Performance Improvement by Coordinating Configurations of Independently-managed NIDS

Miyuki Hanaoka[†] , Kenji Kono[†, ††], Toshio Hirotsu[†††], and Hirotake Abe[††††],

[†]Keio University, Yokohama, Japan
[††]CREST, Japan Science and Technology Agency, Saitama, Japan
[†††]Hosei University, Tokyo, Japan [††††]Osaka University, Osaka, Japan

Summary
Because of today's increased traffic volume and sophisticated attacks, implementing a network intrusion detection/prevention system (NIDS/NIPS) with a single workstation has been challenging. In this paper, we propose Brownie, a system for improving performance by coordinating configurations of already-existing, independently-managed NIDSs in an organization. Instead of installing one expensive hardware or parallel NIDSs at a network entry point, Brownie achieves performance improvement by 1) offloading overloaded NIDS, and 2) eliminating redundant rules. First, Brownie exchanges NIDSs' load status and transfers some rules from overloaded to light-loaded NIDSs, which prevents the overloaded NIDSs from bottlenecking the network. Second, if some NIDSs on a network path enable the same rules, Brownie eliminates the redundant rules, which reduces the aggregate overhead of the NIDSs. The experimental results with a web server benchmark suggest that Brownie increases the benchmark throughput by more than 10%. In addition, Brownie running with a university full-packet trace successfully offloads overloaded NIDS and eliminates redundant rules.

Key words:
Network Security, Network Intrusion Detection/Prevention System, Performance

1. Introduction

Network intrusion detection/prevention systems (NIDSs/NIPSs) are widely used for detecting malicious attacks by monitoring the incoming and outgoing traffic for suspicious payloads. Most NIDSs are signature-based, which relies on a set of rules (or signatures) that are scanned over network packets. Although signature matching is a highly computationally intensive process, many NIDSs, both open-source and commercial, are based on inexpensive commodity hardware. Consequently, implementing a NIDS/NIPS with a single workstation is becoming more challenging. This is because 1) network traffic volumes and rates are rising exceedingly compared to computer processor speeds, and 2) attacks are becoming more sophisticated and thus require more complex and in-depth analysis to detect. Furthermore, a NIPS has more performance impact than a NIDS. This is because it suspends a packet until it finishes checking it [1].

Faced with this performance gap, we have two options to take. One is turning to expensive, custom hardware such as ASICs or FPGAs. Although it gains effective performance improvement, hardware implementation is expensive and lacks flexibility. Another research efforts have explored the use of parallel processing among multiple commodity computers, such as NIDS Cluster [2] and Active Splitter [3]. With these approaches, a frontend divides the traffic stream among the analysis nodes, each of which receives a share of the total network traffic to analyze in depth. Although these approaches are quite effective, introducing these is not trivial. The primary reason is the cost of introducing many machines. Not only the cost for the machines, but also the configuration and maintenance cost might be expensive. In addition, the administrator has to find a large space and provide enough electric power supply for these machines.

In this paper, we propose another approach to improve performance. We show that coordinating *already-existing, independently-managed* NIDSs in an organization network is worthwhile way to achieve total performance improvement of NIDSs in the network. Our key observation is that most organizations, such as universities or companies, have *several* NIDSs managed by different administrators inside their internal networks, not only at the network entry point. For example at a university, some departments or laboratories introduce and manage their own NIDSs even if the university has a NIDS at the top level of the network. Since these NIDSs are configured by each administrator independently, they are not optimized with each other. For example, several NIDSs enable the same rules, and thus traffic must be checked against them many times. In addition, an overloaded NIDS becomes bottleneck in the entire network while other NIDSs are light-loaded.

We show that exploiting tree-like structure of the NIDSs inside the same network improves the performance of NIDSs in the whole network. By communicating with other NIDSs on a network path, NIDSs exchange their own load status and configuration, and then reconfigure

Manuscript received May 5, 2011
Manuscript revised May 20, 2011

themselves to offload overloaded machines or eliminate redundant rules. By doing so, we can achieve total performance improvement in networks. If we move some rules that are originally on overloaded NIDSs to others, we can offload the overloaded NIDSs. If we eliminate redundant rules among some NIDSs, one packet will be checked only once somewhere in the network path and thus latency can be reduced. Existing approaches seek to improve the performance of *one* NIDS at the network entry point by using several machines or adopting new efficient algorithms. On the other hand, our approach aims to improve the entire performance of many NIDSs, not just one.

Note that our approach does not sacrifice the original security strength for performance. Our approach ensures that all traffic is checked against the same rule set as before. We transfer or disable rules so that all the traffic which passes the NIDS that disables the rules is surely checked by another NIDS that enables the rules. In other words, our approach only changes *where* packets are checked, but does not change *what rules* a packet is checked against on the way to its destination.

We developed Brownie to show the efficiency of our approach. The experimental results with a benchmark workload suggest that Brownie offloads the overloaded NIDSs and eliminates redundant rules, and the benchmark throughput increases by more than 10%. We also ran it with a full-packet trace captured at the network entry point of Toyohashi University of Technology with /16 network. Even with the real traffic trace, Brownie successfully offloads the overloaded NIDS and eliminates redundant rules.

We structure the remainder of this paper as follows. In Section 2, we briefly discuss related work. We give our approach of collaborating NIDS in Section 3, and describe the detailed design and implementation developed in Brownie in Section 4. In Section5, we discuss our benchmark-based and trace-based experiments to show the efficiency of Brownie. Section 6 suggests that Brownie can be applied to other issues, and we conclude in Section 7.

2. Related Work

There has been constant research for improving the performance of NIDS. This research is roughly divided into two categories: parallelizing analysis using many machines and improving the performance of one NIDS.

Parallelizing analysis approach is further divided into two categories depending on what is distributed: traffic or signature. Parallelizing analysis by distributing traffic include NIDS Cluster [2], Active Splitter [3], and a system developed by Kruegel et al. [4]. With this approach, a *frontend* divides the traffic stream among analysis nodes, or often called *sensors*, each of which receives a share of the total network traffic to analyze in depth. By dividing the analyzing process among sensors, we can achieve better performance than a NIDS that consists of a single computer. Kruegel et al. propose a three-stage architecture for dividing traffic into sensors to support in-depth, stateful NIDS on high-speed links [4]. To keep up with the high-speed traffic, the traffic is first captured by a traffic scatter, which equally distributes packets to a set of traffic slicers, in a round-robin fashion. Then to preserve detection semantics, the slicers examine the packets for determining a suitable set of sensors for analysis. Active Splitter [3] presents three performance-enhancing techniques implemented in a frontend: (1) early filtering/forwarding, where a fraction of the packets is processed on the frontend instead of the sensors, (2) locality buffering, where the frontend reorders packets to improve memory access locality on the sensors, and (3) cumulative acknowledgments, which optimizes the coordination between the frontend and the sensors. NIDS Cluster [2] explores schemes and implementation for better communication between sensors. Rather than just aggregating alerts, sensors in NIDS Cluster collaboratively analyze a traffic stream by exchanging low-level analysis state. Although these approaches are quite effective, the cost of introducing one of these systems is not trivial. The administrator needs to buy, set up, and maintain many machines. In some situations, finding a space or supplying enough electric power may be difficult. Our approach utilizes already-existing NIDSs in an organization and improves performance by enabling coordination between them.

Parallelizing analysis by distributing signature is less explored [5],[6]. Salour and Su propose a two-layer NIDS to improve performance [5]. Their system targets passive NIDSs which monitor the same segment. It divides one signature database into two (or can be more) NIDSs and improves performance of each NIDS. Our approach is more general; it targets tree-like structure of NIDSs, which is typical in an organization network, and coordinates signature configurations among NIDSs in different subnetworks for performance. In addition, our approach can apply to in-line NIDSs or NIPSs. The patent [6] claims a system and method for dynamic signature distribution among network devices with intrusion detection functionality (including NIDSs). Its purpose is signature distribution and sharing, not performance improvement of NIDSs; it distributes a signature upon detecting the corresponding attack so that the attack can be detected by other devices, and does not consider the loads of NIDSs.

Improving a single NIDS performance has been researched more over the years from many perspectives. The most common perspective is pattern-matching algorithms, such as Aho-Corasick [7], Wu-Manber [8], Commentz-Walter [9], and others [10]-[12] because this is the most

critical operation that affects the performance of NIDS. Other approaches explore hardware-based implementation of NIDS on network processors [13]-[15] and FPGAs [16]-[19]. The most recent approach exploits the latest architectural support including multi-core processors [20] and graphic processors [21]-[23]. These techniques are complementary to ours, which seeks performance improvement by enabling collaboration between already-existing NIDSs.

3. Coordinating Independently-managed NIDSs

In this paper, we propose an approach to improve NIDS performance by coordinating already-existing, independently-managed NIDSs in a network. In our approach, NIDSs in the same organization network communicate with each other to exchange their load status and configuration, and then reconfigure automatically to offload overloaded machines or eliminate redundant rules. Instead of replacing a top-level network entry-point NIDS with a new high-performance but expensive NIDS, or with a parallel NIDS consisting of many machines, our approach leverages already-existing NIDSs, and our goal is to improve the total NIDS performance within an organization's network.

Our key observation is as follows: many organizations, such as universities or companies, have several NIDSs managed by different administrators inside their networks, not only a NIDS at the network entry point of the organization. As shown in Figure 1, in addition to NIDS A set at the network entry point, there are other NIDSs downstream, which may be set by smaller departments or divisions in the organization. For example at a university, NIDS B1 and B2 may be set by departments, and NIDS C by a laboratory. With our approach, a NIDS communicates with its parent and child NIDSs, which are the closest up- and down-stream NIDSs, respectively. For example, NIDS A communicates with NIDS B1 and B2, and NIDS B2 with NIDS A and C.

We propose two approaches to improve performance: 1) offloading overloaded NIDS by transferring rules, and 2) eliminating redundant rules among NIDSs.

3.1 Offloading Overloaded NIDSs

A NIDS becomes overloaded because of large traffic volume or rate, many enabled rules, or poor NIDS computational power. However, when we look at NIDSs on a network path, all the NIDSs on the path rarely become overloaded. When a NIDS becomes overloaded, we try to offload the overloaded NIDS by distributing the load among the light-loaded NIDSs on the network paths. This alleviates or even prevents the overloaded NIDS from

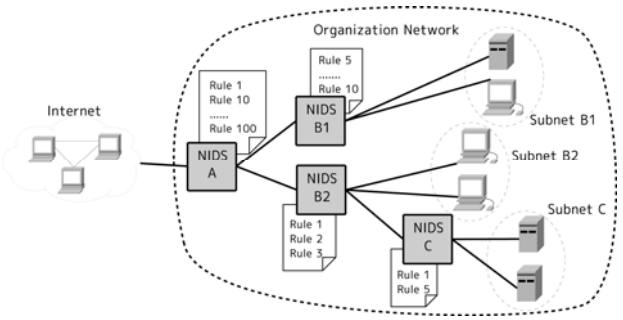


Fig. 1 Example Setting of NIDS

bottlenecking the network, and thus improves the throughput and reduces network latency of the network path. To offload the overloaded NIDS, we transfer a certain number of rules from the overloaded to lighter-loaded parent or child NIDSs. Transferring rules means disabling rules in the overloaded NIDSs and enabling them in the light-loaded NIDSs. We transfer rules and change the number of enabled rules to offload because other choices for offloading, such as changing the traffic volume or rate, or the computational power of the NIDS, are more difficult to change immediately.

In Figure 1, suppose that NIDS A becomes overloaded because it has many rules. On the other hand, the downstream NIDS B1 and B2 do not have many rules and are not overloaded. In this situation, NIDS A can transfer some rules, say rules 70-100, to NIDS B1 and B2. In other words, rules 70-100 become disabled in NIDS A and enabled in NIDS B1 and B2 instead. If this offloads NIDS A, the network throughput increases and the latency decreases.

Note that the original security level is preserved even after transferring a certain number of rules. This is because our approach ensures that all traffic is checked against the transferred rules somewhere on the network path before arriving at its destination. For example if some rules are transferred from NIDS A to NIDS B1 and B2 as described above, all the packets that pass the NIDS A always pass either NIDS B1 or B2. This ensures that all the packets are checked against the transferred rules at either NIDS B1 or B2. We discuss this security issue in more detail in Section 4.1.3.

3.2 Eliminating Redundant Rules

Some NIDSs on a network path usually enable a set of identical rules. This is because the NIDSs are managed independently by the administrators of each department. If some rules are enabled on several NIDSs on a network path, a packet passing along the path is checked several times against those rules. Eliminating these redundant rules will decrease network latency because packets are

checked only once. Redundant rules can be eliminated by disabling them in all except one NIDS on the network path.

In Figure 1, rule 1 is enabled in three NIDSs A, B2, and C. Packets destined to subnet C have to be checked against rule 1 three times. If only NIDS A enables rule 1 and NIDS B2 and C disable it, network latency will decrease.

Again, eliminating redundant rules causes no security degradation because NIDS disables the rules only if they are redundantly enabled on the network path. Other NIDSs on the path check the traffic against them. In the above example, even if NIDS B2 and C disable rule 1, the traffic from the Internet is checked by NIDS A against rule 1. For internal traffic between subnet B2 and subnet C, which cannot be checked by NIDS A, NIDS B2 still checks only the internal traffic against rule 1. We revisit this issue in Section 4.2.

4. Design and Implementation of Brownie

To demonstrate the feasibility of our approach, we design and implement Brownie[1], a NIDS coordination system. Each instance of Brownie is attached to and manages each NIDS, and communicates with other instances of Brownie. We achieve collaboration between NIDSs by means of collaboration between Brownies and managing each NIDS by each Brownie. A NIDS which a Brownie is attached to is called *the managing NIDS* of the Brownie. We will use the term Brownie to refer to both the entire system and each instance of the system.

To use a NIDS with Brownie, an administrator configures the information (such as IP address) of the parent NIDS. At the time of booting the NIDS, the corresponding Brownie establishes a connection with a Brownie attached to the configured parent NIDS for exchanging load and rule status. At the first communication, the Brownies exchange enabled/disabled rules.

For NIDSs, we used Snort, a widely used open-source NIDS. Although we used only Snort for all the NIDSs in the current implementation, we plan to extend our system to heterogeneous NIDS environments. Another NIDS candidate is Bro [24], another open-source NIDS. Since Bro provides mechanisms for dynamic rule rewriting and coordinating between sensors, it may be more suitable for our system. In the future, we will seek ways to address commercial and heterogeneous NIDSs environments to make our approach more effective.

[1] A Brownie is a legendary good-natured elf that performs helpful services.

4.1 Procedure for Offloading Overloaded NIDSs

The basic idea for offloading NIDSs is that a Brownie compares the loads of the managing NIDS and that of the child NIDSs, and transfers rules from higher-loaded one(s) to lower-loaded one(s). In other words, if the load of the managing NIDS is higher than that of all the child NIDSs, the Brownie transfers a certain number of rules from the managing NIDS to all the child NIDSs, and if the load of the managing NIDS is lower than that of all the child NIDSs, the Brownie transfers rules from all the child NIDSs to the managing NIDS. The Brownie repeats transferring rules until the load of the managing NIDS and that of the child NIDSs are balanced. Now we describe detailed procedure to determine 1) when offloading is done, and 2) which rules are transferred.

4.1.1 Determining When Offloading is Done

To determine whether offloading is necessary, a Brownie first measures the load of the managing NIDS. The load of a NIDS is characterized well by its resource consumption, especially by its CPU usage since most NIDS operation is CPU-intensive pattern matching. Even if all the NIDSs do not have the same machine configuration, the CPU usage represents the load of each NIDS.

Then the Brownie decides whether offloading should be done based on the measured CPU usage. Since our primary goal is to offload overloaded NIDS, the naive approach is that Brownie starts offloading if a NIDS becomes overloaded (its CPU usage reaches near 100%), and stops offloading if its CPU usage goes under a configured value. However, this approach arises some problems. First, it is difficult to decide the CPU usage for stopping the NIDS offloading. It should be low enough to effectively offload, but if it is too low, other NIDSs become overloaded by transferring too many rules. Second, if all the NIDSs are overloaded, they transfer rules to each other but offloading cannot be achieved. Finally, it is preferable that Brownie starts offloading before the NIDS becomes completely overloaded if other NIDSs are light-loaded.

Therefore, we adopt a load-balancing approach. A Brownie tries to equalize the CPU usage of the managing NIDS and that of the child NIDSs. The Brownie transfers rules if there is a considerable difference between the CPU usage of the managing NIDS and that of the child NIDSs, and stops transferring rules when the CPU usages are balanced. Unlike the naive approach, even if all the NIDSs are overloaded, no rule transfer occurs. In addition, Brownie can transfer rules to lighter-loaded NIDS before the managing NIDS gets overloaded. To use Brownie, we set a configurable parameter *Diff* for the acceptable CPU difference between NIDSs. By default, *Diff* is 5.

More precisely, a Brownie collects the CPU usage of the managing and the child NIDSs every T seconds. Let

c_{my} be the CPU usage of the managing NIDS, and c_i ($0 \le i < n$, where n is the number of the child NIDSs) be that of the child NIDSs. If c_{my} - max(c_i) > *Diff* (and because *Diff* is positive, this means c_{my} > max(c_i), then the Brownie transfers rules from the managing NIDS to all the child NIDSs. If min(c_i) − c_{my} > *Diff* (similarly, this means min(c_i) > c_{my}), then the Brownie transfers rules from the all the child to the managing NIDSs. Otherwise, i.e., if min(c_i) - *Diff* ≤ c_{my} ≤ max(c_i) + *Diff*, the CPU usages are balanced and the Brownie does not transfer any rules. The configurable parameter *T* affects the sensitivity to the load change, and we set *T* to 30 by default.

4.1.2 Determining Which Rules are Transferred

If we select an appropriate set of rules, Brownie can balance the load soon. Otherwise, the Brownie has to repeat the transfer many times, and thus it takes a longer time to balance the load. Because Brownie repeats offloading until the loads are balanced, it is acceptable not to select rules for offloading just once. Nevertheless, it is preferable to balance the load in a shorter time.

The simplest way to select rules is randomly selecting a constant number of rules. However, it is difficult to decide the constant number of transferring rules. If it is too small (e.g., one) it takes an extremely long time to offload. On the other hand, if the number of transferring rules is too large, the lighter-loaded NIDS may become overloaded and then has to transfer the same rules back to the originally higher-loaded NIDS.

Therefore, Brownie decides the number of transferred rules depending on the difference of the CPU usages of the managing and the child NIDSs; the larger the difference, the more rules are transferred. And then Brownie randomly selects the decided number of rules. Although each rule does not have the same load, we can assume that more rules may bring more load. In our setting, the number of transferred rules is *Factor* times the difference in CPU usage. *Factor* is a configurable parameter to decide how much the difference in CPU usage reflects the number of rules. More precisely, the number of transferred rules is *Factor* × (c_{my} - max(c_i)) or *Factor* × (min(c_i) − c_{my}) depending on whether rules are transferred from or to the managing NIDS, respectively. We use 10 as *Factor* by default.

4.1.3 Keeping Security

As briefly described in Section 3.1, Brownie does not degrade the original security by transferring rules. In this section, we discuss this issue in more detail. To ensure the security level after the rule transfer, we consider four cases along two axes (Table 1): 1) whether some of downstream machines which are directly connected to the upstream NIDS, are not NIDS or all are NIDSs, and 2) whether the rules are transferred from or to downstream NIDS.

In the first case, all the machines under upstream NIDS are NIDSs, and rules are transferred from upstream to downstream NIDSs. For example in Figure 1, only NIDSs (NIDS B1 and B2) exist under NIDS A and suppose rules are transferred from NIDS A to B1 and B2. In this case, all the packets that pass the upstream NIDS (NIDS A) always pass one of the downstream NIDSs (either NIDS B1 or B2). This ensures that all the packets are checked against the transferred rules at the downstream NIDS.

Table 1 Four Cases for Considering Security

		Rule transfer direction	
		from up- to downstream	from down- to upstream
Downstream machines	all are NIDS	1st case	2nd case
	some are not NIDS	3rd case	4th case

In the second case, all the machines under upstream NIDS are NIDSs as in the first case. But unlike the first case, rules are transferred from downstream to upstream NIDSs. For example, rules are transferred from NIDS B2 to A. In this case, traffic from the Internet to the downstream NIDS (NIDS B2) always passes and is checked by the upstream NIDS (NIDS A). However, care must be taken with internal traffic within the subnet under the downstream NIDS. For example, if rule 3 is transferred from NIDS B2 to A, the traffic between subnet B2 and subnet C cannot be checked by NIDS A. To check this traffic, the downstream NIDS B2 keeps the transferred rules 3 and checks only the traffic between a host in subnet B2 and a host in subnet C against rule 3, based on source and destination IP addresses of the packets. Because the traffic from the Internet is usually much larger than the internal traffic, we can offload NIDS B2 even if rule 3 is still enabled.

In the third case, normal hosts and downstream NIDSs are co-located under a NIDS. In Figure 1, hosts in subnet B2 and NIDS C are co-located under NIDS B2. In this case, rules are transferred from upstream to downstream NIDSs, for example from NIDS B2 to C. In this case, if rule 3 is transferred from NIDS B2 to C, disabling the rule in the upstream NIDS B2 results in the traffic destined for normal hosts in subnet B2 not being checked. To avoid this, the upstream NIDS B2 keeps enabling rule 3 for the traffic for the subnet B2. Because NIDS B2 does not check the traffic destined for subnet C, we expect NIDS B2 can be offloaded even if rule 3 is still enabled.

In the forth case, normal hosts and downstream NIDSs are co-located under a NIDS as in the third case. But in this case, rules are transferred from downstream to

upstream NIDSs. For example, rules are transferred from NIDS C to B2. This case can be dealt with the same manner as the second case; NIDS C keeps the rules for the traffic between hosts below the downstream NIDS (hosts in subnet C) and other traffic is checked by NIDS B2.

4.2 Procedure for Eliminating Redundant Rules

The procedure for eliminating redundant rules is straightforward. If a Brownie finds both a downstream and the managing NIDSs enable the same rules, it disables the rules in the downstream NIDS. As a result, all the redundant rules are enabled only in the managing NIDS. Because upstream and downstream Brownies exchange their enabled/disabled rules at the time of booting, they can easily find redundant rules.

Disabling redundant rules in downstream NIDS causes no security degradation for traffic from the Internet since the traffic passes the rule-enabled upstream NIDS before the rule-disabled downstream NIDS. In Figure 1, as explained in Section 3.2, even if rule 1 is disabled in NIDS B2 and C, the traffic from the Internet is checked by NIDS A against rule 1. For internal traffic between subnet B2 and subnet C, which cannot be checked by NIDS A, NIDS B2 still checks it against rule 1 in the same way as described in Section 4.1.3.

It is possible to eliminate redundant rules by enabling them in downstream instead of upstream NIDS. However, this requires a more complex procedure to ensure the security level. This is because no NIDS checks the traffic from the Internet to a subnet not below the downstream NIDS. For example in Figure 1, suppose NIDS B2 enables rule 1 and NIDS A and C disable it. Before eliminating the redundant rule 1, traffic destined for both subnet B1 and B2 is checked against rule 1 at NIDS A. After eliminating the rules, however, traffic destined for subnet B1 is not checked anywhere. To ensure the traffic is checked, we have to enable the rule 1 in NIDS B1. In contrast, enabling rules only in upstream NIDS does not need for any other NIDS to enable the rules (except for the downstream NIDS for internal traffic).

It is unlikely for the upstream NIDS to become suddenly overloaded even if all the redundant rules are enabled. This is because the rules are *already* enabled, rather than *become* enabled, in the upstream NIDS. Nevertheless, if the upstream NIDS becomes overloaded, the Brownie offloads it by transferring a certain number of rules to the downstream NIDSs. Because the downstream NIDSs disable the redundant rules, their loads may be lighter than before, and thus have room to accept more rules from the overloaded upstream NIDS.

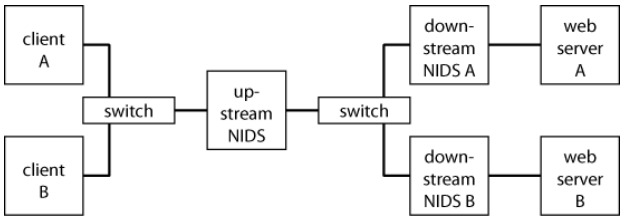


Fig. 2 Experimental Network Setting

4.3 Collecting Alert Logs

Because Brownie enables and disables rules automatically, some alerts are raised on a NIDS different from the one that originally enabled the rule. To notify the administrator of the NIDS that originally enabled it, Brownie keeps track of which rules are transferred to or from which NIDS. When a NIDS alerts an attack, the Brownie attached to the NIDS forwards the alert to a Brownie attached to the NIDS which originally enabled the rule. The Brownie that received it raises the alert on behalf of the managing NIDS. For example, it writes the received alert to the NIDS's alert log. The administrators of the NIDS can notice the alert even if the rule for the alert is not actually enabled in the NIDS.

5. Experiments

5.1 Synthetic Workload

5.1.1 Experimental Setup

To show that Brownie improves network performance, we first conducted experiments with synthetic workload, produced by a web server benchmark. We used seven machines: three NIDSs, two clients, and two servers, connected as shown in Figure 2. All the machines were connected via 1Gbps Ethernet. The upstream NIDS machine was equipped with two Intel Dual-Core Xeon 2.33GHz CPUs (only one core was enabled), 2GB memory, and a 250GB 7200rpm HDD, and all other machines were equipped with a Pentium 4 2.8GHz CPU, 512MB memory, and a 36GB 7200rpm HDD. In this setting, the upstream NIDS has higher performance than the downstream NIDSs. We used Fedora 8 (Linux 2.6.24) as the operating systems for all machines, Apache 2.2.8 as the web server, and Snort 2.8.0.1 [25] as NIDS with the rule set published on January 28th, 2008. We executed WebStone 2.5 [26], a standard benchmark for web servers, on each client with concurrency 10. The default configurations were used for Apache, WebStone, and Snort.

We measured the number of rules enabled in each NIDS, CPU usage of each NIDS, and the benchmark throughput every 10 seconds. For 30 minutes from the

start of each experiment, all the Brownies do not transfer or eliminate rules to see the performance with the initial rule setting.

5.1.2 Results of Offloading Overloaded NIDS

To show the effectiveness of offloading, we configured the initial Snort rule setting as **DOWN**; downstream NIDSs with the default rule set (# of rules is 8676) and upstream NIDS with no rules (# of rules is 0). Since the default rule set is overwork for the downstream NIDS, the downstream NIDSs get overloaded and become bottlenecks with this setting.

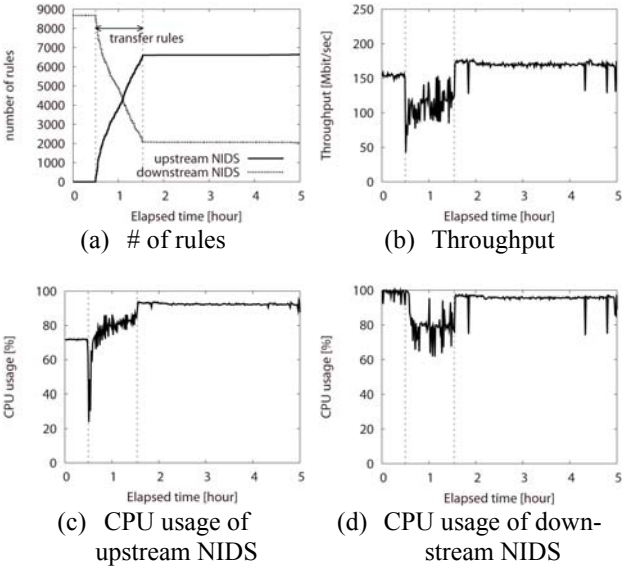
Figure 3 shows the experimental results. Two vertical lines show the times when the Brownies begin and stop offloading (or rule transfer). Figure 3 (a) shows the number of rules enabled in each NIDS. At 30 minutes after starting, the Brownie begins to offload and starts transferring rules. The number of rules in the upstream NIDS increases while that in the downstream NIDS decreases. Figures 3 (c) and (d) show the CPU usages of the upstream and downstream NIDSs, respectively. Because the CPU usages of the two downstream NIDSs do not show any noticeable difference, we only show one of them. With the initial setting, the CPU usages of the downstream NIDSs are constantly at 100%, whereas that of the upstream NIDS is less than 80%. After about an hour, the CPU usage of all the NIDSs become nearly the same and thus Brownie stops rule transfer. After this, the CPU usage of all the NIDSs reaches a little less than 100%, and thus they are equally loaded.

Figure 3 (b) shows the benchmark throughput. After the offloading finishes, the throughput increases from 154 Mbit/sec with the initial setting to 174 Mbit/sec, resulting in a 13% increase. Since the NIDSs need to be restarted for reconfiguration to take effect in the current implementation, the throughput drops temporarily during the transferring of rules. This can be mitigated if we use Elephant [27], which modifies the reloading sequence of Snort to reduce 20% of the time required for the rule reloading.

The numbers of rules after the rule transfer are 6600 at the upstream NIDS and 2076 at the downstream NIDS. Because the downstream NIDSs have lower performance than the upstream NIDS, the upstream NIDS enables much more rules than the downstream NIDS.

5.1.3 Results of Eliminating Redundant Rules

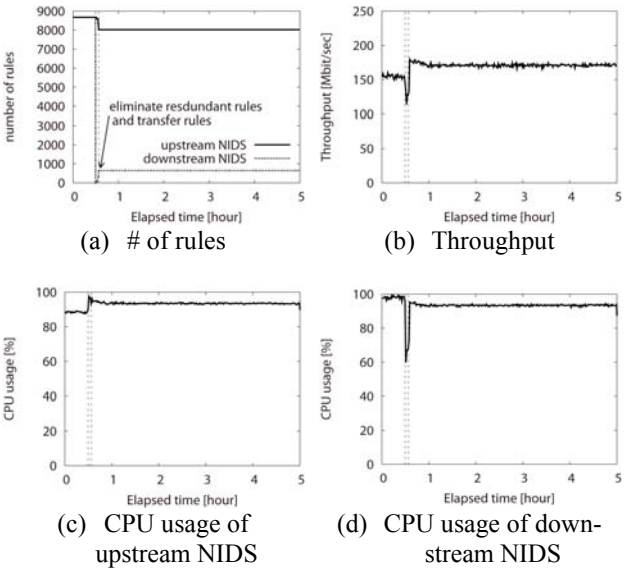
To measure the effectiveness of eliminating redundant rules by Brownie, we configured initial Snort rule setting as **BOTH**; all (up- and down-stream) NIDSs with the default rule set. Because all the NIDS have the same rule set, traffic must always be checked against the same rules twice. Figure 4 shows the experimental results.



(a) # of rules (b) Throughput

(c) CPU usage of upstream NIDS (d) CPU usage of downstream NIDS

Fig. 4 Results of Offloading Overloaded NIDS (Synthetic Workload: DOWN)



(a) # of rules (b) Throughput

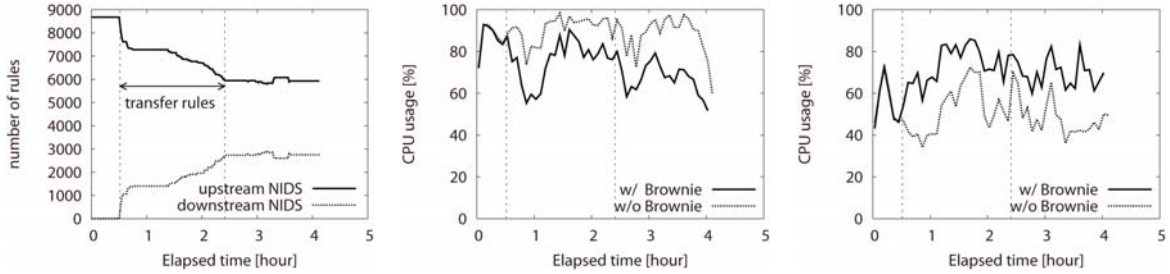
(c) CPU usage of upstream NIDS (d) CPU usage of downstream NIDS

Fig. 4 Results of Eliminating Redundant Rules (Synthetic Workload: BOTH)

Figure 4 (a) shows the number of rules. At 30 minutes after starting, the Brownie first eliminates all the redundant rules. Because the default rules are enabled in all the NIDSs, the rules in the downstream NIDSs are totally disabled and all the rules are enabled in the upstream NIDS. After that, a certain number of rules are transferred to the downstream NIDSs, and the offloading finishes in about 5 minutes. As shown in Figure 4 (b), the throughput increases from 155 Mbit/sec with the initial setting to 173 Mbit/sec, resulting in a 12% increase. This

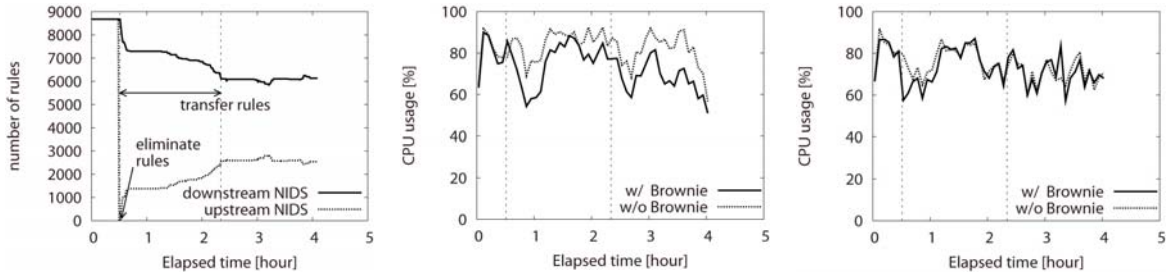
improvement is because multiple checks against the same rules become unnecessary.

Figures 4 (c) and (d) show the CPU usages of the upstream and downstream NIDSs, respectively. Since all the rules are enabled, the downstream NIDSs are overloaded in the initial setting and its CPU usage is near 100% for the first 30 minutes. After Brownie eliminates the redundant rules, the CPU usage of the downstream NIDS reduces and all the NIDSs show almost identical CPU usage.



(a) # of rule with Brownie (b) CPU usage of upstream NIDS (c) CPU usage of downstream NIDS

Fig. 6 Results of Offloading Overloaded NIDS (Real Workload: UP)



(a) # of rule with Brownie (b) CPU usage of upstream NIDS (c) CPU usage of downstream NIDS

Fig. 6 Results of Eliminating Redundant Rules (Real Workload: BOTH)

5.2 Real Workload

5.2.1 Experimental Setup

To assess Brownie with real network traffic, we operated Brownie on a captured packet trace. We captured a 4-day full-packet trace at the network entry point of Toyohashi University of Technology with /16 network starting from March 23rd, 2008, comprising 673GB, and 220,919,190 inbound and 168,454,019 outbound packets.

We used four machines: three NIDSs and a sink machine that replays and receives the captured packets. The NIDSs were connected in the same way as shown in Figure 2, except that the sink machine was connected to all the NIDSs. All the machines were connected via 1Gbps Ethernet. The upstream NIDS was configured with an Intel Quad-Core Xeon 2.33GHz CPU and 4GB memory, the downstream NIDSs with an Intel Pentium Dual-Core 2GHz CPU and 1GB memory, and the sink machine with an Intel Core2 Duo 2.4GHz CPU and 2GB memory. The software configurations were the same as those described in Section 5.1.1. To make the NIDSs well-loaded, we replayed the captured packets at top speed.

We measured the number of rules and the CPU usage. We could not measure throughput or latency because the workload was a captured packet trace. For comparison, we also ran the experiments and measured CPU usage without Brownie. Without Brownie, the initial rule setting does not change in all the NIDSs throughout the experiment.

5.2.2 Results of Offloading Overloaded NIDS

To show the effectiveness of offloading, we configure the initial Snort rule setting as **UP**; downstream NIDSs with no rules (# of rules is 0) and upstream NIDS with the default rule set (# of rules is 8676). Since the machine configuration is different from the one with synthetic workload, the initial Snort rule setting is different from the one in Section 5.1.2 to get either of NIDSs overloaded with the initial setting.

Figure 5 shows the experimental results. Figure 5 (a) shows the numbers of enabled rules in each NIDS with Brownie. At 30 minutes after staring, the Brownie begins to offload and starts transferring rules. The number of rules in the downstream NIDSs increases, while that in the

upstream NIDS decreases. After about 1.9 hours, rule transfer stops.

Figure 5 (b) shows the mean CPU usage of the upstream NIDS every 5 minutes. Figure 5 (c) shows that of the downstream NIDS. Without Brownie (in this case all the rules are in upstream NIDS), the CPU usage of the upstream NIDS sometimes reaches near 100%. With Brownie, the CPU usage of the upstream NIDS decreases and that of the downstream increases soon after the Brownie starts offloading. After finishing rule transfer, the CPU usages of all the three NIDS are well-balanced.

5.2.3 Results of Eliminating Redundant Rules

To measure the effectiveness of eliminating redundant rules, we configured initial Snort rule setting as **BOTH**; all (up- and down-stream) NIDSs with the default rule set. Because all the NIDS have the same rule set, traffic must always be checked against the same rules twice.

Figure 6 shows the experimental results. As shown in Figure 6 (a), all the rules in the downstream NIDSs are first disabled. After that, rules are transferred from the upstream to downstream NIDSs, similar to the initial setting UP, because the setting after eliminating the redundant rules is as the same as the initial setting UP. Figures 6 (b) and (c) show the CPU usage of the CPU usage of the upstream and downstream NIDS. Without the Brownie, the CPU usage of the upstream NIDS sometimes reaches near 100%. With the Brownie, the CPU usage of the upstream NIDS decreases as the rules are transferred while those of the downstream NIDSs do not increase compared to those without Brownie.

6. Applying Brownies to Other Issues

We proposed Brownie, mainly targeting on performance improvement, but we believe Brownie can provide other functionalities.

Fault-Tolerant NIDS: A Brownie can detect and cover failures of other NIDSs controlled by the collaborating Brownies. Since a Brownie periodically exchanges messages with the up/down-stream Brownies, it can detect the failure of the NIDSs. In addition, because it knows what rules are enabled in the failed NIDS, it can cover that failure; it enables the rules enabled by the failed NIDS. Even if a NIDS fails, another NIDS takes it over and continues to check network traffic and detect attacks.

Signature Synchronization: When a NIDS updates its signature database by downloading or automatically generating signatures [28]-[31], Brownie can distribute the update to other NIDSs. Then all the NIDSs in an organization are updated if only one of the NIDSs is up-to-date; the maintenance cost of signature database dramatically decreases.

7. Conclusion

Because of today's increased traffic volume and sophisticated attacks, it becomes difficult to implement a high performance network intrusion detection/prevention system (NIDS/NIPS). We proposed an approach for improving network performance by coordinating NIDSs independently-placed on the path of an organizational network. Our approach is less expensive to introduce; it does not need expensive hardware or many machines. With our approach, a NIDS exchanges its load status and configuration with other NIDSs to reconfigure the NIDSs for better performance. By offloading a certain number of rules from overloaded NIDS and eliminating redundant rules, we can balance loads and obtain better performance.

For the future, we plan to develop more efficient schemes for load-balancing. For example, by analyzing characteristics of traffic, we can estimate which rules produce heavier loads and transfer them before others. If we generate a resource consumption model like described by Dreger et al. [32], we can immediately estimate the best rule distribution. In addition, we will seek other applications for collaborating NIDSs as described in Section 6.

References

[1] V. Paxson, K. Asanovic, S. Dharmapurikar, J. Lockwood, R. Pang, R. Sommer, and N.Weaver, "Rethinking Hardware Support for Network Analysis and Intrusion Prevention," Proc. of the 1st USENIXWorkshop on Hot Topics in Security (HotSec ’06), pp.63--68, 2006.

[2] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney, "The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware," Proc. of the 10th Int’l Symp. on Recent Advances in Intrusion Detection (RAID ’07), pp.107--126, 2007.

[3] K. Xinidis, I. Charitakis, S. Antonatos, K.G. Anagnostakis, and E.P. Markatos, "An Active Splitter Architecture for Intrusion Detection and Prevention," IEEE Transactions on Dependable and Secure Computing, vol.3, no.1, pp.31--44, 2006.

[4] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer, "Stateful Intrusion Detection for High-Speed Networks," Proc. of the 2002 Symp. on Security and Privacy (S&P ’02), pp.285--293, 2002.

[5] M. Salour and X. Su, "Dynamic Two-Layer Signature-Based IDS with Unequal Databases," Proc. of the 4th Int’l Conf. on Information Technology (ITNG ’07), pp.77--82, 2007.

[6] J.J. Roese and R.W. Graham, "System and Method for Dynamic Distribution Of Intrusion Signatures." Patent:WO 2005/036339, 2005.

[7] A.V. Aho and M.J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," Communications of the ACM, vol.18, no.6, pp.333--340, 1975.

[8] S. Wu and U. Manber, "A Fast Algorithm for Multi-pattern Searching," tech. rep., TR-94-17, 1994.
[9] B. Commentz-Walter, "A String Matching Algorithm Fast on the Average," Proc. of the 6th Colloquium on Automata, Languages and Programming, pp.118--132, 1979.
[10] C.J. Coit, S. Staniford, and J. McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort," Proc. of DARPA Information Survivability Conf. &Exposition II (DISCEX '01), pp.367--373, 2001.
[11] M. Fisk and G. Varghese, "Applying Fast String Matching to Intrusion Detection." Technical Report In Preparation, successor to UCSD TR CS2001-0670, 2002.
[12] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection," Proc. of the IEEE Infocom Conf. 2004, pp.2628--2639, 2004.
[13] C. Clark,W. Lee, D. Schimmel, D. Contis, M. Kon´e, and A. Thomas, "A Hardware Platform for Network Intrusion Detection and Prevention," Proc. of the 3rd Workshop on Network Processors & Applications (NP3), 2004.
[14] H. Bos and K. Huang, "Towards Software-Based Signature Detection for Intrusion Prevention on the Network Card," Proc. of the 8th Int'l Symp. on Recent Advances in Intrusion Detection (RAID '05), pp.102--123, 2005.
[15] W. de Bruijn, A. Slowinska, K. van Reeuwijk, T. Hruby, L. Xu, and H. Bos, "SafeCard: A Gigabit IPS on the Network Card," Proc. of the 9th Int'l Symp. on Recent Advances in Intrusion Detection (RAID '06), pp.311--330, 2006.
[16] R. Sidhu and V.K. Prasanna, "Fast Regular Expression Matching using FPGAs," Proc. of the 9th Symp. on Field-Programmable Custom Computing Machines (FCCM '01), pp.227--238, 2001.
[17] Z.K. Baker and V.K. Prasanna, "Time and Area Efficient Pattern Matching on FPGAs," Proc. of the 12th Int'l Symp. on Field Programmable Gate Arrays (FPGA '04), pp.223--232, 2004.
[18] H. Song, T. Sproull, M. Attig, and J. Lockwood, "Snort Offloader: A Reconfigurable Hardware NIDS Filter," Proc. of the 15th Int'l Conf. on Field Programmable Logic and Applications (FPL '05), pp.493--498, 2005.
[19] J.M. Gonzalez, V. Paxson, and N. Weaver, "Shunting: A Hardware/Software Architecture for Flexible, High-Performance Network Intrusion Prevention," Proc. of the 14th Conf. on Computer and Communications Security (CCS '07), pp.139--149, 2007.
[20] R. Sommer, V. Paxson, and N. Weaver, "An architecture for exploiting multi-core processors to parallelize network intrusion prevention," Conurrency and Computation: Practice and Experience, vol.21, no.10, pp.1255--1279, 2009.
[21] N. Jacob and C. Brodley, "Offloading IDS computation to the GPU," Proc. of the 22nd Annual Computer Security Applications Conf. (ACSAC '06), pp.371--380, 2006.
[22] G. Vasiliadis, S. Antonatos, M. Polychronakis, E.P. Markatos, and S. Ioannidis, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors," Proc. of the 11th Int'l Symp. on Recent Advances in Intrusion Detection (RAID '08), pp.116--134, 2008.
[23] G. Vasiliadis, M. Polychronakis, S. Antonatos, E.P. Markatos, and S. Ioannidis, "Regular Expression Matching on Graphics Hardware for Intrusion Detection," Proc. of the 12th Int'l Symp. on Recent Advances in Intrusion Detection (RAID '09), 2009.
[24] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," Computer Networks, vol.31, no.23.24, pp.2435--2463, 1999.
[25] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," Proc. of the 13th USENIX Systems Administration Conf. (LISA '99), pp.229--238, 1999.
[26] Mindcraft, Inc., "WebStone." http://www.mindcraft.com/webstone/.
[27] M.G. Merideth and P. Narasimhan, "Elephant: Network Intrusion Detection Systems that Don't Forget," Proc. of the 38th Annual Hawaii Int'l Conf. on System Science (HICSS '05), pp.309c--309c, 2005.
[28] C. Kreibich and J. Crowcroft, "Honeycomb . Creating Intrusion Detection Signatures Using Honeypots," Proc. of the 2nd Workshop on Hot Topics in Networks (HotNets-II), 2003.
[29] H.A. Kim and B. Karp, "Autograph: Toward Automated, Distributed Worm Signature Detection," Proc. of the 13th USENIX Security Symp., pp.271.286, 2004.
[30] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically Generating Signatures for Polymorphic Worms," Proc. of the 2005 Symp. on Security and Privacy (S&P '05), pp.226--241, 2005.
[31] X. Wang, Z. Li, J. Xu, M.K. Reiter, C. Kil, and J.Y. Choi, "Packet Vaccine: Black-box Exploit Detection and Signature Generation," Proc. of the 13th Conf. on Computer and Communications Security (CCS '06), pp.37--46, 2006.
[32] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer, "Predicting the Resource Consumption of Network Intrusion Detection Systems," Proc. of the 11th Int'l Symp. on Recent Advances in Intrusion Detection (RAID '08), pp.135--154, 2008.

Miyuki Hanaoka received her B.E. degree from the University of Electro-Communications in 2005, and M.E. from Keio University in 2007. Her research interests include network security and system software. She is a member of IEEE, ACM, and USENIX.

Kenji Kono received the B.Sc. degree in 1993, M.Sc. degree in 1995, and Ph.D. degree in 2000, all in computer science from the University of Tokyo. He is an associate professor of the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software and Internet security. He is a member of the IEEE/CS, ACM and USENIX

Toshio Hirotsu received Ph.D. degree in computer science from Keio University in 1995. From 1995 to 2004, he worked in NTT Laboratories, Japan. He was in the Department of Information and Computer Science at Toyohashi University of Technology as an Associate Professor from 2004 to 2009. He is currenty a Professor of the faculty of the Computer and Information Science at Hosei University. His research interests include system software for Internet and the ubiquitous environment.

Hirotake Abe received the B.Eng. degree in 1999, the M.Eng. degree in 2001, and the Ph.D. degree in 2004, all from University of Tsukuba, Japan. From 2004 to 2007, he was a research staff of Japan Science and Technology Agency. From 2007 to 2010, he was an Assistant Professor in Toyohashi University of Technology, Japan. He is currently an Assistant Professor in Cybermedia Center, Osaka University, Japan. His research interests include system software, distributed systems and computer security. He received the distinguished paper award from IPSJ (Information Processing Society of Japan) in 2005.