USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

From Intention to Practice: Towards Systematic Validation of NIDS Rule Enforcement

Huan Liu, *Huazhong University of Science and Technology;* Haoyu Chen, *Zhejiang Lab;*
Biang Xu, *Huazhong University of Science and Technology and Jinyinhu Laboratory;*
Jingyao Zhou, *Huazhong University of Science and Technology;*
Bin Yuan, *Huazhong University of Science and Technology and Songshan Laboratory;*
Qiankun Zhang, *Huazhong University of Science and Technology;*
Deqing Zou, *Huazhong University of Science and Technology and Jinyinhu Laboratory;*
Hai Jin, *Huazhong University of Science and Technology*

https://www.usenix.org/conference/nsdi26/presentation/liu-huan

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

From Intention to Practice: Towards Systematic Validation of NIDS Rule Enforcement

Huan Liu[1,*,‡], Haoyu Chen[2,*], Biang Xu[1,3], Jingyao Zhou[1,‡], Bin Yuan[1,4,†,‡], Qiankun Zhang[1], Deqing Zou[1,3,‡], and Hai Jin[5,‡]

[1]*School of Cyber Science and Engineering, Huazhong University of Science and Technology, China*
[2]*Zhejiang Lab, China*
[3]*Jinyinhu Laboratory, China*
[4] *Songshan Laboratory, China*
[5]*School of Computer Science and Technology, Huazhong University of Science and Technology, China*

Abstract

Rule-based Network Intrusion Detection Systems (NIDS) are integral to contemporary cybersecurity, relying on the rule matching mechanism to identify malicious activities within network traffic. However, there is no inherent assurance that the deployed rules are enforced as intended due to factors regarding the composition of the rules and implementation flaws of NIDS. Unfortunately, administrators lack appropriate means to validate the gap between rule definition and enforcement as existing testing approaches towards NIDS are often rule irrelevant and lack systematic methodologies. To address this issue, this paper presents **NIDSFuzz**, a systematic fuzzing approach designed to validate the enforcement of rules within NIDS, which is rule-oriented so that it employs tailored mutation strategies to generate test traffic based on the very ruleset deployed. In this manner, it becomes feasible to validate the targeted rulesets with guarantee of coverage. An NIDS-specific fuzzing framework is proposed, incorporating an appropriate test traffic injection method to perform fuzzing and carefully designed approaches of sanitization and analysis to effectively identify rule enforcement issues. Experimental results show that **NIDSFuzz** is able to uncover over 10,000 rule enforcement issues. We classified the discovered issues into different categories and explored corresponding countermeasures in terms of both rules and NIDS implementation. Moreover, performance evaluation confirms the efficiency of **NIDSFuzz** and comparison to other tools highlights the significant advantage of **NIDSFuzz** in evaluating rules of NIDS. We have made our code publicly available.

1 Introduction

Rule-based Network Intrusion Detection Systems (NIDS) plays a pivotal role in the detection and prevention of attacks relying on the *rule matching mechanism*, which employs a set of predefined rules that contain signatures of vulnerabilities to classify network traffic and identify indicators of compromise. Typically, administrators either author these rules manually, or select from rulesets that are continuously updated by domain experts to reflect emerging threats and ensure applicability to specific network environments [6].

However, there is no inherent guaranty that these rules would be enforced in accordance with the intention by which they are designed or selected. The gap between intention and enforcement of rules could be induced due to both rule- or implementation-related issues. On the one hand, definition of a rule requires meticulous attention to detail and strict adherence to syntactic and structural conventions. Minor inadvertent errors, such as overlooking the significance of whitespace within matching patterns [25, 34], can fundamentally undermine the intended detection function. Not only that, poorly designed rules can be evaded by crafted packets that alter the appearance of malicious payloads to fool the rule matching engine [40]. On the other hand, interactions between rules [31, 35, 44, 53] can introduce unexpected consequences, where a rule that performs correctly in isolation may fail when integrated into a broader ruleset, due to conflicts or overlapping issues. For instance, a flood of false alarms that overwhelm network administrators so that they cannot cope with the attacks undergoing in time [36, 55] can be achieved by arbitrarily triggering multiple detection rules at the same time, which is especially exaggerated when a packet matched by one rule can also trigger other rules. Furthermore, *implementation flaws* are another impediment to effective rule enforcement. NIDS becomes incapable of evaluating packets against a rule in a required time window due to performance degradation even if the ruleset is syntactically and semantically correct, despite coding errors. This is especially common in regular expressions and string sanitization [37] when the algorithms meet the worst-case of complexity due to poor rule designs or algorithm flaws [42, 43]. Moreover, NIDS may have inconsistent implementations in system behaviors compared to communication endpoints that unexpectedly compro-

* Both authors contributed equally to this work.
† Bin Yuan is the corresponding author, and is acting as a visiting researcher with the Lion Rock Labs of Cyberspace Security, Hong Kong, China.
‡ Hubei Engineering Research Center on Big Data Security, Hubei Key Laboratory of Distributed System Security, National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab.

mise the overall efficacy of the intrusion detection process. For instance, inconsistencies in the protocol parsing logic between NIDS and the communication endpoints could result in malicious traffic deemed invalid by NIDS that requires no evaluation while still functions normally to exert attacks at the endpoints [17,32,39,51]. Therefore, a crucial point stands out naturally: *How to validate whether rules deployed can be enforced as expected in practice?*

Existing research has not been able to adequately address this point. Some studies have proposed static analysis methods for rulesets, in order to identify potential issues using theoretical models or formal verification methods [31,35,53]. However, even after such analysis, the ruleset may still fail to function correctly during enforcement due to implementation flaws in the NIDS. Additionally, other studies have focused on validating the implementation of NIDS, but are generally concerned with the aspects unrelated to rule enforcement, such as the implementation of modules for parsing packet headers [17] or specific protocols [39]. Several studies also explore the impact of packet mutation on the evaluation of rules, modifying specific packet content to bypass rules [26], thereby causing rule failure. However, these studies are only applicable to validating single-type rule enforcement issue and lack a systematic validation methodology.

To address this issue, we propose **NIDSFuzz**, a systematic rule-oriented fuzzing approach. *Fuzzing* has emerged as a promising testing technology in various domains such as software and network testing. It feeds the targeted system with a large amount of data generated to uncover potential vulnerabilities. However, existing network fuzzing tools are inadequate for NIDS testing as they primarily target specific network protocols rather than NIDS rulesets. This limitation reveals a critical gap in generating effective test traffic that accounts for both protocol diversity and semantic richness inherent in NIDS rules. **NIDSFuzz** adopts the fuzzing technique to generate test traffic based on the very ruleset deployed in NIDS to validate whether the rules are enforced as intended. In the meantime, rule-based mutation strategies are provided to achieve precise and thorough evaluation of rules deployed, while avoiding the generation of redundant and ineffective random traffic that could slow down fuzzing.

As a first step towards systematic fuzzing of NIDS rule enforcement, the design and implementation of **NIDSFuzz** are non-trivial. Some critical challenges have been addressed. First of all, the generated traffic should be capable of conducting comprehensive fuzzing against the targeted ruleset. To achieve this goal, four tailored mutation strategies are provided in **NIDSFuzz**, which contain effective means that reflect practical rule enforcement issues. Moreover, how to efficiently inject traffic into NIDS remains an unsolved problem. NIDS monitors bidirectional traffic from both endpoints of communication. Therefore, bilateral traffic injection is required while none of the existing approaches provide this ability. To solve this problem, **NIDSFuzz** is equipped with an initiator and a responder working together to send requests and responses, respectively. Furthermore, accurately diagnosing rule enforcement issues solely through NIDS-generated alerts presents significant challenges. To address this challenge, **NIDSFuzz** introduces a novel sanitization approach to identify potential rule enforcement issues. This is accomplished through a differential checking mechanism that identifies alert inconsistencies across different NIDS platforms when processing identical rulesets.

Experiments have been carried out to evaluate the prototype implementation of **NIDSFuzz**. Our experimental results proved the effectiveness of **NIDSFuzz** in uncovering over 10,000 rule enforcement issues, which have been analyzed and classified. We have reported the discovered issues to NIDS officials. We also noted that a CVE (CVE-2025-29915) [11] has recently been reported by Suricata that coincides with one issue discovered by **NIDSFuzz**. In addition, performance evaluation shows that **NIDSFuzz** is also efficient in fuzzing.

Our main contributions are summarized as follows:

- **Systematic NIDS rule enforcement fuzzing approach**: An NIDS-dedicated fuzzing framework is proposed for validating rule enforcement, including support of diverse NIDS platforms and rules of different network protocols, tailored mutation strategies for targeted rules, efficient generation and injection of traffic, as well as effective identification of potential rule enforcement issues.
- **Prototype Implementation**: A prototype system of **NIDSFuzz** has been implemented, with thorough evaluations carried out with respect to performance and comparison with other tools. The source code is publicly available [14].
- **Discovery of considerable numbers of issues**: Substantial numbers of issues are uncovered, which prove the effectiveness of **NIDSFuzz**. We have reported those issues, with some of them confirmed by NIDS officials. Also, some countermeasures are proposed.

2 Background

In this section, we introduce background of **NIDSFuzz** to inform the necessary knowledge for ease of understanding the present work, including concise introduction on NIDS and the fuzzing technique used by **NIDSFuzz**.

2.1 Network Intrusion Detection System

NIDS has evolved into a highly complex system [46,51,54]. On the one hand, NIDS must be equipped with the ability of sophisticated Deep Packet Inspection (DPI) to process network traffic [17,43,50,51]; on the other hand, a complex set of detection rules is deployed to strive for advanced, tailored security purposes [16,46,48]. NIDS utilizes DPI to inspect the monitored network traffic and then matches the traffic characteristics against user-configured detection rules. If the traffic complies with the rules, it is deemed suspicious and corresponding alerts are triggered.

Typically, a detection rule consists of two parts: the *rule*

```
1  alert tcp $EXTERNAL_NET any -> $HOME_NET 80
2    (msg: "Attack attempt!";
3     service: http;
4     http_raw_body;
5     content: "malicious payload", fast_pattern;
6     http_uri;
7     pcre: "/[?&]user=(intruder|attacker)/i";
8     gid:1; sid:10000; rev:1;)
```

Figure 1: An example of a fully structured Snort 3 rule with a properly defined rule header (Line 1) and rule options (Line 2-8).

header and the *rule options*. The header dictates which traffic to evaluation, while the options specify how the traffic is evaluated. Some commonly used *Rule options* are listed in Table 4 in Appendix A, which can be classified as below.

- *General options:* These options merely provide basic information about the rule. We use the combination of *gid*, *sid* and *rev* options to identify a rule, which appear in the generated alert in the format of "*[gid:sid:rev]*".
- *Sticky buffer options:* These options denote in which part of the packet the signatures are evaluated. Note that *sticky buffer options* are not mandatory; by default, NIDS evaluates signatures against *pkt_data* (the entire packet) [5].
- *Payload detection options:* These options define patterns indicative of suspicious packets.

Figure 1 presents an example rule from Snort 3 [6] to identify traffic accessing internal hosts on port 80 from external network (Line 1). It evaluates whether the packet payload contains "*malicious payload*" (Line 4-5) and whether the request URI matches the regular expression defined by the *pcre* option (Line 6-7). If both conditions are met, an alert is triggered as "*Attack attempt!*" (Line 2).

2.2 Fuzzing

Fuzzing, a widely adopted software testing technique, has demonstrated exceptional capabilities in uncovering vulnerabilities across network protocol implementation and complex software systems [28, 29, 38, 56, 58].

It operates by automatically generating varying inputs and monitoring behaviors of the system under test for anomalies to identify potential vulnerabilities. Based on the granularity of semantics observed during execution, fuzzing is categorized into three paradigms [30]: black-, white-, and grey-box fuzzing. Black-box fuzzing observes only the input/output behavior of the system, while requiring no knowledge of the internal codes or structure. White-box fuzzing requires access to all information about the target system, including the source code or binary. Grey-box fuzzing utilizes partial information, employing lightweight static analysis on the target system to obtain coarse runtime feedback.

In practice, network administrators only need to focus on the rules enforced in NIDS without knowing any specific implementation details of NIDS. Therefore, the choice comes naturally for **NIDSFuzz** to be designed and implemented in the form of black-box fuzzing.

3 Exploring Fuzzing for NIDS Rule Enforcement

As a first step towards systematic validation of rule enforcement in NIDS, achieving this goal is non-trivial. To inform the general insights, we outline the key requirements confronted in the design and explore their address.

RQ1: How to design test packet generation strategies to achieve comprehensive validation?

It is required to generate test cases that not only guaranty coverage of the rules to be evaluated but also be capable of uncovering potential issues of both the rules and implementation flaws in the NIDS. The former is achieved by a rule-based generation approach and the latter is addressed with tailored mutation strategies.

Existing test case generation approaches [17, 26, 32, 39, 49, 51, 57] are protocol-based, which mutate a specific field in the header or payload of a given packet according to the definitions of the network protocol. In this way, there is little chance that the generated packets can trigger the targeted rules accurately. In contrast, **NIDSFuzz** takes the targeted rules as input and extracts signatures they use to identify attacks as well as other parts necessary for packet generation (Section 4.2). Signatures are then mutated and transformed into concrete packets that adhere to the correct protocol syntax and semantically align with the input signature to ensure that they will not be prematurely discarded by the NIDS. Furthermore, tailored mutation strategies are proposed (Section 4.3). These strategies reflect the means that can trigger rule enforcement issues of both rule and implementation flaws, which have been shown to be effective in experiments.

RQ2: How to efficiently inject test traffic into a NIDS?

Traditional test case injection techniques involve only unidirectional traffic, like in firewall testing [49, 52]. However, as an all-round player of traffic surveillance, NIDS monitors bidirectional communications between endpoints, including requests and responses. This necessitates injecting bidirectional test traffic into NIDS without incurring significant overhead.

A strawman solution is to use real protocol programs (e.g., HTTP clients or servers) as endpoints, which is also adopted in [19, 29]. However, it leads to limited fuzzing efficiency and lacks flexibility. On the one hand, our goal is to rapidly perform the validation without executing packet processing logic; yet real protocol programs inherently process incoming packets, resulting in extra time overhead. On the other hand, it is cumbersome to modify the source code of real programs in order to tune their response as needed.

To address this issue, **NIDSFuzz** proposes a tunable traffic injection method (Section 4.4), which employs an initiator and a responder that are responsible for injecting requests and responses, respectively. Specifically, the initiator collects both the request and response generated in a given test case, and passes the response to the responder sideways. Then the responder simply forwards back the response to the initiator

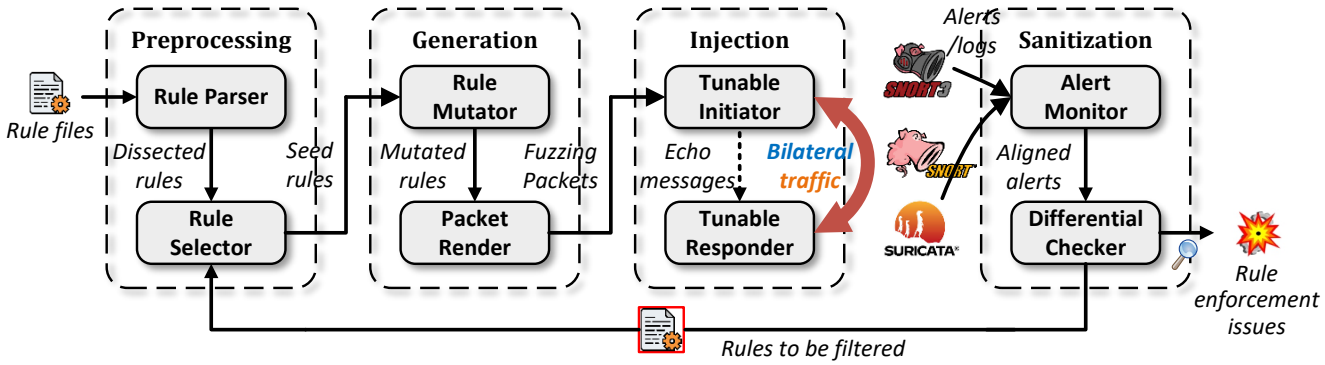

Figure 2: The workflow of **NIDSFuzz**.

through the network pathway monitored by NIDS to mimic full communication session. In this way, extra time overhead induced by real programs for processing the request and generating responses has been waived. In addition, it becomes more flexible that the response can be conveniently tuned.

RQ3: How to identify potential rule enforcement issues based on alerts triggered by NIDS?

Sanitizers contribute to informing abnormalities in fuzzing, such as AddressSanitizer [41, 58] identifies data corruption and memory overflows in the context of memory-related fuzzing. Prior studies [17, 26, 32, 57] concerning network fuzzing implement the sanitizers at the communication endpoints to discover exceptions, for instance, by identifying dropping or injection of packets in requests and responses. Unfortunately, they do not apply to **NIDSFuzz** since NIDS is transparent to the communication endpoints that it does not interfere with but just monitor the network traffic. Therefore, observing the alerts becomes the only measure to perform sanitization in NIDS, which requires new insights.

To achieve this, we propose employing a sanitizer (Section 4.5) to automatically identify potential rule enforcement issues, which works as follows. The sanitizer serves based on a differential checking mechanism. First, different NIDS with the same ruleset deployed within are fed with the identical testing traffic. Then the sanitizer monitors their alerts and uses an alignment algorithm to associate each alert with the exact packet that triggers it. The sanitizer analyzes the alerts across different NIDS for discrepancies. Any inconsistency in alert generation—such as divergent alerts or missing detections—indicates potential rule enforcement issues in the rules responsible for the test traffic. Rules flagged for alert inconsistencies will be automatically excluded from the active ruleset used in subsequent fuzzing iterations. This exclusion prevents redundant testing of problematic rules, thereby enhancing the overall efficiency of the fuzzing process.

4 Design and Implementation

In this section, we demonstrate the detailed designs and implementations of **NIDSFuzz** in workflow-wise. A prototype has been implemented for **NIDSFuzz** [14], which supports evaluation towards three representative NIDS that have publicly available rulesets (i.e., Snort3 [6], Snort2 [3], and Suricata [8]) among rulesets of a variety of network protocols, including HTTP, FTP, DNS, SIP, etc. It is worth noting that NIDSs are deployed behind the perimeter of network in practice (e.g., proxies, load balancers) where the encryption/decryption services (e.g., TLS) locate. Therefore, it actually inspects the traffic in plaintext. This explains why the rules of HTTP make up the largest portion of expert-crafted rulesets and are most actively maintained instead of encrypted protocols (e.g., HTTPS). Therefore, we exemplify the HTTP protocol for demonstrating the design and implementation rationales.

4.1 Overview

Figure 2 illustrates the workflow of **NIDSFuzz**, which can be functionally divided into four phases.

- **Preprocessing :** This phase parses the ruleset under validation. The *Rule Parser* (Section 4.2.1) dissects each rule and represents it with a specific data structure for use in the subsequent phases. The *Rule Selector* (Section 4.2.2) then selects the detection rules for the coming mutation and generation based on particular aspects.
- **Generation :** This phase aims to derive packets of mutations from the selected rules. The *Rule Mutator* (Section 4.3.1) imposes specific mutations on the detection rules. The *Packet Render* (Section 4.3.2) then generates bilateral test packets that conform to the grammars of the given network protocol based on the mutations.
- **Injection :** This phase focuses on injecting the bilateral test packets into the NIDS. To this end, a *Tunable Initiator* and *Tunable Responder* (Section 4.4) are deployed to transmit packets of requests and responses through the network pathway that NIDS is monitoring, respectively.
- **Sanitization :** This phase is tasked with identifying suspicious alerts as rule enforcement issues. The *Alert Monitor* (Section 4.5.1) gathers alerts generated by different NIDS. The *Differential Checker* (Section 4.5.2) then identifies the inconsistencies in the gathered alerts, which would be reported as rule enforcement issues. The related rules will be filtered out in the following iterations to avoid unnecessary repetitive evaluation.

4.2 Preprocessing Phase

In this phase, **NIDSFuzz** implements two modules named *Rule Parser* that parses rules in the given ruleset, and *Rule Selector* that selects rules for the coming generation phase.

4.2.1 Rule Parser

The *Rule Parser* parses the syntax of the detection rules in a given ruleset. Although the rule syntax differs slightly between various NIDS, they share a similar *header-body* structure [1]. Notably, a rule can be either activated or commented on, depending on whether the rule line starts with #. The *Rule Parser* only parses the activated rules.

Buffer Constraints. A detection rule usually contains multiple signatures combined to denote the payload of malicious packets. The *Rule Parser* utilizes a data structure named *buffer constraint* to describe each signature specified in a rule. In essence, a *buffer constraint* is implemented as a dictionary item where the key is a *sticky buffer option*, and the value is a list of *payload detection options*. For example, the signatures of the rule presented in Figure 1 are parsed and described as two *buffer constraints*, as follows.

http_raw_body : { *content* : ["malicious payload", fast_pattern]},
http_uri : { *pcre* : "/[?&]user=(intruder|attacker)/i" }

The first constraint applies to *http_raw_body* with a *content* option. The second constraint applies to *http_uri*, containing a *pcre* option. These *buffer constraints* will be mutated and translated into concrete packets in the subsequent phase.

4.2.2 Rule Selector

The *Rule Selector* filters rules in which the *service* option contains "*http*" or the *rule header* includes ports 80 or 8080 in terms of HTTP protocol. Additionally, the *Rule Selector* identifies activated rules, as commented-out rules are not enforced in the NIDS and thus do not require testing. These selected rules will serve as seeds to perform fuzzing.

Selection Algorithms. The *Rule Selector* provides a set of selection algorithms that return a specified number of rules per call. Callings for all selection algorithms share the same interface with different parameters. Specifically, the interface takes three parameters: ① the rule pool that includes all selected rules, ② the number of rules to be returned per iteration and ③ whether to repeat the selection algorithm. Three selection algorithms are implemented to pick up rules for mutation, among which the *sequential* and *random* selection can be used to evaluate a single rule while the *combination* selection can be used for testing a given number of rules simultaneously:

- *Sequential Selection* : This algorithm selects a specified number of rules in the order they appear in the file;
- *Random Selection* : This algorithm first shuffles the order of rules in the rule pool, and then selects and returns a specified number of rules in sequence;
- *Combination Selection* : This algorithm combines all rules based on the specified number and returns a combination at a time, where the order of the rules does not matter (i.e., $\{A,B,C\}$ and $\{C,B,A\}$ are considered the same combination);

[1] There are some open-source tools that feature the conversion of rules between different NIDS [7]. In addition, **NIDSFuzz** also provides a script that adapts the Snort 3 rules to the corresponding Suricata rules.

4.3 Generation Phase

In this phase, **NIDSFuzz** generates test packets based on the detection rules selected in the preprocessing phase. To this end, two modules are implemented: the *Rule Mutator*, which mutates the rules and passes to the *Packet Render*, which transforms the mutated rules into concrete packets.

4.3.1 Rule Mutator

Rule mutations are performed by the *Rule Mutator* among the buffer constraints of each parsed rule. Currently, four mutation strategies are implemented, as detailed below.

Pass-Through Strategy. This strategy selects a single rule at a time and directly passes it to the *Packet Render* without applying any manipulations. The purpose of this strategy is to generate test packets that strictly adhere to the semantics of the detection rules. This strategy represents that rules should at least be triggered by the NIDS with their corresponding non-mutated packets.

Obfuscation Strategy. This strategy selects a single rule at a time and applies obfuscation actions to the *payload detection options* of its buffer constraints. In particular, two types of obfuscation are imposed: ① *Character Encoding* : This action substitutes certain characters with their encoded equivalents. More specifically, it randomly selects a specified number of characters from the *payload detection options*, and replaces these characters of plaintext with alternative encodings. For example, "`:`" can be replaced with "`%3A`", and "`/`" with "`%2F`" [1], among others. ② *Path Shifting* : This action inserts given characters at particular positions within the URI. Specifically, this action inserts "`/.`" before each "`/`" in a URI and inserts "`./`" after it. For instance, given the original URI "`/res/exec.dll`" and the insertion number specified to 1, the obfuscated URI would be "`/././/res/././/exec.dll`". This strategy tries to verify the robustness of the rule matching mechanism of the NIDS.

Repetition Strategy. This strategy selects a single rule at a time, applies repetition actions to its buffer constraints. For now, two repetition modes are implemented: ① *Element-wise Repetition* : For each *buffer constraint*, this mode individually repeats each *payload detection option*. ② *Block-wise Repetition* : For each *buffer constraint*, this mode repeats the *payload detection option* as a whole. For clarity, Figure 3 illustrates how the two repetition modes operate. The *element-wise* repetition mode repeats cnt_1, cnt_2, and *pcre* individually, whereas the *block-wise* repetition mode treats cnt_1 and cnt_2 as a unit for repetition (since they all belong to *http_uri*), and then

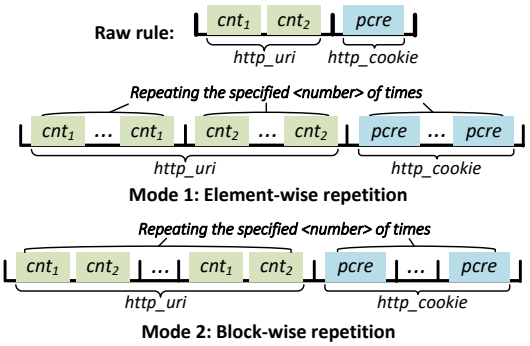

Figure 3: The mutated buffer constraints obtained by applying the two repetition modes to the raw rule. Note that the raw rule has two buffer constraints: one for *http_uri*, containing two *content* options, and another for *http_cookie*, containing a single *pcre* option.

repeats *pcre* separately. This strategy can be used to stress the rule matching mechanism of NIDS with worst cases.

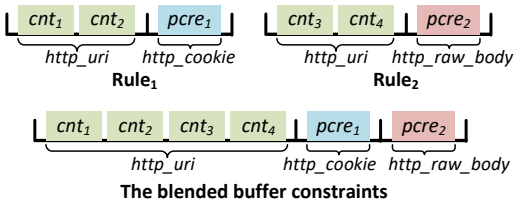

Figure 4: The resulting buffer constraints obtained by blending $rule_1$ and $rule_2$. Note that both $rule_1$ and $rule_2$ have a buffer constraint for *http_uri*. Additionally, $rule_1$ also has a constraint for *http_cookie*, while $rule_2$ has a constraint for *http_raw_body*.

Blending Strategy. This strategy selects multiple rules at a time and blends their buffer constraints. Specifically, it combines the *buffer constraints* with identical keys into a new constraint, keeping the key unchanged while combining the associated values. As an example, Figure 4 illustrates how the blending action is applied to $rule_1$ and $rule_2$. It combines the constraints for *http_uri* from both rules, while leaving the constraints for *http_cookie* and *http_raw_body* unchanged. Notably, the blending strategy accepts a list of rules and is order-sensitive, meaning that during blending, the constraints from rules in the front of the list will be placed accordingly at the front of the blended buffer constraints. This strategy tries to dig out enforcement issues of overlapping and conflicts across multiple rules.

4.3.2 Packet Render

After the rule-based mutation, the *Packet Render* generates test packets based on the mutated buffer constraints, including bidirectional requests and responses. The generated packets are compliant to the grammars of the protocol in question. The *Packet Render* has already supported the generation of packets from various network protocols in our prototype implementation [14] and can be easily adapted to new ones. To demonstrate the packet generation process, take HTTP as an

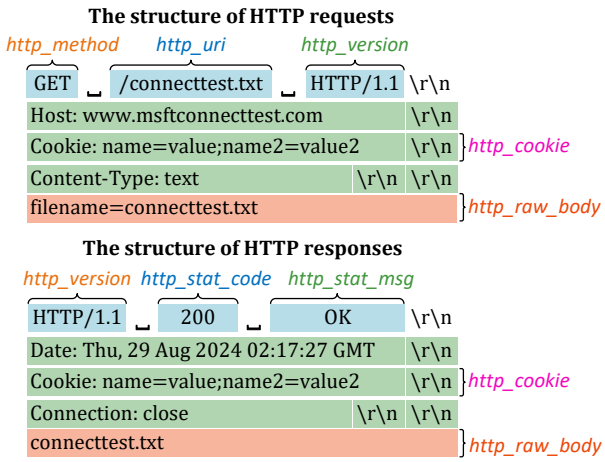

Figure 5: An example of HTTP request and response packets. For clarity, some optional headers are not fully illustrated. The symbol "␣" denotes positions where whitespace is required.

example. The packet generation follows three steps.

Step 1: Framework Setup. Template packets of both requests and responses that serve as the framework must be provided. For example, we extracted the grammar of HTTP requests and responses from relevant RFC documents [20–24] as a reference for *Packet Render* to generate packets. Overall, both HTTP requests and responses consist of a start-line (the *request-line* in requests and the *status-line* in responses) followed by multiple *header fields*, and end with a *message body*, as shown in Figure 5. There exist various optional header fields. Each line in the HTTP header fields is separated by "`\r\n`". The last line ends with "`\r\n\r\n`" as a delimiter from the message body. The *Packet Render* will construct HTTP packets in compliance with these HTTP grammars.

Step 2: Buffer Constraint Fitting Up. The buffer constraints of each rule are building blocks that ought to fit in the appropriate position of the packet framework to construct valid packets. To achieve this, the *Packet Render* maps each buffer constraint as per its key to a specific field. For instance, for HTTP requests in Figure 5, a buffer constraint with the key being *http_uri* and the value being "`/connecttest.txt`" is accordingly mapped to the *http_uri* field of the HTTP request packet. In addition, multiple buffer constraints that share identical keys are mapped to the same field. In this way, it is assured that the signature described by the buffer constraint can be placed in the exact position denoted by the rule.

Step 3: Context Alignment. In addition to the mutated values in the buffer constraint, the value of the remaining fields must be determined accordingly and align with the overall context. The specific value of a certain field attributes to three types of limits defined in the buffer constraint, which are extracted from the *payload detection options* of the rule and exemplified with HTTP as follows. ① *Length limit* : It specifies the maximum or minimum length of a field. For example, given a buffer constraint with two length limits of

"*bufferlen*:<100;" and "*isdataat*:30;", the field length must be between 30 and 100 bytes. ② *Data limit*: It designates the data value and relative position in a field. For example, a data limit of *content*:"ABC",offset 5; specifies that the data ABC has a 5-byte offset from the beginning of the field. Another example in which the limit is defined in a regular expression is *pcre*:"/[?&]user= (A|B)/i"; . In this case, an optional valid value is "&user=B". We employ the exrex library [2] to generate matching values for regular expressions. ③ *Padding limit*: It denotes the value used to pad the field after applying the other two limits. For example, *content*:!"DEF"; indicates that the field cannot contain a value of "DEF". Therefore, the padding value must not contain "DEF". An algorithm is designed to orchestrate these limits for value determination, as detailed in Algorithm 1 in Appendeix B. The remaining fields will use the default valid values embedded in the template packets, or will be dynamically calculated (e.g., Content-Length) to align with the context of the message body.

4.4 Injection Phase

To inject the generated bilateral packets into the targeted NIDS, **NIDSFuzz** implements *Tunable Initiator* and *Tunable Responder* to send requests and responses respectively.

Tunable Initiator and Responder. As illustrated in Figure 6, the injection of bilateral packets is carried out in three steps: ① *Echo Transmission*: The *Tunable Initiator* first sends an *echo message* that contains the generated response to the *Tunable Responder* through a side-way path that NIDS is not monitoring. ② *Request Transmission*: The *Tunable Initiator* transmits the generated request to the *Tunable Responder* through the exact pathway NIDS is monitoring. ③ *Response Transmission*: The *Tunable Responder* extracts the response from the received *echo message* and sends it back to the *Tunable Initiator* upon receiving the request from the *Tunable Initiator*. Notably, the *Tunable Initiator* selects a port from its maintained pool of available ports (ranging from 1024 to 65535) when sending requests (step ①), rather than relying on the system's random port allocation in order to prevent the reuse of ports within a short period.

In this way, the *Tunable Initiator* and *Tunable Responder* equip **NIDSFuzz** with the ability to tune both the request and response packets, such as adjusting the ports or the order of packets, which provides flexibility in terms of traffic injection.

4.5 Sanitization Phase

In principle, **NIDSFuzz** sanitizes the rule enforcement of NIDS with a differential method. Specifically, different NIDS instances are deployed with detection rules that are semantically identical. **NIDSFuzz** injects the same generated test packets into each NIDS instance, and then collects the alerts via the *Alert Monitor*. Then the *Differential Checker* examines the collected alerts to identify suspicious ones that may indicate a rule enforcement issue.

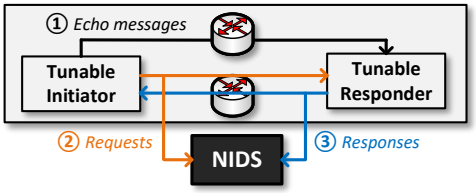

Figure 6: The workflow of the tunable initiator and responder.

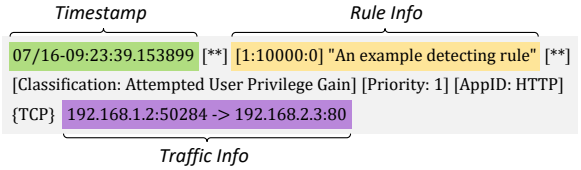

Figure 7: An example alert produced by Snort 3.

4.5.1 Alert Monitor

In prototype implementation, the *Alert Monitor* supports collecting alerts from Snort 3 and Suricata.

Alert Format. When collecting alerts from varied NIDS that have different formats, the *Alert Monitor* tells them apart based on specific sections of the alert. For example, Figure 7 presents an alert generated by Snort 3. The *Alert Monitor* concentrates on three sections of it: ① *Timestamp*, which indicates when NIDS fired this alert. ② *Rule Info*, which denotes the triggered rule with the rule id and message. ③ *Traffic Info*, which records the metadata of the malicious flow, including the direction, IP address, ports, etc. Although the alert formats of different NIDS vary slightly, they all include the three sections mentioned above.

The *Alert Monitor* is implemented with multiple threads, where each NIDS is assigned a thread that monitors changes in the alert/log files and extracts the newly appended alerts.

Port Based Alignment. Another task of the *Alert Monitor* is to align alerts to the test packets. It is achieved using a port-based approach. After receiving an alert, the *Alert Monitor* examines whether the port of the *traffic info* in the alert matches the ports used by the tunable initiator. If a match found, the alert is attributed to the packet sent from this port. However, *delayed alerts* lead to failures in matching. Since the tunable initiator uses continuously increasing ports to send request packets, the alerts regarding the port that fall behind the latest matched alert are considered delayed. In order to identify the delayed alerts, a port window is set. The port window defines the scope to which the delay is tolerated. If an alert is delayed but its port does not fall behind the latest matched alert exceeding the port window, it can still be matched; otherwise, it is classified as a *delayed alert*. A detailed algorithm is provided in Algorithm 2 in Appendix B to demonstrate the process of such alignment.

4.5.2 Differential Checker

NIDSFuzz implements the *Differential Checker*, which is responsible for identifying and reporting suspicious alerts that are referred as *alert discrepancies* to serve as indicators of

rule enforcement issues. The *Differential Checker* follows two oracles to sanitize fuzzing results.

1. If alerts are found attributing to rules that do not belong to rules selected in the current iteration, it means the generated packets happen to unexpectedly trigger rules that are not selected in the current fuzzing iteration. Therefore, this case is considered an alert discrepancy.
2. If any two NIDS produce inconsistent alerts, it implies an exposure of potential rule enforcement issue. Specifically, for the case where a test packet targeting a certain rule fires different numbers of alerts in different NIDS, it is deemed as an alert discrepancy.

If the fuzzing output satisfies at least one of these two oracles, the *Differential Checker* identifies it as a rule enforcement issue and logs the corresponding test packet(s), the inconsistent alert(s), and the related detection rule(s).

5 Evaluation and Findings

This section introduces our experiments and enforcement issues discovered, including the experiment setup, findings, corresponding countermeasures, and performance evaluation.

5.1 Experiment Setup

NIDS and Rules. We evaluated **NIDSFuzz** on three widely used NIDS platforms, each running its latest version available at the time of writing, and utilized the most up-to-date Snort community ruleset, as shown in Table 1. Our evaluation targeted rules for four protocols—HTTP, FTP, DNS and SIP. All rules were adapted as needed to ensure compatibility and proper deployment on each NIDS during testing.

Table 1: Basic information about tested NIDS and rules.

NIDS Version			# of Rules			
Snort3	**Suricata**	**Snort2**	**HTTP**	**FTP**	**DNS**	**SIP**
3.6.0	7.0.8	2.9.20	4033	297	112	219

Network Topology. Our evaluation was conducted on a machine equipped with an AMD EPYC 7742 CPU and 128 GB RAM. As shown in Figure 8, the network topology, managed by Docker Compose [12], included three subnets (left, mid, right). **NIDSFuzz** operated from *Container* ① (left), targeting the *tunable responder* in *Container* ② (right). *Container* ③ functioned as a gateway, routing traffic between all subnets. In the mid subnet, three NIDS were deployed in promiscuous mode to inspect packets mirrored from *Container* ③. For the purpose of replicability, the entire experimental environment is fully automated through a purpose-built script and accompanying configuration files.

Experiment Methodology. We ran our experiments over four mutation strategies: pass-through, obfuscation, repetition and blending. Specifically, we first applied pass-through strategy to every rule in Table 1, and removed any rule that exhibited issues from the ruleset. The remaining rules were then evaluated for obfuscation, repetition, and blending strategies.

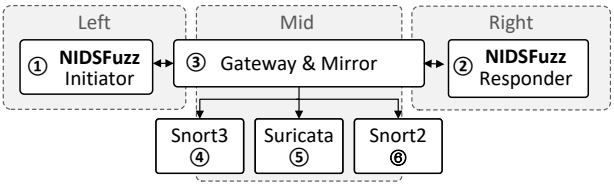

Figure 8: The experiment network topology.

- **Obfuscation**: This strategy is only applicable to text-based protocol, namely HTTP, FTP, and SIP. With sequential selection, we performed 50 obfuscation tests on each rule.
- **Repetition**: For this strategy, each rule was tested 50 times in both element-wise and block-wise modes.
- **Blending**: This strategy was evaluated in a progressive manner. We began by iterating through all possible two-rule combinations, removing those implicated in any enforcement issues, and then proceeded to evaluate all possible triplet combinations. This "test-and-prune" cycle continued until no enforcement issues were reported. In particular, an exhaustive search for combinations of three or more rules is computationally infeasible, so we fixed the test duration at 24 hours and employed random selection to sample the required number of rules.

Post-Hoc Analysis. After the experiments, we conducted a manual analysis to investigate the rule enforcement issues discovered by **NIDSFuzz** and performed manual experiments to explore their underlying causes and latent risks. This process took around a week to complete.

5.2 Findings

After running the experiments, 10,873 in total unique rule enforcement issues have been reported by **NIDSFuzz**. The time consumption to execute each mutation strategy ranged from 20 minutes to 24 hours. The detailed execution time for each mutation strategy with the specific selection algorithm and the number of unique rule enforcement issues discovered in different network protocols are given in Table 5 in Appendix C. To reproduce our findings, a script [14] is also provided that replays these rule enforcement issues.

5.2.1 Severity

To highlight the significance of the reported issues, we assessed their severity based on whether they can be exploited as well as the consequences resulting from such exploitations. The evaluation process comprises four steps: (1) We extract the detection rules associated with the reported issue (see Section 4.5.2) and deploy them in the NIDS exclusively, ensuring that no other rules are active to eliminate noise. (2) We craft malicious packets by analyzing the rule signatures (e.g., *content*, *pcre*) and relevant *references* (e.g., CVEs) to ensure the payload targets the intended vulnerability. (3) We inject these packets into the NIDS. When the evaluated issue is rate- or throughput-dependent, we control the injection frequency and send traffic repeatedly to satisfy the necessary triggering

Table 2: Taxonomy of rule enforcement issues discovered by **NIDSFuzz**.

	Category	Issue & Severity	Stat	Strategy				Affected NIDS			Affected Rules			
				P	O	R	B	S3	Su	S2	HTTP	FTP	DNS	SIP
Rule Induced Issues	Intra-Rule	Redundant Sticky Buffers (H)	●	+	–	–	–	✗	✓	✗	✓	✗	✗	✗
		Outsized Option Values (H)	●	+	–	–	–	✓	✓	✓	✓	✓	✗	✗
		Misused Keywords (H)	●	+	–	–	–	✗	✓	✓	✗	✓	✗	✓
		Overbroad Signatures (M)	○	+	–	–	–	✓	✓	✓	✓	✓	✓	✓
	Inter-Rule	Duplicate Rules (M)	●	+	–	–	–	✓	✓	✓	✓	✗	✓	✓
		Overlapping Rules (M)	●	+	–	+	+	✓	✓	✓	✓	✓	✓	✓
Implementation Induced Issues	Protocol Parsing	Invalid Status Code (H)	●	–	–	–	+	✓	✗	✓	✓	✗	✗	✗
		Corrupted Content Length (H)	●	–	+	–	+	✗	✓	✗	✓	✗	✗	✗
		Mis-stripped Spaces (L)	●	–	–	–	+	✗	✓	✗	✓	✗	✗	✗
		Omitted Checks (L)	○	–	+	+	+	✓	✗	✗	✗	✗	✓	✗
	Rule Matching	Complex Regex (H)	●	+	+	+	+	✗	✓	✗	✓	✓	✓	✓
		Inconsistent Interpretation (H)	○	+	–	–	–	✓	✓	✓	✓	✓	✗	✓
		Reduced Reliability (M)	○	+	+	+	+	✗	✓	✗	✓	✓	✓	✓
		Direction Mismatch (M)	○	+	+	+	+	✓	✗	✗	✗	✗	✓	✗
		Transport Layer Mismatch (M)	●	+	–	–	+	✓	✗	✗	✗	✗	✓	✓
		Obfuscated Characters (H)	○	–	+	–	–	✓	✗	✗	✓	✓	✗	✓
		Limited Packet Size (H)	●	–	–	+	+	✓	✓	✓	✓	✓	✓	✓
		Long Segmentation (M)	●	–	–	+	–	✓	✓	✓	✗	✓	✓	✓

P represents the pass-through strategy. **O** represents the obfuscation strategy. **R** represents the repetition strategy. **B** represents the blending strategy.
S3 stands for Snort3. **Su** stands for Suricata. **S2** stands for Snort2.
○ indicates the issue was reported to the official and is still under investigation, while ● means it was reported and confirmed.
+ indicates that the mutation strategy reported related alert discrepancies, while – means no relevant discrepancies were found in the evaluation.
✓ indicates that the NIDS or the rules of the protocol is affected by the issue, while ✗ indicates neither is affected.
(H) denotes high-severity issues, (M) denotes medium-severity issues, and (L) denotes low-severity issues.

conditions. (4) Finally, we analyze the generated alerts to classify the severity of the reported issue as follows:

- **High** ((H)): If no alerts are generated, we conclude that the malicious packets have successfully evaded detection, and thus classify the issue as a high-severity tier.
- **Medium** ((M)): If false alerts are generated or alerts are partially missed, we consider that the issue degrades the overall usability of the NIDS, and thus assign it a medium-severity tier.
- **Low** ((L)): If the packets that trigger the issues in NIDS are actually not functional in the communication endpoints, suggesting that the issue is impossible to be weaponized by attackers in practice, we assign it to a low-severity tier.

We manually evaluated all reported issues, and calculated their proportions as summarized in Table 6.

5.2.2 Issues and Countermeasures

Based on the root causes identified by manual analysis, we organized these issues into two categories: *rule-induced* and *implementation-induced*, as shown in Table 2. Specifically, rule-induced issues denote enforcement issues rooted in the rules themselves, whereas implementation-induced issues arise from unintended aspects in NIDS implementations. To illustrate, we exemplify the symptoms of each issue, analyze its security impact, and discuss potential countermeasures. Due to space limitations, we focus on representative issues in this section; additional cases are available in Appendix D.

Rule-Induced Issues. Rule-induced issues can be classified into *intra-rule* and *inter-rule* issues, with the former caused by an individual rule and the latter involving interference between multiple rules.

• *Redundant Sticky Buffers*: We observed that rules containing redundant *sticky buffers*—specifically, two or more occurrences of the same *sticky buffer*—do not function correctly in Suricata, but work normally in Snort2 and Snort3. For example, Rule 1:41408:3 in Figure 11 contains two *file_data*. These rules are completely useless in Suricata, leaving the system vulnerable to evasion.

• *Outsized Option Values*: We found that some rules include "outsized" option values, hindering their correct enforcement. For example, Rule 1:18809:13 in Figure 12 specifies the option "*isdataat:5000*", meaning that the HTTP response must contain at least 5,000 bytes of data. However, such packet size far exceeds the default inspection limits of Snort3, Suricata, and Snort2, rendering the rule ineffective and allowing

malicious packets to bypass detection.

• *Misused Keywords*: As illustrated in Figure 13, a Snort2 SIP rule applies a *pcre* option with the "H" modifier. However, the official manual [3] states that this modifier "*matches the normalized HTTP request/response header (similar to http_header)*", indicating that it is designed for HTTP rather than SIP traffic. In contrast, the corresponding Snort3 rule removes the "H" modifier and instead uses "*sip_header*", ensuring the *pcre* check is restricted to the SIP header only. Consequently, the semantics of these rules are distorted due to keyword misuse, inevitably leading to a failure in detecting the intended malicious traffic.

• *Overbroad Signatures*: Some rules use overbroad signatures, making them prone to unintended false positive alerts. For example, Rule 1:2417:17 in Figure 10 is designed to match packets with two "%" characters. In our experiments, thousands of test packets triggered this pattern, generating a large number of meaningless repetitive alerts.

• *Duplicate Rules*: Several duplicate rules that have different *sid* but share identical signatures. For example, in Figure 14, Rule 1:23609:9 and Rule 1:31469:2 are duplicates. Although duplicate rules do not result in evasion, they produce redundant alerts that impede security analysis.

• *Overlapping Rules*: We discovered dozens of overlapping rules with signatures that are not exactly the same but highly similar. For example, Figure 15 illustrates the overlap between Rule 1:44978:3 and Rule 1:29579:3. In Rule 1:44978:3, "*within 62;*" means that the pattern will be searched within 62 bytes after the preceding content match. In contrast, Rule 1:29579:3 uses "*within 60;*", indicating that the pattern will be searched within 60 bytes. Notably, the impact of *overlapping rules*, *duplicate rules*, and *overbroad signatures* is cumulative. This allows attackers to exploit these combined issues to launch squealing attacks [31, 36, 55], where a flood of false alerts overwhelms security operators. For instance, in our experiments, we demonstrated that a single spoofed packet could be easily crafted to trigger as many as 9 rules.

Countermeasures

The rule-induced issues highlight the complexity and error-prone nature of detection rule authoring and integration. We advocate for stricter verification practices within the community. Specifically, individual rules must be screened to preclude *redundant sticky buffers* and *misused keywords*, and the ruleset should be validated to remove *duplicate* and *overlapping rules*. Additionally, we suggest that vendors explicitly annotate rules with *outsized option values* to indicate the requisite configuration settings.

Implementation-Induced Issues. Implementation-induced issues are classified into *protocol parsing* and *rule matching* issues, attributing to the specific phase they being processed by NIDS.

• *Invalid Status Code*: We found that Snort3 does not inspect HTTP responses that contain invalid status codes, allowing such packets to evade detection. As shown in Figure 17, we utilized a packet with an invalid status code "`302302`" to test Rule 1:56449:2 and Rule 1:60282:2. We observed that Snort 3 failed to trigger any alerts for this packet, whereas real-world HTTP clients successfully processed it. This client-side tolerance aligns with RFC 7230 [24], which implies a general expectation of compatibility with poorly implemented servers. Crucially, attackers can weaponize this issue by crafting malicious responses with invalid status codes; this allows them to deliver payloads that evade Snort 3's inspection while remaining fully effective against the victim client.

• *Corrupted Content Length*: An interesting phenomenon was observed in Suricata: a malformed "Content-Length" header can cause the engine to hang until a timeout occurs, resulting in significantly delayed alerts. Through step-by-step debugging, we found that if Suricata has not received enough data from packet shards meeting "Content-Length", it continues to wait for additional data until a timeout is triggered. Based on this insight, we tried to periodically send small amounts of data to Suricata to keep it in a perpetual reception state without triggering a timeout, which could delay alert generation by even tens of hours.

• *Complex Regex*: We found that Suricata struggles to enforce rules with complex *pcre* patterns. In Figure 19, Rule 1:16767:9 defines a complex regex composed of two intricate subpatterns joined by alternation ("`|`"). In our tests, packets that match this regex nonetheless failed to trigger a Suricata alert. An attacker could exploit two evasion strategies by padding semantically insignificant locations—for example, positions where a JavaScript parser tolerates multiple consecutive spaces [4]—to evade the rule: abusing "`\s*`" to insert more than 60 spaces, or inserting spaces mid-string and appending extra letters at the end.

• *Inconsistent Interpretation*: We found that some rule keywords are interpreted differently across NIDS platforms. For example, the FTP packet in Figure 20 triggers Rule 1:38950:4 in Snort2 but not in Snort3 and Suricata because Snort2 scopes the "*file_data*" buffer differently. Consequently, this leads to rules that are enforced correctly on specific NIDS platforms while failing to accurately detect the intended malicious traffic on others.

• *Obfuscated Characters*: We found that obfuscating specific characters can bypass Snort3's string normalization, leading to false negatives. As shown in Figure 22, a packet that triggers Rule 1:18508:6 no longer matches after the "`[`" and "`]`" characters are encoded (e.g., percentage-encoding). In contrast, both the original and encoded packets still trigger Rule 1:18508:6 in Suricata and Snort2.

• *Limited Packet Size*: We found that when packets exceed certain sizes, all NIDS platforms perform no inspection. Specifically, packets with application-layer payloads exceeding 1452 bytes (Snort3), 1464 bytes (Suricata), or 1448 bytes (Snort2) were not inspected. A similar phenomenon has also been re-

ported in national-level censorship systems [26], which may be due to the use of similar DPI techniques in both NIDS and these censorship mechanisms. Coincidentally, we note that CVE-2025-29915 [11] reported by Suricata recently has similar conclusions. By inserting extra data into a malicious packet to push it beyond the inspection size threshold, an attacker can evade the NIDS. Although these NIDS claim to provide configurable parameters to set the maximum packet size for inspection (Snort3 and Snort2 uses "*snaplen*", Suricata uses "*default-packet-size*"), we found that configuring these parameters is non-trivial. First, these systems use specialized configuration formats—Snort3 uses "snort.lua", Suricata uses "suricata.yaml", and Snort2 uses "snort.conf"—which require some familiarity to modify correctly. Second, the configuration files span hundreds of lines with numerous parameters, making it difficult to locate and adjust the relevant settings.

Countermeasures

Most implementation-induced issues, such as *invalid status code* and *direction mismatch*, can be addressed by refining the codebase. Alternatively, certain issues can be mitigated through rule optimization. A practical workaround for the *complex regex*, for instance, is to split a complex rule into multiple simpler segments. Figure 23 provides an example of how Rule 1:14266:12, with a complex *pcre*, can be rewritten as two simple rules that are not impacted by the issue, while preserving the original intent. Beyond the specific mitigations, we argue that the community should devote more effort to establishing rule specifications and implementation paradigms to fundamentally eliminate inconsistent rule enforcement across different NIDS platforms.

5.3 Comparison with Existing Approaches

For comparison of the capability in uncovering rule enforcement issues between the existing approaches and **NIDSFuzz**, we applied different approaches to generate test traffic for evaluating the NIDSs and rules listed in Table 1 and compared the results. Specifically, two state-of-the-art fuzzing tools were adapted, including Geneva [17] and Boofuzz-HTTP [9]. In addition, we replayed the real-world attack traffic dataset CICIDS2017 [13] and CICIDS2018 [10]. The *rule coverage* were computed for each tool and traffic trace to compare with **NIDSFuzz**, as shown in Table 3. Generally, the existing approaches failed to achieve sufficient rule coverage for assessing rules compared to **NIDSFuzz**. This is because the fuzzing tools either could only operate at the TCP layer while leaving the application layer untouched (Geneva), or lacked the ability to generate rule-oriented test packets (Boofuzz-HTTP). Besides, these tools generate only unidirectional traffic, leaving many response-targeted rules untested. Moreover, the traffic traces focused on a narrow set of attack families (e.g., port scans, XSS) instead of the specific targets of the deployed rules. **NIDSFuzz**, with rule-oriented mutation strategies and bidirectional traffic generation, achieved near-complete rule coverage, excluding rules that are inherently untriggerable uncovered in Section 5.2.

Table 3: The rule coverage achieved by existing methods.

Method	Snort3	Suricata	Snort2
Geneva	0.02%	0%	0.02%
Boofuzz-HTTP	1.17%	1.21%	1.19%
CICIDS2017	1.18%	1.54%	1.18%
CICIDS2018	2.61%	2.92%	2.79%
NIDSFuzz	94.45%	91.72%	94.15%

5.4 Performance

In experiments, the preprocessing and sanitization phases impose negligible performance overhead, therefore, We mainly evaluated the performance of the generation and injection phases. To evaluate the packet generation time in the generation phase, We applied different mutation strategies to generate 20,000 test packets and recorded the time taken. The results are shown in the left part of Figure 9. The *pass-through* strategy incurred the least time cost (6.54 s), while the *repetition* strategy incurred the most (89.71 s). This is because *pass-through* merely renders the original options into HTTP packets without mutation overhead. In contrast, *repetition* dramatically increases the number of options (often hundreds), significantly increasing the rendering complexity. It also applied to the *blending* strategies as the time increased with the number of blended rules. As for tunable communication in the injection phase, we compared the performance of **NIDSFuzz**'s tunable communication architecture with that of using a real HTTP program. We issued 20,000 requests to both the *tunable responder* and the apache server, and measured the total time to complete all request-response cycles. The result is given in the right part of Figure 9, which shows that the *Tunable Responder* significantly outperforms the real HTTP program.

The performance evaluation demonstrates that, during testing, the time spent on injection markedly exceeds that of packet generation, further highlighting the efficiency of **NIDSFuzz**'s tunable communication architecture.

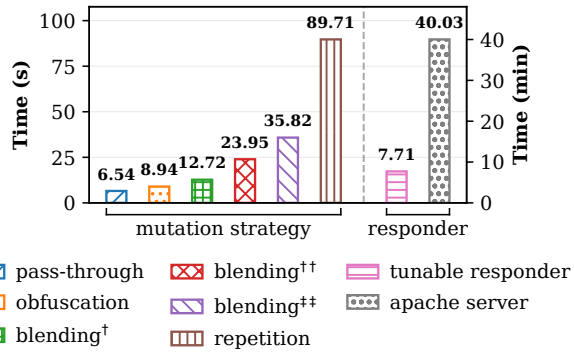

Figure 9: Results of performance evaluation. † indicates blending two rules at a time; †† blends three rules; and ‡‡ blends four rules.

6 Discussion

Extending NIDSFuzz to support more NIDS systems and other protocols. It is easy to adopt more NIDS systems and protocols to **NIDSFuzz**. To validate other NIDS systems, the developer should download and deploy the rulesets of the NIDS to be tested and then set up each NIDS system by adding a few lines of Docker Compose commands in the provided script [14]. To apply to more protocols, the developer only need to provide the definition of template packets using the interface provided.

Automating the analysis of alert discrepancies reported by NIDSFuzz. Currently, we identify the root causes of alert discrepancies through half-automated manual analysis with a script running to automatically categorize alert discrepancies into different rule enforcement issues. We notice that large language models (LLMs) have demonstrated strong automation capabilities in the security domain and can complement traditional vulnerability analysis techniques [18, 27]. In the future, we plan to explore how the power of LLMs can be leveraged to assist in automatically analyzing the root causes of alert discrepancies.

7 Related Work

This section reviews studies akin to **NIDSFuzz**, which can be categorized into rule-related and implementation-related evaluation of NIDS based on their specific concerns.

Rule-related Evaluation. A wealth of research has put effort into evaluating detection rules of NIDS with various measures, whose purpose is to uncover potential risks related to rules. One risk point is the rules, as it can be faulty that they conflict with each other. For detection of such issue, tools known as *IDS stimulators* [33, 36] have been proposed to evaluate the performance of rulesets in NIDS, through directly translating open-source signatures of attacks into synthetic traffic datasets. However, the generated datasets are bound to suffer from imbalances and low test coverages as the testing datasets are not tailored for the rulesets under evaluation, which motivates **NIDSFuzz**'s rule-based fuzzing. The other risk point is the rule matching mechanism, which can be exploited by carefully crafted traffic to avoid firing the detection rules. To detect this issue, a tool [47] has been designed to generate a large number of variants for given exploits, while the variation is based on the payload of traffic instead of detection rules, which in turn has no guarantees on rule evaluation coverage. Furthermore, rules for stateful protocols has been exploited to evade NIDS with improper configurations of flowbits [15, 45].

These studies focus on standalone issues, lacking systematic methodologies that proposed by **NIDSFuzz**. It is worth noting that **NIDSFuzz** has taken the aforementioned work into account when designing the mutation strategies and is adaptable to more mutation strategies.

Implementation-related Evaluation. Recent research has focused on exploiting implementation flaws to evade inspection of DPI, which is an integral component of NIDS, thereby falling within the scope of the present paper. The core idea can be summarized as leveraging the discrepancies of implementation in protocol parsing between DPI and end hosts, but differs in terms of techniques used in different studies. StateDiver [57] assessed the effectiveness of test cases by comparing the internal state changes of different DPI systems, but required pre-instrumenting the TCP layer, which falls out of the scope of black-box fuzzing. SymTCP [51] employed symbolic execution to identify state discrepancies of TCP implementations based on paths that lead to the acceptance or dropping of packets. Pryde [32] proposed using automata learning to infer the TCP state machine that guides the symbolic definition of evasion attacks, which is then used to synthesize custom evasion attacks. Geneva [17] presented the first automated evasion discovery tool based on genetic principles, which continuously evolves four basic packet manipulation primitives to generate sophisticated TCP evasion strategies. DPIFuzz [39] proposed a differential fuzzing framework to detect DPI evasion strategies for QUIC. Furthermore, more efforts have been done in later work by extending Gnenva with instrumenting the tested DPI [57] or revising the manipulation primitives [26] in order to improve its performance.

Although these studies have different concerns from **NIDSFuzz** that they concentrate on rule-irrelevant implementation flaws, their automated testing methodologies offer valuable insights that inspired the proposal of **NIDSFuzz**.

8 Conclusion

This paper presents **NIDSFuzz**, a systematic fuzzing approach designed to validate the enforcement of rules within Network Intrusion Detection Systems (NIDS). By addressing the limitations of existing methods, **NIDSFuzz** focuses on rule-oriented fuzzing, leveraging tailored mutation strategies to detect potential enforcement issues. Experimental results confirm **NIDSFuzz**'s efficiency and effectiveness in identifying a wide range of rule enforcement issues, some of which were previously unreported.

Acknowledgment

We would like to thank our shepherd Prof. David Lie and the anonymous reviewers for their insightful comments. This work was supported by the National Natural Science Foundation of China (No. 62372191, 62302183), the Open Topics from The Lion Rock Labs of Cyberspace Security (No. LRL24013), the Songshan Laboratory (No. 241110210200), the Open Foundation of Key Laboratory of Cyberspace Security, Ministry of Education of China (No. KLCS20240401), and CCF-DiDi GAIA Collaborative Research Funds (No. CCF-DiDi GAIA 202412 and No. CCF-DiDi GAIA 202522).

References

[1] Rfc 3986 - uniform resource identifier (uri): Generic syntax. https://datatracker.ietf.org/doc/html/rfc3986, 2005. Accessed: 2025-01.

[2] Irregular methods on regular expressions. https://github.com/asciimoo/exrex, 2012. Accessed: 2025-01.

[3] Snort users manual. http://manual-snort-org.s3-website-us-east-1.amazonaws.com/snort_manual.html, 2020. Accessed: 2025-01.

[4] Ecmascript 2023 language specification. https://262.ecma-international.org/14.0/#sec-white-space, 2023. Accessed: 2025-01.

[5] Payload detection rule options - sticky buffers. https://docs.snort.org/rules/options/payload/, 2024. Accessed: 2025-01.

[6] Snort - network intrusion detection & prevention system. https://www.snort.org/snort3, 2024. Accessed: 2025-01.

[7] snort2bro, convertes snort signatures automatically into zeek's (then called "bro") signature syntax. https://github.com/ewust/telex/blob/master/old-telex-station-backup/bro-1.5.1/scripts/s2b/bin/snort2bro, 2024. Accessed: 2025-01.

[8] Suricata: Open source ids/ips/nsm engine. https://suricata.io/, 2024. Accessed: 2025-01.

[9] boofuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz, 2025. Accessed: 2025-01.

[10] Cse-cic-ids2018 on aws. https://www.unb.ca/cic/datasets/ids-2018.html, 2025. Accessed: 2025-01.

[11] CVE-2025-29915. https://nvd.nist.gov/vuln/detail/CVE-2025-29915, 2025. Accessed: 2025-09.

[12] Docker compose. https://docs.docker.com/compose/, 2025. Accessed: 2025-01.

[13] Intrusion detection evaluation dataset (cic-ids2017). https://www.unb.ca/cic/datasets/ids-2017.html, 2025. Accessed: 2025-01.

[14] Nidsfuzz: a rule-orenited differential fuzzing framework for nids. https://github.com/nidsfuzz/nidsfuzz, 2025. Accessed: 2025-01.

[15] Issam Aib, Tung Tran, and Raouf Boutaba. Characterization and solution to a stateful ids evasion. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, pages 597–604, 2009.

[16] Lucas Alcantara, Guilherme Padilha, Rui Abreu, and Marcelo d'Amorim. Syrius: Synthesis of rules for intrusion detectors. *IEEE Transactions on Reliability*, 71(1):370–381, 2022.

[17] Kevin Bock, George Hughey, Xiao Qiang, and Dave Levin. Geneva: Evolving censorship evasion strategies. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security*, pages 2199–2214, 2019.

[18] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the 19th European Conference on Computer Systems*, pages 674–688, 2024.

[19] Felix Erlacher and Falko Dressler. How to test an ids? genesids: An automated system for generating attack traffic. In *Proceedings of the 3rd Workshop on Traffic Measurements for Cybersecurity*, pages 46–51, 2018.

[20] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Rfc2616: Hypertext transfer protocol–http/1.1. https://datatracker.ietf.org/doc/html/rfc2616, 1999. Accessed: 2025-01.

[21] Roy Fielding, M Nottingham, and J Reschke. Rfc 7234: hypertext transfer protocol (http/1.1): Caching. https://datatracker.ietf.org/doc/html/rfc7234, 2014. Accessed: 2025-01.

[22] Roy Fielding and Julian Reschke. Rfc 7231: Hypertext transfer protocol (http/1.1): Semantics and content. https://datatracker.ietf.org/doc/html/rfc7231, 2014. Accessed: 2025-01.

[23] Roy Fielding and Julian Reschke. Rfc 7232: hypertext transfer protocol (http/1.1): Conditional requests. https://datatracker.ietf.org/doc/html/rfc7232, 2014. Accessed: 2025-01.

[24] Roy Thomas Fielding and Julian Reschke. Rfc 7230: Hypertext transfer protocol (http/1.1): Message syntax and routing. https://datatracker.ietf.org/doc/html/rfc7230, 2017. Accessed: 2025-01.

[25] Prahlad Fogla, Monirul Islam Sharif, Roberto Perdisci, Oleg M Kolesnikov, and Wenke Lee. Polymorphic blending attacks. In *Proceedings of the 15th USENIX Security Symposium*, pages 241–256, 2006.

[26] Michael Harrity, Kevin Bock, Frederick Sell, and Dave Levin. Get/out: Automated discovery of application-layer censorship evasion strategies. In *Proceedings of the 31st USENIX Security Symposium*, pages 465–483, 2022.

[27] Zongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. Towards understanding the effectiveness of large language models on directed test input generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1408–1420, 2024.

[28] Yinxi Liu and Wei Meng. Dsfuzz: Detecting deep state bugs with dependent state exploration. In *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security*, pages 1242–1256, 2023.

[29] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. Bleem: Packet sequence oriented fuzzing for protocol implementations. In *Proceedings of the 32nd USENIX Security Symposium*, pages 4481–4498, 2023.

[30] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021.

[31] Frédéric Massicotte and Yvan Labiche. An analysis of signature overlaps in intrusion detection systems. In *Proceedings of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 109–120, 2011.

[32] Soo-Jin Moon, Milind Srivastava, Yves Bieri, Ruben Martins, and Vyas Sekar. Pryde: A modular generalizable workflow for uncovering evasion attacks against stateful firewall deployments. In *Proceedings of the 45th IEEE Symposium on Security and Privacy*, pages 4440–4458, 2024.

[33] Darren Mutz, Giovanni Vigna, and Richard Kemmerer. An experience developing an ids stimulator for the black-box testing of network intrusion detection systems. In *Proceedings of the 19th Annual Computer Security Applications Conference*, pages 374–383, 2003.

[34] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 26th IEEE Symposium on Security and Privacy*, pages 226–241, 2005.

[35] Piyawat Noiprasong and Assadarat Khurat. An ids rule redundancy verification. In *Proceedings of the 17th International Joint Conference on Computer Science and Software Engineering*, pages 110–115, 2020.

[36] Samuel Patton. An achilles' heel in signature-based ids: Squealing false positives in snort. In *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection*, 2001.

[37] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 24th ACM SIGSAC conference on Computer and Communications Security*, pages 2155–2168, 2017.

[38] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification*, pages 460–465, 2020.

[39] Gaganjeet Singh Reen and Christian Rossow. Dpifuzz: A differential fuzzing framework to detect dpi elusion strategies for quic. In *Proceedings of the 36th Annual Computer Security Applications Conference*, pages 332–344, 2020.

[40] Mikko Särelä, Tomi Kyöstilä, Timo Kiravuo, and Jukka Manner. Evaluating intrusion prevention systems with evasions. *International Journal of Communication Systems*, 30(16):e3339, 2017.

[41] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 17th USENIX Annual Technical Conference*, pages 309–318, 2012.

[42] Govind Sreekar Shenoy, Jordi Tubella, and Antonio Gonz'lez. Hardware/software mechanisms for protecting an ids against algorithmic complexity attacks. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1190–1196, 2012.

[43] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking algorithmic complexity attacks against a nids. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 89–98, 2006.

[44] Natalia Stakhanova and Ali A Ghorbani. Managing intrusion detection rule sets. In *Proceedings of the 3rd European Workshop on System Security*, pages 29–35, 2010.

[45] Tung Tran, Issam Aib, Ehab Al-Shaer, and Raouf Boutaba. An evasive attack on snort flowbits. In *Proceedings of the 12th IEEE Network Operations and Management Symposium*, pages 351–358, 2012.

[46] Mathew Vermeer, Michel van Eeten, and Carlos Gañán. Ruling the rules: Quantifying the evolution of rulesets, alerts and incidents in network intrusion detection. In *Proceedings of the 17th ACM on Asia Conference on Computer and Communications Security*, page 799–814, 2022.

[47] Giovanni Vigna, William Robertson, and Davide Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *Proceedings of the 11th ACM conference on Computer and Communications Security*, pages 21–30, 2004.

[48] Todd Vollmer, Jim Alves-Foss, and Milos Manic. Autonomous rule creation for intrusion detection. In *Proceedings the 2nd IEEE Symposium on Computational Intelligence in Cyber Security*, pages 1–8, 2011.

[49] Qi Wang, Jianjun Chen, Zheyu Jiang, Run Guo, Ximeng Liu, Chao Zhang, and Haixin Duan. Break the wall from bottom: Automated discovery of protocol-level evasion vulnerabilities in web application firewalls. In *Proceedings of the 45th IEEE Symposium on Security and Privacy*, pages 185–202, 2024.

[50] Zhongjie Wang, Yue Cao, Zhiyun Qian, Chengyu Song, and Srikanth V Krishnamurthy. Your state is not mine: A closer look at evading stateful internet censorship. In *Proceedings of the 17th Internet Measurement Conference*, pages 114–127, 2017.

[51] Zhongjie Wang, Shitong Zhu, Yue Cao, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, Kevin S. Chan, and Tracy D. Braun. Symtcp: Eluding stateful deep packet inspection with automated discrepancy discovery. In *Proceedings of the 27th Network and Distributed System Security Symposium*, 2020.

[52] Mingshi Wu, Jackson Sippe, Danesh Sivakumar, Jack Burg, Peter Anderson, Xiaokang Wang, Kevin Bock, Amir Houmansadr, Dave Levin, and Eric Wustrow. How the great firewall of china detects and blocks fully encrypted traffic. In *Proceedings of the 32nd USENIX Security Symposium*, pages 2653–2670, 2023.

[53] Yi Yin, Yun Wang, and Naohisa Takahashi. Set-based calculation of topological relations between snort rules. In *Proceedings of the 2nd International Symposium on Computing and Networking*, pages 617–619, 2014.

[54] Xingliang Yuan, Xinyu Wang, Jianxiong Lin, and Cong Wang. Privacy-preserving deep packet inspection in outsourced middleboxes. In *Proceedings the 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, 2016.

[55] William Yurcik. Controlling intrusion detection systems by generating false positives: Squealing proof-of-concept. In *Proceedings of the 27th Annual IEEE Conference on Local Computer Networks*, pages 134–135, 2002.

[56] Cen Zhang, Yuekang Li, Hao Zhou, Xiaohan Zhang, Yaowen Zheng, Xian Zhan, Xiaofei Xie, Xiapu Luo, Xinghua Li, Yang Liu, et al. Automata-guided control-flow-sensitive fuzz driver generation. In *Proceedings of the 32nd USENIX Security Symposium*, pages 2867–2884, 2023.

[57] Zhechang Zhang, Bin Yuan, Kehan Yang, Deqing Zou, and Hai Jin. Statediver: Testing deep packet inspection systems with state-discrepancy guidance. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 756–768, 2022.

[58] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. Tcp-fuzz: Detecting memory and semantic bugs in tcp stacks with fuzzing. In *Proceedings of the 2021 USENIX Annual Technical Conference*, pages 489–502, 2021.

A Rule Options

The rule options of a detection rule define its intent, based on which **NIDSFuzz** generates test packets. Table 4 presents several rule options along with their descriptions of functionality.

Table 4: Some rule options used in this paper and their descriptions.

Name	Description
msg	indicates the generated alert message
identifiers	identify this rule by numeric values
service	denotes the service(s) this rule applies to
http_uri	targets HTTP request URIs for matching
http_raw_body	targets HTTP message data for matching
http_cookie	targets HTTP cookies for matching
content	denotes attack signatures in plain text
bufferlen	checks the length of a given buffer
isdataat	verifies payload at a specified location
pcre	defines attack signatures using regex

B Algorithms

Algorithm 1: Derive a field value from a buffer constraint.

Input: $\{k, V\}$, the buffer constraint, where k represents the keys and V denotes the values
Output: S, a string that satisfies the buffer constraint

```
1  ζ ← (min, max)        ▷ initialize the valid length range;
2  Σ ← ∅                 ▷ initialize a list to store the data;
3  Ω ← init(Ω)           ▷ initialize the valid padding;
4  for each limits lmt in V do
5      if lmt limits the length of S then
6          adjust ζ to comply with opt;
7      if lmt forbids certain characters in S then
8          remove those characters from Ω;
9      if lmt explicitly defines the data in S then
10         store the data and positions in Σ;
11 length ← select a valid length from ζ;
12 padding ← generate padding using Ω;
13 S ← assemble the padding and Σ;
14 return S;
```

Algorithm 1 demonstrates how to orchestrate rule's buffer constraints to determine valid values. First, it initializes a tuple to represent the valid length range ζ (Line 1), a list to store the data Σ (Line 2), and a set to store the valid padding Ω (Line 3). Then the values of ζ, Σ, and Ω are updated according to the limits defined in the buffer constraint (Line 4-10). Finally, the value of a field is determined by the length, data, and padding (Line 11-13).

Algorithm 2: Correlate the alerts to the test packets.

Input: Q, a queue that stores newly appended alerts; ω, a 2-tuple including the source and destination ports of the test packet; $\Delta\omega$, a port window that defines the delay tolerance
Output: Π, a list that contains alerts triggered by the analyzed packet

```
1  Π ← initialize an empty alert list for the analyzed packet;
2  do
3      α ← retrieve an alert α from Q;
4      ω^α ← extract the traffic info of α;
5      if ω^α is equal to ω then
6          append the alert α to Π;
7      else if ω^α is not in Δω then
8          report by recording the delayed alert α;
9      else
10         Π' ← find the alert list of the preceding packet that uses ω^α;
11         append the alert α to Π';
12 while no alert in Q;
13 return Π;
```

Algorithm 2 demonstrates how to align alerts to the rules based on ports. For each thread that listens to an NIDS, after receiving an alert(Line 3), the thread examines whether the port of the *traffic info* in the alert matches the ports used by the tunable initiator ω . If a match is found, the alert is attributed to the test packet sent from this port (Line 6). However, delayed alerts are a common occurrence, leading to failures in matching. Since the tunable initiator uses continuously increasing ports to send request packets, alerts regarding the port that fall behind the latest matched alert can be considered delayed. In order to identify delayed alerts, we set a port window $\Delta\omega$. The port window defines the scope to which the delay is tolerated due to network fluctuations. If an alert is delayed, but its port does not fall behind the latest matched alert exceeding the port window, it can still be matched (Line 10- 11); otherwise, it is classified as a *delayed alert* (Line 8). By this algorithm, the *Alert Monitor* achieves real time alert-packet alignment and report of delayed alerts that suggest potential issues.

C Experiment Statistics

Table 5 presents the experiments we carried out and the number of unique rule enforcement issues reported in each. We employed four different mutation strategies and three selection algorithms to ensure comprehensive coverage during the evaluation. Table 6 presents the proportion of each category

Table 5: The execution time of each experiment and the number of unique rule enforcement issues (REIs) reported.

Mutation Strategy	Selection Algorithm	Time Cost	# of unique REIs			
			HTTP	FTP	DNS	SIP
Pass-Through	Sequential	20 min	556	241	83	109
Obfuscation	Sequential	5 h	135	51		11
Repetition (Element-wise)	Sequential	7 h	911	124	34	15
Repetition (Block-wise)	Sequential	7 h	2,130	157	38	23
Blending	Combination (n=2)	8 h	1,621	3,390	437	390
Blending	Random (n=3)	24 h	367	0	50	0
Blending	Random (n=4)	24 h	0		0	

of reported cases.

Table 6: The distribution of reported issues by category.

Severity	Issue Name	Prop.	
High (H)	Redundant Sticky Buffers	0.8%	65.7%
	Outsized Option Values	0.8%	
	Misused Keywords	0.7%	
	Invalid Status Code	5.3%	
	Corrupted Content Length	4.3%	
	Complex Regex	21.1%	
	Inconsistent Interpretation	1.2%	
	Obfuscated Characters	1.8%	
	Limited Packet Size	29.8%	
Medium (M)	Overbroad Signatures	1.1%	21.2%
	Duplicate Rules	0.5%	
	Overlapping Rules	0.8%	
	Reduced Reliability	10.9%	
	Direction Mismatch	0.8%	
	Transport Layer Mismatch	1.5%	
	Long Segmentation	5.8%	
Low (L)	Mis-stripped Spaces	8.0%	13.1%
	Omitted Checks	5.1%	

D Other Issues

This section provides supplementary discussion of the remaining issues listed in Table 2.

• *Mis-stripped Spaces*: We found that Suricata automatically strips the trailing whitespace from the HTTP header field values. As an example, Figure 18 shows an HTTP packet crafted by combining Rule 1:32525:5 and Rule 1:24474:2. The packet contains content of "`User-Agent: tnftp/X-Puffin-UA: `", which satisfies Rule 1:24474:2. As expected, Snort3 raised the corresponding alert, whereas Suricata did not. Debugging shows that Suricata trims trailing spaces in HTTP header values; here it removes the space in the content, so the content no longer matches Rule 1:24474:2. Although it remains unclear whether this behavior could be exploited by attackers for malicious purposes, we firmly believe that such inconsistencies in rule enforcement across different NIDS platforms should be addressed.

• *Omitted Checks*: We found that Snort3 does not enforce strict DNS protocol conformance checks prior to rule evaluation. As a result, certain DNS packets that are not well-formed can still match DNS signatures and raise alerts, while they are not evaluated in Suricata and Snort2.

• *Reduced Reliability*: Our evaluation showed that Suricata may not reliably generate alerts under high-load conditions. It occasionally failed to trigger alerts for malicious packets.

• *Direction Mismatch*: It is revealed that the direction operator ("->") does not take effect for DNS traffic in Snort3. Packets triggered the rule whenever the attack signature matched, regardless of flow direction, which could cause false positives. We did not observe this behavior in Snort2 or Suricata.

• *Transport Layer Mismatch*: Similar to *Direction Mismatch*, the "*tcp*" and "*udp*" keywords in the rule header do not take effect for SIP and DNS traffic in Snort3. As illustrated in Figure 21, Rule 1:51495:2 and Rule 1:51751:2 differ only in the transport layer protocol ("*tcp*" vs. "*udp*"), yet both are triggered by the same TCP-encapsulated packet.

• *Long Segmentation*: After setting the inspection threshold to 100,000 bytes, we observed another interesting behavior: all NIDS engines segmented packets into 10,000 byte slices and re-applied rule matching to each slice.

E Supporting Materials

This section presents detailed examples that illustrate the issues identified by **NIDSFuzz** and the proposed countermeasures.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21
  ( ......
  content:"%",fast_pattern,nocase;
  pcre:"/\s+.*?%.*?%/ims";
  sid:2417; rev:17;
  ...... )
```

Figure 10: The signature of Rule 1:2417:17.

```
.....
  file_data; file_data;
  content:"CustomEvent",nocase;
  content:"connect",within 50,nocase;
  content:"CustomEvent",nocase;
  content:"message",within 50,nocase;
  content:"message_type",nocase;
  content:"launch_meeting",within 50,nocase;
  content:"GpcComponentName",fast_pattern;
  content:!"YXRtY2NsaS5ETEw=",within
20,nocase;
  sid:41408; rev:3;
  ......)
```

Figure 11: The signature of Rule 1:41408:3.

```
......
  file_data;
  content:"<applet",nocase;
  isdataat:5000,relative;
  content:!"</applet",within 5000,nocase;
  content:"<param",within 500,distance
1500,nocase;
  content:"<param",within 50,nocase;
  content:"<param",within 50,nocase;
  sid:18809; rev:13;
  ......)
```

Figure 12: The signature of Rule 1:18809:13.

```
......
  content:"CSeq|3A|"; fast_pattern:only;
  pcre:"/^CSeq\x3A[^\r\n]+[\x01-\x08\x0B\
x0C\x0E-\x1F\x80-\xFF]/Hsmi";
  sid:20306; rev:4;
  ......)
```

```
......
  content:"CSeq|3A|",fast_pattern,nocase;
  sip_header;
  pcre:"/^CSeq\x3A[^\r\n]+[\x01-\x08\x0B\
x0C\x0E-\x1F\x80-\xFF]/ims";
  sid:20306; rev:4;
  ......)
```

Figure 13: The signature of Rule 1:20306:4 for Snort2/Suricata (upper) and Snort3 (lower).

```
......
  file_data;
  content:".getClientRects|28 29|",fast_pattern,no
case;
  content:"for|28|n=0|3B|n<tList.length|3B|n++
|29 7B|";
  content:"tList|5B|n|5D|.tBodies|5B|0|5D|.appen
dChild|28|document.createElement|28 27|tr|27
29 29|";
  content:"tList|5B|n|5D|.removeChild|28|tList|5
B|n|5D|.children|5B|0|5D 29|";
  sid:23609; rev:9;
  ......)
```

```
......
  file_data;
  content:".getClientRects|28 29|",fast_pattern,no
case;
  content:"for|28|n=0|3B|n<tList.length|3B|n++
|29 7B|";
  content:"tList|5B|n|5D|.tBodies|5B|0|5D|.appen
dChild|28|document.createElement|28 27|tr|27
29 29|";
  content:"tList|5B|n|5D|.removeChild|28|tList|5
B|n|5D|.children|5B|0|5D 29|";
  sid:31469; rev:2;
  ......)
```

Figure 14: Rules 1:23609:9 and 1:31469:2 are duplicate rules.

```
......
  file_data;
  content:"|3A|first-letter { float|3A| ",fast_patte
rn,nocase;
  content:"|5B 22|setAttribute|22 5D 28|'style', '
display|3A| table-cell'";
  content:"|5B 22|style|22 5D 5B 22|display|22
5D|= 'none'",within 62;
  sid:44978; rev:3;
  ......)
```

```
......
  file_data;
  content:"|3A|first-letter { float|3A| ",fast_patte
rn,nocase;
  content:"|5B 22|setAttribute|22 5D 28|'style', '
display|3A| table-cell'";
  content:"|5B 22|style|22 5D 5B 22|display|22
5D|= 'none'",within 60;
  sid:29579; rev:3;
  ......)
```

Figure 15: Rules 1:44978:3 and 1:29579:3 are overlapping rules.

```
.....
  file_data;
  content:"document.getElementById(";
  content:"path",within 10;
  content:".pathSegList.getItem(",fast_pattern;
  content:"-",within 5;
  sid:15164; rev:10;
  .....)
```

```
.....
  http_header;
  content:!"mozilla";
  http_uri;
  content:".xpi",nocase; pcre:"/\.xpi$/i";
  sid:26659; rev:4;
  .....)
```

Figure 16: Rules 1:15164:10 and 1:26659:4 are orthogonal rules.

Rule 1:56449:2

```
1  alert tcp $EXTERNAL_NET $FILE_DATA_PORTS -> $HOME_NET any
2  (
3    msg:"BROWSER-CHROME Microsoft Teams Electron
   framework command injection attempt";
4    flow:to_client,established;
5    file_data;
6    content:"msteams:";
7    content:"--browser-subprocess-path=",within 150,nocase;
8    metadata:policy max-detect-ips drop;
9    service:ftp-data,http,imap,pop3;
10   reference:cve,2018-1000006;
11   reference:url,electronjs.org/blog/protocol-handler-fix;
12   classtype:attempted-user;
13   sid:56449; rev:2;
14 )
```

Rule 1:60282:2

```
1  alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any
2  (
3    msg:"BROWSER-CHROME Intent handling downgrade
   attempt";
4    flow:to_client,established;
5    http_header;
6    content:"com.sec.android.app.sbrowser",fast_pattern,nocase;
7    http_stat_code;
8    content:"302";
9    http_header;
10   content:"Location:",nocase;
11   content:"intent:",nocase;
12   metadata:policy max-detect-ips drop;
13   service:http;
14   reference:cve,2021-38000;
15   reference:cve,2022-2856;
16   reference:url,chromereleases.googleblog.com/2021/10/
   stable-channel-update-for-desktop_28.html;
17   reference:url,chromereleases.googleblog.com/2022/08/
   stable-channel-update-for-desktop_16.html;
18   classtype:attempted-user;
19   gid:1; sid:60282; rev:2;
20 )
```

The Packet for both Rule 1:56449:2 and Rule 1:56449:2

```
1  HTTP/1.1 302302 OK\r\n
2  Content-Length: 34\r\n
3  com.sec.android.app.sbrowserLocation:intent:com.sec.android.
   app.sbrowserLocation:intent:\r\n
4  \r\n
5  msteams:--browser-subprocess-path=
```

Figure 17: Packets with an invalid status code can evade detection.

Rule 1:32525:5

```
1  alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS
2  (
3    msg:"BROWSER-OTHER FreeBSD tnftp client detected";
4    flow:to_server,established;
5    content:"User-Agent|3A| tnftp/"; fast_pattern:only;
6    http_header;
7    flowbits:set,tnftp;
8    flowbits:noalert;
9    metadata:policy max-detect-ips alert, service http;
10   classtype:protocol-command-decode;
11   sid:32525; rev:5;
12 )
```

Rule 1:24474:2

```
1  alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS
2  (
3    msg:"BROWSER-OTHER Puffin Browser usage detected";
4    flow:to_server,established;
5    content:"X-Puffin-UA|3A| "; fast_pattern:only;
6    http_header;
7    metadata:policy max-detect-ips drop, service http;
8    reference:url,www.puffinbrowser.com;
9    classtype:policy-violation;
10   sid:24474; rev:2;
11 )
```

The Packet for both Rule 1:32525:5 and Rule 1:24474:2

```
1  GET /connecttest.txt HTTP/1.1\r\n
2  Content-Length: 0\r\n
3  User-Agent: tnftp/X-Puffin-UA: ␣ \r\n
4  \r\n
5
```

Figure 18: Suricata automatically strips the trailing spaces in HTTP header fields.

Rule 1:16767:9

```
1  alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any
2  (
3    msg:"BROWSER-PLUGINS AwingSoft Web3D Player
   SceneURL ActiveX clsid access";
4    flow:to_client,established;
5    file_data;
6    content:"17A54E7D-A9D4-11D8-9552-00E04CB09903";
   nocase;
7    pcre:"/(<object\s*[^>]*\s*id\s*=\s*(?P<m1>\x22|\
   x27|)(?P<id1>.+?)(?P=m1)(\s|>)[^>]*\s*classid\s*=\
   s*(?P<q1>\x22|\x27|)\s*clsid\s*\x3a\s*{?\s*17A54E7D-
   A9D4-11D8-9552-00E04CB09903\s*}?\s*(?P=q1)(\
   s|>).*(?P=id1)\s*\.\s*(SceneURL)|<object\s*[^>]*\
   s*classid\s*=\s*(?P<q2>\x22|\x27|)\s*clsid\s*\x3a\s*{?\
   s*17A54E7D-A9D4-11D8-9552-00E04CB09903\s*}?\
   s*(?P=q2)(\s|>)[^>]*\s*id\s*=\s*(?P<m2>\x22|\
   x27|)(?P<id2>.+?)(?P=m2)(\s|>).*(?P=id2)\.(SceneURL))/
   siO";
8    metadata:policy max-detect-ips drop, service http;
9    reference:cve,2009-4588;
10   reference:cve,2009-4850;
11   classtype:attempted-user;
12   sid:16767; rev:9;
13 )
```

The packet for Rule 1:16767:9

```
1  HTTP/1.0 200 OK\r\n
2  Server: SimpleHTTP/0.6 Python/3.10.12\r\n
3  Date: Wed, 09 Apr 2025 12:21:32 GMT\r\n
4  Content-Type: text/plain\r\n
5  Content-Length: 145\r\n
6  \r\n
7  <object classid='clsid:17A54E7D-A9D4-11D8-9552-
   00E04CB09903' id=testid> ␣␣␣ testid.SceneURLaaaaa
   aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 19: Rules with complex pcre patterns cannot be enforced as intended in Suricata.

Rule 1:38950:4 for Snort3

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any
(
   msg:"MALWARE-CNC Win.Trojan.PassStealer passwords exfiltration attempt";
   flow:to_server;
   file_data;
   content:"Passwords Recorded On ",fast_pattern;
   content:"Time of Recording:",within 20,distance 22;
   content:"IP Address",within 12,distance 15;
   metadata:impact_flag red,ruleset community; service:ftp;
   sid:38950; rev:4; )
```

Rule 1:38950:4 for Snort2 and Suricata

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any
(
   msg:"MALWARE-CNC Win.Trojan.PassStealer passwords exfiltration attempt";
   flow:to_server;
   file_data;
   content:"Passwords Recorded On "; fast_pattern;
   content:"Time of Recording:"; within:20; distance:22;
   content:"IP Address"; within:12; distance:15;
   metadata:impact_flag red, ruleset community, service ftp;
   sid:38950; rev:4;)
```

The Packet for Rule 1:38950:4

```
USER ftp\r\n
Passwords Recorded On ;tzB%x*[cjx4<:vjY+~s4CTime of Recording:67}OP(;otKOET*kIP Address
```

Figure 20: Rule 1:38950:4 is enforced inconsistently across Snort3, Suricata, and Snort2 due to their differing interpretations of *file_data*.

Rule 1:51495:2

```
alert udp $EXTERNAL_NET any -> $HOME_NET $SIP_PORTS
(
   msg:"PROTOCOL-VOIP SIP Torture negative Content-Length attempt";
   flow:to_server;
   content:"SIP/2.0";
   content:"|0D 0A|Content-Length: -",fast_pattern,nocase;
   service:sip;
   sid:51495; rev:2;
   ......)
```

Rule 1:51751:2

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $SIP_PORTS
(
   msg:"PROTOCOL-VOIP SIP Torture negative Content-Length attempt";
   flow:to_server,established;
   content:"SIP/2.0";
   content:"|0D 0A|Content-Length: -",fast_pattern,nocase;
   service:sip;
   sid:51751; rev:2;
   ......)
```

The Packet for Rule 1:51495:2

```
INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/TCP client.atlanta.example.com:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
To: Bob <sip:bob@biloxi.example.com>
Call-ID: 3848276298220188511@atlanta.example.com
CSeq: 2 INVITE
Content-Type: application/sdp
Content-Length: 26

SIP/2.0
Content-Length: -
```

Figure 21: Single TCP data packet simultaneously activates TCP-based Rule 1:51495:2 and UDP-based Rule 1:51751:2.

Rule 1:18508:6

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any
(
   msg:"BROWSER-WEBKIT Apple Safari WebKit ParentStyleSheet exploit attempt";
   flow:to_client,established;
   file_data;
   content:".sheet.rules[",fast_pattern,nocase;
   pcre:"/getElementById\(\x22(.*?)\x22\)\.sheet\.rules\[\d+\].*?([A-Z\d_]+)\s*=\s*document\.getElementById\(\x22\1\x22\).*?\s+\2\.parentElement/ims";
   service:http;
   sid:18508; rev:6;
   ......)
```

The packet triggering Rule 1:18508:6

```
GET /connecttest.txt HTTP/1.1\r\n
Host: www.msftconnecttest.com\r\n
Connection: Close\r\n
User-Agent: Microsoft NCSI\r\n
Content-Type: text/plain\r\n
Content-Length: 95\r\n
\r\n
getElementById(\"test\").sheet.rules[0] VAR = document.getElementById(\"test\"); VAR.parentElement;
```

The obfuscated packet not triggering Rule 1:18508:6

```
GET /connecttest.txt HTTP/1.1\r\n
Host: www.msftconnecttest.com\r\n
Connection: Close\r\n
User-Agent: Microsoft NCSI\r\n
Content-Type: text/plain\r\n
Content-Length: 99\r\n
\r\n
getElementById(\"test\").sheet.rules%5B0%5D VAR = document.getElementById(\"test\"); VAR.parentElement;
```

Figure 22: Obfuscation of specific characters can evade Snort3 detection.

Rule 1:14266:12

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any
(
   msg:"BROWSERPLUGINS Microsoft Windows Image
Acquisition Logger ActiveX clsid access";
   flow:to_client,established;
   file_data;
   content:"A1E75357-881A-419E-83E2-BB16DB197C68";
fast_pattern:only;
   pcre:"/(<object\s*[^>]\sid\s*=\s*(?P<m1>\x22|\
x27|)(?P<id1>.+?)(?P=m1)(\s|>)[^>]\sclassid\s*=\
s*(?P<q1>\x22|\x27|)\sclsid\s\x3a\s*{?\sA1E75357-881A-
419E-83E2-BB16DB197C68\s}?\s*(?P=q1)(\s|>).(?P=id1)\
s.\s*(Save)|<object\s*[^>]\sclassid\s*=\s*(?P<q2>\x22|\
x27|)\sclsid\s\x3a\s*{?\sA1E75357-881A-419E-83E2-
BB16DB197C68\s}?\s*(?P=q2)(\s|>)[^>]\sid\s*=\
s*(?P<m2>\x22|\x27|)(?P<id2>.+?)(?P=m2)(\
s|>).(?P=id2).(Save))\s*(/Osi";
   metadata:policy max-detect-ips drop, service http;
   reference:bugtraq,31069;
   reference:cve,2008-3957;
   classtype:attempted-user;
   sid:14266; rev:12;
)
```

Rule 1:99999:1, obtained by splitting Rule 1:14266:12

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any
(
   msg:"BROWSERPLUGINS Microsoft Windows Image
Acquisition Logger ActiveX clsid access";
   flow:to_client,established;
   file_data;
   content:"A1E75357-881A-419E-83E2-BB16DB197C68";
fast_pattern:only;
   pcre:"/(<object\s*[^>]*\s*id\s*=\s*(?P<m1>\x22|\
x27|)(?P<id1>.+?)(?P=m1)(\s|>)[^>]*\s*classid\s*=\
s*(?P<q1>\x22|\x27|)\s*clsid\s*\x3a\s*{?\s*A1E75357-
881A-419E-83E2-BB16DB197C68\s*}?\s*(?P=q1)(\
s|>).*(?P=id1)\s*\.\s*(Save))\s*\(/Osi";
   metadata:policy max-detect-ips drop, service http;
   reference:bugtraq,31069;
   reference:cve,2008-3957;
   classtype:attempted-user;
   sid:99999; rev:1;
)
```

Rule 1:99998:1, obtained by splitting Rule 1:14266:12

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any
(
   msg:"BROWSERPLUGINS Microsoft Windows Image
Acquisition Logger ActiveX clsid access";
   flow:to_client,established;
   file_data;
   content:"A1E75357-881A-419E-83E2-BB16DB197C68";
fast_pattern:only;
   pcre:"/(<object\s*[^>]*\s*classid\s*=\s*(?P<q2>\x22|\
x27|)\s*clsid\s*\x3a\s*{?\s*A1E75357-881A-419E-83E2-
BB16DB197C68\s*}?\s*(?P=q2)(\s|>)[^>]*\s*id\s*=\
s*(?P<m2>\x22|\x27|)(?P<id2>.+?)(?P=m2)(\
s|>).*(?P=id2)\.(Save))\s*\(/Osi";
   metadata:policy max-detect-ips drop, service http;
   reference:bugtraq,31069;
   reference:cve,2008-3957;
   classtype:attempted-user;
   sid:99998; rev:1;
)
```

Figure 23: Splitting complex rules into multiple simpler rules.