inclined to believe that σ' cannot derive any instance that is part of $Cl_\Phi(\sigma) \cap \Sigma^k$. However, in a general deduction system, each rule might have an arbitrary effect. So, even though σ' is too long, it might derive a shorter instance that is part of the closure.

The observation above suggests that a Φ^k-complete and Φ-sound *TG* requires to define how rules change the size of a sequence. This is the intuition behind a uniform and reversible system.

4.2 Uniform and Reversible Deduction Systems

We describe two additional properties of a deduction system: uniformity and reversibility. Then, given a uniform and reversible system, we provide an algorithm that simplifies an attack to its most concise form.

4.2.1 Uniformity and Reversibility

Let $\preceq$ be a partial order on the set $\Sigma^\star$. Recall that $\preceq$ is a partial order if it satisfies the following conditions for all $\sigma, \beta, \alpha \in \Sigma^\star$:

1. $\sigma \preceq \sigma$.
2. $\sigma \preceq \beta$ and $\beta \preceq \sigma$ implies that $\sigma = \beta$,
3. $\sigma \preceq \beta$ and $\beta \preceq \alpha$ implies that $\sigma \preceq \alpha$.

These conditions are referred to, respectively, as *reflexivity*, *antisymmetry*, and *transitivity*. We say that $\sigma \prec \beta$ if and only if $\sigma \preceq \beta$ and $\sigma \neq \beta$. For a sequence σ the *down set* of σ, denoted $\sigma\downarrow_{\preceq}$, is the set of all elements that are $\preceq$ than σ. Given a finite set $S \subseteq \Sigma^*$, $S\downarrow_{\preceq}$ is defined as the set $\{\sigma \mid \text{there exists } \sigma' \in S \text{ such that } \sigma \preceq \sigma'\}$.

We assume that for all $\sigma \in \Sigma^*$ the set $\sigma\downarrow_{\preceq}$ is finite. Since $S\downarrow_{\preceq}$ is equal to $\cup_{\sigma\in S}(\sigma\downarrow_{\preceq})$, the set $S\downarrow_{\preceq}$ is also finite. This assumption implies that any descending chain starting with $\sigma \in \Sigma^*$ is finite[1]. In section 5.2.1, we show that this assumption holds for our exemplary attack-deduction system, and that it is likely to hold for other systems as well.

Given a sequence σ and partial order $\preceq$, the *height of* σ, denoted $ht_{\preceq}(\sigma)$, is the length of the longest descending chain starting at σ. Notice that $ht_{\preceq}(\sigma)$ is bounded by the size of the set $\sigma\downarrow_{\preceq}$ and hence is finite.

A rule r is called a *shrinking rule* if for all σ and σ' such that $\sigma \xrightarrow{r} \sigma'$ we have that $\sigma \succ \sigma'$. A rule r is called an *expanding rule* if for all σ and σ' such that $\sigma \xrightarrow{r} \sigma'$ we have that $\sigma \prec \sigma'$. A derivation $\sigma_0 \xrightarrow{r_1} \sigma_1 \cdots \xrightarrow{r_k} \sigma_k$ is called a *uniform derivation* if there does not exist an $i < j$ such that r_i is an expanding rule and r_j is a shrinking rule (shrinking rules are applied before expanding rules). Now we define a uniform deduction system.

Definition 2 *A deduction system Φ is called* **uniform** *if there exists a partial order $\preceq$ on Σ^* such that the following conditions hold:*

- *With respect to $\preceq$ each rule in Φ is either a shrinking or expanding rule.*
- *For all σ and σ', such that $\sigma \overset{\Phi}{\Rightarrow} \sigma'$ there exists a uniform derivation from σ to σ', i.e., any derivation in Φ has a corresponding uniform derivation.*

Given a uniform deduction system Φ, Φ^- and Φ^+ denote the deduction systems consisting of shrinking and expanding rules in Φ, respectively. Inverse of a rule r, denoted r^{-1}, is a rule such that for all σ and σ' $\sigma \xrightarrow{r} \sigma'$ if and only if $\sigma' \xrightarrow{r^{-1}} \sigma$. Notice that inverse of a shrinking rule is an expanding ruled and vice-versa. A uniform deduction system Φ is called *reversible* if every rule in Φ has an inverse.

4.2.2 Computing Atoms.

Given a uniform deduction system Φ, a sequence σ is called a Φ-*atom* if there does not exist a shrinking rule r in Φ such that $\sigma \xrightarrow{r} \sigma'$; in other words, no shrinking rule from Φ can be applied to a Φ-atom. The set of Φ-atoms is denoted by $Atoms(\Phi) \subseteq \Sigma^*$. Given a sequence σ, the set $atoms_\Phi(\sigma)$ is the set of Φ-atoms that have a derivation from σ consisting only of shrinking rules; formally, $atoms_\Phi(\sigma)$ is the set $Cl_{\Phi^-}(\sigma) \cap Atoms(\Phi)$. Similarly, for a set of sequences S the set $atoms_\Phi(S)$ is the set $Cl_{\Phi^-}(S) \cap Atoms(\Phi)$.

Consider Algorithm 1, which works as follows:

[1]A chain $\sigma_0, \sigma_1, \cdots$ is called descending if and only if $\sigma_i \succ \sigma_{i+1}$ for $i \geq 0$.

```
input : A sequence $\sigma$
output: $atoms_\Phi(\sigma)$
$currentSequence = \sigma$;
while $true$ do
    if a shrinking rule cannot be applied to $currentSequence$ then break;
    else Pick a rule $r$ from $\Phi^-$ that can be applied to $currentSequence$.
    $currentSequence = r(currentSequence)$;
end
return $currentSequence$;
```

Algorithm 1: Algorithm for computing $atoms_\Phi(\sigma)$.

- Initially, the algorithm sets $currentSequence$ to σ.
- Each time in the while loop, a shrinking rule r is applied to $currentSequence$. If a shrinking rule cannot be applied to $currentSequence$, the algorithm breaks out of the while loop. (Given a rule r and a sequence $currentSequence$, $r(currentSequence)$ is the sequence obtained by applying the rule to $currentSequence$.)

Claim 1 Given a sequence σ, Algorithm 1 computes $atoms_\Phi(\sigma)$.

Proof: Algorithm 1 only computes descending chains. Since any descending chain is finite, the algorithm must terminate. It is clear that the algorithm computes an atom. Theorem 1 proves that for a uniform and reversible deduction system, the set $atoms_\Phi(\sigma)$ is a singleton set. Hence, the algorithm computes the set $atoms_\Phi(\sigma)$. The time complexity of Algorithm 1 is $O(ht_\preceq(\sigma))$ or linear in the height of the input sequence.

Notice that given a finite set of sequences S, $atoms_\Phi(S)$ is equal to $\cup_{\sigma\in S}\{atom_\Phi(\sigma)\}$. Therefore, $atoms_\Phi(S)$ can be computed by invoking Algorithm 1 on each element of the set S.

Theorem 1 If Φ is a uniform and reversible deduction system, then for every sequence σ, the set $atoms_\Phi(\sigma)$ is a singleton set.

Proof: We prove the theorem by contradiction. Suppose there are two sequences σ_A and σ_B in the set $atoms_\Phi(\sigma)$. By definition, there are derivations $\langle r_1, r_2, \ldots, r_i\rangle$ and $\langle r'_1, r'_2, \ldots, r'_j\rangle$ from σ to σ_A and from σ to σ_B, respectively. Since the deduction system is reversible, $\langle r_i^{-1}, \ldots, r_1^{-1}\rangle$ is derivation from σ_A to σ. Therefore, the following sequence of rules is a derivation from σ_A to σ_B:

$$\langle r_i^{-1}, \ldots, r_1^{-1}, r'_1, r'_2, \ldots, r'_j\rangle$$

Hence, σ_B is derivable from σ_A. Since the deduction system Φ is uniform, there is a uniform derivation $\langle r''_1, r''_2, \cdots, r''_l\rangle$ from σ_A to σ_B. If r''_1 is a shrinking rule, then a shrinking rule can be applied to σ_A, violating

```
input  : A set of sequences $S$
output: A set of test sequences
worklist = $\emptyset$;
// compute $atoms_\Phi(S)$.
forall $\sigma \in S$ do
    Compute $atoms_\Phi(\sigma)$ using Algorithm 1;
    worklist = worklist $\cup$ $atoms_\Phi(\sigma)$
end
// Compute the closure.
tests = worklist;
while worklist $\neq \emptyset$ do
    Pick $\alpha \in$ worklist;
    worklist = worklist $-$ $\{\alpha\}$;
    Compute $M = \Phi^+(\{\alpha\}) \cap \Sigma^k$;
    forall elements $\beta$ of $M$ do
        if $\beta \notin$ tests then
            worklist = worklist $\cup$ $\{\beta\}$
        end
    end
    tests = tests $\cup$ $M$
end
return tests;
```

Algorithm 2: A sound and k-complete testing algorithm.

```
input  : Two sequences $\sigma$ and $\sigma'$
output: true/false
Compute $\sigma_A = atoms_\Phi(\sigma)$ using Algorithm 1;
Compute $\sigma_B = atoms_\Phi(\sigma')$ using Algorithm 1;
if ($\sigma_A = \sigma_B$) return true;
else return false;
```
Algorithm 3: A linear-time algorithm for the forensics problem.

the fact that σ_A is an atom. Suppose that all rules r''_i, $1 \le i \le l$, are expanding rules. In this case, $(r''_l)^{-1}$ is a shrinking rule that can be applied to σ_B, violating the fact that σ_B is an atom. □

4.3 A Φ^k-Complete and Φ-Sound *TG*

We describe a new test-generation algorithm for a uniform and reversible deduction system Φ that is k-complete and sound. The basic idea is as follows:

- Generate $atoms_\Phi(S)$.
- Next, apply expanding rules from Φ^+ to all sequences in $atoms_\Phi(S)$ to generate additional sequences. Notice that when a sequence α is picked from the $worklist$, only its successor sequences that are of length $\le k$ denoted $\Phi^+(\{\alpha\}) \cap \Sigma^k$, are generated.

For closure generation purposes, we assume that $\preceq$ is *length preserving*: if $\alpha \preceq \beta$ then *length*$(\alpha) \le$ *length*(β). As we show in Section 5, this assumption holds for any attack transformations that we are familiar with. Theorem 2 shows that if Φ is a reversible and uniform deduction system and $\preceq$ is length preserving, then Algorithm 2 is sound and k-complete.

Theorem 1 also leads to an efficient algorithm for the forensics problem. Assume that we are given a reversible and uniform deduction system Φ. We observe that given two sequences σ and σ', $\sigma \overset{\Phi}{\Rightarrow} \sigma'$ if and only if $atoms_\Phi(\sigma) = atoms_\Phi(\sigma')$ (this follows straight from the observation that $atoms_\Phi(\sigma)$ is derivable from $atoms_\Phi(\sigma')$ and vice-versa). This immediately leads to the linear time Algorithm 3. Notice that the time complexity of this forensics algorithm is $O(ht(\sigma) + ht(\sigma'))$.

Theorem 2 Let Φ be a uniform and reversible deduction system based on a length preserving partial order, then Algorithm 2 is Φ-sound and Φ^k-complete test-generation algorithm.

Proof: Soundness of Algorithm 2 follows from the fact that we only apply rules from Φ to generate test cases. For showing k-completeness, we have to show that every sequence σ' in the set $Cl_\Phi(S) \cap \Sigma^k$ is generated by Algorithm 2.

Consider an arbitrary $\sigma' \in Cl_\Phi(S) \cap \Sigma^k$. There is a uniform derivation from a sequence $\sigma \in S$ to σ'. Let the uniform derivation be of the following form:

$$\sigma \ = \sigma_0 \xrightarrow{r_1^-} \sigma_1 \cdots \xrightarrow{r_k^-} \sigma_k \xrightarrow{r_1^+} \sigma_{k+1} \cdots \xrightarrow{r_m^+} \sigma_{k+m} \ = \ \sigma'$$

In the derivation given above, $r_1^-, \cdots, r_k^-$ are shrinking rules and $r_1^+, \cdots, r_m^+$ are expanding rules. If σ_k (the last sequence obtained after applying shrinking rules) is an atom, then $\sigma_k \in atoms_\Phi(S)$ because there is a derivation consisting of shrinking rules from $\sigma \in S$ to σ_k. In this case, σ' will be generated by Algorithm 2 because (i) all expanding rules are applied from $atoms_\Phi(S)$, and (ii) since *length*$(\sigma') \le k$ and $\preceq$ is length preserving, then $\sigma_k, \ldots, \sigma_{k+m}$ have length $\le k$.

If $\sigma_k \notin atoms_\Phi(S)$, then we will construct a new uniform derivation that derives σ' from σ and "passes through" an instance in $atoms_\Phi(S)$. Note that showing such a derivation implies that Algorithm 2 generates σ'.

	Name	*pre*	*post*	Description
TCP Rules	frag$^+$	$\sigma = \langle s_1, \ldots, s_i, \ldots, s_n \rangle$	$\tau = \langle s_1, \ldots, s_{i-1}, r_1, r_2, s_{i+1} \ldots, s_n \rangle \wedge$ $\mathit{frag\_seg}(s_i) = (r_1, r_2)$	Fragmentation (*frag*$^+$) corresponds to string splitting.
	frag$^-$	$\sigma = \langle s_1, \ldots, s_i, s_{i+1} \ldots, s_n \rangle$	$\tau = \langle s_1, \ldots, s_{i-1}, r_1, s_{i+2} \ldots, s_n \rangle \wedge$ $\mathit{frag\_seg}(r_1) = (s_i, s_{i+1})$	Defragmentation (*frag*$^-$) corresponds to concatenation.
	swap$^+$	$\sigma = \langle s_1, \ldots, s_i, \ldots, s_j, \ldots, s_n \rangle \wedge$ $s_i.seq < s_j.seq$	$\tau = \langle s_1, \ldots, s_j, \ldots, s_i, \ldots, s_n \rangle$	Swaps two segments such that they are sent our-of-order[a]. See definition of *frag_seg* in Appendix A.
	swap$^-$	$\sigma = \langle s_1, \ldots, s_i, \ldots, s_j, \ldots, s_n \rangle \wedge$ $s_i.seq > s_j.seq$	$\tau = \langle s_1, \ldots, s_j, \ldots, s_i, \ldots, s_n \rangle$	Swaps two segments such that they are sent in-order.
TCP Rules	*http*$^+$	$\sigma = \langle s_1, \ldots, s_i, \ldots, s_n \rangle$	$\tau = \langle r_1, \ldots, r_i, \ldots, r_n \rangle \wedge$ $\mathit{url\_encode}(s_i) = (r_i) \wedge$ $\forall j \neq i(r_j.payload = s_j.payload) \wedge$ $\forall j > i(r_j.seq = s_j.seq + 2) \wedge$ $\forall j < i(r_j.seq = s_j.seq) \wedge$	Replaces one ASCII character in a URL with its hexadecimal encoding (see *url_encode* definition in Appendix A). The predicate updates the sequence numbers of segments that their sequence numbers are larger than sequence number of the modified segment (as required by the TCP specification [7]).
	http$^-$	$\sigma = \langle s_1, \ldots, s_i, \ldots, s_n \rangle$	$\tau = \langle r_1, \ldots, r_i, \ldots, r_n \rangle \wedge$ $\mathit{url\_encode}(r_i) = (s_i) \wedge$ $\forall j \neq i(r_j.payload = s_j.payload) \wedge$ $\forall j > i(r_j.seq = s_j.seq - 2) \wedge$ $\forall j < i(r_j.seq = s_j.seq) \wedge$	

Table 1: A uniform deduction system for TCP-based attacks. *rule_name*$^+$ denotes an expanding transformation and *rule_name*$^-$ denotes shrinking one.

[a]In TCP jargon, out-of-order means not in the order of their sequence numbers.

Recall that $\sigma \overset{\Phi}{\Rightarrow} \sigma'$ if and only if $atoms_\Phi(\sigma) = atoms_\Phi(\sigma')$. Hence, there is a derivation from σ_k to a sequence $\alpha \in atoms_\Phi(\sigma) \subseteq atoms_\Phi(S)$. Let the derivation be

$$\sigma_k = \alpha_0 \xrightarrow{r'_1} \alpha_1 \cdots \xrightarrow{r'_i} \alpha_i \;=\; \alpha \in atoms_\Phi(S)$$

All the rules in the derivation given above are shrinking. Now, using the inverse rules we obtain a uniform derivation from σ_k to σ_k, which passes through α, the sequence in $atoms_\Phi(S)$. This derivation looks as follows:

$$\sigma_k = \alpha_0 \xrightarrow{r'_1} \alpha_1 \cdots \xrightarrow{r'_i} \alpha_i \xrightarrow{(r'_i)^{-1}} \alpha_{i-1} \cdots \xrightarrow{(r'_1)^{-1}} \alpha_0 = \sigma_k$$

Notice that all the rules $(r'_i)^{-1}$ are expanding rules. Now, insert this derivation after σ_k in the original derivation from σ to σ'. We obtained a uniform derivation that passes through a sequence in $atoms_\Phi(S)$. $\square$

5 A Practical Attack Deduction System

We prove that a set of common attack transformations forms a uniform and reversible attack deduction system. First, we define the rules in our system (Section 5.1). Second, we prove that these rules form a uniform and reversible system (Section 5.2). Last, we discuss the uniformity and reversibility of transformations that are not mentioned in our formal proofs (Section 5.3).

5.1 Deduction System for HTTP Attacks

Our system derives HTTP attacks. It comprises a set of TCP transformations and two HTTP transformations. We start with a formal definition for a TCP stream. This definition, and the TCP rules that use it, is also a basis for deduction systems for other application-level protocols (e.g., FTP).

stream	length	disorder
α=⟨(0,'GET perl.exe')⟩	(1,12)	0
β=⟨(0,'GET %70erl.exe')⟩ δ=⟨(0,'GET perl.e%78e')⟩	(1,14)	0
γ=⟨(0,'GET perl'),(8,'.exe')⟩ φ=⟨(0,'G'),(1,'ET perl.exe')⟩	(2,12)	0
ε=,(8,'.exe'), (0,'GET perl')⟩	(2,12)	1
ϕ=⟨(0,'GET perl'),(7, '.ex%65')⟩	(2,14)	0

Figure 2: Partial order over TCP streams: $\alpha \prec \beta, \delta \prec \gamma, \varphi \prec \varepsilon \prec \phi$ (for brevity, full URLs are not shown).

A TCP stream represents the communication between the attacker and the victim. A TCP stream comprises a sequence of segments,$\langle s_1, \ldots, s_n \rangle$, where each segment represents a single message that the attacker and victim exchange. Each segment is formulated as a pair, $(seq, payload)$, where seq represents the sequence number of the segment and *payload* represents the message (in bytes) that the segment contains.

The position of a segment in a sequence determines the time this segment is sent: s_i is sent only after s_j have been sent for all $j < i$, and before s_k for all $k > i$. For brevity, our TCP stream definition only includes the segments sent by the attacker.

Table 1 defines the set of rules in our attack deduction system. As mentioned (Section 4.1), each rule has the form of $\frac{\sigma, pre(\sigma)}{\tau, post(\sigma,\tau)}$ where *pre* is a precondition for applying the rule and *post* is a postcondition that holds after we applied the rule. Our deduction system includes rules that fragment a TCP stream and rules that deliver TCP segments out-of-order. Our HTTP rules define the URL encoding transformation mentioned in the Introduction. The superscript $^+$ denotes expanding rules and the superscript $^-$ denotes shrinking rules.

5.2 Proving Reversibility and Uniformity

Let Φ be the set of rules in Table 1. To prove that Φ is uniform and reversible, we need to show:

1. **Partial order for TCP streams**. There exists a partial order such that for any two TCP streams, σ, τ: (i) if $\sigma \xrightarrow{r^-} \tau$ then $\sigma \succ \tau$, and (ii) if $\sigma \xrightarrow{r^+} \tau$, then $\sigma \prec \tau$.
2. **Reversibility**. For any two TCP streams σ, τ and a rule $r \in \Phi$: if $\sigma \xrightarrow{r} \tau$ then there exists $r^{-1} \in \Phi$ such that $\tau \xrightarrow{r^{-1}} \sigma$.

   It is easy to see that each rule in our system has a reverse form: each *rule_name*$^+$ can be reversed by *rule_name*$^-$, and vice versa (Table 1).
3. **Uniform derivation**. For any two TCP streams σ, τ if $\sigma \overset{\Phi}{\Rightarrow} \tau$ then there exist a uniform derivation from σ to τ.

5.2.1 A Partial Order for TCP Streams

We order TCP streams according to their *complexity*. Informally, complexity of a TCP stream is based on the stream length and the order of the stream segments. We say that σ is more complex than τ, if it delivers a longer payload, delivers the same payload but uses more segments, or delivers the same payload with the same number of segment but the segment in σ are more disordered, as we define below, than the segments in τ.

Figure 2 illustrates how *complexity* orders TCP streams. The simplest stream has a single TCP segment and a URL only containing ASCII characters (i.e., α). The most complex stream contains one segment per byte and encodes a URL using hexadecimal values (not shown). A sequence with two segments is longer than any sequence with a single segment; non-identical streams might have the same length and disorder-level but are still incomparable (e.g., β, δ) and we only consider the number of segments rather than the way the segments are split (e.g., γ, ϕ). Note that other definitions of *complexity* are also possible.

We now turn to a formal definition of *complexity*.

Definition 3 (Length of a TCP stream) *Let* $\sigma = \langle s_1, \dots s_n \rangle$ *be a TCP stream. Define* $length(\sigma)$ *to be the tuple* $(n, \sum_{i=1}^{n} size\_of(s_i.payload))$.

Let $length\ (\sigma) = (n, k)$ *and* $length(\tau) = (m, j)$ *then:*

1. *We say that* $length(\sigma) = length(\tau)$ *if and only if* $n = m$ *and* $k = j$.
2. *We say that* $length(\sigma) < length(\tau)$ *if and only if* $(n < m)$ *or* $(n = m \wedge k < j)$.

The next component the complexity of a TCP stream is the stream disorder-level. Intuitively, the disorder level of σ counts the number of segment pairs that are sent out-of-order. For example, the disorder level of a stream that sends segments ordered according to their sequence numbers is zero. Similarly, the disorder level of a stream that sends the segments in their reverse order is $\frac{n(n-1)}{2}$.

Definition 4 (TCP stream disorder level) *Let* $\sigma = \langle s_1, \dots, s_i, \dots, s_j, \dots, s_n \rangle$ *be a TCP stream. Define:*

1. $not\_in\_order(\sigma, s_i, s_j) = 1$ *if and only if* $i < j$ *and* $s_i.\mathrm{seq} > s_j.\mathrm{seq}$.
2. $disorder(\sigma) \equiv \sum_{1 \le k < l \le n} not\_in\_order(\sigma, s_k, s_l)$.

Definition 5 (Complexity of a TCP stream) *Let* $\sigma = \langle s_1, \dots, s_n \rangle$ *be a TCP stream. Define* $complexity(\sigma) \equiv (length(\sigma), disorder(\sigma))$.

1. *We say that* $complexity(\sigma) = complexity(\tau)$ *if and only if* $\sigma = \tau$.
2. *We say that* $complexity(\sigma) < complexity(\tau)$ *if and only if* $(length(\sigma) < length(\tau))$ *or* $(length(\sigma) = length(\tau) \wedge (disorder(\sigma) < disorder(\tau))$.

We say that σ *is less complex than* τ, *denoted* $\sigma \prec \tau$, *if* $complexity(\sigma) < complexity(\tau)$.

Note that *complexity* is a partial order; it ranks streams using *length* as the primary index and *disorder* as a secondary one. As required in Section 4.2.1, the down set of *complexity* is finite. Intuitively, this means that we cannot simplify an attack instance infinitely; at some point, we derive an atom that cannot be simplified anymore. Furthermore, *complexity* is length preserving as required by Theorem 2, so Algorithm 2 is sound and k-complete with respect to our rules (Table 1).

5.2.2 Shrinking and Expanding Transformations.

We prove that applying an expanding transformation increases the complexity of an instance.

1. Let $length(\sigma) = (n, k)$. Assume $\sigma \xrightarrow{frag^+} \tau$. Since *frag*$^+$ adds a segment to a stream, then $length(\tau) = (n+1, k)$. So, regardless the values of $disorder(\tau)$ and $disorder(\sigma)$, $complexity(\sigma) < complexity(\tau)$, that is, $\sigma \prec \tau$.
2. Let $length(\sigma) = (n, k)$. Assume $\sigma \xrightarrow{http^+} \tau$. Since *http*$^+$ increases the length of a payload, then $length(\tau) = (n, k+2)$. Note that $disorder(\tau) = disorder(\sigma)$, so $complexity(\sigma) < complexity(\tau)$, that is, $\sigma \prec \tau$.
3. Let $\sigma = \langle s_1 \dots p \dots q \dots s_n \rangle$. Assume $\sigma \xrightarrow{swap^+} \tau$ and $\tau = \langle s_1 \dots q \dots p \dots s_n \rangle$ such that $\sigma[j] = \tau[k] = p$ and $\sigma[k] = \tau[j] = q$. From the definition of *swap*$^+$ we know that $p.seq < q.seq$ (Table 1). Note the following properties:
   (a) For all i such that $i > k$, $not\_in\_order(\sigma, q, s_i) = 1$ if and only if $not\_in\_order(\tau, q, s_i) = 1$.

(b) For all i such that $i > k$, $\mathit{not\_in\_order}(\sigma, p, s_i) = 1$ if and only if $\mathit{not\_in\_order}(\tau, p, s_i) = 1$.

(c) For all i such that $i < j$, $\mathit{not\_in\_order}(\sigma, s_i, p) = 1$ if and only if $\mathit{not\_in\_order}(\tau, s_i, p) = 1$.

(d) For all i such that $i < j$, $\mathit{not\_in\_order}(\sigma, s_i, q) = 1$ if and only if $\mathit{not\_in\_order}(\tau, s_i, q) = 1$.

(e) For all i such that $j < i < k$:

   i. Assume $p.seq < s_i.seq < q.seq$. Then $\mathit{not\_in\_order}(\sigma, p, s_i) = 0$ and $\mathit{not\_in\_order}(\sigma, s_i, q) = 0$. However, $\mathit{not\_in\_order}(\tau, s_i, p) = 1$ and $\mathit{not\_in\_order}(\tau, q, s_i) = 1$. This means that the swap operation contributes to the value of $\mathit{disorder}(\tau)$.

   ii. Assume $p.seq < q.seq < s_i.seq$. Then $\mathit{not\_in\_order}(\sigma, p, s_i) = 0$ and $\mathit{not\_in\_order}(\sigma, s_i, q) = 1$, but $\mathit{not\_in\_order}(\tau, s_i, p) = 1$ and $\mathit{not\_in\_order}(\tau, q, s_i) = 0$. This means that $\mathit{disorder}(\tau)$ is at least as $\mathit{disorder}(\sigma)$.

   iii. Assume $s_i.seq < p.seq < q.seq$. Then $\mathit{not\_in\_order}(\sigma, p, s_i) = 1$ and $\mathit{not\_in\_order}(\sigma, s_i, q) = 0$, but $\mathit{not\_in\_order}(\tau, s_i, p) = 0$ and $\mathit{not\_in\_order}(\tau, q, s_i) = 1$. This means that $\mathit{disorder}(\tau)$ is at least as $\mathit{disorder}(\sigma)$.

   iv. Note that other orderings of p, q and s_i are impossible because $p.seq < q.seq$.

(f) Since we used swap^+, $\mathit{not\_in\_order}(\sigma, p, q) = 0$ but $\mathit{not\_in\_order}(\tau, q, p) = 1$.

From properties (a) to (f), we conclude that $\mathit{disorder}(\tau) \geq \mathit{disorder}(\sigma) + 1$. Since the swap^+ transformation does not change the length of a stream, $\mathit{complexity}(\sigma) < \mathit{complexity}(\tau)$ and $\sigma \prec \tau$.

Proofs that shrinking transformations reduce complexity is analogous to the proofs above. Hence, our complexity order is suitable for a uniform and reversible attack deduction system.

5.2.3 Uniform Derivations.

As mentioned at the beginning of Section 5.2, our goal is show that if $\sigma \overset{\Phi}{\Rightarrow} \tau$ then there exists a uniform derivation from σ to τ.

We prove uniformity by induction on the number of *uniformity violations* in a derivation; a uniformity violation is an occurrence of an expanding transformation before a shrinking one. Intuitively, the proof converts a long derivation into a uniform one by replacing the original subsequences that violate uniformity with uniform derivations of length two. Since the proof of the induction base requires a case-by-case analysis of all derivation sequences of length two, we first present the induction-step.

Definition 6 (Uniformity Violation) *Let $\langle r_1, \ldots, r_n \rangle$ be a derivation. The Uniformity Violation of $\langle r_1, \ldots, r_n \rangle$, denoted $UV(\langle r_1, \ldots, r_n \rangle)$, is the number of subsequences $\langle r_i, r_{i+1} \rangle$ such that $r_i \in \Phi^+$ and $r_{i+1} \in \Phi^-$.*

Induction hypothesis. Let $\langle r_1, \ldots, r_n \rangle$ be a derivation from σ to τ. Then, there exists a corresponding uniform derivation from σ to τ.

Induction step. Assume that the induction hypothesis holds for any sequence with UV equals N. Let $\langle r_1, \ldots, r_m \rangle$ be a derivation such that $UV(\langle r_1, \ldots, r_m \rangle) = N+1$. Let $\langle r_i, r_{i+1} \rangle$ be the first subsequence such that $UV(\langle r_i, r_{i+1} \rangle) = 1$, denote $\sigma_i \xrightarrow{r_i} \sigma_{i+1} \xrightarrow{r_{i+1}} \sigma_{i+2}$.

According to the induction base, there exists a uniform derivation from σ_i to σ_{i+2}. Denote this sequence $\langle \bar{r}_1, \ldots, \bar{r}_k \rangle$. Now, consider the sequence $\langle r_1, \ldots, r_{i-1}, \bar{r}_1, \ldots, \bar{r}_k, r_{i+2}, \ldots, r_m \rangle$. This sequence derives τ from σ and its UV equals N. So, according to our hypothesis, this sequence has a corresponding uniform derivation. □

Induction base. We show that for any derivation of the form $\sigma \xrightarrow{r^+} \rho \xrightarrow{r^-} \tau$ there exist a uniform derivation from σ to τ.

Our induction-base proof requires the assumption that segments in a TCP stream do not overlap (otherwise, Claim 4 and Claim 5 below do not hold). Essentially, each byte in a TCP stream has a unique identifier: the sequence number of the byte's segment plus the byte's index within the segment. We say that segments of a TCP stream do not overlap if each byte in the stream has a unique identifier. Note that

σ	ρ	τ	**We know that:**	**Equivalent uniform derivation:**
$\langle p,q,r\rangle$	$\langle r,q,p\rangle$	$\langle p,q,r\rangle$	*swap*$^+$ and *swap*$^-$ reverse each other.	empty derivation.
$\langle p,q,r\rangle$	$\langle p,r,q\rangle$	$\langle p,q,r\rangle$		
$\langle p,q,r\rangle$	$\langle q,p,r\rangle$	$\langle p,q,r\rangle$		
$\langle p,q,r\rangle$	$\langle q,p,r\rangle$	$\langle q,r,p\rangle$	$r.seq < p.seq < q.seq$	$\langle p,q,r\rangle \xrightarrow{swap^-} \langle r,q,p\rangle \xrightarrow{swap^+} \langle q,r,p\rangle$
$\langle p,q,r\rangle$	$\langle q,p,r\rangle$	$\langle r,p,q\rangle$	$(p.seq < q.seq) \wedge (r.seq < q.seq)$	$\langle p,q,r\rangle \xrightarrow{swap^-} \langle p,r,q\rangle \xrightarrow{swap^{+/-}} \langle r,p,q\rangle$
$\langle p,q,r\rangle$	$\langle r,q,p\rangle$	$\langle r,p,q$	$(p.seq < r.seq) \wedge (p.seq < q.seq)$	$\langle p,q,r\rangle \xrightarrow{swap^-} \langle q,p,r\rangle \xrightarrow{swap^{+/-}} \langle r,p,q\rangle$
$\langle p,q,r\rangle$	$\langle r,q,p\rangle$	$\langle q,r,p\rangle$	$p.seq < r.seq < q.seq$	$\langle p,q,r\rangle \xrightarrow{swap^-} \langle p,r,q\rangle \xrightarrow{swap^+} \langle q,r,p\rangle$
$\langle p,q,r\rangle$	$\langle p,r,q\rangle$	$\langle q,r,p\rangle$	$(q.seq < r.seq) \wedge (q.seq < p.seq)$	$\langle p,q,r\rangle \xrightarrow{swap^-} \langle q,p,r\rangle \xrightarrow{swap^{+/-}} \langle q,r,p\rangle$
$\langle p,q,r\rangle$	$\langle p,r,q\rangle$	$\langle r,p,q\rangle$	$q.seq < r.seq < p.seq$	$\langle p,q,r\rangle \xrightarrow{swap^-} \langle r,q,p\rangle \xrightarrow{swap^+} \langle r,p,q\rangle$

Table 2: Uniform derivation for the non-uniform derivation: $\sigma \xrightarrow{swap^+} \rho \xrightarrow{swap^-} \tau$. *swap*$^{+/-}$ denotes that either *swap*$^+$ or *swap*$^+$ are possible.

our rules (Table 1) do not create overlapping segments; given an non-overlapping TCP stream they always return a stream with non-overlapping segments. In Section 5.3, we discuss how to prove the uniformity of the TCP-retransmission transformation that does create overlapping TCP segments.

Claim 2 *There exists a uniform derivation for the sequence* $\sigma \xrightarrow{swap^+} \rho \xrightarrow{swap^-} \tau$.

Proof: The only non-trivial case for this derivation is when the two rules swap three segments of a TCP stream. Table 2 presents uniform derivation for all possible derivations involving three segments.

Claim 3 *There exist a uniform derivation for the sequence* $\sigma \xrightarrow{frag^+} \tau \xrightarrow{frag^-} \rho$.

Proof: Assume that $\langle s,r\rangle \xrightarrow{frag^+} \langle s_1,s_2,r\rangle \xrightarrow{frag^-} \langle s_1,s_2\cdot r\rangle$ (we denote $s\cdot r$ as the defragmentation of s and r, see definition 7). Remember that fragmentation (*frag*$^+$) is analogous to string splitting while defragmentation (*frag*$^+$) is analogous to string concatenation. Hence, like in the case of strings, we can reverse the order of the two operations.

Claim 4 *There exist a uniform derivation for the sequence* $\sigma \xrightarrow{frag^+} \tau \xrightarrow{swap^-} \rho$.

Proof:

1. Assume that $\langle q,s\rangle \xrightarrow{frag^+} \langle q,s_1,s_2\rangle \xrightarrow{swap^-} \langle s_1,q,s_2\rangle$. Then $s_1.seq = s.seq < q.seq$. Hence, $\langle q,s\rangle \xrightarrow{swap^-} \langle s,q\rangle \xrightarrow{frag^+} \langle s_1,s_2,q\rangle \xrightarrow{swap^+} \langle s_1,q,s_2\rangle$ (we know that $q > s_2.seq$ because of the non-overlapping assumption).

2. Assume that $\langle q,s\rangle \xrightarrow{frag^+} \langle q,s_1,s_2\rangle \xrightarrow{swap^-} \langle s_2,s_1,q\rangle$. Then $s.seq = s_1.seq < s_2.seq < q.seq$. Then $\langle q,s\rangle \xrightarrow{swap^-} \langle s,q\rangle \xrightarrow{frag^+} \langle s_1,s_2,q\rangle \xrightarrow{swap^+} \langle s_2,s_1,q\rangle$.

Claim 5 *There exist a uniform derivation for the sequence* $\sigma \xrightarrow{swap^+} \tau \xrightarrow{frag^-} \rho$.

Proof:

1. Assume $\langle s,r,q\rangle \xrightarrow{swap^+} \langle r,s,q\rangle \xrightarrow{frag^-} \langle r,s\cdot q\rangle$. Then $(s.seq < r.seq) \wedge (q.seq = s.seq + size\_of(s.payload))$. Since in our deduction systems TCP segments cannot overlap we get: $r.seq > q.seq$. Then $\langle s,r,q\rangle \xrightarrow{swap^-} \langle s,q,r\rangle \xrightarrow{frag^-} \langle s\cdot q,r\rangle \xrightarrow{swap^+} \langle r,s\cdot q\rangle$

2. Assume $\langle s,r,q\rangle \xrightarrow{swap^+} \langle s,q,r\rangle \xrightarrow{frag^-} \langle s\cdot q,r\rangle$. Then $(r.seq < q.seq) \wedge (q.seq = s.seq + size\_of(s.payload))$. Since TCP segments do not overlap we get: $r.seq < s.seq$. Then: $\langle s,r,q\rangle \xrightarrow{swap^-} \langle r,s,q\rangle \xrightarrow{frag^-} \langle r,s\cdot q\rangle \xrightarrow{swap^+} \langle s\cdot q,r\rangle$

3. Assume $\langle s, r, q\rangle \xrightarrow{swap^+} \langle q, r, s\rangle \xrightarrow{frag^-} \langle q, r{\cdot}s\rangle$. Then $(s.seq < q.seq) \wedge (s.seq = r.seq{+}size\_of(r.payload))$. Then $\langle s, r, q\rangle \xrightarrow{swap^-} \langle r, s, q\rangle \xrightarrow{frag^-} \langle r \cdot s, q\rangle \xrightarrow{swap^{-/+}} \langle q, r \cdot s\rangle$.

Claim 6 *There exist a uniform derivation for the sequence* $\sigma \xrightarrow{http^+} \tau \xrightarrow{frag^-} \rho$.

Proof: Assume $\langle s, r\rangle \xrightarrow{http^+} \langle s', r\rangle \xrightarrow{frag^-} \langle s' \cdot r\rangle$ and $url\_encode(s) = s'$. Since $http^+$ works on a payload of a segment, it can work also on a larger payload. So, we can reassemble first and then apply the http encoding. $\langle s, r\rangle \xrightarrow{frag^-} \langle s \cdot r\rangle \xrightarrow{http^+} \langle s' \cdot r\rangle$.

Claim 7 *There exist a uniform derivation for the sequence* $\sigma \xrightarrow{http^+} \tau \xrightarrow{swap^-} \rho$.
Proof: Clearly, we can swap first and then apply the http encoding.

5.3 Modeling Other Transformations

In Section 5.2, we proved that our system (Table 1) is uniform and reversible. Here, we discuss the uniformity and reversibility of transformations that are not part of our system. Since these additional transformations are similar in nature to the transformations in our system, we only provide informal arguments for the uniformity of these additional transformations.

Modeling other TCP transformations. We identified two additional types of TCP transformations: *header change* and *TCP retransmission.*

Header change transformations operate on the header of a TCP segment; for example, they modify the TCP flags [7, 19]. The uniformity proof of such transformations, is similar to the proofs we provided for the TCP transformations in our system. First, we extend the representation of a TCP stream (Section 5.2.1) by adding a TCP header in each segment. Second, we define a partial order for TCP headers. For example, we define setting the RESET TCP flag [7] as increasing the instance complexity and unsetting it as decreasing the complexity. Third, we define *complexity* (Definition 5) as the order imposed by *length* (Definition 3), *disorder* (Definition 4), and the new order for TCP headers. Last, to prove uniformity, we extend our induction base (Section 5.2.3) to include short sequences that contain header change rules.

Modeling TCP retransmission is more challenging. Essentially, TCP retransmission complicates an attack instance by resending a TCP segment [19, 22]. Hence, we model *expanding-retransmission* as a transformation that duplicates a TCP segment, and *shrinking-retransmission* as a transformation that removes a duplicated one. Unfortunately, expanding-retransmission violates our non-overlapping assumption, so it breaks our uniformity proof (e.g., Claim 4).

TCP-retransmission breaks uniformity only when it is mixed with other rules (e.g., $frag^+$). Hence, to convert a non-uniform derivation that contains TCP-retransmissions into a uniform one, we separate them from the other transformations: in the uniform derivation we group all the shrinking-retransmissions at the beginning of the derivation and all the expanding-retransmissions at the end. More particulary, to construct a uniform derivation from σ to τ, we first use shrinking-retransmissions to remove all duplicated segments from σ. Then, we shrink the resulting instance into an atom and expand the atom without using expanding-retransmissions. Last, we use expanding-retransmissions to duplicate segments in a way that matches τ.

Modeling application-level transformations. Application-level transformations operate on the attack payload. For example, *FTP padding* [12, 22] adds benign commands before the malicious commands of an FTP attack. Such transformations are analogous to our $http^{+/-}$ transformations (Table 1), which also add or remove bytes from the payload. Hence, we can prove their uniformity in the same way we proved the uniformity of $http^{+/-}$.

Modeling network-level transformation. Network-level transformations (e.g., IP, UDP) change the way the attack is delivered; for example, IP transformations [19] might split IP packets. Such transfor-

Attack (size in bytes)	Max pack-ets/ instance	Execution time (sec)		Memory consumption (MB)	
		Naive	Forensics	Naive	Forensics
finger (8)	8	37	0.05	35	1
perl-in-cgi (30)	30	10800[a]	0.11	>1500[a]	1.2
pro-ftpd(1500)	150	10800[a]	16	>1500[a]	3

[a]Did not terminate after three hour.

Table 3: Solving the forensics problem. Comparing the performance of a naive algorithm to that of our forensics algorithm (Algorithm 3).

mations are similar in nature to our TCP transformations and their uniformity proofs are similar to the uniformity proofs of the TCP transformations above.

6 Empirical Evaluation

We empirically evaluate the execution time and memory consumption of our algorithms. To do so, we compare the algorithms to a naive closure-generation algorithm, an algorithm that recursively applies the rules until no new attack instances are found[2]. First, we use this naive algorithm to solve the forensics problem (as discussed in Section 4.1.2). We show that the naive algorithm does not scale beyond attacks with more than 20 bytes, while our forensics algorithm (Algorithm 3) can handle attack with thousands of bytes. Second, we applied the naive algorithm to generate a closure; we compared its performance to the performance of our testing algorithm (Algorithm 2). Although both algorithms require exponential memory to store the closure, we show that our testing algorithm is faster because it only uses the expanding rules during closure generation.

We implemented all algorithms using XSB [28], a highly optimized evaluation engine for Prolog programs. Unlike standard Prolog [26], XSB uses tabled resolution [2] which is useful for closure generation; it allows recursive programs to terminate correctly in many cases where Prolog does not. Essentially, the evaluation engine of XSB implements the naive algorithm: to avoid derivation cycles, the engine store each instance it derives in an internal data structure. All the experiments were performed on a 2.4GHz Pentium 4 processor with 1Gb of RAM.

The Forensics Experiment. To illustrate the performance gap between the naive and forensics algorithms, we performed the experiment using the rules $\{frag^{+/-}, swap^{+/-}\}$. We performed the experiment three times using three different attacks: the *finger-root* attack (CVE-1999-0612 [15]) that requires up 10 bytes, *perl-in-cgi* (CAN-1999-0509 [15]) that requires up to 30 bytes, and the *pro-ftpd* attack (CAN-2003-0831 [15]) that requires almost 1500 bytes. Using three different attacks enables us to compare the performance of the algorithms over attacks with considerably different lengths.

The experiment comprises two steps. First, we manually generated two random instances of each attack; then, we measured the average time and memory it took for the naive and forensics algorithms to determine that these instances are equivalent. Even though we know that these instances are equivalent, this experiment provides insight to the performance of the algorithms. We also repeated the experiment with pairs of instances that are not equivalent, and got similar results.

Table 3 presents our measurements for this experiment. The naive algorithm does not scale, both in terms of execution time and memory consumption, beyond short attacks. However, our forensics algorithm can efficiently handles even long attacks.

[2]Since in these experiments, the closure is finite, this algorithm terminates.

The Testing Experiment. We used the naive algorithm and our testing algorithm (Algorithm 2) to generate the closure of the finger-root attack with respect to our rules. The naive algorithm generates the closure in 297 seconds while it took only 215 seconds for our testing algorithm. The memory consumption of both algorithms was the same (35Mb) since they store the closure in memory. The testing algorithm is superior because it only uses the expanding rules rather than the expanding and shrinking. We believe that the memory consumption of our testing algorithm can be further improved, because we can apply the rules in a way that avoids derivation cycles. When we can prevent derivation cycles, we no longer need to store the closure in memory. However, this optimization is left for future work.

7 Conclusion

Attack mutation is an effective method for NIDS testing. We formally investigate the underlying principle of attack mutation, the idea that attack instances are derivable from a few exemplary ones. We prove that when the transformations are uniform and reversible, all attack instances are derivable from a few atoms. We show that common transformation are indeed uniform and reversible. Therefore, the algorithms we developed can be immediately deployed in current NIDS testing tools.

References

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

[2] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, **43**(1), January 1996.

[3] Y. Chevalier, R. Ksters, M. Rusinowitch, and M. Turuani. An NP decision procedure for protocol insecurity with XOR. In *IEEE Symp. on Logic in Computer Science*, Ottawa, Canada, June 2003.

[4] E. M. Clarke, S. Jha, and W. R. Marrero. Verifying security protocols with Brutus. *ACM TOPLAS*, **9** (4), Oct. 2000.

[5] H. Comon-Lundh and V. Shmatikov. Intruder deductions, constraint solving and insecurity decision in presence of exclusive or. In *IEEE Symp. on Logic in Computer Science*, Ottawa, Canada, June 2003.

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

[7] 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.

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

[9] L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson. CSI/FBI computer crime and security survey. Computer Security Institute, 2004.

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

[11] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, **110**, 1994.

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

[13] C. Meadows. The NRL protocol analysis tool: A position paper. In *IEEE Computer Security Foundations Workshop*, Franconia, NH, June 1991.

[14] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, **51**, 1991.

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

[16] 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, Dec. 2003.

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

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

[19] 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.

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

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

[22] S. Rubin, S. Jha, and B. P. Miller. Automatic generation and analysis of NIDS attacks. In *Annual Computer Security Applications Conference*, Tucson, AZ, Dec. 2004.

[23] N. Shankar. Proof search in the intuitionistic sequent calculus. In *Proceedings of 11th International Conference on Automated Deduction (CADE-11)*, Saratoga Springs, NY, June 1992.

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

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

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

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

[28] The XSB Research Group. The XSB system, version 2.7. Available at `xsb.sourceforge.net`.

[29] 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, Oct. 2004.

A Definitions Used Throughout the Paper

Definition 7 (Fragmentation of a TCP Segment) *Let* $s = \{seq, payload\}$ *be a TCP segment.* $seg\_frag(s) = \langle s_1, s_2 \rangle$ *such that*

1. $s_1.seq = s.seq$,
2. $payload = s_1.payload \cdot s_2.payload$,
3. $s_2.seq = s.seq + size\_of(s_1.payload)$.

Note that if a TCP stream contains the segments s_1 and s_2, and if seg_frag(s)=$\langle s_1, s_2 \rangle$ then the payload of a TCP stream contains the string $s_1.payload \cdot s_2.payload$.

We say that a segment r is a defragmentation of s_1 and s_2 if $seg\_frag(r) = \langle s_1, s_2 \rangle$.

Definition 8 (URL encoding) *Let* $s = \{seq, payload\}$ *be a TCP segment, let payload be the string* $a_1 \cdot a_2 \cdots a_n$, *and let* a_i *by part of a URL.*

Then: $url\_encode(s) = \{seq, a_1 \cdot \ldots a_{i-1} \cdot \%h \cdot a_{i+1} \ldots a_n\}$ *where* h *is the hexadecimal value of the ASCII character* a_i.

Technically, to formally express that a_i *is part of a URL, we can use two alphabets: one for URLs and one for the other parts of the attack.*