Practical and Accurate Runtime Application Protection Against DoS Attacks

Mohamed Elsabagh[1(✉)], Dan Fleck[1], Angelos Stavrou[1], Michael Kaplan[2], and Thomas Bowen[2]

[1] George Mason University, Fairfax, VA 22030, USA
{melsabag,dfleck,astavrou}@gmu.edu
[2] Vencore Labs, Red Bank, NJ 07701, USA
{mkaplan,tbowen}@vencorelabs.com

Abstract. Software Denial-of-Service (DoS) attacks use maliciously crafted inputs aiming to exhaust available resources of the target software. These application-level DoS attacks have become even more prevalent due to the increasing code complexity and modular nature of Internet services that are deployed in cloud environments, where resources are shared and not always guaranteed. To make matters worse, many code testing and verification techniques cannot cope with the code size and diversity present in most services used to deliver the majority of everyday Internet applications. In this paper, we propose *Cogo*, a practical system for early DoS detection and mitigation of software DoS attacks. Unlike prior solutions, *Cogo* builds behavioral models of network I/O events in linear time and employs Probabilistic Finite Automata (PFA) models to recognize future resource exhaustion states. Our tracing of events spans then entire code stack from userland to kernel. In many cases, we can block attacks far before impacting legitimate live sessions. We demonstrate the effectiveness and performance of *Cogo* using commercial-grade testbeds of two large and popular Internet services: Apache and the VoIP OpenSIPS servers. *Cogo* required less than 12 min of training time to achieve high accuracy: less than 0.0194% false positives rate, while detecting a wide range of resource exhaustion attacks less than seven seconds into the attacks. Finally, *Cogo* had only two to three percent per-session overhead.

Keywords: Software DoS · Early detection · Slow-rate attacks · Probabilistic Finite Automata

1 Introduction

Software availability is a major concern for the success of today's interconnected Internet services. As technologies become more advanced and complex, servicing

Electronic supplementary material The online version of this chapter (doi:10.1007/978-3-319-66332-6_20) contains supplementary material, which is available to authorized users.

M. Dacier et al. (Eds.): RAID 2017, LNCS 10453, pp. 450–471, 2017.
DOI: 10.1007/978-3-319-66332-6_20

an ever increasing number of users and devices, they become much harder to properly design and test against inputs and runtime conditions that may result in resource exhaustion and, eventually, denial-of-service (DoS). Recent surveys clearly indicate that business owners are concerned about DoS attacks over other security concerns [6,8]. A system is vulnerable to resource exhaustion attacks if it fails to properly regulate the resources that can be allocated to individual user sessions and the service overall. Resource DoS attacks can target system resources such as memory, computing (CPU), and I/O including file access and traditional network resources [21,23,37]. Contrary to the common belief, resource exhaustion attacks are increasing in numbers, becoming even more prevalent and risky when compared to network layer attacks [1,21].

Recent work by Elsabagh et al. [25] proposed Radmin, a system for detecting DoS attacks at the application layer. Radmin operated by learning (offline) and enforcing (online) resource consumption patterns of programs. Radmin showed promising results; however, it had a quadratic training time complexity in the training data size that makes it prohibitive to apply to large code bases. Moreover, Radmin was tested on stateless traffic and synthetic attacks rather than on live traffic and known attacks used in practice. Radmin also did not cover network state and I/O which are common targets for attacks. Another limitation was that Radmin was heavily dependent on "normal" patterns of pure resource utilization without modeling the rate at which individual resources were acquired and released. As we show in our experiments, lack of taking into consideration when individual resources were allocated can lead to prolonged evasion by Slow-rate [11] attacks, violating the early detection goal of Radmin.

In this paper, we propose *Cogo* as a novel Probabilistic Finite Automata (PFA) based system for runtime detection and mitigation of software resource exhaustion DoS attacks. *Cogo* fully addresses all the aforementioned limitations of Radmin, enabling early detection of real-world attacks in many cases before they are able to affect the service operation or quality. Our approach operates in two phases: offline and online. In the offline phase, *Cogo* monitors the entire resource consumption behavior of the target program — including its network I/O — and builds PFA models that characterize the program's resource behavior over time. *Cogo* monitors network I/O at the individual socket level and supports monitoring of containerized processes. To reduce modeling complexity, we introduce an efficient PFA learning algorithm that operates in linear time. During the online phase, *Cogo* actively monitors the program and detects deviations from the learned behaviors. It attributes anomalies to the specific threads and connections causing them, allowing for selectively limiting resource utilization of individual sessions that may violate the models.

We built a working prototype implementation of *Cogo* by extending the code base of Radmin [25] which offered several integrated user/kernel tracing capabilities and an extensible PFA detection engine. We extended Radmin by supporting new low-level network I/O monitoring, process migration, and monitoring containerized processes. Extending Radmin allowed us to benchmark our approach

in a unified way and provide comparative results.[1] We discuss two case studies using real-world attacks and commercial-grade testbeds against The Apache HTTP Server [2] and the VoIP OpenSIPS [9] server. In our experiments, *Cogo* achieved a significant improvement in training time over Radmin, requiring only few minutes instead of days to train and build the models. This is significant since in real-world systems training data are expectantly large in size. In addition to short training time, *Cogo* achieved a low false positive rate (FPR) (0.019% for Apache, 0.063% for OpenSIPS) using small models (76 MB for Apache, 55 MB for OpenSIPS). Moreover, *Cogo* swiftly detected the attacks in less than seven seconds into their execution, resulting in **zero** downtime in some cases. Its runtime overhead is negligible. it increased the latency by 0.2 ± 0.3 ms per request on average, resulting in two to three percent per-session overhead.
To summarize, this study makes the following contributions:

- Demonstrates *Cogo* as a system for early detection and mitigation of resource exhaustion DoS attacks against real-word complex Internet services. Our approach extends prior work on Radmin [25] by enabling network stack tracing from the application to the kernel, monitoring containerized processes, and attaching to running processes.
- Presents and discusses a linear time training algorithm that reduces the training and model building time complexity.
- Studies the effectiveness of *Cogo* using realistic testbeds with real-world attacks on Apache and the VoIP OpenSIPS server. The results demonstrate that *Cogo* is suitable for large-scale deployment as it is scalable, accurate, has low false positives, and can mitigate real-world attacks.

2 Assumptions and Threat Model

Cogo focuses on DoS attacks that occur at the application layer such as algorithmic, state, and protocol-specific attacks. Volumetric attacks targeting the network and transport layers, as well as other attack vectors such as code execution and memory exposure are outside the scope of this work. We assume that attackers have full knowledge of the internals of the attacked program and can craft benign-looking inputs that prevent the attacked program from serving legitimate clients (a DoS attack). To protect a program with *Cogo*, we assume the availability of benign training inputs that cover the typical desired behavior of the program. *Cogo* uses kernel tracing; our prototype currently supports only Linux and Unix-like operating systems since they power the majority of servers.[2] However, the approach itself does not place restrictions on the runtime environment and can be ported to other operating systems with little effort.[3]

[1] By building on Radmin, *Cogo* inherits other monitoring sensors from Radmin such as CPU and memory sensors.

[2] Market share of operating systems by category: https://en.wikipedia.org/wiki/Usage_share_of_operating_systems.

[3] For Microsoft Windows, kernel tracing can be implemented using the Event Tracing for Windows (ETW) kernel-mode API: https://msdn.microsoft.com/en-us/windows/hardware/drivers/devtest/adding-event-tracing-to-kernel-mode-drivers.

We only focus on detection; proper remediation strategies after attack detection should be implemented by the operator and are outside the scope of this work. Nevertheless, *Cogo* offers the option to migrate the offending process or session to another server, reduce its resource priority, or terminate it based on a configurable policy. Finally, we assume that attackers can be local or remote, but cannot overwrite system binaries or modify the kernel.

3 The *Cogo* System

Cogo operates in two phases: offline training phase and online detection phase. In the offline phase, *Cogo* monitors the behavior of the target program on benign inputs and collects a trace of network I/O measurements. The measurements are sequences of raw data that include the event type (socket open, close, send, receive), the consumption amount of the related resource (number of owned sockets, traffic rate per socket), and meta data such as the PID, the socket inode number, and timestamps.

The raw resource consumption amounts are encoded (quantized) over a countable finite alphabet Σ (a finite set of symbols). $|\Sigma|$ is a tuning parameter, typically less than 16 for a maximum of 16 different consumption levels. Encoding is done by mapping (many-to-few) each raw resource consumption value to one symbol from Σ. This is necessary since the PFAs (state machines) only work with a finite set of values. Since encoding is a typical step in constructing finite automata from arbitrary values, and due to space constraints, we refer interested readers to [25,26] for more detail.[4]

Cogo constructs multiple PFAs from the measurements, one PFA per resource type. The PFAs capture both the spatial and temporal network I/O patterns in the measurements. In the online phase, *Cogo* executes the PFAs as shadow state machines along with the target program and raises an alarm if a deviation of the normal behavior is detected. *Cogo* detects anomalous behavior using the statistical properties of the PFAs — namely the transition probabilities on the PFA edges. In the following, we discuss how *Cogo* monitors network I/O and its PFA learning and detection algorithms.

3.1 Network Tracing

Cogo monitors the network activity of the target program by intercepting the traffic and socket events that happen in the context of target processes inside the kernel. Specifically, it monitors all socket creation and destruction events triggered by the target processes and tracks traffic sent or received on those sockets. *Cogo* computes the transmit (TX) and receive (RX) rates per second from the size and direction of the monitored traffic.

Cogo differentiates sockets from regular file descriptors inside the kernel as follows: First, it retrieves a target process task structure in kernel space using

[4] We use "measurements" to refer to encoded measurements in the rest of this paper.

the global process identifier (PID). (The task structure is the actual structure that represents the process inside the kernel.) It traverses the task structure and extracts the file descriptors table owned by the process. For each file descriptor, *Cogo* extracts the inode object associated with the file descriptor. (The inode object is a kernel structure that contains all needed information to manipulate and interact with a file descriptor. An inode represents each file in a file system, including regular files and directories, as well as special files such as sockets, devices, and pipes.) *Cogo* checks if the inode object contains an embedded (allocated member) socket object. If found, *Cogo* marks the corresponding file descriptor of the inode as a socket descriptor. *Cogo* tracks all identified sockets by their low-level unique inode numbers throughout their lifetime.

For each identified socket, *Cogo* extracts the socket Internet protocol family from the socket kernel structure. (The protocol family defines the collection of protocols operating above the Internet Protocol (IP) layer that utilize an IP address format. It can be one of two values: INET6 and INET for the IPv6 and IPv4 protocol families, respectively.) This is essential for determining how to interpret the socket network addresses. Given a socket protocol family, *Cogo* extracts the local and foreign addresses and port numbers, if available. Foreign port numbers may not be available if the socket is a listening or a datagram socket.

Cogo intercepts all transmit and receive socket events that occur in the context of the monitored process in kernel space, including regular I/O operations such as streamed and datagram I/O, asynchronous I/O (AIO) operations, and operations utilizing a socket iterator. *Cogo* collects the direction (TX or RX) and size of the traffic, and associates them with the corresponding socket inode number. The TX and RX rates are computed periodically per socket. The period length is configurable (defaults to 1 s). To minimize memory and runtime overhead, *Cogo* installs a kernel timer that ticks once per period length, requiring minimal memory per socket as only the last tick timestamp and total traffic size need be kept in memory. It also minimizes runtime overhead by avoiding unnecessary context switches to compute the rates. *Cogo* also monitors the socket status: connected or disconnected. When a socket disconnects or is freed by the kernel, *Cogo* purges any structures associated with that particular socket from its kernel memory.

3.2 Training and Learning

Cogo employs Probabilistic Finite Automata (PFA) based learning and detection. *Cogo* builds one PFA for each monitored resource: one PFA for socket creation and destruction, one PFA for TX rate, and one PFA for RX rate. *Cogo* uses the PFAs to compute the probability of observed measurements in the online phase. In the following, we present a training algorithm that runs in time linear in the measurements length, making *Cogo* attractive and realistic for real-world deployment.

Constructing Bounded Generalized Suffix Trees. To construct each resource PFA, first, *Cogo* builds a bounded Generalized Suffix Tree (GST) from the resource measurements. (A suffix tree is a tree containing all suffixes of a given string. A GST is a suffix tree for a *set* of strings.) Given a set of strings S over an alphabet Σ (a finite set of symbols), a GST over S contains a path from the root to some leaf node for each suffix in S. Each edge in the GST is labeled with a non-empty substring in S; the labels of outgoing edges from the same node must begin with unique symbols. A GST can be constructed in linear time and space $O(n)$ where n is the total number of symbols in S, using Ukkonen's algorithm [36]. A GST allows efficient implementations of several string query operations over sets of strings such as linear time substring searching and finding the longest common substring among all the strings in the set. *Cogo* limits the depth of the GST by processing the measurements into non-overlapping subsequences of maximum length L.[5] This bounds the depth of the GST to L and the space requirements per GST to $O(|S|L)$.

After constructing the bounded GST, *Cogo* counts the number of occurrences of each substring in the tree. This corresponds to the number of leaf nodes in the subtree rooted at each node in the tree. These counts are computed in a single depth-first traversal of the GST. For each parent-child nodes in the tree, the ratio between the child's count to the parent's count gives the conditional probability of seeing the first symbol of the corresponding child substring after the parent's. More formally, the prediction probability of a symbol s_j after a substring $s_i s_{i+1} \ldots s_{j-1}$ can be computed as:

$$P(s_j|s_i s_{i+1} \ldots s_{j-1}) = \frac{count(s_i s_{i+1} \ldots s_{j-1} s_j)}{count(s_i s_{i+1} \ldots s_{j-1})}, \tag{1}$$

which *Cogo* computes on-the-fly during the depth-first traversal of the GST to count the substrings, and stores it in each child node in the tree.

Inferring the PFAs. *Cogo* infers a PFA from the GST. Each PFA is a 5-tuple $(\Sigma, Q, \pi, \tau, \gamma)$, where: Σ is a finite set of symbols processed by the PFA; Q is a finite set of states, and $q^\circ \in Q$ is the start state; $\tau \colon Q \times \Sigma \to Q$ is the state transition function; and, $\gamma \colon Q \times \Sigma \to [0, 1]$ is the transition probability function.

To infer a PFA from the GST, *Cogo* starts by creating a forest of unconnected PFA nodes where each node has a unique ID and corresponds to exactly one node in the GST. It then traverses the GST in depth-first order: For each edge between each parent (source) and child (destination) nodes in the GST, *Cogo* checks the length of the edge label. If the label has exactly one symbol, *Cogo* adds a transition between the corresponding source and destination nodes in the PFA, sets the transition probability to the child node probability in the GST, and sets the transition symbol to the edge label. If the edge has a label of length

[5] We found that non-overlapping subsequences were sufficient for large-scale deployments. However, it may be desired to overlap subsequences to maximize fidelity of very small datasets.

greater than one, i.e., the label is a substring consisting of multiple symbols, *Cogo* adds nodes to the PFA corresponding to each inner symbol in the label; adds a PFA transition from the source state to the node corresponding to the first symbol in the label; and adds another transition from the last inner symbol in the label to the destination node. Formally put, given the edge $u \xrightarrow{s_i s_{i+1} \ldots s_j} v$ in the GST, *Cogo* adds the following path to the PFA: $u' \xrightarrow{s_i, count(u[s_i])/count(u)} \bullet \xrightarrow{s_{i+1}, 1.0} \ldots \xrightarrow{s_{j-1}, 1.0} \bullet \xrightarrow{s_j, 1.0} v'$ where u' and v' are the corresponding nodes in the PFA of u and v. Recall that transitions in the PFA hold both a transition symbol and an emitted probability.

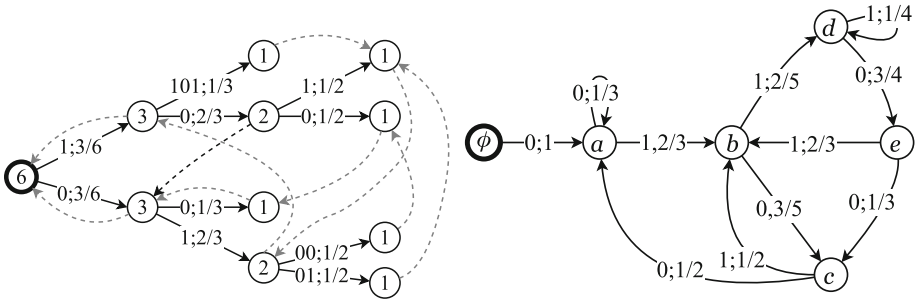
At this stage, this initial PFA contains paths that correspond to the substrings from the GST, and can be used for prediction so long as the *entire* substring is in the tree. However, if the next symbol following some substring is not in the tree, then a Markovian decision need be made since it may still be possible to predict the symbol using a shorter suffix. For this, the GST suffix links are used to find the next immediate suffix. In a GST, the node corresponding to the string $s_i \ldots s_j$ has a suffix link (a pointer) to the internal node corresponding to the string $s_{i+1} \ldots s_j$, i.e., its immediate suffix. This enables *jumping* to the next available context (history) in constant time. *Cogo* utilizes the suffix links to complete the PFA construction in the following manner: For each node u (visited during the depth-first traversal) and for each symbol $\sigma \in \Sigma$ that does not mark any outgoing edge from u, *Cogo* follows the suffix links starting from u until:

1. An internal node v is reached where the first symbol of the substring represented by that node equals σ. In this case, *Cogo* adds a transition between the corresponding two nodes to u and v in the PFA. It sets the transition symbol to σ and the transition probability to that stored in v in the GST.
2. The root of the GST is reached and it has an edge with a label that begins with σ to some child node v. Here, *Cogo* adds a transition between the corresponding u and v nodes in the PFA. It sets the transition symbol to σ and the transition probability to that stored in v.
3. The root is reached but it has no outgoing edges for σ. In this case, a loop-back transition on σ from u to itself is added and the transition probability is set to ρ_{min} (a small predefined value for the minimum transition probability).

Since the GST contains all suffixes, the resulting PFA would contain outgoing edges from the start state that never prefixed the training sequences. This can result in the PFA accepting anomalous behavior if an attack occurs at the very beginning of execution of a target process. *Cogo* eliminates those spurious transitions by keeping a set of the initials of the training sequences and pruning outgoing start state transitions from the PFA that do not correspond to those initials. This is done in constant time ($|\Sigma|$ comparisons). Using a single depth-first traversal, *Cogo* also removes any transitions that have a probability less than or equal to ρ_{min} and replaces them with loop-back transitions with ρ_{min} probability. During the same traversal, *Cogo* normalizes the probabilities across outgoing edges from each node.

Minimizing the PFAs. The PFA may contain redundancy such as unreachable states (because of eliminated transitions) or overlapping paths, resulting in unnecessary space overhead. To overcome this, *Cogo* minimizes the PFA using the following greedy approach. The goal is to reduce the size of the PFA as much as possible without incurring excessive training overhead, i.e., reduction time has to be linear in the size of the PFA. The minimization algorithm is based on the insight that paths farther away from the PFA root (the start state) are more likely to overlap sine they represent longer substrings.

Cogo iterates over the PFA in breadth-first order. Each time it visits a new state u, it searches for all previously visited states that are fully equivalent to the u. Two states are fully equivalent if they have the same outgoing transitions with the same transition symbols, probabilities, and destination states for each transition. *Cogo* groups all the equivalent states into a single state set. This process continues till all states in the PFA are visited, producing a set family of states. After that, all equivalent states set are removed and replaced with a single state in the PFA. The process is repeated on the resulting PFA till any of the following conditions occur: (*1*) Tthe PFA stops changing. (*2*) The minimization ratio, i.e., the size of the resulting PFA divided by the size of the old PFA, drops below some user defined threshold θ (defaults to 0.1). (*3*) The number of repetitions exceeds a user chosen threshold ζ (defaults to 100). The 2nd condition terminates the minimization stage once a diminishing returns point is reached. The 3rd condition gives the user the ability to control the hidden constant c of the minimization complexity $O(cn)$. This completes the construction of the PFA. Figure 1 illustrates an example of a bounded GST and the PFA inferred by *Cogo* from the set $\{01001101, 01010100\}$ where $L = 4$, i.e., the effective set is $\{0100, 1101, 0101, 0100\}$. The figure also shows how to compute the probability of the sequence 010 using the PFA.



(a) GST. (b) PFA.

Fig. 1. Bounded GST and final PFA produced by *Cogo* from the strings {01001101, 01010100} with maximum depth $L = 4$. Each edge in the GST has a substring and a transition probability. Dotted edges are suffix links in the GST. Each edge in the PFA has one symbol and a transition probability. Low probability edges are not shown for simplicity. To compute $P(010)$, we walk $\phi \rightarrow a \rightarrow b \rightarrow c$, giving $1 * 2/3 * 3/5 = 2/5$.

3.3 Detection

In the online phase, *Cogo* executes the PFAs as shadow state machines to the monitored program. Each measurement symbol results in a transition in the corresponding PFA of that measured resource type. Computing the probability of a sequence of symbols using a PFA reduces to walking the path corresponding to the symbols in the PFA, one transition at a time. This enables constant time online detection with minimal state keeping overhead, since only the current state and the transition symbol determine the next state.

For a sequence of n measurements, a PFA allows us to compute the prediction probability in $O(n)$ time and $O(1)$ space. Given a PFA M and a string of measurements $s = s_1 \dots s_l$, and assuming that M is currently in state q_j, we walk M (for each $s_i \in s$) where each transition *emits* the transition probability. The prediction probability of s by M is computed as the multiplication of all emitted probabilities along the walked path. *Cogo* decides that the sequence s is anomalous if the sequence resulted in at least t low probability transition in the PFA. Specifically, *Cogo* performs the following test:

$$\left|\left\{\gamma(q_j, s_i) \leq \rho_{min}, i \in 1 \dots l\right\}\right| \begin{cases} \leq t \rightarrow accept \\ > t \rightarrow reject \end{cases} \tag{2}$$

where $\gamma(q_j, s_i)$ is the transition probability of symbol s_i outgoing from state q_j, $q_{j+1} = \tau(q_j, s_i)$ gives the next PFA state, and t is the tolerance level. Recall that *Cogo* builds the PFAs such that low probability transitions are loop-back transitions, therefore they do not result in a state change in the PFA. This allows *Cogo* to offer tolerance by *forgetting* up to t low probability transitions. If a sequence results in more than t low probability transitions, *Cogo* raises an alarm.

3.4 Attaching to a Running Process

It is desirable in practice to be able to attach *Cogo* to a running process rather than starting a program under *Cogo*. For instance, attaching to running processes is essential for on-demand monitoring of processes that migrate among a cluster of servers. The main challenge in attaching to a run process in our context is that *Cogo* would not know in which states in the PFAs the process might be, nor how it got to those states. In other words, the process and the PFAs would not be in sync.

To resolve this, we developed the following non-deterministic PFA executor: First, *Cogo* attaches to the running program and starts monitoring at any arbitrary point in its execution. As measurements arrive, for each PFA for the target program, *Cogo* executes the PFA in a non-deterministic fashion by finding all paths that correspond to the incoming measurements, producing a set of potential paths $\mathcal{P}$ that the monitored process might have executed along. As more measurements arrive, *Cogo* extends each path in $\mathcal{P}$ by one transition at a time and checks if the detector accepts or rejects the new paths. A rejected path is

eliminated from $\mathcal{P}$. Eventually, either all paths in $\mathcal{P}$ are eliminated or only a single path remains. If all paths are eliminated, meaning the process has deviated, *Cogo* raises an alarm. If a single path remains, then the PFA and the process have been successfully synchronized and *Cogo* returns to normal operation.

3.5 Seeing Through Containers

It is typical that web applications are deployed in isolated instances, i.e., multiple instances of the web server would be running in isolation from each other on the same host. Each instance gets its own isolated view of the systems resources — including file system, CPU, RAM, and network interfaces. Common isolation techniques are either based on full virtualization (e.g., virtual machines) or operating-system-level virtualization using software containers (e.g., OpenVZ, LXC, and Docker). Full virtualization does not pose an issue for *Cogo* since *Cogo* can be deployed inside the web server VM itself. On the other hand, containers abstract out the OS kernel, making it impossible to deploy *Cogo* inside an isolated container since *Cogo* requires kernel access. Therefore, *Cogo* needs to be deployed on the host (outside the containers) yet monitor processes running inside isolated containers.

The main hurdle of seeing through containers is that PIDs inside a container are *local* to that container, i.e., they only identify the process inside that container PID namespace. Quoting from the Linux kernel manual, "a namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource."[6] The local PID serves no meaning outside the container where a process is running. Instead, the process is identified by a different *global* PID only known to the host running the container. Without knowledge of the global PID of a process, *Cogo* cannot attach and monitor that process in kernel space since the global PID is the PID seen by the kernel tracer in kernel space. Note that there are no containers or namespaces in kernel space.

We implemented a container-aware global PID resolver to be able to identify processes running in namespaces. First, *Cogo* starts the process in a *suspended* state *inside* the container and gets the process id in the container namespace (NSPID). (The NSPID from the loader process is the PID local to the container where the process is running.) This is possible by creating a custom loader process that outputs its NSPID and its namespace identifier (NSID), then sends a stop signal to *itself*. (The NSID is a unique namespace identifier.) When the loader process receives a continue signal, it loads the desired target program via a call to the `exec` system call. Given the NSPID and NSID, *Cogo* searches all namespaces on the host system for a matching child NSID that contains a matching NSPID. Once identified, *Cogo* extracts the global PID of the process from the identified child namespace. It then attaches to that process (the loader) using the global PID and sends it a continue signal. Upon receiving the continue signal

[6] The Linux kernel manpage for namespaces is available at: http://man7.org/linux/man-pages/man7/namespaces.7.html.

by the loader, it loads and executes the desired target using the `exec` system call, replacing the process image but retaining any PIDs. *Cogo* then continues normal operation.

4 Implementation

We implemented *Cogo* by extending the code base of Radmin [25]. Radmin offered several integrated kernel space and user space tracing capabilities and an extensible PFA engine, which allowed us to implement and benchmark *Cogo* in a unified way. Figure 2 illustrates the architecture of *Cogo* within Radmin. We extended Radmin's kernel tracer to support network I/O monitoring, and implemented *Cogo*'s learning and detection algorithms by extending Radmin's PFA engine which originally only supported a quadratic time PFA construction (q-PFA in the figure). We also extended the framework to support attaching to running processes and monitoring containerized processes.

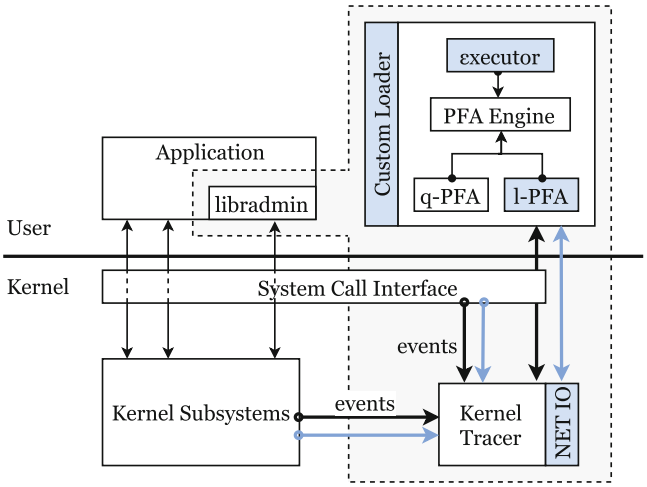


Fig. 2. *Cogo*'s architecture within Radmin. *Cogo* extends Radmin with a network I/O monitoring module, the linear PFA construction component, a non-deterministic PFA executor, and a custom loader to resolve namespace PIDs.

We extended Radmin's kernel tracer to support network I/O monitoring by attaching handlers to the relevant tracepoints [24] in the kernel. Kernel tracepoints are special points in the executable kernel memory that provide hooks to various events in the kernel. The hooks call functions (probes) that are provided at runtime by kernel modules. *Cogo* provided a handler for each tracepoint where it collected and reported the measurements to the rest of Radmin as needed. Each tracepoint executes in the context of the process that triggered the event. *Cogo* filters out process contexts using the global PIDs of the monitored processes.

Table 1. Kernel tracepoints hooked by *Cogo* for network I/O monitoring.

Kernel tracepoint	Description
`socket.create`	A socket is allocated
`socket.close`	A socket is closed and released
`socket.sendmsg`, `socket.writev`, `socket.aoi_write`, `socket.write_iter`	Data is being sent on a socket
`socket.recvmsg`, `socket.readev`, `socket.aoi_read`, `socket.read_iter`	Data is received on a socket

It supports monitoring a single process, all processes part of one program, or all processes in a process tree. Table 1 lists the relevant tracepoints that *Cogo* hooked to monitor network state.

5 Evaluation

We measured the detection accuracy, earliness, and overhead of *Cogo* on two large-scale server applications that are commonly targeted by application layer DoS attacks: Apache [2], the world's most used web server software; and OpenSIPS [9], the famous free VoIP server and proxy implementation of the session initiation protocol (SIP) [33]. The testbeds used Docker containers for isolation and CORE [15] for network emulation.

5.1 HTTP Attacks on Apache

Our Apache testbed is depicted in Fig. 3. It consisted of a server running Apache, one User Agent (UA) node for benign clients, and one Weaponized User Agent (W-UA) node for attackers. UA and W-UA consisted of Docker containers running HTTP clients. We generated benign traffic using an HTTP client model derived from Choi-Limb [22]. From the mean and standard deviation for various model parameters for the data set reported in [22], we used a nonlinear solver to calculate approximate distribution parameters for the distribution found to be a good fit in Choi-Limb. We represented each client using one instance of the HTTPerf [7] benchmark. For each client, we generated a workload session using a unique seed and the distilled distribution parameters. The session consisted of a series of requests with a variable think time between requests drawn from the client model. We generated the workload session in HTTPerf's workload session log format (wsesslog). Each client request contained as URL parameters a random request length padding and a requested response size drawn from the client model. The Apache server hosted a CGI-bin web application that simulated real deployments. For each client HTTP request, the server responded with content equal in byte length to the requested response size.

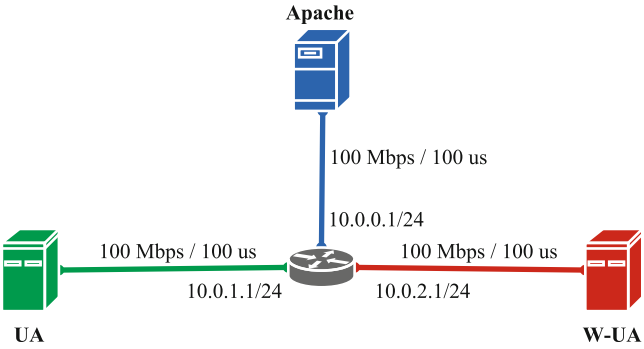


Fig. 3. HTTP DoS testbed used in our experiments, including the Apache server, a User Agent (UA) node where benign clients reside, and a Weaponized User Agent (W-UA) node where attacks originate from.

Attack traffic originated from the W-UA node. We used the HTTP application layer DoS benchmark SlowHTTPTest [4] which bundles several Slow-rate [11] attack variants. (Slow-rate attacks are low-bandwidth application layer DoS attacks that use legitimate albeit slow HTTP requests to take down web servers.) Two famous examples of Slow-rate attacks are Slowloris [12] and Slowread [5]. In Slowloris, attackers send the HTTP request headers as slowly as possible without hitting the connection timeout limit of the server. Its Slowread variant sends the headers at normal speeds but reads the response as slowly as possible. If enough of these slow requests are made in parallel, they can consume the entire server's application layer connections queue and the server becomes unable to serve legitimate users. Slow-rate attacks typically manifest in an abnormally large number of relatively idle or slow sockets.

We built *Cogo* model for Apache using 12 benign traffic runs, each of which consisted of one hour of benign traffic. We set the number of benign clients to 100. Note that each benign client is a whole workload session. For testing, we performed several experiments using blended benign and attack traffic by injecting attack requests at random points while serving a benign load. Testing is performed by running Apache under *Cogo* in detection mode, serving one hour worth of benign requests from 100 benign clients and 100 Slow-rate clients (attackers). The number of attackers represents the total concurrent SlowHTTPTest attack connections. We limited the attack duration to 15 min. We configured Apache to serve a maximum of 100 concurrent connections at any moment in time.

We performed each experiment with and without *Cogo*. We configured *Cogo* to kill the offending Apache worker process when an attack is detected.[7] Finally, we experimented with two types of attackers: non-aggressive attackers that seep

[7] More advanced remediation policies can be used, such as blocking offending source IPs, rate limiting, or protocol-specific recovery. We opted for process termination for simplicity as remediation is not the focus of *Cogo*.

in the server at a very slow rate, and aggressive attackers that bombard the server with as many concurrent connections as possible. For non-aggressive attackers, we set the SlowHTTPTest connection rate to one connection per second. For aggressive attackers, we set the connection rate to the server capacity, i.e., 100 connections per second.

Detection Results. Table 2 summarizes the results. It took *Cogo* only about 12 min to build a model for Apache from the benign measurements. This is about a 505× improvement over Radmin which took more than four days to construct a model from the same measurements. The savings in training time came at the expense of a slight increase in the model size (from 34 MB to 76 MB) which is acceptable and does not pose a bottleneck. The model is only loaded once at startup of *Cogo*; detection time is invariant of the model size as each measurement point results in exactly one transition in one of the PFAs.

Cogo achieved a very low false positive rate (FPR) at 0.0194% (about 91% better than Radmin). We believe the reason for this reduction in FPR is that *Cogo* retains longer low-probability paths in the PFA as the detection algorithm limits transition probabilities rather whole path probabilities as in Radmin. For the most part, false positives (FPs) were encountered during startup or shutdown of Apache which from experience has shown considerable variability.

Table 2. Summary of results for Apache. The number of requests was 473,558.

Item	Radmin	*Cogo*	Improvement
Training time (sec.)	379,661	**752**	▼ 505×
Model size (MB)	**34**	76	▲ 0.45×
FPs, FPR	1,116, 0.2357%	**92, 0.0194%**	▼ 12×
Downtime (sec; non-aggressive)	137	**0**	▼ ∞
Downtime (sec; aggressive)	58	**7**	▼ 8.3×

Figures 4 and 5 depict the availability of Apache against non-aggressive and aggressive attacks. *Cogo* successfully prevented Apache from going down against non-aggressive attacks. As the attack connections were idling at the server side, *Cogo* detected anomalous transmit and receive rates and terminated the attacked Apache workers. This occurred within seven seconds from connection establishment. Against the same attacks, Apache under Radmin remained down for longer than two minutes. For aggressive attacks, Apache protected with Radmin was down for one minute, compared to only seven seconds under *Cogo*.

5.2 VoIP Attacks on OpenSIPS

Next, we considered detection of resource attacks on VoIP servers as telephony systems have increasingly become targets of DDoS attacks evidenced during the

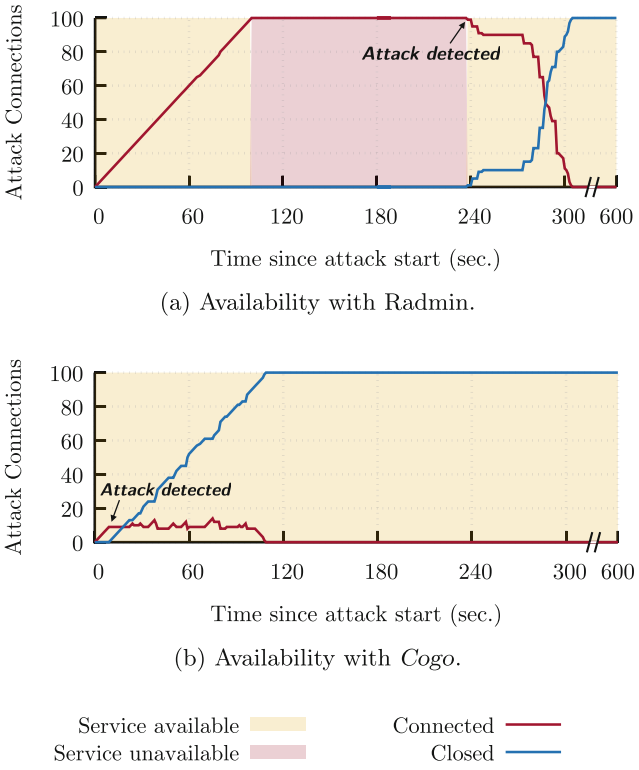


Fig. 4. Apache server availability against non-aggressive Slow-rate attacks. With Radmin, the server was down for more than two minutes. There was no downtime under *Cogo*.

2015 attack on the Ukrainian power grid [13]. To establish and manage calls, VoIP servers rely on Session Initiation Protocol (SIP) [33] which is known to be vulnerable to exhaustion and overload, even under benign conditions [29]. Overload can be caused by a variety of legitimate SIP behaviors such as response duplication, call forwarding, and call forking (conference calls) which result in large numbers of control packets that may congest servers. Similarly, excessive transactions cause system resource exhaustion in stateful servers when the number of requests exceeds the finite memory available to track each call state machine. An adversary who wishes to cause DoS can do so by initiating calls that exercise these legitimate but atypical resource intensive behaviors and thus degrade server performance — all while blending in with normal traffic (without malformed packets or specification violations) to circumvent defenses such as scrubbing or bandwidth limitation. In the following we evaluate *Cogo* against these protocol attacks on a representative SIP testbed based on OpenSIPS [9].

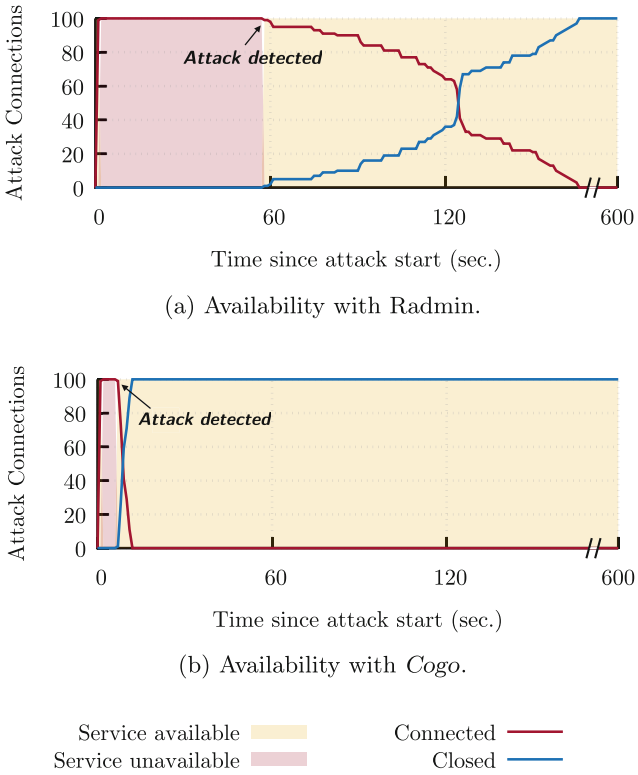


Fig. 5. Apache server availability against aggressive Slow-rate attacks. *Cogo* reduced the server down time by at least a factor of eight, down from 58 s to only seven seconds.

Testbed and Procedure. Our SIP DDoS testbed, shown in Figure 6, consisted of a SIP server and pairs of SIP user agents and weaponized agents that serviced simultaneous callers and attackers. The SIP server ran OpenSIPS 2.2 and was configured using the residential configuration generated by the OpenSIPS configuration tools. OpenSIPS used fixed-size shared and private memory across its child processes (32 MB and 16 MB respectively). To exacerbate memory exhaustion at the server, we adjusted the *wt_timer* of the OpenSIPS to 32 s (the recommended value in the RFC) which corresponds to the length of time a transaction is held in memory after it has completed. Though intended to help absorb delayed messages after the transaction completed, it also inadvertently reserves memory that could otherwise be made available to handle new calls. For the following experiments, we considered a small enterprise or large residential deployment, thus end-to-end delays from UA to server were minimal (ten ms) and link bandwidth was isolated to SIP traffic at 100 Mbps.

Pairs of UA nodes were used to represent benign SIP callers (UA-1) and callees (UA-2). These nodes ran instances of the SIP Proxy (SIPp) [10]: a SIP

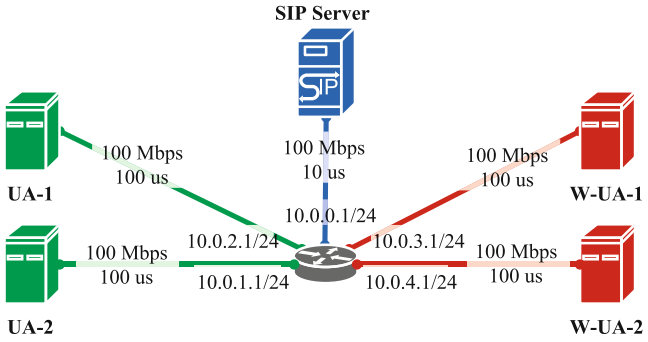


Fig. 6. SIP DDoS testbed used in our experiments. UA-1 and UA-2 are benign user agents. W-UA-1 and W-UA-2 are attack (weaponized) agents.

benchmarking tool to generate SIP caller/callee workloads. While we did not model the audio portion of the call, we leveraged the log-normal feature of SIPp to insert a random, lognormal distributed pause between call setup and hang up to simulate variability among call lengths. Our call length distribution was log-normal with a mean of 10.28 and variance of one ms equating to an average call length of 30 s. Each call consisted of an INVITE transaction followed by the variable pause, and then terminated with a BYE transaction. SIPp can initiate calls in parallel, allowing us to model many users from a single node.

Attacks were initiated from the W-UAs at caller W-UA-1 and callee W-UA-2. We staged attacks by repurposing SIPp as an attack tool, supplying it with scenario files that specify malicious caller/callee behaviors such as flooding requests or excessive duplication of responses. For example, a BYE flood attack equates to W-UA-1 initiating a number of spurious BYE transactions, each with a new call id to represent a new transaction. Because SIP does not associate a BYE with a prior INVITE, the BYE is accepted and transaction memory is wastefully reserved while the attack is in process. W-UA-2 colludes with W-UA-1 by purposefully not responding to the request, which adds to the time transaction memory is held at the server. Like the benign workload, we can tune the amplitude of SIPp to control the number of simultaneous attack calls.

The *Cogo* model for OpenSIPS was built from five benign observation collecting runs, totaling 8 h of benign measurements. During this observation run OpenSIPS was subjected to a benign load between the SIPp clients (UA-1, UA-2) and the SIP server. The clients initiated calls to the server. Call setup and call disconnect were specified using XML files input to SIPp and followed standard SIP call setup conventions for invite, ringing, bye, and appropriate response and status messages. Call hold used the SIPp log-normal distribution. The SIPp maximum calls per second rate was set to 10 and call limit to 200. This combination of SIPp settings produced a steady call rate of $\sim$7 calls every second. Several additional benign observation runs were made during which OpenSIPS was started and then terminated to ensure the observations captured startup

Table 3. Summary of OpenSIPS results. The number of benign call requests was 6,342.

Item	Radmin	*Cogo*	Improvement
Training time (sec.)	43,493	**258**	▼ 169×
Model size (MB)	**41**	55	▲ 0.75×
FPs, FPR	9, 0.1419%	**4, 0.0631%**	▼ 2.25×
Bye flood detection delay (sec.)	∞	**6**	▼ ∞
Invite flood detection delay (sec.)	∞	**4**	▼ ∞

and shutdown which from experience has shown considerable variability. The total size of the observation data was 515 MB. Processing these observations resulted in a model of 55 MB. Several test runs were made using the model and with it *Cogo* exhibited virtually zero false positives under a load with the same characteristics as that used for the observation runs.

Detection Results. Table 3 summarizes the results. *Cogo* reduced training time from about 12 h to only 4 min (greater than 169× reduction). The model size increased by a factor of only 0.75×, from 41 MB to 55 MB. In terms of accuracy, *Cogo* only had 4 FPs throughout the experiment. The four FPs all occurred at startup time of OpenSIPS. Radmin triggered 9 FPs also at startup time. The impact of BYE and INVITE floods on OpenSIPS, and the detection behavior of *Cogo* is shown in Fig. 7. The attacks were not detectable by Radmin since OpenSIPS uses a fixed size memory pool, therefore preventing memory exhaustion by the attack calls. In other words, without monitoring network I/O, it is impossible to early detect BYE and INVITE floods. *Cogo*, on the other hand, detected the attacks almost immediately, within less than six seconds after the attacks onset. Note that we did not implement any remediation policy for OpenSIPS; proper remediation requires a protocol-specific solution that times out or hangs up the attack calls.

5.3 Performance Overhead

Cogo effectively had a negligible overhead. We measured the throughput of Apache and OpenSIPS on the benign workloads with and without *Cogo*. Apache maintained a steady rate of 130 requests per second. We benchmarked Apache with HTTPerf and experienced a very marginal 0.2 ± 0.3 ms response time increase per request. The average response time increased from 10.3 ms to 10.5 ms. For OpenSIPS, it maintained a steady call rate of 200 calls per second. We experimented with call rates from 300 to 1000 calls per second and did not observe any degradation in throughput.

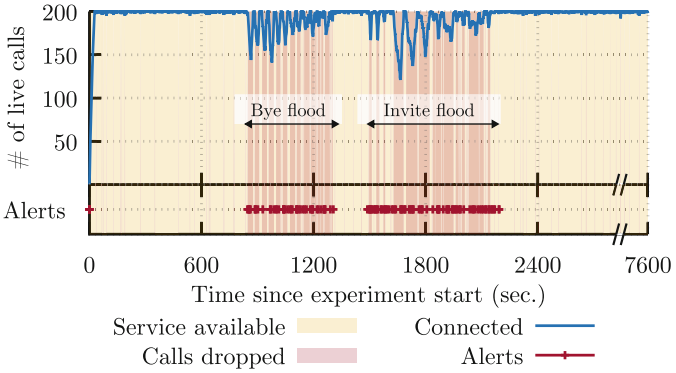


Fig. 7. *Cogo* detection of bye and invite floods against OpenSIPS.

6 Related Work

Modern operating systems have a number of threshold-based facilities to limit resource consumption (`ulimit, AppArmor` [3]). However, these limits are static upper bounds and disregard different consumption of different program segments for different inputs. This enables attackers to maximize the DoS time by crafting inputs that trigger prolonged resource consumption or starvation, such as Slow-rate attacks [21,23,30]. Several static and dynamic instrumentation tools exist for profiling, such as Valgrind [32] and Intel Pin [31]. However, the instrumentation overhead is often too high to enable their continuous usage, especially when detection of exhaustion is the goal [34,35]. Apostolico [18] presented a theoretic study for linear prediction using a Generalized Suffix Trees (GST). However, to the best of our knowledge, there is no implementation or a quantitative study of [18]. Our approach, builds a simpler model using a PFA construction that provided tight detection time and space guarantees instead of a GST.

In [14,30] there is a survey of different approaches for anomalous traffic detection which is not connected directly or indirectly to resource consumption at the application layer. Antunes et al. [17] proposed a system for testing servers for exhaustion vulnerabilities using fuzzed test cases from user-supplied specs of the server protocol. Groza et al. [27] formalized DoS attacks using a protocol-specific cost rules. Aiello et al. [16] formalized DoS resilience rules that protocols should meet but they are not feasible requirements in practice [37]. Chang et al. [20] proposed a static analysis system for identifying source code sites that may result in uncontrolled CPU time and stack consumption. The system employed taint analysis and control-dependency analysis to identify source code sites that can be influenced by untrusted input. Several similar approaches that required manual code annotation were also developed [28,38]. Closely related, Burnim et al. [19] used symbolic execution to generate inputs that exhibit worst case complexity.

Cogo substantially differs from those systems in that it does not require access to the source code or any side information and it covers network resources

used by an application, not only CPU and memory. Elsabagh et al. [25] proposed Radmin, a system for early detection of application layer DoS attacks. This is the system we used as a starting point for *Cogo*. The system showed good accuracy and low overhead. However, it did not monitor network I/O, had a prohibitive quadratic training time, and could not monitor containerized processes or attach to a running process.

7 Conclusions

This paper presented *Cogo*, a practical and accurate system for early detection of DoS attacks at the application layer. Unlike prior solutions, *Cogo* builds a PFA model from the temporal and spatial resource usage information in linear time. *Cogo* monitors network state, supports containerized processes monitoring and attaching to running processes. *Cogo* detected real-world attacks on Apache and OpenSIPS, both are large-scale servers. It achieved high accuracy, early detection, and incurred negligible overhead. *Cogo* required less than 12 min of training time, incurred less than 0.0194% false positives rate, detected a wide range of attacks less than seven seconds into the attacks, and had a negligible response time overhead of only 0.2 ± 0.3 ms. *Cogo* is both scalable and accurate, suitable for large-scale deployment.

Acknowledgments. We thank the anonymous reviewers for their insightful comments and suggestions. This material is based upon work supported in part by the National Science Foundation (NSF) SaTC award 1421747, the National Institute of Standards and Technology (NIST) award 60NANB16D285, and the Defense Advanced Research Projects Agency (DARPA) contract no. HR0011-16-C-0061 in conjunction with Vencore Labs. Opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, NIST, DARPA or the US Government.

References

1. myths of ddos attacks. http://blog.radware.com/security/2012/02/4-massive-myths-of-ddos/
2. The apache http server project. https://httpd.apache.org/
3. Apparmor. http://wiki.apparmor.net/index.php/Main_Page
4. Application layer DoS attack simulator. http://github.com/shekyan/slowhttptest
5. Are you ready for slow reading? https://blog.qualys.com/securitylabs/2012/01/05/slow-read
6. Availability overrides security. http://hrfuture.net/performance-and-productivity/availability-over-rides-cloud-security-concerns.php?Itemid=169
7. Httperf - http performance measurement tool. http://linux.die.net/man/1/httperf
8. Mobile users favor productivity over security. http://www.infoworld.com/article/2686762/security/mobile-users-favor-productivity-over-security-as-they-should.html
9. OpenSIPS: the new breed of communication engine. https://www.opensips.org/

10. Sipp: traffic generator proxy for the sip protocol. http://sipp.sourceforge.net/
11. Slow-Rate Attack. https://security.radware.com/ddos-knowledge-center/ddospedia/slow-rate-attack/
12. Slowloris - apache server vulnerabilities. https://security.radware.com/ddos-knowledge-center/ddospedia/slowloris/
13. When the lights went out: Ukraine cybersecurity threat briefing. https://www.boozallen.com/insights/2016/09/ukraine-cybersecurity-threat-briefing/
14. Denial of service attacks: A comprehensive guide to trends, techniques, and technologies. ADC Monthly Web Attacks Analysis 12 (2012)
15. Ahrenholz, J.: Comparison of core network emulation platforms. In: Military Communications Conference (2010)
16. Aiello, W., Bellovin, S.M., Blaze, M., Ioannidis, J., Reingold, O., Canetti, R., Keromytis, A.D.: Efficient, DoS-resistant, secure key exchange for internet protocols. In: 9th ACM Conference on Computer and Communications Security (2002)
17. Antunes, J., Neves, N.F., Veríssimo, P.J.: Detection and prediction of resource-exhaustion vulnerabilities. In: International Symposium on Software Reliability Engineering (2008)
18. Apostolico, A., Bejerano, G.: Optimal amnesic probabilistic automata. J. Comput. Biol. **7**(3–4), 381–393 (2000)
19. Burnim, J., Juvekar, S., Sen, K.: Wise: automated test generation for worst-case complexity. In: 31st International Conference on Software Engineering (2009)
20. Chang, R.M., et al.: Inputs of coma: static detection of denial-of-service vulnerabilities. In: 22nd Computer Security Foundations Symposium (2009)
21. Chee, W.O., Brennan, T.: Layer-7 ddos. (2010). https://www.owasp.org/images/4/43/Layer_7_DDOS.pdf
22. Choi, H.K., Limb, J.O.: A behavioral model of web traffic. In: 7th International Conference on Network Protocols (1999)
23. Crosby, S., Wallach, D.: Algorithmic dos. In: van Tilborg, H.C.A., Jajodia, S. (eds.) Encyclopedia of Cryptography and Security, pp. 32–33. Springer, USA (2011)
24. Desnoyers, M.: Using the linux kernel tracepoints. https://www.kernel.org/doc/Documentation/trace/tracepoints.txt
25. Elsabagh, M., Barbará, D., Fleck, D., Stavrou, A.: Radmin: early detection of application-level resource exhaustion and starvation attacks. In: 18th International Conference on Research in Attacks, Intrusions and Defenses (2015)
26. Gray, R.M., Neuhoff, D.L.: Quantization. IEEE Trans. Inform. Theory **44**(6), 2325–2383 (1998)
27. Groza, B., Minea, M.: Formal modelling and automatic detection of resource exhaustion attacks. In: Symposium on Information, Computer and Communications Security (2011)
28. Gulavani, B.S., Gulwani, S.: A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In: Computer Aided Verification, pp. 370–384 (2008)
29. Hilt, V., Eric, N., Charles, S., Ahmed, A.: Design considerations for session initiation protocol (SIP) overload control (2011). https://tools.ietf.org/html/rfc6357
30. Kostadinov, D.: Layer-7 ddos attacks: detection and mitig. InfoSec Institute (2013)
31. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 190–200. ACM, New York (2005). http://doi.acm.org/10.1145/1065010.1065034

32. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM Sigplan Notices, vol. 42, pp. 89–100. ACM (2007)
33. Rosenberg, J., et al.: SIP: Session initiation protocol (2002). https://www.ietf.org/rfc/rfc3261.txt
34. Ruiz-Alvarez, A., Hazelwood, K.: Evaluating the impact of dynamic binary translation systems on hardware cache performance. In: International Symposium on Workload Characterization (2008)
35. Uh, G.R., Cohn, R., Yadavalli, B., Peri, R., Ayyagari, R.: Analyzing dynamic binary instrumentation overhead. In: Workshop on Binary Instrumentation and Application (2007)
36. Ukkonen, E.: Online construction of suffix trees. Algorithmica **14**(3), 249–260 (1995)
37. Zargar, S.T., Joshi, J., Tipper, D.: A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks. IEEE Commun. Surv. Tutorials **15**(4), 2046–2069 (2013)
38. Zheng, L., Myers, A.C.: End-to-end availability policies and noninterference. In: 18th IEEE Workshop Computer Security Foundations (2005)