pSnort: Parallel Intrusion Detection System Based on Multi-Core Processors (Postprint)

Authors: He Peng, Jiang Haiyang, Xie Gaogang

Date: 2017-03-09T00:00:00+00:00

Abstract

Network intrusion detection and defense systems constitute a critical component in the contemporary IP network security landscape. The exponential growth of Internet traffic coupled with the inherent processing bottlenecks of single-core architectures has rendered conventional single-threaded network intrusion detection systems inadequate for meeting the evolving requirements of network development. To address this limitation, this paper presents pSnort, a parallel intrusion detection system based on software pipelining, which leverages the mainstream single-threaded Snort software as its foundation. The proposed architecture partitions Snort into two distinct phases, parallelizing the most computationally intensive processing stage to achieve enhanced performance. Additionally, through deliberate program design, pSnort effectively circumvents severe synchronization and mutual exclusion issues typically introduced by parallelization. Experimental results demonstrate that pSnort attains packet processing throughput exceeding 1 Gbps on a general-purpose Intel Quad-core Xeon platform. Relative to traditional Snort, pSnort achieves a maximum performance improvement of 147% and a speedup ratio of 2.5×.

Full Text

Preamble

pSnort: A Parallel Intrusion Detection System Based on Multi-Core Processors

Peng He, Haiyang Jiang, Gaogang Xie

Abstract: Network intrusion detection and prevention systems play a crucial role in contemporary IP network security. The exponential growth of Internet traffic and the processing bottlenecks inherent in single-core processors have rendered traditional single-threaded network intrusion detection and prevention

systems running on single-core platforms far inadequate for modern network demands. To address this challenge, this paper presents pSnort, a parallel intrusion detection system based on software pipelining, built upon the mainstream single-threaded Snort system. By partitioning the traditional Snort architecture into two phases and parallelizing the most time-consuming processing stage, pSnort achieves significant performance improvements. Furthermore, through careful program design, pSnort avoids severe synchronization and mutual exclusion issues that typically arise from parallelization. Experimental results demonstrate that pSnort achieves packet processing speeds exceeding 1 Gbps on a general-purpose Intel Quad-core Xeon platform, delivering up to 147% performance improvement and a 2.5x speedup over traditional Snort.

Keywords: multi-core, Snort, software pipelining, parallel architecture

1 Introduction

With the widespread development and deployment of the Internet, network attacks such as buffer overflow attacks, denial-of-service attacks, and backdoor exploits have become increasingly prevalent. Attackers leverage various backdoor software to gain administrative privileges on target systems and launch denial-of-service attacks against other hosts. Although mainstream operating systems and application software regularly release security patches to address vulnerabilities, the pace of updates lags far behind the speed of new vulnerability discoveries, making Network Intrusion Detection Systems (NIDS) increasingly vital in the cybersecurity landscape.

Network intrusion detection systems are security appliances deployed at network edges to detect anomalous traffic. Snort[1], an open-source software, stands as one of the mainstream network intrusion detection systems, operating on general-purpose CPU platforms and employing signature-based anomaly detection through regularly updated anomaly signature databases to maintain network security. Other NIDS solutions, such as Bro, adopt similar architectures. Currently, Snort faces three major challenges. First, Snort stores its anomaly signature database as a set of rules that characterize protocol information, port details, and suspicious content of malicious traffic, requiring packets to be matched against this rule set. Typically, each attack type corresponds to one identification rule. As the variety of network attacks continues to grow, the number of rules in Snort' s rule base has increased substantially, intensifying the computational load for detection. Measurements show that on a single-core platform (2.0 GHz AMD Opteron processor), Snort' s throughput reaches a maximum of less than 500 Mbps[4]. Second, attackers exploit characteristics of target operating systems' TCP/IP protocol stacks to fragment or overlap attack code across multiple packets, attempting to evade detection by network intrusion detection systems. To identify these potential threats, Snort provides complex functions such as IP fragment reassembly and stream reassembly to reconstruct network data flows, which introduce significant computational and storage overhead. Third, while chip performance improvements roughly follow

Moore' s Law (doubling every 18 months), Gilder' s Law posits that backbone network bandwidth will double every six months over the next 25 years. According to Gilder' s Law, bandwidth growth exceeds the CPU processing speed growth predicted by Moore' s Law by a factor of three. Consequently, single-threaded network intrusion detection systems like traditional Snort face severe performance challenges.

The emergence of multi-core processor platforms presents new opportunities for implementing high-speed network intrusion detection systems on general-purpose CPU platforms. However, traditional NIDS were developed for single-core platforms—Snort operates in single-threaded mode, and Bro' s data analysis component also runs single-threaded—preventing them from fully leveraging multi-core architectures for performance gains. Multi-threaded network intrusion detection systems have consequently become an active research area. The Snort development team is working on SnortSP 3.0 to support multi-threaded packet processing[5], while Bro' s main developers have also identified multi-threaded packet processing as a priority for future work[6]. Nevertheless, network packet processing applications are stateful[3], with tightly coupled modules and high data dependencies, making parallelization particularly challenging. Although Snort processes packets individually, its internal IP fragment reassembly and stream reassembly mechanisms jointly process multiple packets belonging to the same IP fragment or flow to reconstruct application streams for Deep Packet Inspection (DPI). When inspecting packets, Snort must rely on state changes from previous packets in the same flow, making packet-based parallelization strategies extremely difficult to apply.

Addressing these challenges, this paper experimentally analyzes the processing time of Snort' s various detection components and proposes pSnort, a parallel intrusion detection system employing flow-based parallelism and software pipelining. pSnort focuses on parallelizing the most time-consuming portions of Snort' s processing pipeline to maximize utilization of multi-core computational resources while eliminating potential synchronization/mutual exclusion overhead through architectural design. Experimental results demonstrate that pSnort achieves processing speeds of 1 Gbps, delivering up to 147% performance improvement and approximately 2.8x speedup over the original Snort.

The remainder of this paper is organized as follows: Section 2 reviews related work in parallel network intrusion detection systems, including various parallel execution approaches. Section 3 presents the pSnort parallel architecture based on analysis of Snort' s processing stages. Section 4 describes implementation techniques for pSnort. Section 5 provides experimental results and performance analysis. Section 6 concludes and outlines future work.

2 Related Work

Traditional network intrusion detection systems have been thoroughly optimized for single-core platforms. However, due to complex CPU design logic and limi-

tations in manufacturing processes and power consumption[2], performance can no longer sustain exponential growth, prompting the industry to shift toward multi-core platforms for performance improvements. This transition has sparked research into parallelization methods for NIDS, with various parallel execution models proposed for contemporary multi-core platforms. Among these, reference[3] summarizes two fundamental multi-threaded execution models: Run-To-Complete (RTC) and Software Pipeline (SPL).

The RTC model, as its name suggests, replicates multiple execution units and distributes different tasks among them to achieve performance gains. Each thread performs identical functions: decoding, fragment reassembly, flow lookup and reassembly, rule matching, etc. With one thread bound to each core, increasing the number of cores directly improves packet processing throughput. The software pipeline model, in contrast, divides the target system into several stages, with each stage implemented as a separate thread running on a dedicated core. Different threads communicate through queues or buffer pools. Borrowing from pipeline principles, this model enhances system throughput by completing the overall processing flow across different cores in stages, where each core handles only a portion of the complete process and buffer pools connect the stages. The primary drawback of the software pipeline model is increased inter-thread communication overhead. In practical engineering design, stage partitioning requires careful consideration to prevent synchronization/mutual exclusion overhead from negating the performance benefits of multi-threading.

Based on software pipelining principles, B. Haagdorens et al. proposed five multi-threaded Snort architectures and evaluated their performance. Their study found that on a dual-core platform with hyper-threading enabled (equivalent to four logical cores), only two of the five designs achieved 11% to 16% performance improvement, while the remaining three designs resulted in varying degrees of performance degradation[7]. This research demonstrates that when partitioning Snort using software pipelining, inter-thread synchronization overhead must be carefully considered; otherwise, the performance loss from such overhead often outweighs the gains from multi-threading.

Intel also proposed a solution for improving Snort performance using multi-core technology in a technical report[8]. They evaluated throughput and L2 cache hit rates for two parallel approaches: complete parallelization and flow-distribution-based parallelization. Their experiments on a Dual-Core Intel Xeon LV processor platform showed that with a packet trace containing only 175 TCP connections, both approaches achieved 540-560 Mbps throughput and approximately 80% cache hit rate. However, with a trace containing 25,000 TCP connections, the flow-distribution-based approach achieved 188 Mbps throughput and 70% cache hit rate, while the complete parallelization approach only reached 29 Mbps throughput and 42% cache hit rate.

To achieve even higher-speed network intrusion detection systems (exceeding 10 Gbps), many researchers have proposed hardware-based string matching acceleration for deep packet inspection. In these approaches, Ternary Content

Addressable Memory (TCAM) has become a mainstream hardware acceleration solution due to its support for Longest Prefix Matching. Yu Fang et al. categorized intrusion detection string matching into simple and complex patterns, implementing both on TCAM. Simulation results demonstrated that this approach provides 2 Gbps processing capability[9]. Jung-Sik Sung et al. improved the TCAM lookup algorithm, achieving 10 Gbps processing capability[10]. However, TCAM solutions suffer from high cost, high power consumption, and low storage efficiency, and cannot meet the demands of ever-growing rule sets[11].

Network processors have also attracted significant attention as a platform for implementing high-performance network intrusion detection systems. Seok-Min Kang et al. implemented a worm and virus defense system based on a network processor platform using TCAM hardware technology, capable of operating on 10 Gbps high-speed links[12]. However, using network processors increases programming complexity, and such solutions are typically only suitable for low-level packet processing near the network protocol stack. Complex layer 4-7 application processing often involves frequent memory accesses, making network processors less suitable for these tasks.

3 pSnort Architecture

3.1 Snort Processing Flow

[Figure 1: see original paper] illustrates a typical Snort packet processing flow diagram. Other network intrusion detection systems feature similar architectures. As shown, Snort packet processing comprises four components: decoding, preprocessing, rule analysis, and output. The preprocessing stage includes functions such as fragment reassembly, flow record management, stream reassembly, as well as optional features like port scanning and HTTP[2] protocol analysis. For the back-end rule detection module, fragment reassembly and flow management are essential, while other functions are optional. Within Snort' s codebase, fragment reassembly is primarily implemented by the Frag3 module, while flow record management and stream reassembly are handled by the Stream5 module. Across the entire preprocessing stage, the Frag3 and Stream5 modules consume the vast majority of processing time.

When designing a parallel Snort architecture using the RTC model, minimizing synchronization/mutual exclusion overhead introduced by multi-threading is critical. This overhead typically stems from access to shared data. [Figure 2: see original paper] depicts an RTC-based parallel Snort architecture. As illustrated, synchronization overhead from shared flow record access is difficult to avoid. While employing separate flow tables for each thread could eliminate this overhead, it would prevent cross-flow detection since related flows might be distributed across different threads.

When applying the software pipeline model to parallelize Snort, partitioning Snort' s stages too finely must be avoided. Since contemporary general-purpose CPUs generally lack hardware thread migration support and fast inter-core

communication, software pipeline architectures often incur significant message-passing overhead that impacts overall performance.

3.2 pSnort Architecture Design

[Figure 3: see original paper] Based on the analysis above, we can partition Snort' s data processing flow into two major phases: (1) protocol processing and (2) rule matching. In the protocol processing phase, Snort focuses on information carried in packet headers, with the primary task being to calculate and store the positions of each layer's protocol headers within the packet for subsequent stages. In the rule matching phase, Snort examines the packet payload, searching the rule database to determine whether the packet may carry attack information.

Experimental data shows that on a server equipped with an Intel Xeon E5420 CPU and 8GB DDR2 667 memory, the protocol processing phase incurs a latency of 3 s, while the rule matching phase incurs 21 s[14], accounting for 87.5% of total processing time due to costly string matching operations. Clearly, rule matching represents the bottleneck in Snort' s processing pipeline. Therefore, parallelizing this phase across multiple threads promises significant performance improvements.

Based on this analysis, we propose pSnort, a software-pipelined parallel Snort architecture shown in [Figure 3: see original paper]. Packet processing is collaboratively performed by one protocol processing thread and multiple rule matching threads. With only a single protocol processing thread, flow record access requires no locking, eliminating mutual exclusion overhead from concurrent multi-threaded access. For the rule database accessed by multiple rule matching threads, since all operations are read-only, no locking is required either. Thus, our architecture inherently avoids severe synchronization/mutual exclusion overhead.

Although rule matching processes packets individually, our architecture cannot employ packet-based parallelism when applied to Snort. Snort' s rule description language provides the flowbit keyword, which only functions correctly when packets belonging to the same flow pass through the same rule matching thread in order; otherwise, some attacks may be missed. Therefore, we employ a micro-decoding module before the protocol processing stage to perform minimal decoding for extracting the flow quadruple, ensuring packets from the same flow are processed by the same rule matching module in sequence.

Snort's packet processing code utilizes a global Packet structure, with one Packet instance sufficient for the entire processing flow since Snort processes packets sequentially. In pSnort, however, each thread completes only a portion of the overall process and immediately moves to the next packet upon completion. This means multiple packets are being processed concurrently within the pSnort system, necessitating a Packet structure buffer pool to cache multiple Packet instances.

The pSnort processing flow after acquiring a packet from the network interface proceeds as follows:

1. Obtain a Packet instance from the buffer pool and perform protocol processing, which assigns corresponding address values to fields in the Packet structure.
2. Rule matching threads retrieve Packet instances from their respective message queues for rule matching. Upon completion, the Packet instance is returned to the Packet buffer pool.

Both protocol processing and rule matching stages may generate alerts, requiring output modules in each thread.

3.3.1 Packet Buffer Pool

The Packet buffer pool is accessed by both processing phases, requiring lock protection. However, we can reduce the conflict domain by assigning each rule matching thread its own Packet buffer pool, thereby decreasing lock contention probability. Since each rule matching thread has a dedicated buffer pool and packets from the same flow must be processed on the same thread, flow information must be obtained before acquiring a Packet instance. As shown in [Figure 3: see original paper], the system performs micro-decoding to extract flow information before accessing the Packet buffer pool and selects the appropriate buffer based on this information.

The Packet buffer pool stores pre-allocated Packet instances in a pointer array, using an index value for allocation and deallocation operations. Pseudocode is shown in [Figure 4: see original paper].

3.3.2 Message Passing Queue

Each rule matching thread includes a message passing queue, creating a "single-producer-single-consumer" relationship between the protocol processing thread and each rule matching thread. This enables the use of lock-free circular queues for message passing.

Due to the lock-free circular queue implementation, queue length must be predetermined. The appropriate length depends on network traffic burstiness, which is difficult to estimate in advance. In our design, we experimentally use a value of 1024. When the message queue is full, threads continuously retry until a free slot becomes available.

3.4 Global Variables and Data Dependencies

Parallelizing Snort requires addressing two issues: (1) privatization of global variables to ensure correct anomaly detection, and (2) data dependency problems.

The first issue can be easily resolved using GCC' s[3] `__thread` extension attribute. The data dependency problem arises because packets experience queuing delay between protocol processing and rule matching stages. Information needed by the rule matching stage may become inconsistent. For example, when the final FIN packet of a TCP flow completes protocol processing, the corresponding flow record may be deleted, yet some packets from that flow might still be queued awaiting rule matching. When the rule matching stage processes these packets, it cannot locate their flow records, causing processing logic errors.

Data dependency issues can be divided into two sub-problems: (1) when the back-end stage requires information from the front-end stage, that information may have been updated; (2) when the back-end stage requires information from the front-end stage, that information may have been deleted. If stage partitioning achieves complete data isolation—where different stages require different information—data dependency issues can be avoided. Unfortunately, Snort does not satisfy this condition. Furthermore, since processing speeds cannot remain perfectly synchronized, data dependency issues cannot be completely eliminated.

If sub-problem (1) existed, the protocol processing stage would need to record flow state information for each packet for back-end query, consuming enormous memory space. If only sub-problem (2) existed, the protocol processing stage would simply need to preserve flow records until they could be safely deleted. Fortunately, Snort does not exhibit sub-problem (1). Therefore, we employ the following solution for sub-problem (2):

- Assign a reference count to each flow record. Increment the reference count atomically when the protocol processing stage enqueues a packet, and decrement it when rule matching completes. Reference count operations must be atomic.
- When the protocol processing stage needs to delete a flow record, mark its state, remove it from the flow table, and place it in a deletion list. This list is invisible to the protocol processing thread, preventing interference with normal processing while allowing rule matching threads to safely access required flow records. When a flow record is marked for deletion and its reference count reaches zero, it can be safely removed from the deletion list.

5 Experimental Evaluation

5.1 Experimental Environment

The hardware parameters of our experimental platform are provided in . We designed a hardware acceleration card as the network interface for pSnort. This NIC uses Direct Memory Access (DMA) to deliver packets directly to the application, significantly reducing memory copies, CPU interrupts, and user/kernel context switching overhead, thereby freeing the CPU from packet capture tasks and improving system performance. The card features four Gigabit ports capa-

ble of handling up to 4 Gbps of traffic; we used one port for our experiments. Test packet traces were captured from a GE link at a research institute' s network exit. shows the parameters of three packet traces. We used tcpreplay[13] on another server to replay traces and simulate external network environments, employing the `-t` parameter to achieve maximum replay rate. On our platform, tcpreplay achieved a maximum transmission rate of 918.18 Mbps.

Three rule sets were selected for evaluation. Rule set 1 contained Trace parameters Trace1 Trace2 Trace3 Average packet size Size (GB) Experimental hardware parameters Intel Xeon E5420 2 sockets, 4 cores each, total 8 cores; per-core L1 cache 32KB: 16KB (instruction) + 16KB (data); four cores share 6MB L2 cache in pairs DDR2 667 Total 8GB Hardware acceleration card with DMA 4 ports, handles up to 4 Gbps traffic, using one port for experiments.

5.2 Performance Metrics

The performance metrics used in our experiments are throughput and speedup. Define T_i as pSnort' s throughput (Mbps) with i rule matching threads enabled, P_i as the number of packets processed by pSnort, P_{total} as the total number of packets in the trace file, and S as tcpreplay' s transmission rate (Mbps). Throughput is calculated by the following formula: pSnort：基于多核处理器的并行入侵检测系统 (Mbps) total (Mpbs) Define A_i as the speedup with i rule matching threads. Speedup is calculated by the following formula: (Mpbs) (Mpbs)

5.3 Experimental Results and Analysis

[Figure 5: see original paper] shows the relationship between pSnort throughput and thread count using rule set 2. A thread count of 0 represents the throughput achieved by original Snort. The results indicate that with one thread, pSnort performs worse than original Snort due to lock overhead. However, as thread count increases, pSnort' s throughput grows accordingly. With seven threads, pSnort can process all traffic in Trace1 and Trace2, reaching the maximum achievable throughput on our platform.

[Figure 6: see original paper] shows the corresponding speedup. With rule set 2, pSnort achieves 99%, 107%, and 88% performance improvement over original Snort under the three different traces.

The speedup does not grow linearly because our architecture parallelizes only a portion of the entire processing pipeline. With equal per-packet processing latency across threads, speedup can be calculated using the following formula, where D_p represents per-packet processing latency in the protocol stage and D_R represents per-packet processing latency in the rule matching stage. According to[14], setting $D_R = 21$ yields the ideal speedup curve.

As shown, with $i = 7$, pSnort' s actual speedup differs from the ideal by approximately 1.5. This gap stems from three factors: First, actual traffic distribution is unbalanced—pSnort' s flow distribution hash algorithm cannot guarantee

perfectly balanced traffic across threads. Second, as thread count increases, per-packet processing latency in both pSnort stages gradually increases, further limiting speedup. Third, due to rule specificity, a small number of flows have large numbers of matching rules (e.g., HTTP traffic on port 80). Even with balanced traffic distribution, if a thread is assigned a flow with many matching rules, its processing latency increases, resulting in unequal processing delays across threads.

According to formula (3), when $i \geq D_p/D_R$, the protocol processing stage becomes the primary performance bottleneck, and increasing thread count yields diminishing returns. With $D_R = 21$, when $i \geq 7$, the protocol processing stage becomes the bottleneck, and further thread count increases do not bring significant performance improvements. Nevertheless, we can enhance pSnort' s performance through improved load balancing.

[Figure 8: see original paper] shows the relationship between thread count and throughput for different rule sets using Trace2. A thread count of 0 represents original Snort' s throughput. Since rule set 3 contains more rules than the other two sets, throughput is lower. The other two rule sets successfully processed all packets in Trace2.

[Figure 9: see original paper] shows the corresponding speedup. Rule set 3 exhibits greater speedup due to its larger rule set and increased matching workload. Using Trace2, pSnort achieves 115%, 107%, and 147% performance improvement over original Snort with the three rule sets, respectively.

6 Conclusion

To fully exploit the computational capabilities of multi-core platforms and meet the demands of intrusion detection on high-speed links, this paper proposes pSnort, a software-pipelined parallel Snort architecture. Our architecture carefully considers synchronization/mutual exclusion issues inherent in multi-threading, employing software pipelining to eliminate lock contention overhead from shared data access such as flow records while preserving support for cross-flow detection. Experiments demonstrate that pSnort achieves up to 147% performance improvement over original Snort. Due to the similar processing flows in most network applications, our proposed architecture exhibits high generality.

Future work includes: (1) Extending our architecture to incorporate additional preprocessors beyond Frag3 and Stream5. (2) Optimizing pSnort' s load balancing algorithm to achieve higher speedup. (3) Identifying and optimizing pSnort' s performance bottlenecks. The software pipeline model incurs significant inter-core communication overhead from data passing. Current CPU architectures, particularly cache systems, generally lack support for efficient inter-core communication. We aim to carefully measure and analyze pSnort' s performance parameters, especially cache miss impacts, to identify system bottlenecks and propose methods to reduce data transfer volume for performance improvement.

(4) Since the protocol processing stage primarily involves memory operations with minimal computation and modest CPU frequency requirements, we can design a new hardware acceleration card that integrates protocol processing onto the card' s onboard CPU, further reducing CPU load and achieving higher performance.

References

[1] M. Roesch. Snort -Lightweight Intrusion Detection for Networks. In Proceedings of the 13th USENIX Conference on System Administration, pages 229–238, 1999.

[2] Gelsinger P. Microprocessors for the New Millennium: Challenges, Opportunities, and New Frontiers. Proceedings of the International Solid State Circuits Conference, 2001, 22-25.

[3] J Verdu, et al. MultiLayer Processing - An execution model for parallel stateful packet processing. ANCS' 08, November 6-7, 2008, San Jose, CA, USA.

[4] Derek L. Schuff and Vijay S. Pai. Design Alternatives for a High-Performance Self-Securing Ethernet Network Interface. Parallel & Distributed Processing Symposium (IPDPS), 2007 IEEE International; Long Beach,CA .

[5] Snort Security Platform (SnortSP) 3.0 Beta http://www.snort.org/dl/snortsp, 2007

[6] V Paxson, R. Sommer. An Architecture for Exploiting Multi-Core Processors to Parallelize Network Instrusion Prevention. Proceedings of IEEE Sarnoff Symposium, 2007, 1-7.

[7] B Haagdorens, et al. Improving the performance of signature-based network intrusion detection sensors by multi-threading. Proceedings of the 5th International Workshop on Information Security Applications, 2004, 188-203.

[8] Intel. Supra-linear Packet Processing Performance with Intel® Multi-core Processors, 2006.

[9] Yu Fang, Randy H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using tcam. In ICNP, pages 174–183, 2004.

[10] Jung-Sik Sung, eok Min Kang, Youngseok Lee, Taeck-Geun Kwon, and Bong-Tae Kim. A multi-gigabit rate deep packet inspection algorithm using tcam. In GLOCOM), pages 453–457, 2005.

[11] David E. Taylor. Survey and taxonomy of packet classification techniques. ACM Comput. Surv. 37(3):238–275, 2005.

[12] Seok-Min Kang et al. Design and Implementation of a Multi-gigabit Intrusion and Virus/Worm Detection System. ICC ' 06. IEEE International Conference on Communication. 2006.

[13] A.Turner. tcpreplay. http://tcpreplay.synfin.net/trac/

[14] Haiyang Jiang et. al. Performance Measurement and Analysis of Software Pipeline IDS Model on Multi-core Processor. Submitting to EuroNF Workshop on Traffic Management and Traffic Engineering for the Future Internet. 2009

Author Biographies:

Peng He: Ph.D. candidate, Network Technology Research Center, Institute of Computing Technology, Chinese Academy of Sciences, hepeng@ict.ac.cn

Haiyang Jiang: Ph.D. candidate, Network Technology Research Center, Institute of Computing Technology, Chinese Academy of Sciences

Gaogang Xie: Director and Professor, Network Technology Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Ph.D. supervisor

Note: Figure translations are in progress. See original paper for figures.