SafeCard: a Gigabit IPS on the network card

Willem de Bruijn[†], Asia Slowinska[†], Kees van Reeuwijk[†], Tomas Hruby[†], Li Xu[*], and Herbert Bos[†]

[†]Vrije Universiteit Amsterdam
[*]Universiteit van Amsterdam

Abstract. Current intrusion detection systems have a narrow scope. They target flow aggregates, reconstructed TCP streams, individual packets or application-level data fields, but no existing solution is capable of handling all of the above. Moreover, most systems that perform payload inspection on entire TCP streams are unable to handle gigabit link rates. We argue that network-based intrusion detection systems should consider *all* levels of abstraction in communication (packets, streams, layer-7 data units, and aggregates) if they are to handle gigabit link rates in the face of complex application-level attacks such as those that use evasion techniques or polymorphism. For this purpose, we developed a framework for network-based intrusion prevention at the network edge that is able to cope with all levels of abstraction and can be easily extended with new techniques. We validate our approach by making available a practical system, *SafeCard*, capable of reconstructing and scanning TCP streams at gigabit rates while preventing polymorphic buffer-overflow attacks, using (up to) layer-7 checks. Such performance makes it applicable in-line as an intrusion *prevention* system. *SafeCard* merges multiple solutions, some new and some known. We made specific contributions in the implementation of deep-packet inspection at high speeds and in detecting and filtering polymorphic buffer overflows.

1 Introduction

Network intruders are increasingly capable of circumventing traditional Intrusion Detection Systems (IDS). Evasion and insertion techniques blind the IDS by spoofing the datastream, while polymorphism cloaks malicious code to slip past the filter engine [1, 2]. Besides hiding the attack, however, attackers employ another weapon to thwart network defence systems: raw speed [3]. Less sophisticated attacks travelling over Gigabit links may be as difficult to stop as more complex attacks spreading more slowly. This leads to an interesting dilemma. On the one hand, systems that handle evasion and polymorphism are either too slow for in-line deployment (and are often host-based) or not sufficiently accurate (e.g. [4]). On the other hand, fast in-line solutions are not able to detect and stop sophisticated attacks (e.g., [5]). Our goal is to build a network card that can be deployed in the datastream as an Intrusion Prevention System (IPS) at the edge of the network and that handles many forms of attack at Gigabit rates.

Like [6], we advocate distributed firewalls. Briefly, centralised firewalls do not protect against attacks from inside an organisation, and are less able to analyse in detail complete TCP streams at link rate and to exploit knowledge about specific configurations of end-hosts. Host-based solutions are problematic also, because they depend on correct configuration of users' PCs, which has proved elusive in the past.

As a result, we prefer network administrators to have full control and security measures to be physically removed from users. A network device (such as a switch, or a router) *close* to the users' machines is the sweet spot for positioning the IPS system. The firewall could even reside in the network card of an end-host [7]. However, physically removing safety measures from the user's machine has the advantage that they cannot be tampered with, which from a security viewpoint may be preferred by administrators.

Unlike much existing work on distributed firewalls, the focus of our work is on *enforcing* security policies on all levels of the protocol stack, rather than specification of policies, distribution of rules, etc., for which we intend to build on existing solutions like [6]. *SafeCard* provides a single IPS solution that considers many levels of abstraction in communication: packets, streams, higher-level protocol units, and aggregates (e.g., flow statistics). We selected state-of-the-art methods for the most challenging abstractions (streams and application data units) and demonstrate for the first time the feasibility of a full IPS on a network card containing advanced detection methods for all levels of abstraction in digital communication. To support in-depth analysis in higher-level protocol layers and still achieve performance at Gigabit rates, we target specialised hardware as might be found in common router line cards. In particular, we aim for a truly low-level implementation on network processors. For the same reason as in [7] we evaluated the system on a slightly outdated processor to make it price competitive[1].

Besides combining many levels of abstraction in our IPS, we also make contributions to individual components. In particular, we developed a high-performance pattern matching language, *Ruler*, that offers functionality similar to that of Snort but is amenable to implementation on low-level hardware. In addition, we developed a protocol-specific detector, *Prospector*. Finally, we developed fast, zero-copy TCP reassembly that proves crucial for performance.

We offer a full network IPS implemented as a pipeline on a single network card. Each stage in the pipeline drops traffic that it perceives as malicious. Thus, the compound system works as a sieve, applying orthogonal detection vectors to maximise detection rate. In stage 1, we filter packets based on header fields (e.g., protocol, ports). Stage 2 is responsible for reconstructing and sanitising TCP streams. In stage 3, we match the streams against Snort-like patterns using Ruler. Unmatched traffic is inspected further in stage 4 by *Prospector*, an innovative protocol-specific detection method capable of stopping polymorphic buffer overflow attacks. This method is superior to pattern-matching for the detection of exploits in known protocols. Against other types of malicious traffic,

[1] In terms of manufacturing costs, not necessarily in current retail prices

such as trojans, it is ineffective, however. The two methods therefore complement each other: an indication of the strength of our sieve-based approach. Stage 5 further expands this idea by taking into account behavioural aspects of traffic. It generates alerts when it encounters anomalies in flow aggregates (e.g., unusual amounts of traffic) and subsequently drops the streams. Stage 6, at last, transmits the traffic if it is considered clean.

The conservative prevention strategy that we adopted may also drop benign traffic due to false positives. We take the position that occasional dropped connections outweigh the cost of even a single intrusion. That said, we have taken care to minimise false positives in the individual filtering steps.

The remainder of this paper is structured as follows: we begin with examining the shortcomings of existing IDSs in Section 2, after which we discuss our novel features individually in Section 3. The implementation of the complete system is described in Section 4 and subsequently put to the test in Section 5. We discuss limitations of our system in Section 6. Conclusions are drawn in Section 7.

2 Related work

In this paper we address the issue of deploying a practical IPS capable of scanning traffic at line rate. For some of our previous work on signature generation we refer to [8]. Current solutions for stopping intrusions often focus on two layers of defence, namely (network) intrusion *detection* and host-based intrusion *prevention* (exemplified by such approaches as Snort [9] and [10–13], respectively). We argue that both of them are lacking and propose a third approach: application-aware network intrusion prevention.

Most network IDSs (nIDS) search for malicious code in network packets, but, apart from simple firewalls, they are often not suitable as in-line IPS and prove vulnerable to insertion and evasion. Even though some systems, like Snort, have the required functionality for in-line deployment, this is hardly ever used on fast links since both TCP stream reassembly and pattern matching are prohibitively expensive. In previous work, CardGuard [7], we achieved 100s Mbit Ethernet performance when scanning payloads for simple strings after TCP reassembly on an IXP1200 network processor. Others, like EarlyBird [5] were specifically designed to allow in-line deployment on high-speed links as IDS solutions, but still do not lend themselves for prevention, because of the high ratio of false positives.

Work at Georgia Tech uses IXP1200s for TCP stream reconstruction in an IDS for an individual host [14], using both an IXP1200 and a completely separate FPGA board. Like [7], it limits itself to simple signature matching and achieves similar performance. Like *SafeCard* these systems do not exhibit the 'fail-open' flaw [1], because the IDS/IPS *is* the forwarding engine.

The inadequacy of pattern matching techniques as applied by Snort was also demonstrated by the recent WMF exploit, for which the pattern was so costly to inspect that IDS administrators were initially forced to let it pass or setup

a completely separate configuration[2]. While later attempts yielded fairly reliable (although not 100% accurate) signatures that could be handled by Snort, the issue is symptomatic of a flaw in Snort-like approaches for certain attacks. In essence, they are too costly when they must handle huge or complex signatures that can be applied to any traffic stream. As a result, nIDSs often limit themselves to per-packet processing, which renders them useless for detecting application level (layer 7) attacks. Note that we do not dismiss Snort-like pattern matching out of hand. It can be used for all sorts of malware (spyware, trojans) that do not use protocol exploits to enter the system. Also, many Snort-like rules exist and we can use these rules to filter out a plethora of known attacks. This saves us from having to develop and check protocol-specific signatures for each of these attacks. As a result, a snort-like pattern matcher is one of the pillars that underlie the *SafeCard* architecture.

Pattern matching engines are also weak in the face of polymorphism. In response, detection techniques were developed that look at aggregate information, e.g., triggering alerts when an unusual number of outgoing connections to unique IP addresses is detected [15] or looking at anomalies in webtraffic [16]. Doing so probably incurs too many false positives to be used for IPS by itself. On the other hand, it may detect suspect behaviour that would otherwise go unnoticed.

Host-based intrusion prevention blocks attacks based on local information. Many different measures fall in this category, including address space and instruction set randomisation (ASR and ISR [11,12]), non-executable memory [10, 17, 18], systrace [19], language approaches [20, 21], anti-virus software, host firewalls, and many others. Note that simple measures (like non-executable memory) are easy to circumvent [22, 23] and may break normal code (e.g., Linux depends on executable stacks for trampolines and signals). An advantage of host-based protection is that knowledge about the configuration can be exploited. We need to install specific filters only for the software running on the host which in turn makes signature generation easier. Also, all traffic that is classified as harmful to the local configuration can be safely dropped without worrying about hurting related applications (e.g., a request that hurts IIS, but not Apache can be dropped on the edge if we use IIS).

Most firewalls are restricted in their cycle budget and limit themselves to flow-based detection (e.g., port-filtering). This is a crude measure at best that fails to detect many types of malicious data, such as malformed requests sent to a vulnerable webserver, or all sorts of services deliberately implemented on top of port 80 to bypass firewall rules. Like nIDS, most anti-virus software is good at scanning for known patterns, but often less so at recognising polymorphic attacks.

Host-based filters may check protocol fields up to layer 7. Recent work has explored the use of protocol-specific approaches in detection of buffer overflow attacks [24]. In this approach the address that causes an alert is traced to a specific protocol field by the signature generator which then determines the maximum size M for the protocol field. We believe this is a promising approach and we

[2] source: http://isc.sans.org/diary.php?storyid=992

show how we improved the method to be more accurate. Other approaches look at executable code in traffic [25]. We did not opt for this method because it seems less reliable if fields are encoded (e.g., URL encoding).

Perhaps the greatest challenge in host-based protection is the need for user cooperation. If users are slow to update, unwilling to pay for anti-virus software, or if they disable firewalls, host-based protection breaks down. The past has shown that security policies that hinge on proactive users who secure and update their systems in a timely fashion are problematic.

In summary, the problems we face are twofold: existing solutions both do not handle many attacks and are already too slow to be able to scale to Gigabit rates. To deal with both issues and move from weak intrusion detection to stronger intrusion prevention we present *SafeCard*, a practical filter engine that (1) is fast enough to be placed in-line as an Intrusion Prevention System (IPS), (2) can handle polymorphism through smarter matching, (3) offers (up to) layer-7 detection of intrusions through stream reconstruction and application-level signatures and (4) coalesces the flow-based and payload-based approaches to increase each other's effectiveness. When connected to Argos [8], a signature generating honeypot, it can even stop (some) zero-day exploits.

Kerschbaum [26] uses in-kernel *sensors* to place an IDS in the datapath. An important difference with *SafeCard* is that sensors require a reconfiguration of kernel code and are therefore more OS-specific. Paxson's Bro [27] is another well-known IDS. Bro focuses on event handling and policy implementation. It relies on other libraries (e.g., `libpcap`) for its datapath and thus suffers from their performance problems.

3 Architecture

SafeCard must process at network, transport, and application protocol levels, as well as handle aggregates. For this reason we designed it as a compound, pipelined IPS built from independent functions elements (FEs). Each FE takes as input a stream of data and generates as output a stream of classification results. As side-effect it may also generate derived data streams. For example, an IP-header filter takes as input a stream of IP packets, and generates a binary output stream of per-packet pass or drop instructions. More complex is the TCP translation FE, which takes as input a stream of TCP segments and generates a set of continuous streams of application data, while using the classification result for signalling to which stream data belongs.

The FEs are interconnected in a directed acyclic graph (DAG), such that an FEs classification results plus one or more data streams serve as input to another. Each FE can have multiple such IO ports. The architecture that is used to place, connect, instantiate and run FEs is known as *Streamline*, a complete overhaul of its predecessor, the fairly fast packet filter (FFPF [28]). *Streamline* extends FFPF in many ways, for instance by adding stream reassembly, distributed processing, packet mangling and forwarding.

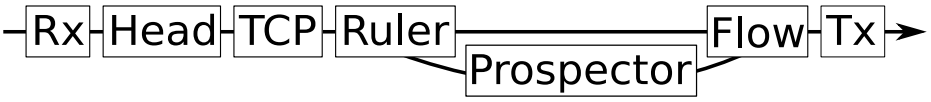

Fig. 1. Functional architecture of the intrusion prevention system

Before continuing with implementational details we discuss the FEs that form the stages in the pipeline. *SafeCard* combines 4 stages of defence: header-based filtering, payload inspection, flow-based statistical processing and application level protocol reconstruction. Supporting these methods are 3 additional stages: packet receive, packet transmit, and TCP stream reassembly. The full 7-stage pipeline is shown in Figure 1. FEs forward traffic from left to right, but each FE can drop what it perceives as malicious data. Only safe traffic reaches the last stage, where it is transmitted to its destination. The *Prospector* stage can only be applied to a a limited set of protocols (currently only HTTP) and is therefore bypassed by other traffic.

The first practical stage, header-based filtering, is implemented using FPL-3 [29]. Its functionality is run-of-the-mill and roughly equivalent to `pcap`. We will not discuss it further. The other FEs are explained in the order in which they are encountered by incoming packets.

3.1 Zero-Copy TCP Stream Reassembly

Recreating a continuous stream of data from packets is expensive because in the common case it incurs a copy of the full payload. TCP is especially difficult to reconstruct, as it allows data to overlap and has many variants. These features have been frequently misused to evade IDSs. We have developed a version of TCP reassembly that is both efficient and secure. We reassemble in-place, i.e. in *zero-copy* fashion, and take a conservative view of traffic by dropping overlapping data.

In terms of performance, we win by reducing memory-access costs. In the common case, when packets do not overlap and arrive in-order, our method removes the cost of copying payload completely. Instead, we incur a cost for bookkeeping of the start and length of each TCP segment. Due to the (growing) inequality between memory and CPU speed this cost is substantially smaller.

Our TCP reassembly design is based on the insight that consumers of TCP streams do not need access to the streams continuously. They only need to receive blocks in consecutive order. Applications generally use the Socket `read(..)` call for this. We have slightly modified this call to return a pointer to a block, whereas it normally receives one from the caller. We exploit this change to implement zero-copy transfer as follows. First, we never supply more data than fits in a single TCP segment. `read(..)` is allowed to return a smaller block than was requested. Second, instead of allocating a transfer buffer and copying data into it we return a pointer directly into the original segment. The transport architecture used to support this is not standard. Packets are stored in one large circular *packet*

buffer. TCP streams have private circular *pointer* buffers, which store references to the start and end of TCP segments. References are valid only as long as the pointed-to elements in the shared packet buffer exist.

Our method is not just fast, but also secure, because it drops potentially harmful TCP streams. IPSs are inherently more capable in dealing with malformed TCP options than IDSs: because they work in-line they can operate as a proxy, reassembling a stream of data as they see fit, checking it, and then re-encoding the cleansed data in a new TCP stream. Full re-encoding scrubs [2,30] payload from abused transport protocol features and thus protects the hosts, but is very expensive, and incurs multiple checksum computations. The cheap alternative that we use, dropping malicious streams, will equally deal with malformed payloads, but at much lower cost. In essence, we perform a first protocol scrub of the traffic [2]. Later on we will see that higher-layer protocols are scrubbed as well.

In relation to security, TCP segment overlap is worth mentioning individually because it has frequently been abused. Meant to circumvent IDS detection, overlap is powerless against in-line scrubbing. Overlapping traffic may be indicative of broader malicious intent, especially when the overlapping segments differ in content. For this reason its appearance should be notified to the flow-based filtering unit, as well. Flow-based detection is discussed further in Section 3.4.

Another security issue concerns out-of-order packet arrival. Received data must be buffered in a reconstruction window until missing data arrives, dramatically increasing memory footprint on links with large bandwidth-delay products. Arrival of many out-of-order segments can lead to memory exhaustion, a situation that is potentially exploitable.

One solution is to check payloads per-packet and then pass them on immediately. Feasibility of immediate processing depends on whether filtering algorithms can checkpoint and move around in the datastream. Special care must be taken not to let an exploit slip through because the signature is larger than the minimal malicious payload and happens to span two packets.

Even when immediate processing is not possible *SafeCard* is not subvertible through memory exhaustion. Since all packets are kept in a single circular buffer there is no memory allocation in the datapath at all. This advantage is offset by the increased chance of packet drop due to a full buffer. Overwriting a single packet may invalidate an entire (benign) TCP stream if used with in-place reassembly. Therefore we have to keep buffers large enough to deal with incidental delays.

3.2 Payload Inspection

Static string matching (as for example provided by hardware CAMs and our own CardGuard [7]) is too limited for detecting most intrusion attempts. Pattern matching in *SafeCard* is therefore implemented using superior regular expression matching. Our engine, *ruler*, is innovative in that it matches packets against the whole set of regular expressions in parallel, which allows it to sustain high

traffic rates. Matched substrings can be accepted or rejected entirely, or rewritten to an altered output packet. Rewriting is of use in address translation or anonymisation, but here we are interested only in Ruler's high-speed selection mechanism.

Regular expression matching has been used in IDSs before, but only in special cases. Traditionally, the cost of matching scales linearly with the number of signatures. Because there are thousands of known signatures, scanning at high-speed is infeasible. Instead, string-matching algorithms whose runtime complexity remains constant regardless of patternset-size have to be used in the common case. Ruler *can* completely replace string-matching because it is a generalisation of and therefore provably as efficient as Aho-Corasick (AC), a popular constant-time pattern-matching algorithm that is employed for instance in Snort.

Ruler's internal design is based on a Deterministic Finite Automaton (DFA). This allows it to merge many patterns—or more precisely their DFA state machines—into a single state machine. Each state in the Ruler DFA encodes a character in a pattern. Patterns that share prefixes will reuse subpaths in the DFA and thus do not impose additional burden apart from their unique tails. One caveat is that states themselves become more costly to compute when the number of outgoing connections grows, because internal control-flow is that of a switch statement. Ruler reverts to an AC automaton when run with only static strings, but it can be extended, for instance with (unbounded) repetitions.

Compiling regular expressions like those in Ruler is a well-studied problem[3] [32]. The standard approach is to first generate a Non-deterministic Finite Automaton (NFA) from the regular expressions. This NFA contains state transitions for all matching possibilities of all regular expressions in the filter. The NFA is then converted to its DFA form using the *subset* algorithm. This algorithm traces execution paths through the NFA, and lists the sets of NFA states that can be reached for each known NFA state set and each possible input character. Each distinct NFA state set is a distinct state in the DFA.

To the DFA we apply general optimisations: (i) we merge overlapping parts of the patterns as much as possible, (ii) we eliminate unreachable patterns and machine states, (iii) we stop the state machine as soon as a verdict can be reached, and (iv) we use a standard state minimisation algorithm [33] to construct a DFA with the smallest number of states.

The Ruler DFA is further optimised for traffic processing. Network packets often contain fixed-length stretches of bytes that need not be inspected at all, such as header fields. Instead of having the state machine go through the motions for these bytes, we support 'jump' states that skip past them. Also, we have provisions for content-dependent field lengths, such as IP headers, whose length is defined in the header itself. Continuous streams place further demands on the matching engine. An engine must be able to handle multiple streams concurrently, each of a-priori unknown length. Ruler is capable of checkpointing

[3] In fact, things are not this simple. Ruler also supports packet rewriting, which requires *tagging* of positions in the regex for which we need a generalisation of the DFA construction algorithm [31]. This is beyond the scope of this paper.

its state so that it can switch between streams at will. When data arrives for a stream it will resume exactly where it left off.

A second performance benefit stems from the method of Ruler's execution. Instead of running in an interpreter, Ruler code is compiled straight to assembly. Back-ends exist for network stream processors, kernel modules and application code. Even Verilog (for FPGAs) can be produced, although this is currently in its infancy. When used within *Streamline*, Ruler automatons can be compiled, shipped and instantiated at runtime on the supported hardware with minimal intervention.

With the help of our `snort2ruler` compiler most Snort signatures can be automatically incorporated in the Ruler DFA, but Ruler also has its own high-level input language. This supports protocol-specific constructs such as TCP options and variable-sized fields to aid signature generation. Let us illustrate the language with an example: scanning for the Slammer worm. Slammer is a 376 byte payload encapsulated in a UDP packet destined for port 1434. To find Slammer-based intrusion attempts in a packet stream, we would use the following filter:

```
include "layouts.rli"
filter slammer [accept_reject]
    IPv4_Ethernet_header
    IPv4_header with [protocol=17]
    UDPv4_header with [dest=1434,length=376]
    4 1 1 1 1 1 * "." "D"|"d" "L"|"l" "L"|"l" * => accept;
```

We require that packets start with Ethernet, IPv4 and UDP headers. The layout of these headers is defined in the include file `layouts.rli`, which is not shown here. We then scan the payload of such packets for the signature `"04 01 01 01 01 01.*[.][Dd][Ll][Ll]"`. In *SafeCard* the packet is dropped when a match is made.

3.3 Protocol-specific detection of polymorphic attacks

Scanning streams for known signatures using regular expressions catches a large class of known and immutable attacks. However, future worms are expected to be increasingly polymorphic. While exploits are less likely to exhibit advanced polymorphism than payloads, simple variations will be used. Snort-like pattern matching is not suitable for stopping such attacks. Rather, we use protocol-specific detection methods requiring up to layer 7 messages.

Like the Covers approach [24], we protect hosts from buffer overflow attacks by tracing the address that causes the control flow diversion to a specific (higher-level) protocol field and capturing characteristics (such as the length of the field) that are subsequently used as an attack signature. Briefly, Covers uses ASR to detect an attack, and any exploit that attempts to divert the control flow will, with high probability, crash the process with a memory fault. If so, it queries the OS to find the address M_t that caused the crash (see also Figure 2). Next, it will look for the address A and some bytes in its vicinity in the (logged) traffic trace,

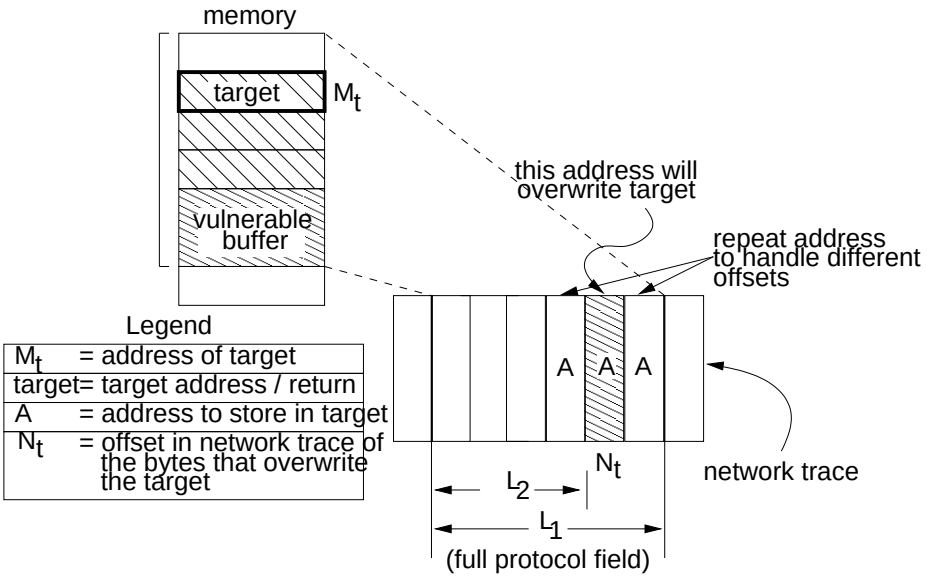

Fig. 2. Memory and network traces of a simple buffer overflow attack

thus approximating location N_t. Using knowledge about the protocol governing the interaction, Covers subsequently determines the protocol field that caused an overflow. Next, it uses the length of this protocol field as a signature, as all messages of the same protocol with this length will lead to the same overflow, regardless of the contents. By focusing on properties like field length, the signatures are independent of the actual content of the exploit and hence resilient to polymorphism.

In *SafeCard*, we developed the *Prospector*, a protocol-specific detector that builds on the same principles, but differs from Covers in important aspects. First, we moved the filter out of the host and into an Intel IXP2400 network processor. By moving the filter away from the host to the first router or switch connected to the end-user's PC, administrators keep tight control over the security software. At the same time, not moving it all the way to a centralised firewall permits the network device to exploit application specific knowledge. For instance, we keep track of which applications (and which versions) are running on the servers connected to each port. Whether the applications are discovered automatically (e.g., by port scanning) or administered explicitly is beyond the scope of this paper.

Second, rather than the crude and somewhat error-prone address space randomisation, we use a more reliable method based on taint analysis for detecting intrusions [34]. The *Argos* IDS used for *SafeCard* is an efficient and reliable emulator that tags and tracks network data and triggers alerts whenever the use of such data violates security policies (e.g., when it is used as a jump target). Argos is not part of our high-speed datapath. It is a signature generating honeypot

that listens to background traffic on a separate machine. Whenever it observes an intrusion attempt it generates a signature. *Prospector* then uses these signatures for filtering on high-speed links. We will not repeat the full explanation of Argos here (interested readers are referred to [8]), but we do note that Argos is more reliable in finding the address that causes the control diversion than ASR. After all, with ASR there is a non-negligible chance that the attack does not cause a memory fault immediately, but crashes after executing a few random instructions. In that case, the address would be bogus. Moreover, by keeping track of the origin in the traffic trace of tainted data, as provided by the next release of Argos, the correlation with network data will be very accurate. Even if the probability of not producing an address with ASR is small, in our experience the odds of making the wrong guess as to the origins N_t of the address A that exactly overflows M_t in the network trace is much greater [8]. Worse, if protocol fields are encoded in the network trace (e.g., URL encoding), scanning traces for occurrences of the target will fail altogether. In contrast, tracking the origins of tainted data handles these cases well.

Third, sophisticated overflows are caused by more than one field. An example is chunking and multiple host headers in HTTP, where multiple chunks or headers end up in the same buffer. While Covers is unable to figure out that it should watch the total length of all chunks/headers together, rather than a single field, *Prospector* handles such cases correctly. The importance of this improvement is demonstrated for instance by attacks like the Apache-Knacker exploit [35] which consists of a GET request with multiple host headers that end up in the same buffer. Such attacks frequently lead to false positives in Covers, but are correctly identified by *Prospector*.

Fourth, we do not necessarily consider the whole field. The work described in [24] always uses up to L_1, the length of the entire protocol field containing the jump target, even though the jump target is often not found at the end of the protocol field. It seems the authors use statistics of legitimate messages received in the past to help estimate the maximum length that the field may have. Doing so may cause false negatives, e.g., if the jump target is followed by a variable number of bytes in the same protocol field. A signature generated for a long version of the protocol field is unable to find attacks with shorter protocol fields, even if they contain the same exploit. Such behaviour is quite common, especially if part of the payload is stored in the same vulnerable buffer. Instead, our *Prospector* uses L_2, the exact distance between the start of the protocol field and N_t. We speculate that the reason for taking the whole field is that Covers is unable to accurately pinpoint N_t, as jump targets are often repeated in the exploit in order to handle minor differences in offset (as indicated by multiple occurrences of A in Figure 2).

Fifth, the way multiple signatures are used in [24] is not specified. We have an efficient tree-like structure for dealing with large numbers of signatures. Briefly, every signature consists of a sequence of *value fields* and *critical fields*. A value field specifies that a field in the protocol should have this specific value. For instance, in the HTTP protocol a value field may specify that the method should

be `GET` for this signature to match. Critical fields, on the other hand, should collectively satisfy some condition. For instance, in the current implementation the critical fields should collectively have a length that is less than L_2. The signatures are organised in memory like a tree, so that common prefixes are checked only once. Because our signature recognition is stateful, the *Prospector* is able to check whether a TCP segment matches a signature efficiently (i.e., without having to traverse the whole tree each time a segment comes in).

Sixth, *Prospector* has an option to scan for and reject malformed protocol messages. Since we have protocol-specific knowledge, it was easy to extend *Prospector* to also check whether the application-level interaction conforms to the protocol. In other words, we scrub higher-layer protocols in this FE.

The *Prospector* module in *SafeCard* allows us to scan for a large class of polymorphic buffer overflows at application-level. Both stack and heap overflows are already handled in the current version. However, given an accurate location of N_t, one may detect format string attacks in a similar way. We are currently extending the *Prospector* with such a format string handler. The details are beyond the scope of this paper as the mechanism is not yet thoroughly evaluated. *Prospector* is at the moment further limited by its support for only a single protocol: HTTP. We will add support for more protocols as well.

3.4 Flow-based behavioural detection

Flow-based detection complements payload-scanning and (header-based) protocol reconstruction as the three detection vectors are orthogonal. We have already seen one method of flow-based detection: arrival of overlapping segments. Another group of methods detects unexpected variations in incoming or outgoing connections (e.g., number per time-unit, address-space entropy, or length), for example HP's VirusThrottle technology [15], or [16].

As a demonstrator, *SafeCard* incorporates a filter that is similar to VirusThrottle, but works on incoming traffic. The filter limits per-service traffic spikes to protect servers against flash mobs. The algorithm is admittedly naive, and serves mostly as a placeholder. Irrespective of the algorithm(s) used, statistical processing is based on *Streamline*'s support for datastream correlation: multiple classification streams enter a single aggregation function that forward data only once, when a threshold is reached. The runtime-constructed data path combined with the cost-effectiveness of flow-based detection, encourages further experimentation with these methods.

4 Implementation

We have implemented the discussed architecture on a programmable NIC, the Radisys ENP2611 board built around the Intel IXP2400 network processor (NPU). The IXP2400 is controlled by a 600 MHz general purpose processor, the XScale. This processor is not nearly fast enough for line-rate data inspection. The NPU therefore also embeds 8 specialised stream processors on the same

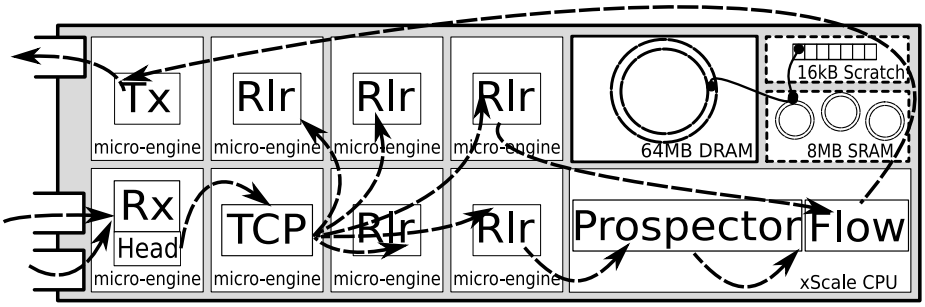

Fig. 3. Implementation of *SafeCard* on an Intel IXP2400 NPU

die that run without any OS whatsoever. These so called *micro-engines* each support up to 8 hardware threads, with zero-cycle (i.e., *free*) context switching. We have moved most processing to these resources. One micro-engine is used by the receiver and header-based filter, one by the transmitter, and one by the TCP reassembler. The remaining 5 are available to Ruler. The other FEs have not been ported yet and must run on the on-board control processor.

Figure 3 shows the functional architecture from Figure 1 again, but now overlaid over the hardware resources. Data enters the NPU on two one-Gigabit ports and leaves on the third. All processing is on the network card, there is no intervention from the connected host beyond loading and starting the IPS. It is therefore easy to see that this device can run independent of a host-processor as well. Intel's IXDP2850 board is such a stand-alone network processor, to which *SafeCard* has also been ported. Alternatively, the current setup could easily be changed to forward data over the peripheral bus to the local host. We have discussed our experiences with designing high-performance programmable NICs for this mode of operation elsewhere [36].

Dataplane operations must be cheap if we are to scale to high datarates. In theory, a non-superscalar 600 MHz processor with single-cycle instruction costs would be able to scan traffic at close to 5 Gigabit per second. However, instructions are factors more costly, and more importantly memory-access costs are generally two orders of magnitude slower. We rely on a few heuristics to be able to scale to multi-Gigabit rates regardless of these obstacles.

First, we use zero-copy transport where efficient. In *SafeCard* the only copy incurred is from the Gigabit ports to the memory and back *once* thanks to the zero-copy TCP reassembly FE.

Second, we minimise synchronisation, including locking. Most synchronisation is through per-stream circular buffers. Polling on these buffers is essentially free in dedicated processors like the micro-engines. For processing on the XScale we have a dual polling/interrupt based mechanism. Micro-engines raise an interrupt for each newly processed piece of data, but the CPU masks these interrupts while it processes its backlog, in NAPI style. The dual scheme ensures both

timely operation under low load as well as graceful degradation under strain (as opposed to thrashing due to livelock).

An embedded device like the IXP network processor adds its own complexity to the general architecture. The board contains 5 different layers of memory, with each layer trading off increased capacity at the expense of throughput. Communication can take place through interrupts, shared registers or shared memory. The top entry IXP2850 even comes with 2 hardware cryptography units and content-addressable memory. The need to take into account such hardware details is inherent to embedded design, where implementational choices (e.g., to use the cryptographic units) greatly influence overall performance.

As memory access is the bottleneck in high-volume traffic processing, the memory hierarchy features should be optimally exploited. In *SafeCard* we optimise placement of structures based on access frequency and structure size. The packet buffer is placed in the largest (64MB), but slowest memory, DRAM. Because their smaller size permits this, pointer buffers are placed in faster 8MB SRAM.

TCP stream metadata sits in the even faster, but far more scarce 16 KByte *scratch* memory. Communication occurs through two datastructures: a hardware-accelerated FIFO queue that holds per-segment work orders and a hashtable that keeps per-stream metadata. For each segment the TCP reassembly unit places a work order in the queue, where it is fetched by Ruler. Ruler then looks up the correct stream in the hashtable and restore its DFA to the checkpointed state. The two fastest types of memory, 2.5KB per-micro-engine RAM and their 4KB instruction stores, are reserved for function-specific uses.

Ruler makes use of two methods to reduce memory-accessing costs. First, non-preemptive multi-threading enables threads to hand off control while waiting for I/O. Second, asynchronous I/O allows individual threads to interleave processing and I/O operations. As the computation versus I/O ratio changes, so does the number of concurrent threads needed to hide memory latency. For computation-bound applications such as Ruler, threading is not necessary at all.

Resource allocation also encompasses layering the pipeline across the distributed processors. As said, the IXP micro-engines each support up to 8 hardware threads. Having more threads (e.g., one per TCP flow) introduces software scheduling overhead. The opposite, a centralised event-handling mechanism, adds parallelisation overhead and then reverts to a master-work threading model. The optimal solution is therefore to create a thread-pool of functions of the same size as the hardware resources [4]. A threadpool of interchangeable worker threads can only be applied when workers can attach to and detach from a stream at will, i.e. checkpoint their state, as Ruler can. The size of the pool can be scaled by incorporating more or fewer micro-engines. For example the IXP28xx has 7 more micro-engines that the Ruler pool can use without changes.

High-volume traffic must be processed on the micro-engines. However, because these processors are scarce and hard to program, some processing will usually take place on the slower XScale. We have implemented *Prospector* and flow-

[4] this mechanism is also known as I/O Completion Ports

based detection on the XScale embedded in *Streamline*. If data is not matched on the XScale it is sent back to the fast path on the micro-engines by writing an entry in the transmission unit's pointer buffer.

The volume of traffic that is handled by the XScale must be considerably smaller than the Gigabit traffic handled in the fast path. As *Prospector* currently checks HTTP request headers only, and flow-based methods do not touch payload either, volume is indeed small.

5 Evaluation

As *SafeCard* is a compound system each function can prove to be the bottleneck. Some operations are obviously more expensive than others, such as pattern-matching, but this heuristic is of limited value when functions are implemented on different hardware resources. Indeed, as we discussed before, the Ruler engine can be scaled across multiple fast micro-engines, while flow-detection must compete with *Prospector* for cycles on the XScale.

For this reason, to evaluate *SafeCard* and all its constituent parts, we conduct experiments that both measure the performance of individual FEs (micro-benchmarks) as well as the overall throughput (a macro-benchmark).

5.1 Micro-benchmarks

We can get an indication of the per-stage processing overhead by running the micro-engines in single-thread mode and measuring the cycle count in isolation. Table 1 shows the cost in cycles per protocol data unit (PDU, e.g., IP packet, TCP segment) with minimal payload and the additional cost per byte of payload for each hardware accelerated FE. Figure 4 shows on the left the maximal sustained rate of the FEs as obtained from these numbers. At 600MHz, we can see that all FEs can process common-case traffic at a Gigabit except Ruler. A single Ruler instance can process only 170 Mbit. The 5 combined engines thus top at 850Mbit, which we've plotted in the figure as *5x Ruler*. Merging Reception and Transmission would give us the additional engine we need for full Gigabit processing.

Description	**PDU**	**Byte**
Reception	313	1.5
TCP reassembly	1178	0
Ruler	628	26
Transmission	740	2

Table 1. Single threaded cycle counts of individual FEs

TCP reassembly A single threaded cycle count presents a lower-bound on the per-segment overhead as it omits memory contention costs. Nevertheless, for TCP its performance represents the worst-case scenario for overall throughput, because a single thread spends much of its time waiting for memory. Since multi-threading enables latency hiding throughput will improve dramatically.

Independent of maximal obtainable throughput is the question how indirect stream reassembly measures up to regular copy-based reassembly. For this reason we have compared them head-to-head. As we have no copy-based method available on the micro-engines we ran this comparison in a host based Streamline

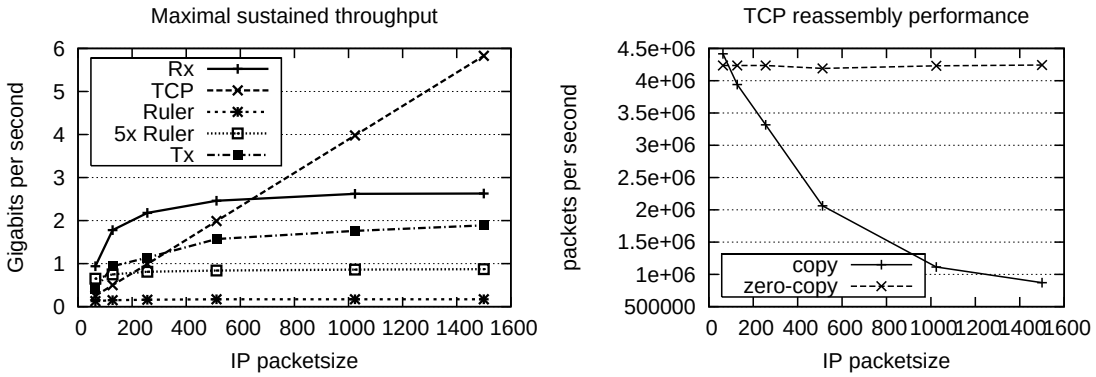

Fig. 4. Theoretical sustained throughput & TCP Reassembly performance

function. The two functions share the majority of code, only differing in their actual data bookkeeping methods. Figure 4(right) shows that indirect reassembly easily outperforms copy-based reassembly. Only for the smallest packets can the computational overhead be seen.

Ruler The third row in Table 1 shows the overhead in cycles of *Ruler*. As expected, costs scale linearly with the amount of data; the cost per PDU is negligible. The function is computation-bound: fetching 64 bytes from memory costs some 120 cycles, but processing these costs an order of magnitude more. For this reason multi-threading is turned off.

Prospector We have to benchmark *Prospector* on the XScale, because it is not yet ported to the micro-engines. Figure 5(left) compares throughput of *Prospector* to that of a payload-scanning function (we used Aho-Corasick). We show two versions of *Prospector*: the basic algorithm that needs to touch all header data, and an optimised version that skips past unimportant data (called `Pro+`). The latter relies on HTTP requests being TCP segment-aligned. This is not in any specification, but we expect it is always the case in practise.

Each method processes 4 requests. These are from left to right in the figure: a benign HTTP GET request that is easily classified, a malicious GET request that must be scanned completely, and two POST requests of differing lengths. In the malicious GET case all bytes have to be touched. Since AC is faster here than both versions of *Prospector* we can see that under equal memory-strain we suffer additional computational overhead.

However, all three other examples show that if you do not have to touch all bytes —the common case— protocol-deconstruction is more efficient than scanning. Looking at the right-most figure, the longest POST request, we can see that the gap quickly grows as the payload grows. The benign GET learns us additionally that skipping remaining headers when a classification has been made can result in a dramatic (here 2-fold) increase in worst-case performance.

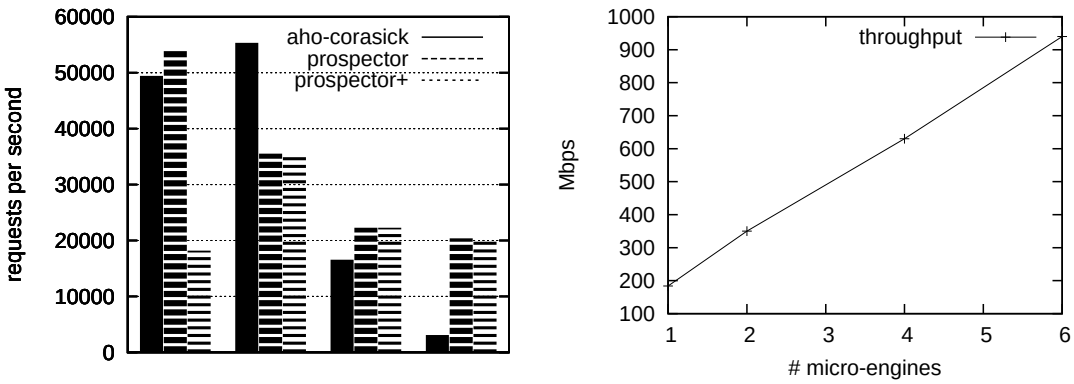

Fig. 5. *Prospector* throughput & Macro benchmark

Note that none of these example requests carry a message body. This would also be skipped by *Prospector*, of course. Even without message bodies, performance is continuously above 18,000 requests per second, making the function viable for in-line protection of many common services.

5.2 Macro Benchmark

Our final experiment evaluates the pipeline in hardware. We do not include results for the FEs on the XScale again, because their throughput is not measurable in bitrate and we have already computed an upper bound. For this test we connected our board to a mirror image of communication between three computers. By using mirroring we were able to test peak throughput without interfering with the active TCP control flow. The traffic was generated using `ab`, a benchmarking tool for Apache. When ran against two servers at the same time our maximally obtainable rate was 940Mbits. The results are shown in Figure 5(right).

From the Figure we can see that with 6 micro-engines we can process all traffic. To free up the 6th micro-engine we had to remove the transmission unit temporarily. The presented numbers are worst-case estimations as a result of crude dropped traffic statistics. Actual performance could be up to 20% higher.

6 Discussion

Limitations The presented solution is an amalgam of solutions. While fairly powerful as a whole, we are aware of improvements that could be made to its parts. For starters, while Ruler accepts most Snort rules through our `snort2ruler` compiler, there is a subset of expressions that we cannot handle yet. In *Prospector*, we do not currently block format string attacks, although this is possible

in principle. We are currently implementing this feature and expect to have it available soon. Also, the flow-based IDS (stage 5) is currently rather naive and should be improved.

Finally, while we have tried to implement a powerful set of network-based intrusion prevention methods, we have clearly not exhausted the options. For instance, as we operate close to the end-hosts with application-awareness, we are still considering filters such as those generated by Vigilante [13]. We opted for protocol-aware filtering because Vigilante does not handle polymorphism well.

Hardware Acceleration An obvious way of increasing network throughput is to switch to expensive specialised hardware. Although implemented on an embedded device, *SafeCard* is expressly not meant to explore that option. The IXP2400 is 5 years old and no longer supported by Intel. It was expensive, but mostly because of its low volume sales. The trend toward multi-core CPUs at the network edge could bring a cheap equivalent, if memory latency-hiding is also provided for.

To illustrate our point more clearly, we compare performance to that of the IXDP2850, a dual processor variant with 32 micro-engines in total, that runs at 1.4GHz. Cycle-for-cycle this device can process more than 9 times as much traffic. As the bottleneck in our pipeline is computationally bound and inherently scalable, this will directly translate into better *SafeCard* performance. We decided not to show those results, however, because installing IXDP2850s at the network edge is not viable in the near future.

7 Conclusion

In this paper, we have described *SafeCard*, a full intrusion prevention system (IPS) on an embedded network processor. *SafeCard* is unique in that it includes detection techniques at all levels of abstraction in communication: packets, reassembled TCP streams, application protocol units, and flow aggregates. Moreover, *SafeCard* is capable of handling close to a Gigabit per second of TCP traffic, making it a viable option for the edge of the network. The IPS is implemented as a pipeline on a single Intel IXP2400 network processor embedded on a network card. Its task is to enforce security policies on incoming traffic by means of in-depth analysis in the last hop toward the host. The system first receives traffic in a circular buffer and applies simple header-field filtering to determine which data needs further inspections. TCP streams that are classified as suspect are reassembled with an efficient in-place algorithm and fed into a per-stream pattern matching engine, similar to Snort. For all streams that are not blocked by the pattern matching engine *SafeCard* checks whether higher-level protocol-specific rules exist and if so, checks them against these also. A final detection technique works on flow aggregates (e.g., statistics and number of incoming connections). Our future work looks at combining alerts generated by multiple stages when each individual stage is subject to false positives.

Acknowledgements

We would like to thank Lennert Buytenhek for his invaluable help during development of the IXP2400 code and installation of the testbed. This research was made possible by grants from the EU Lobster and Noah projects.

References

1. Ptacek, T.H., Newsham, T.N.: Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks Inc. (1998)
2. Handley, M., Paxson, V., Kreibich, C.: Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In: USENIX-Sec'2001, Washington, D.C., USA (2001)
3. Stuart Staniford, V.P., Weaver, N.: How to 0wn the internet in your spare time. In: Proc. of the 11th USENIX Security Symposium. (2002)
4. James Newsome, B.K., Song, D.: Polygraph: Automatically generating signatures for polymorphic worms. In: Proc. of the IEEE Symposium on Security and Privacy. (2005)
5. S. Singh, C. Estan, G. Varghese and S. Savage: Automated worm fingerprinting. In: In Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI). (2004) 45–60
6. Ioannidis, S., Keromytis, A.D., Bellovin, S.M., Smith, J.M.: Implementing a distributed firewall. In: CCS '00: Proceedings of the 7th ACM conference on Computer and communications security, ACM Press (2000) 190–199
7. Bos, H., Huang, K.: Towards software-based signature detection for intrusion prevention on the network card. In: Proc of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID). (2005)
8. Portokalidis, G., Slowinska, A., Bos, H.: Argos: an emulator for fingerprinting zero-day attacks. In: Proc. ACM SIGOPS EUROSYS'2006, Leuven, Belgium (2006)
9. Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proc. of LISA '99: 13th Systems Administration Conference. (1999)
10. C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle and Q. Zhang: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proc. of the 7th USENIX Security Symposium. (1998)
11. S. Bhatkar, D.C. Du Varney and R. Sekar: Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: In Proc. of the 12th USENIX Security Symposium. (2003) 105–120
12. E. G. Barrantes, D.H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovix and D.D. Zovi: Randomized instruction set emulation to disrupt code injection attacks. In: In Proc. of the 10th ACM Conference on Computer and Communications Security (CCS). (2003) 281–289
13. M. Costa, J. Crowcroft, M. Castro, A Rowstron, L. Zhou, L. Zhang and P. Barham: Vigilante: End-to-end containment of internet worms. In: In Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP), Brighton, UK (2005)
14. Clark, C., Lee, W., Schimmel, D., Contis, D., Koné, M., Thomas, A.: A hardware platform for network intrusion detection and prevention. In: Third Workshop on Network Processors and Applications, Madrid, Spain (2004)
15. Williamson, M.M.: Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code. In: Proc. of ACSAC Security Conference, Las Vegas, Nevada (2002)

16. Robertson, W., Vigna, G., Kruegel, C., Kemmerer, R.: Using generalization and characterization techniques in the anomaly-based detection of web attacks. In: NDSS'05. (2005)
17. C. Cowan, S. Beattie, J. Johansen and P. Wagle: PointGuard: Protecting pointers from buffer overflow vulnerabilities. In: In Proc. of the 12th USENIX Security Symposium. (2003) 91–104
18. C. Cowan, M. Barringer, S. Beattie and G. Kroah-Hartman: FormatGuard: Automatic protection from printf format string vulnerabilities. In: In Proc. of the 10th Usenix Security Symposium. (2001)
19. Provos, N.: Improving host security with system call policies. In: In Proc. of the 12th USENIX Security Symposium. (2003)
20. U. Shankar, K. Talwar, J. S. Foster, and D. Wagner: Detecting format string vulnerabilities with type qualifiers. In: In Proc. of the 10th USENIX Security Symposium. (2001) 201–216
21. G. C. Necula, S. McPeak, and W. Weimer: CCured: Type-safe retrofitting of legacy code. In: In Proc. of the Principles of Programming Languages (PoPL). (2002)
22. bulba and Kil3r: Bypassing Stackguard and Stackshield. Phrack Magazine **10**(56) (2000)
23. gera, riq: Advances in format string exploitation. Phrack Magazine **11**(59) (2002)
24. Liang, Z., Sekar, R.: Fast and automated generation of attack signatures: A basis for building self-protecting servers. In: Proc. ACM CCS, Alexandria, VA, USA (2005) 213–223
25. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: Proc. of RAID'05, Seattle, USA (2005)
26. Kerschbaum, F., Spafford, E.H., Zamboni, D.: Using embedded sensors for detecting network attack. Technical report, Purdue University (2000)
27. Paxson, V.: Bro: A system for detecting network intruders in real-time. Computer Networks **31(23-24)** (1999) 2435–2463
28. Bos, H., de Bruijn, W., Cristea, M., Nguyen, T., Portokalidis, G.: FFPF: Fairly Fast Packet Filters. In: Proceedings of OSDI'04, San Francisco, CA (2004)
29. Cristea, M., de Bruijn, W., Bos, H.: Fpl-3: towards language support for distributed packet processing. In: Proceedings of IFIP Networking, published as LNCS Volume 3462 / 2005, ISBN: 3-540-25809-4, Waterloo, Ontario, Canada (2005) p.743–755
30. Malan, R., Watson, D., Jahanian, F., Howell, P.: Transport and application protocol scrubbing. In: Infocom'2000, Tel-Aviv, Israel (2000)
31. Laurikari, V.: NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In: SPIRE. (2000) 181–187
32. Aho, A.V., Ullman, J.D.: Foundations of Computer Science. Computer Science Press (1992)
33. Gill, A.: Introduction to the Theory of Finite-state Machines. McGraw-Hill (1962)
34. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proc. of the 12th Annual Network and Distributed System Security Symposium (NDSS). (2005)
35. SecurityFocus: Can-2003-0245 apache apr-psprintf memory corruption vulnerability. http://www.securityfocus. com/bid/7723/discussion/ (2003,)
36. Nguyen, T., Cristea, M., de Bruijn, W., Bos, H.: Scalable network monitors for high-speed links: a bottom-up approach. In: Proceedings of IPOM'04. (2004)