# Radmin: Early Detection of Application-level Resource Exhaustion and Starvation Attacks

Mohamed Elsabagh✉, Daniel Barbará, Dan Fleck, and Angelos Stavrou

George Mason University
{melsabag, dbarbara, dfleck, astavrou}@gmu.edu

**Abstract.** Software systems are often engineered and tested for functionality under normal rather than worst-case conditions. This makes the systems vulnerable to denial of service attacks, where attackers engineer conditions that result in overconsumption of resources or starvation and stalling of execution. While the security community is well familiar with volumetric resource exhaustion attacks at the network and transport layers, application-specific attacks pose a challenging threat. In this paper, we present Radmin, a novel system for early detection of application-level resource exhaustion and starvation attacks. Radmin works directly on compiled binaries. It learns and executes multiple probabilistic finite automata from benign runs of target programs. Radmin confines the resource usage of target programs to the learned automata, and detects resource usage anomalies at their early stages. We demonstrate the effectiveness of Radmin by testing it over a variety of resource exhaustion and starvation weaknesses on commodity off-the-shelf software.

**Keywords:** Resource Exhaustion, Starvation, Early Detection, Probabilistic Finite Automata.

## 1 Introduction

Availability of services plays a major – if not the greatest – role in the survivability and success of businesses. Recent surveys [2, 5] have shown that IT managers and customers alike tend to prefer systems that are more often in an operable state, than systems that may offer higher levels of security at the expense of more failures. This means that any disruption to the availability of a service is directly translated into loss of productivity and profit. Businesses invest in deploying redundant hardware and replicas to increase the availability of the services they offer. However, as software designers often overlook Saltzer-Schroeder's "conservative design" principle [32], systems are often engineered and tested for functionality under normal rather than worst-case conditions. As a result, worst-case scenarios are often engineered by the attackers to over-consume needed resources (resource exhaustion), or to starve target processes of resources (resource starvation), effectively resulting in partial or complete denial of service (DoS) to legitimate users.

A system is exposed to resource exhaustion and starvation if it fails to properly restrict the amount of resources used or influenced by an actor [3]. This includes, but is not limited to, infrastructure resources, such as bandwidth and connection pools, and computational resources such as memory and cpu time. The attacks can operate at the network and transport layers [37], or at the application layer such as algorithmic and starvation attacks [17, 18]. The asymmetric nature of communication protocols, design and coding mistakes, and inherently expensive tasks all contribute to the susceptibility of programs to resource exhaustion and starvation attacks. Attacks targeting the network and transport layers have attracted considerable research attention [21, 23, 31]. Meanwhile, attacks have become more sophisticated and attackers have moved to higher layers of the protocol stack. Since 2010, resource exhaustion attacks that target the application layer have become more prevalent [1, 17] than attacks at the network layer and transport layer.

In this paper, we present Radmin, a system for automatic *early* detection of application-level resource exhaustion and starvation attacks. By application-level attacks we refer to the classes of DoS attacks that utilize small, specially crafted, malicious inputs that cause uncontrolled resource consumption in victim applications. To this end, Radmin traces the resource consumption of a target program in both the user and kernel spaces (see Section 3), builds and executes multiple state machines that model the consumption of the target program.

The key observation is that attacks result in *abnormal* sequences of transitions between the different resource consumption levels of a program, when compared to normal conditions. By modeling the resource consumption levels as multiple realizations of a random variable, one can estimate a conditional distribution of the current consumption level given the history (context) of measurements. Consequently, the statistical properties of the resulting stochastic process can be used to detect anomalous sequences.[1]

Radmin operates in two phases: offline and online. In the offline phase, the monitored programs are executed on benign inputs, and Radmin builds multiple Probabilistic Finite Automata (PFA) models that capture the temporal and spatial information in the measurements. The PFA model is a finite state machine model with a probabilistic transition function (see Section 4). Both the time of holding a resource, and the amount used of that resource are mapped to states in the PFA, while changes in the states over the time are mapped to transitions.

In the online phase, Radmin executes the PFAs as shadow resource consumption state machines, where it uses the transition probabilities from the PFAs to detect anomalous consumption. Additionally, Radmin uses a heartbeat signal to *time out* transitions of the PFAs. Together with the transition probabilities, this enables Radmin to detect both exhaustion and starvation attacks.

Radmin aims at detecting attacks as early as possible, i.e., *before* resources are wasted either due to exhaustion or starvation. Radmin does *not* use any static resource consumption thresholds. Instead, the PFAs capture the transitions between the different consumption levels of different program states, and statistics of the PFAs are used to detect anomalies. The PFAs allow Radmin to *implicitly* map different program states, i.e., program behavior at some execu-

[1] Unless stated otherwise, we use "measurements" and "sequences" interchangeably in the rest of this paper.

tion point given some input, to *dynamic* upper and lower resource consumption bounds.

We quantified the earliness of detection as the ratio of resources that Radmin can save, to the maximum amounts of resources that were consumed in benign conditions (see Section 5). This corresponds to the tightest *static* threshold that traditional defenses can set, without causing false alarms. Radmin has an advantage over all existing defenses that use static thresholds (see Section 7), since exhaustion and starvation attacks can evade those defenses. Exhaustion attacks can consume the highest amounts of resources possible, just below the static threshold [1, 17]. Additionally, starvation attacks, by design, do not aim at directly consuming resources such as attacks that trigger deadlocks or livelocks [17].

To summarize, this study makes the following contributions:

– **Radmin**. A novel system that can detect both resource exhaustion and starvation attacks in their early stages. Radmin employs a novel detection algorithm that uses PFAs and a heartbeat signal to detect both exhaustion and starvation attacks. Radmin takes both temporal and spatial resource consumption information into account and adds minimal overhead.
– **Working Prototype**[2]. We implement a prototype that uses kernel event tracing and user space instrumentation to efficiently and accurately monitor resource consumption of target processes.
– **Evaluation**. We demonstrate the effectiveness of Radmin using a wide range of synthetic attacks against a number of common Linux programs. We show that Radmin can efficiently detect both types of anomalies, in their early stages, with low overhead and high accuracy.

The rest of the paper is organized as follows. Section 2 discusses the assumptions and threat model. Section 3 presents the technical details of Radmin and its implementation. Section 4 describes the models used in Radmin and the detection algorithm. Section 5 evaluates Radmin. Section 6 provides a discussion of different aspects of Radmin and possible improvements. We discuss related work in Section 7, and conclude in Section 8.

## 2 Assumptions and Threat Model

Radmin's main goal is early detection of application-level resource exhaustion and starvation, which may result in full or partial depletion of available resources (CPU time, memory, file descriptors, threads and processes) or in starvation and stalling. We assume that actors can be local or remote, with no privilege to overwrite system binaries or modify the kernel.

We consider the following types of exhaustion and starvation attacks. First, attacks that result in a sudden surprisingly high or low consumption of resources (e.g., an attacker controlled value that is passed to a `malloc` call). Second, attacks that result in atypical resource consumption sequences such as algorithmic and

[2] Source code available under GPLv3 at: https://github.com/melsabagh/radmin

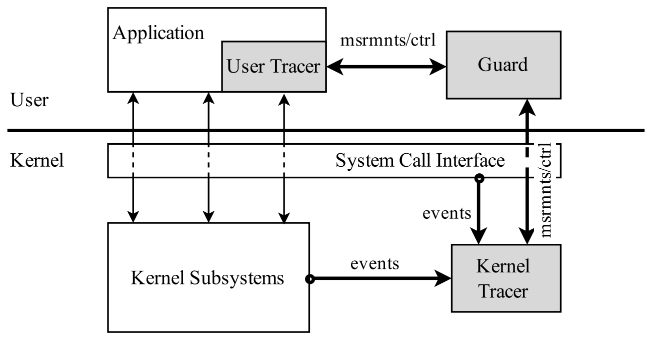

**Fig. 1.** Architecture of Radmin. The User Tracer, and Kernel Tracer, monitor and collect measurements from a target program by binding checkpoints to events of interest in both the user and kernel spaces. They send the measurements to the Guard, where the bulk of processing takes places.

protocol-specific attacks that aim at maximizing (flattening) the amounts of consumed resources. Third, attacks that result in stalling of execution, including triggering livelocks or prolonged locking of resources.

Although, in our experiments, we considered only programs running on x86 Linux systems and following the Executable and Linkable Format (ELF), the proposed approach places no restrictions on the microarchitecture, the binary format, or the runtime environment.

## 3 System Architecture

The major components of Radmin are a kernel space tracing module (Kernel Tracer), a user space tracing library (User Tracer), and a daemon process (Guard) where the bulk of processing takes place. The tracing modules monitor and control a target program by binding checkpoints to events of interest, in the execution context of the target. Checkpoints are functions in the tracing modules that are called when an event of interest is triggered. Each checkpoint communicates measurements and control information to the Guard. We refer to a code site at which an event was triggered as a checkpoint site. Figure 1 shows the system architecture of Radmin.

Radmin takes a target program binary as input, and operates in two phases: offline and online. In the offline phase, Radmin instruments the target binary by injecting calls to the User Tracer into the binary, and writes the instrumented binary to disk. The instrumented program is then executed over benign inputs, while Radmin monitors its execution in both the user and kernel spaces, using the User Tracer and the Kernel Tracer modules, respectively. During that stage, the Guard receives the measurements from the tracers and learns multiple PFAs that capture the resource consumption behavior of the target program. Finally, in the online phase, the Guard executes the PFAs along with the target program, and raises an alarm if a deviation of the normal behavior is detected (see Section 4).

**Table 1.** Checkpoint sites monitored by the tracing modules. Checkpoint sites used by the Kernel Tracer are given in the SystemTap probes notation.

| Checkpoint Site | User/Kernel | Resource Type |
|---|---|---|
| `vm.brk`, `vm.mmap`, `vm.munmap` | Kernel | Memory |
| `kernel.do_sys_open`, `syscall.close` | Kernel | File descriptors |
| Recursive sites<br>Sites that manipulate the stack pointer | User | Stack |
| `scheduler.ctxswitch`, `perf.sw.cpu_clock`<br>Heartbeat every 500 ms. | Kernel<br>Both | CPU |
| `scheduler.wakeup_new`, `kprocess.exec_complete`<br>`kprocess.exit` | Kernel | Child tasks |

Each measurement is a vector of $\langle$*consumed kernel time, consumed user time, consumed resource amount*$\rangle$ associated with a resource type and a task[3] ID. Here, "consumed resource amount" accounts for the total amount of a resource that would be in consumption if the allocation or deallocation request is granted. We tracked parent-child task relationships by recording both the parent and current task IDs, in addition to the process ID. The measurement vectors accurately capture the resource consumption behavior of a process, as they map out both the sequences of resource consumption changes and the time for each change, which effectively captures both the temporal and spacial information in the resource consumption behavior of the process.

We developed the user space components in C/C++, using the Dyninst [4] library for static binary rewriting. The kernel tracer was developed using SystemTap [8]. A number of coordination scripts and a command line interface were also developed in Shell Script.

A summary of the checkpoint sites and the associated resource types is shown in Table 1, which we discuss in the following sections.

## 3.1 Kernel Tracer

The Kernel Tracer binds checkpoints to various kernel events by binding probes to the corresponding kernel tracepoints. Kernel tracepoints provide hooks to various points in the kernel code by calling functions (probes) that are provided at runtime by kernel modules [20]. Binding to the centralized, well-defined, kernel tracepoints associated with resource (de)allocation is more robust than attempting to enumerate and trace, from user space, *all* possible ways a program can (de)allocate resources through library calls. Additionally, kernel tracing gives maximum visibility into the target process, allows for low-penalty monitoring and control of the target.

[3] Unless stated otherwise, we use "task" to indistinguishably refer to child processes and threads spawned by a monitored program.

The Kernel Tracer keeps track of task creation by binding to the kernel scheduler wakeup tracepoint (`scheduler.wakeup_new`), which is triggered when a task is being scheduled for the first time. It monitors task destruction by binding to the task exit tracepoint (`kprocess.exit`). The tracer also monitors processes overlaid by the `exec` call family by binding to the exec completion tracepoint (`kprocess.exec_complete`).

For memory monitoring, the Kernel Tracer install probes for the tracepoints that are triggered upon the allocation of contiguous memory (`vm.brk`), memory regions (`vm.mmap`), and the release of memory to the kernel (`vm.munmap`). For file monitoring, probes are installed for tracepoints that are triggered when file descriptors are allocated (`kernel.do_sys_open`) or released (`syscall.close`). For CPU monitoring, the Kernel Tracer keeps track of the consumed clock ticks by binding to the scheduler tracepoints that trigger when monitored tasks context switch (`scheduler.ctxswitch`), and when the kernel clock ticks (`perf.sw.cpu_clock`) inside the context of a monitored task. The reason for monitoring only those two events is to minimize the overhead of profiling the CPU time.

It is important to note that even though memory is monitored from the kernel module, user space processes can exhaust their stack space without interfacing with the kernel. Therefore, we decided to include additional checkpoints for monitoring the stack in user space.

## 3.2 User Tracer

The User Tracer consists of a user space library, where calls to that library are injected in the target binary at assembly sites of interest. The User Tracer is injected as follows. First, Radmin statically parses the input binary and extracts a Control Flow Graph (CFG) using the Dyninst ParseAPI library. It then analyzes the CFG to identify assembly sites that dynamically operate on the stack such as recursive calls (direct and indirect) and variable length arrays. Radmin injects calls to the tracer library at the marked sites in the binary, and saves the modified binary to disk.

To calculate the stack size consumed by recursive call sites, we first experimented with two options: a) parse the process memory maps from `/proc/pid/smaps`, and b) unwinding the stack. Both options proved unreliable. The obtained values from `smaps` were too coarse to reflect actual stack consumption. Unwinding the stack was very expensive, and required special arrangements at compilation time, such as the usage of frame pointers, that were not feasible to attain since we are directly working with compiled programs. Instead, Radmin implements a workaround by tagging (marking) the stack inside the *caller* function site, at a point directly before the recursive call, then calculating the distance from the entry point of the recursive *callee* function site to the tag. The tag is injected *only* in non-recursive caller function sites, which avoids mistakenly overwriting the tag due to indirect recursion.

Additionally, the User Tracer spawns a *heartbeat* thread that periodically consumes 1 clock tick then switches out. Consequently, the heartbeat tick is captured by the Kernel Tracer whenever the heartbeat thread is scheduled out. It delivers a clock signal from the monitored process to the Guard, which we use

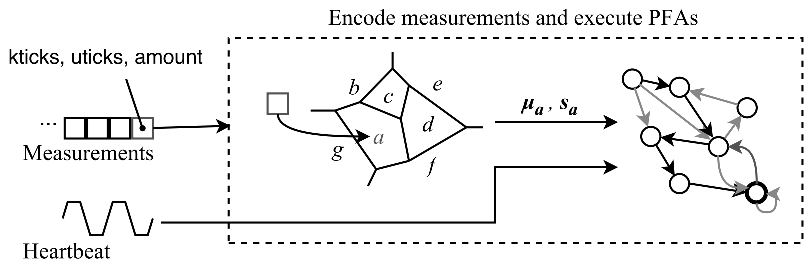

**Fig. 2.** Overview of Radmin Guard.

to detect starvation attacks by testing if the transitions between the PFA states have timed out (see Section 4).

### 3.3 Radmin Guard

Figure 2 shows the underlying architecture of the Guard. In the offline phase, the Guard learns a codebook over a finite alphabet $\Sigma$, and encodes the incoming measurements over $\Sigma$. Encoding the measurements serves two purposes: (1) it discretizes the continuous measurements, making them useful for estimating the conditional probabilities using the PFAs; and (2) it reduces the dimensionality (lossy compression) of the measurements by mapping them to a finite alphabet of a much smaller size. The Guard then builds multiple PFAs over the encoded sequences, one for each monitored resource type. In the online phase, the Guard encodes the incoming stream of measurements, executes the PFAs (per task, per resource type), runs the detection algorithm, and raises an alarm if an anomaly is detected. In our experiments, we only terminated the violating process. However, more advanced recovery can be used such as resource throttling or execution rollback [35].

In the following section, we discuss in more depth how the Guard encodes the measurements, learns and executes the PFAs, and detects attacks.

## 4 Learning and Detection

### 4.1 Encoding

Radmin learns each codebook, used by the encoder, by running a $k$-means quantizer over the raw vectors of measurements, where $k = |\Sigma|$ is the number of desired codewords. In our implementation, we used $k$-means++ [10, 13], which is guaranteed to find a codebook (clusters) that is $O(\lg k)$-competitive with the optimal $k$-means solution [13]. To build the codebook, each measurement (consumed kernel and user time, and resource value) is treated as a point in a three-dimensional space. $k$-means++ starts by selecting one center point at random, from among all measurement points. Then, the distance $d(x)$ between each measurement point $x$ and the *nearest* center point is computed. Next, one more center point is chosen with probability proportional to $d^2(x)$. This seeding process repeats until $k$ centers are chosen. After which, standard $k$-means clustering is performed resulting in $k$ point clusters, the centers of which are the codewords. We refer the interested reader to [13] for a detailed discussion of $k$-means++.

Each codebook $\Sigma$ (one codebook per resource type) stores an indexed list of codewords. Each codeword $\sigma$ is represented by three-dimensional centers $\mu_\sigma$ and spreads $s_\sigma$, where each dimension corresponds to one dimension of the raw measurement vector. The number $|\Sigma|$ of codewords is determined such that each dimension gets at least 1 degree of freedom (level), constrained by a total of 64 degrees of freedom per codeword, i.e., $|\Sigma| \in [3 \dots 64]$. This setup allows at most 4 degrees of freedom per dimension ($4^3$ total), in case that *all* dimensions have the same amount of variance. Finally, encoding is done by mapping a given measurement vector to the *index* of its nearest codeword. If a measurement vector falls outside the coverage of all codewords, an empty codeword $\emptyset$ is returned.

## 4.2 Learning the PFAs

Radmin builds multiple PFAs for each resource type, and uses them to predict the probability of new sequences of measurements given the history of measurements. A PFA is a 5-tuple $(\Sigma, Q, \pi, \tau, \gamma)$, where:

- $\Sigma$ is a finite alphabet (the codebook) of symbols processed by the PFA.
- $Q$ is a finite set of PFA states.
- $\pi\colon Q \to [0,1]$ is the probability distribution vector over the start states.
- $\tau\colon Q \times \Sigma \to Q$ is the state transition function.
- $\gamma\colon Q \times \Sigma \to [0,1]$ is the *emitted* probability function (predictive distribution) when making a transition.

The subclass of PFA used in Radmin is constructed from their equivalent Probabilistic Suffix Tree (PST) model [30], which is a *bounded* variable-order Markov model where the history length *varies* based on the context (statistical information) of the subsequences of measurements, and the tree does not grow beyond a given depth $L$. In other words, the PST captures all statistically significant *paths* between resource consumption levels (encoded measurements), where the path length is at most $L$. In the construction of the PST, a subsequence of encoded measurements $s \in \Sigma^*$ is added to the PST *only if*:

1. $s$ has a significant prediction probability, i.e., there is some symbol $\sigma \in \Sigma$ such that $P(\sigma|s) \geq \gamma_{min}$, where $\gamma_{min}$ is the *minimum prediction probability* of the model.
2. And, $s$ makes a contribution, i.e., the prediction probability is significantly *different* from the probability of observing $\sigma$ after the parent node of $s$, i.e., $\frac{P(\sigma|s)}{P(\sigma|\texttt{Parent}(s))} \geq r_{min}$ or $\leq \frac{1}{r_{min}}$, where $r_{min}$ is the minimum difference ratio.

The PFA model provides tight time and space guarantees since it has a bounded order, and *only* the current state and the transition symbol determine the next state. Those are desirable properties for Radmin since (1) we construct

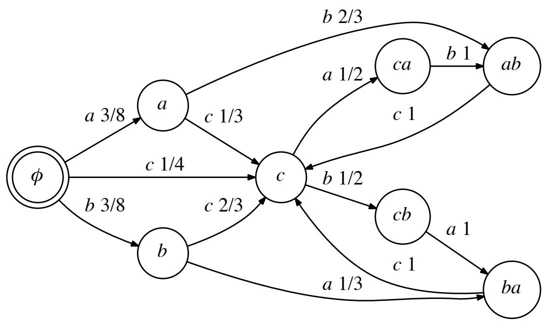

**Fig. 3.** Example of a PFA over the alphabet $\Sigma = \{a, b, c\}$. $\phi$ is the start state. Every edge is labeled by the transition symbol and transition probability. Transition symbols correspond to encoded measurements. Note, transition probabilities were rounded, and low probability transitions were removed for simplicity.

the PFAs without prior knowledge of the dependencies order (the length of statistical history in the measurements produced by target programs); and (2) we want to minimize the execution overhead of Radmin by maintaining a minimal amount of state-keeping information for the PFAs, and calculating the prediction probability for each measurement as fast as possible. For a sequence of $n$ measurements, the PFA model allows us to compute the prediction probability in $O(n)$ time and $O(1)$ space. Due to space constraints, we refer the reader to [14, 19, 30] for detailed discussions of various construction algorithms.

In the subclass of PFA used in Radmin, each state $q \in Q$ has a unique ID corresponding to the subsequence captured by that state, and the PFA has a *single* start state $q^\circ$, where $\pi(q^\circ) = 1$. Given a PFA $M$ and a string of encoded measurements $s = s_1 \dots s_l$, we walk $M$ (for each $s_i \in s$) where each transition $q^{i+1} = \tau(q^i, s_i)$ *emits* the transition probability $\gamma(q^i, s_i)$. The *prediction probability* of $s$ by $M$ is given by:

$$P(s) = \prod_{i=1}^{l} \gamma\left(q^{i-1}, s_i\right) . \tag{1}$$

For example, given the PFA in Fig. 3, the prediction probability of the sequence of encoded measurements "abca" is given by:

$$\begin{aligned} P(abca) &= \gamma(\phi, a) \times \gamma(a, b) \times \gamma(ab, c) \times \gamma(c, a) \\ &= 3/8 \times 2/3 \times 1 \times 1/2 \\ &= 1/8. \end{aligned}$$

Learning the PFAs, for a target program, requires running the target program over benign inputs. The following are some possible ways to handle this:

- **Dry runs and collected benign traffic**. Radmin can be trained through dry runs over benign inputs. This is typical in internal acceptance and pre-release testing. Radmin can also be trained using traffic that has already

been processed by applications and shown to be benign. This is arguably the easiest approach to train Radmin if it is deployed to protect a web-server.
– **Functionality tests**. Radmin can be trained using positive functionality tests. Testing is integral to the software development lifecycle, and Radmin can integrate with the test harness at development time. The main disadvantage is the additional effort needed for integration and debugging.
– **Endusers**. Radmin can be trained by endusers. Even though this causes an increased risk of learning bad behavior, the resulting PFAs can be compared or averaged based on the type and privileges given to each class of users. The PFAs can be averaged, for example, based on the distance between their transition functions. Additionally, the learning algorithm can be modified such that the PFA learns new behavior if the new behavior is statistically similar to old behavior, by using statistics over the frequency of minimum probability transitions.

Once trained, Radmin can continue learning or be locked down, based on the system policy. For example, system administrators may desire to limit guest users to what Radmin already knows, while PFAs for `sudoers` can still adjust and add to what they learned. The PFAs can also be locked after some time of no change, which can be an effective strategy for preventing future attacks from compromised users.

## 4.3 Anomaly Detection

In the online phase, the Guard operates by encoding the received sequences over $\Sigma$, and executing the corresponding PFAs as shadow automata, where each sequence results in a transition in one or more PFA. In addition to the measurements, the Guard uses the received heartbeat signal to *timeout* the transitions of the PFAs.

Algorithm 1 outlines the detection algorithm. Radmin raises an alarm if *any* of the following conditions is satisfied:

1. A foreign symbol is detected (lines 2–4). In this case, the program is requesting some resource amount that is *not* within the spread of *any* of the codewords in the codebook. This typically indicates an overshoot or undershoot signal. A very common example is DoS attacks that use data poisoning to pass a huge value to a `malloc` call, resulting in immediate crashing.
2. The program is requesting a transition that has a very low probability (lines 5–7). This case captures scenarios where attackers craft input that consumes (or locks) resources at program states that differ from benign runs. A common example is attacks that aim at maximizing the amounts of resources consumed by the program.
3. One or more PFAs time out (lines 8–16). In this case, the program has *not* transitioned to any of the next states within an acceptable time, with respect to one or more resource types. This, for example, could indicate the presence of a livelock.

The algorithm takes $O(|\Sigma|)$ time in the worst case, since the number of outgoing edges from any state is at most $|\Sigma|$.

**Algorithm 1:** AcceptMeasurement

```
input  : Measurement vector $\mathbf{v}$, heartbeat signal $t$,
         PFA $M$, Current state $q_i \in M$, Current path probability $p$
output: Accept or Reject
1  $c \leftarrow$ Encode($\mathbf{v}$);
2  if $c = \emptyset$ then
3      Reject                                          ▷ Foreign value
4  end
5  if $p \cdot \gamma(q_i, c) < \gamma_{min}(M)$ then
6      Reject                     ▷ Low probability transition or path
7  end
8  $timedout \leftarrow 1$;
9  foreach outgoing edge $e_i$ from $q_i$ do
10     if $\neg$ Timedout($e_i$, $t$) then
11         $timedout \leftarrow timedout \wedge 0$;
12     end
13 end
14 if $timedout = 1$ then
15     Reject                            ▷ All transitions timed out
16 end
17 Accept                                   ▷ take the transition
```

– $\gamma_{min}(M)$ is the minimum prediction probability of $M$.
– Timedout($e_i$, $t$) tests if the time signal $t$ lies outside the spread of the time dimensions of the codeword corresponding to transition $e_i$.

## 5 Empirical Evaluation

We conducted a series of experiments to evaluate the effectiveness of Radmin. The first set of experiments evaluate the effectiveness of Radmin in detecting attacks that trigger uncontrolled resource consumption. The first experiment uses a web server and a browser, with sufficient input coverage. We then conducted a second experiment using common Linux programs, and only using the functionality tests that shipped with them as a representation of normal inputs. Finally, we conducted a third experiment to evaluate the effectiveness of Radmin in detecting starvation, using starvation and livelock cases that are common in the literature.

We refer to test cases that trigger abnormal behavior by *positive* (malicious), and those that do not by *negative* (benign). Each positive test case can either be correctly detected or missed, giving a true positive (TP) or a false negative (FN), respectively. Each negative test case can either be detected as such or incorrectly detected as an attack, giving a true negative (TN) or a false positive (FP), respectively.

### 5.1 Procedure and Metrics

For every target program, we proceeded as follows. We executed two thirds of the negative test cases to collect benign measurements and build the PFAs. Then, we executed the remaining one third to measure the false positive rate. Finally, we executed all positive test cases to measure the detection rate and earliness of the detection.

We trained the PFAs, and optimized their hyperparameters, using 5-fold cross-validation (CV) over the training sequences (measurements from the two-thirds of negative test cases used in training). For each resource type, we build a PFA and select its hyperparameters from a cross product of all possible values (see Appendix A). Training sequences were divided into five roughly equal segments. Each fold in the CV used the sequences in one such segment for testing, and a concatenation of the rest for training. CV testing is performed by calculating the average log-loss of the prediction probability of sequences, given by $-\frac{1}{T}\sum_{i=1}^{T} \lg P(s_i)$, where $P(s_i)$ is the prediction probability of test sequence $s_i$ and $T$ is the total number of test sequences. This is done for each fold, resulting in five average log-loss values per hyperparameters vector. Finally, the hyperparameter vector with the best median log-loss over the five folds is used for building the PFA over the entire training sequences.

We used the following metrics in our evaluation: False Positive Rate (FPR), True Positive Rate (TPR), and Earliness (Erl.). Earliness is calculated as the percentage of the amount of resources that Radmin *saved* under an attack, to the *maximum* resources used by negative runs. We use Erl. to quantify how quick Radmin detected the attacks. For example, if a program consumed a maximum of 40 MB under benign conditions, and an attack consumed 30 MB before Radmin detected it, the earliness of detecting the attack would be $\frac{40-30}{40} = 25\%$. Erl. reaches its best value at 100 and its worst at 0.

For resource exhaustion detection, we used synthetic attacks (which we discuss in the following section). In the case of starvation and livelocks, we used a number of common cases that appeared in prior livelock detection studies [6, 7, 22, 27]. Note, since the attacks aimed at exhausting system resources, they were always detected once consumed resources more than the maximum of benign runs. Therefore, Radmin always achieved a TPR of 1. The same applies to starvation and livelock test cases.

### 5.2 Synthetic Exhaustion Attacks

One approach to evaluate Radmin against resource exhaustion attacks would be to test it with several known attacks. While such an approach is common in the literature, it suffers from two major drawbacks. First, it is very challenging to identify real exhaustion attacks that exploit different weaknesses, different resource types, and exercise different code paths for each target program. That means the produced results could be biased, because the number of attacks would have little to no correlation with the *variety* of attacks that can be detected. Second, evaluating a defense system against only known attacks limits the scope of the evaluation and the results to *only* the known attacks. As we have seen in the past [15, 25, 29], this may establish a false sense of security against attacks that are possible in practice but have not yet been seen in the wild. Therefore, we decided against using the only few known attacks, and instead opted for generating synthetic attacks that resemble, and even surpass in sophistication and variety, the attacks seen in the wild. Our ultimate goal is to stress the system and find out its limits on a much richer set of attack entries.

To achieve that, we assume that the attacker has successfully identified some exhaustion vulnerability in the target program, and has crafted malicious input that successfully triggers the vulnerability. The nature of the exploit by which the vulnerability is triggered is not pertinent to our evaluation, since we are only concerned about the scope of the exploit (in our case, resource exhaustion) rather than its cause. Therefore, the malicious input that caused the exhaustion can be substituted by attack code that executes to the same effect at some vulnerable code site in the context of the process. Therefore, we generated synthetic attack datasets by separately collecting measurements for exhaustion attack samples, and injecting those measurements in the trace of negative (benign) measurements. The attack measurements are injected once per trace file at a randomly selected location. To account for differences in the total amount of the attacked resource at the injection point, we adjust the injected measurements by adding (summing) the last benign measurement vector of the same resource type to each attack vector in the rest of the trace. Being able to inject the attacks at *any* point in the trace allows us to accurately capture attacks seen in the wild, and even cover more sophisticated cases, including exhaustion attacks at very early or very late stages in the execution of the process. For example, exhaustion may be possible through attacker controlled environment variables that are used by dynamic libraries during process creation or termination.

The attack snippets were designed to enable the attacks to execute stealthily (by slowly harvesting resources) and avoid early detection. This is a worst-case scenario that is much more conservative than current attacks seen in the wild. For attacks that targeted memory, file descriptors, and tasks, we allocated 10 memory pages, 1 file descriptor, and 1 task per each iteration of the attack, respectively. For stack attacks, we used uncontrolled recursion where each stack frame is approximately 512 bytes. CPU attacks were infinite loops that compute `sqrt` and `pow` operations, where each iteration consumed 4 clock ticks on average. In general, the attacks covered the following CWE classes[4]: 400, 401, 404, 674, 770, 771, 772, 773, 774, 775, and 834. Note that the choice of the parameters does not bias our results because they do not, by themselves, alter the outcome of the attack or the pattern at which it occurs.

## 5.3 Resource Exhaustion Results

**Experiment 1** The first experiment replayed a dataset of ∼60K *unique* benign URLs of incoming HTTP GET requests to our school servers. We used the `w3m` browser on the `xterm` terminal, and the host domains were mirrored and served

[4] For details and code samples, please refer to the CWE project at http://cwe.mitre.org

**Table 2.** Detection performance for Experiment 1.

| Prog. | TP | FP | TN | FPR | %Erl. (mean ± std.) | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | CPU | File | Task | Mem. |
| `apache-2.4.7` | 6064 | 5 | 12167 | 0.0004 | $40 \pm 23$ | $85 \pm 19$ | $12 \pm 10$ | $05 \pm 03$ |
| `w3m-0.5.3` | 14245 | 20 | 18684 | 0.0011 | $87 \pm 08$ | $49 \pm 40$ | $25 \pm 23$ | $51 \pm 27$ |

**Table 3.** Detection performance for Experiment 2.

| Prog. | TP | FP | TN | FPR | %Erl. (mean ± std.) | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | CPU | File | Task | Mem. |
| `cmp-3.3` | 9 | 0 | 14 | 0 | $98 \pm 01$ | $62 \pm 32$ | - | $54 \pm 39$ |
| `cpio-2.11` | 24 | 0 | 17 | 0 | $99 \pm 01$ | $49 \pm 35$ | - | $99 \pm 03$ |
| `diff-3.3` | 56 | 0 | 109 | 0 | $90 \pm 01$ | $65 \pm 32$ | - | $55 \pm 41$ |
| `gawk-4.0.1` | 223 | 2 | 389 | 0.0051 | $81 \pm 03$ | $50 \pm 29$ | $76 \pm 15$ | $28 \pm 21$ |
| `gzip-1.6` | 109 | 2 | 201 | 0.0099 | $77 \pm 28$ | $53 \pm 35$ | - | $39 \pm 48$ |
| `openssl-1.0.1f` | 380 | 0 | 594 | 0 | $94 \pm 01$ | $77 \pm 25$ | - | $28 \pm 38$ |
| `rhash-1.3.1` | 22 | 1 | 35 | 0.0278 | $47 \pm 40$ | $62 \pm 33$ | - | $57 \pm 33$ |
| `sed-4.2.2` | 108 | 6 | 194 | 0.0300 | $70 \pm 30$ | $62 \pm 33$ | - | $80 \pm 16$ |
| `tar-1.27.1` | 480 | 3 | 980 | 0.0031 | $98 \pm 02$ | $82 \pm 24$ | $25 \pm 24$ | $70 \pm 19$ |
| `wget-1.5` | 55 | 0 | 79 | 0 | $95 \pm 01$ | $79 \pm 21$ | - | $50 \pm 32$ |

using `apache`. On `xterm`, `w3m` renders tables, frames, colors, links, and images. Radmin monitored both `apache` and `w3m`. In the case of apache, the monitoring was performed per each request handler.

Table 2 shows the results for this experiment. Radmin achieved a FPR of only 11 out of 10,000 requests in the case of `w3m`. For `apache`, the number further decreases to only 4 out of 10,000 requests. In the case of `apache`, Radmin saved more than 85% of the file descriptors (the maximum of negative runs was 10 file descriptors). The memory saving for `apache` is only 5%, which is due to the highly centralized distribution of memory consumption of `apache` during negative runs (1.19GB mean, 1.22GB median, 1.28GB mode). In the case of `w3m`, the maximum saving achieved was 87% for CPU time (maximum of benign runs was 56 ticks). Overall, the results show that Radmin can effectively save resources with very high accuracy.

**Experiment 2** The second experiment used 10 common Linux programs. The functionality test packages that shipped with the programs were used to train Radmin. The major difference between this experiment and Experiment 1 is the lack of input coverage. In Experiment 1, we had sufficient input to build a profile of benign behavior with high confidence. In Experiment 2, the functionality tests were few, and some of the consumption subsequences were not significant to be learned by the model (see Sections 4.2, 5.1), resulting in a higher FPR.

The selected programs cover critical infrastructure services that are often utilized by desktop and web applications — namely, compression, text processing (pattern matching and comparison), hashing, encryption, and remote downloads. Attacks on compression programs can involve highly-recursive compressed files (zip bombs), where decompressing the files would result in uncontrolled consumption of CPU time and file descriptors. Attacks on text processing applications typically use specially crafted regular expressions or data blocks that result in CPU and memory exhaustion. Hashing and encryption are notorious for CPU and memory exhaustion through specially crafted or erroneous messages. Download managers often suffer from exhaustion of file descriptors and CPU time.

Table 3 shows the results of this experiment. As expected, the FPR is higher than Experiment 1. Nevertheless, Radmin achieved a low FPR in most of the cases. For earliness, Radmin achieved high savings for all resources, saving more than 90% of CPU time in most cases. This is mainly due to the high skewness of the CPU time (in clock ticks) distribution of those programs (e.g., 374 mean, 120 median, and 1987 mode for `tar`). Overall, the results demonstrate the effectiveness of our approach, and the feasibility of using functionality tests to train Radmin.

We emphasize that the FPR of Radmin is inverse proportional to input coverage. As higher input coverage is achieved, the PFA models used in Radmin become more complete and the FPR decreases. We discuss this in Section 6.1, along with ways to further increase the earliness of detection.

## 5.4 Starvation and Livelock Results

In this experiment, we used a number of common resource starvation samples [6, 7, 22, 27]. Simplified snippets of the test cases are provided in Appendix B. The test cases spanned the two major resource starvation causes: (1) starvation due to prolonged holding of resources by other processes, and (2) livelocks due to busy-wait locking.

The first test case, `filelock`, is a multi-process program that manages exclusive access to resources by holding a lock on an external file. In this case, starvation can happen when a process holds the lock for a prolonged time, preventing other processes from making progress. In the second test case, `twolocks`, two threads try to acquire two locks, in reversed order, and release any acquired locks if the two locks were not both acquired. This is a fundamental livelock case due to unordered busy-wait locking of resources. Finally, the third test case is a rare bug in `sqlite`, when two or more threads fail, at the same time, to acquire a lock.

In this experiment, we ran each test case a 1000 times, and timed out each run after 20 seconds. Runs that finished before the 20 seconds deadline were considered negative samples, and runs that did not finish by the deadline were considered positive. Table 4 shows the results for this experiment.

Radmin detected the positive samples with high earliness. For `filelock`, Radmin saved 59% of the maximum (8 ticks) of negative `filelock` runs. In the case of `twolocks`, Radmin saved more than 93% of 12 ticks. For `sqlite`, Radmin saved 76% of 19 clock ticks. Additionally, Radmin achieved 0 FPs and 0 FNs,

**Table 4.** Starvation detection performance.

| Prog. | TP | FN | FP | TN | TPR | FPR | %Erl. (mean $\pm$ std.)[†] |
|---|---|---|---|---|---|---|---|
| `filelock` | 570 | 0 | 0 | 143 | 1 | 0 | $59 \pm 26$ |
| `twolocks` | 705 | 0 | 0 | 98 | 1 | 0 | $93 \pm 04$ |
| `sqlite` | 460 | 0 | 0 | 180 | 1 | 0 | $76 \pm 13$ |

[†] Erl. here refers only to percentage of saved clock ticks.

indicating that none of the negative samples spent time in a PFA state more than the spread of the codewords corresponding to all outgoing transitions from that state. This means that the negative runs showed a set of similar timing behaviors that were fully learned by the model. Due to the external factors involved, such as internal parameters of the kernel scheduler, further studies are needed in order to reach a conclusive understanding of such behavior. Overall, the results show the promise of our approach, even in starvation situations that involved multiple processes and threads.

### 5.5 Overhead

We report the overhead incurred by Radmin, in the online phase, for the programs used in our experiments as well as for the UnixBench [9] benchmark. We chose UnixBench because it tests various aspects of the system performance and uses well-understood and consistent benchmarks. Note, Radmin generated no false positives for UnixBench. All experiments were executed on machines running Ubuntu Server 14.04, quad-core 2.83 GHz (base) Intel Xeon X3363 processor and 8 GB of memory. The overhead is summarized in Fig. 4. Radmin incurred less than 16% overhead, with mean overhead (geometric) of 3.1%. The runtime overhead is more pronounced in CPU bound programs that were more frequently interrupted by the heartbeat thread. Overall, since Radmin avoids sampling, uses static rewriting, and selectively traces a particular set of events, the overhead incurred is significantly less than generic dynamic instrumentation and profiling tools (more than 200% runtime increase [36,39]).

## 6 Discussion and Limitations

### 6.1 Higher Accuracy and Earliness

The PFA model used in Radmin learns *only* the subsequences that have significant prediction probability (see Section 4.2), which means that some *benign but rare* subsequences may *not* be learned by the PFA. Such subsequences would be erroneously flagged as attacks (false positives), since they traverse low probability paths in the PFA. Although it is straightforward to force the inclusion of such subsequences in the PFA by adjusting the transition probabilities of their corresponding paths, we decided against doing so in order to give a clear and

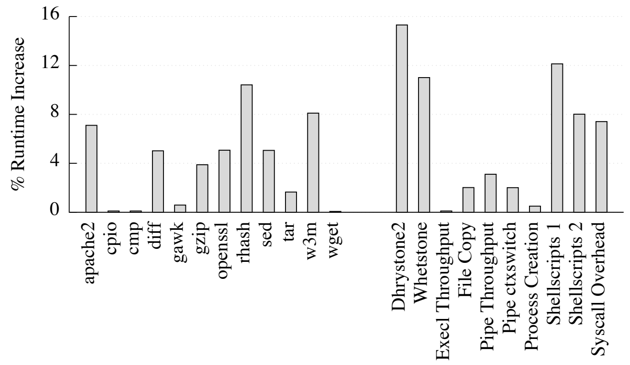

**Fig. 4.** Runtime overhead incurred by Radmin, in the online phase, for theprograms used in Experiments 1 and 2 (left), and the UnixBench benchmark (right).

realistic view of the efficacy of the system. However, Radmin has the nice property that the FPR is inverse proportional to input coverage, i.e., as benign input coverage increases, the number of benign rare subsequences decreases and the PFAs become eventually complete.

Leveraging more information about the target process can allow Radmin to achieve higher earliness. For example, we can associate input values and attributes with paths in the PFAs. The challenges here are reaching a reliable model for representing and matching various input vectors, such as command line arguments, file IO, environment variables, and succinctly associating the input with paths in the PFAs. Given such a model, we can traverse the PFAs without actually executing the program. That would give the *near*-optimum earliness, since traversing the PFAs is much cheaper than running the target program itself. Also, one can synthesize static input filters from the PFAs. We plan to explore these ideas in more details in our future work.

## 6.2 Behavior Confinement

Radmin can be used to confine the behavior of processes to users rather than only detecting anomalous usage. Depending on how each user uses a program, Radmin will learn different behavior that is specific to the user. This can help defend unknown attacks by detecting anomalous, but valid, consumption of resources. Radmin can be easily extended to seal off paths of infrequent or undesired resource usage in protected programs by adjusting the conditional distributions in the PFAs. Similarly, Radmin can be used to construct a profile of specific behavioral aspects of target programs, such as sequences of executed events or files accessed. It can also confine the behavior of protocols, which is currently in our future work.

## 6.3 Attacker Knowledge of Radmin

Attackers could potentially attempt to employ Radmin to learn the PFAs for a target program, then craft input that maximizes the consumption of the program by steering the execution to paths of high resource consumption. We argue that such an attack is not a resource exhaustion attack per se. The reason is that if the PFA contains a path of high resource consumption, that means some typical benign input to the program does exercise that path, and the subsequences of the path are statistically significant. Therefore, the consumed resources cannot amount to an exhaustion, otherwise the input should have not been accepted (as benign) by the program in the first place. In this case, rate limiting techniques can be employed to throttle the rate of requests (whether benign or not) that exercise such paths. Nevertheless, Radmin still limits the potential of the attacks to cause *actual* resource exhaustion damage, by confining them to *only* high probability paths in the PFAs. In other words, the attacker has to identify a PFA path that exhibits high resource consumption, but such path might not be present in many of the programs.

### 6.4 Accuracy of Recursive Sites Identification

Dyninst ParseAPI uses recursive traversal parsing to construct the CFG, and employs heuristics to identify functions that are reached only through indirect flows. The resulting CFG may be incomplete, which might cause the User Tracer to miss some recursive code sites if the recursion is chained using indirect calls that ParseAPI could not resolve. While we argue that such construct is rare in practice, it can be addressed by dynamically tracing indirect calls using a shadow call stack, at the expense of increased runtime overhead. We plan on exploring this option as part of our future work.

### 6.5 Exhaustion Through Separate Runs

The current monitoring approach monitors consumption that lives only within individual processes. This does not allow detection of attacks that span multiple runs of some target program. For example, if a program creates a new file every time it runs, excessively running the program can exhaust the storage space. Extending Radmin to monitor consumption of system resources across separate runs is straightforward.

## 7 Related Work

Modern operating systems offer a number of threshold-based facilities to limit the resource consumption of processes (e.g., `setrlimit`, `ulimit`, `AppArmor`). Those facilities, while widely available, fall short of detecting or mitigating resource exhaustion and starvation attacks, for two reasons. First, the limits are set irrespective of the actual consumption of different program segments for different inputs or users. This enables attackers to exhaust resources by crafting input that consumes the highest possible resources, for prolonged times [17,18,26]. Second, the facilities cannot detect starvation attacks, since only an upper bound is used in detection.

Antunes et al. [12] proposed a system for testing server programs for exhaustion vulnerabilities. The system depended on a user supplied specifications of the server protocol, and automatically generated (fuzzed) test cases and launched them against the server. In [23], Groza et al. formalized DoS attacks using a set of protocol cost-based rules. Aiello et al. [11] formalized a set of specifications that a protocol has to meet to be resilient to DoS attacks. While the idea is promising, the specifications need explicit cost calculation of required computational resources, which is often not feasible in practice [37].

Chang et al. [16] proposed a static analysis system was for identifying source code sites that may result in uncontrolled CPU time and stack consumption. The authors used taint and control-dependency analysis to automatically identify high complexity control structures in the source code, whose execution can be influenced by untrusted input. Similar approaches that required manual source code annotation were also developed [24,38]. Radmin substantially differs from those systems in that it a dynamic solution, does not require access to the source code or any side information, and it covers different types of resources rather than only CPU and stack consumption.

In [33,34], Sekar et al. introduced approaches for detecting abnormal program behavior by building automata from system calls and executing the automata at runtime, flagging invalid transitions as anomalies. Mazeroff et al. [28] described methods for inferring and using probabilistic models for detecting anomalous sequences of system calls. They built a baseline model of sequences of system calls executed by benign programs, a test model of a target program, and compared the distance between the two models to detect anomalies. While approaches based on system call monitoring are easy to deploy, they are prone to mimicry attacks [25,29]. Additionally, they either completely ignore call arguments, which makes them inapplicable for exhaustion detection; or they model the arguments using point estimates, which is insufficient for early exhaustion detection.

Radmin is fundamentally different from all these systems in that it captures both program code and input dependencies of resource consumption, by modeling both the temporal and spatial information in resource consumption behavior of the program. Radmin detects both exhaustion and starvation attacks, and does not use static thresholds. By leveraging temporal information, Radmin also detects when target programs are starving of resources. Additionally, Radmin monitors the target programs by hooking into the kernel tracing facilities, which allows for maximum visibility into the target process and allows for low-penalty monitoring.

## 8 Conclusion

The paper presented Radmin, a system for early detection of resource exhaustion and starvation attacks. Unlike existing solutions, Radmin does not use static limits and utilizes both temporal and spatial resource usage information. Radmin reduces the monitoring overhead by hooking into kernel tracepoints. The Radmin user space library keeps track of stack usage used by target processes, and provides a heartbeat signal that enables Radmin to detect starvation. We showed that Radmin can detect resource exhaustion and starvation attacks with high earliness and accuracy, and low overhead. The implementation of Radmin was discussed along with its limitations and possible areas for improvements.

**Acknowledgements.** We thank Konstantinos Kolias, the anonymous reviewers, and our shepherd Andrei Sabelfeld for their insightful comments and suggestions. We thank Sharath Hiremagalore for technical assistance. This work is supported by the National Science Foundation Grant No. CNS 1421747 and II-NEW 1205453. Opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or the US Government.

## References

1. 4 myths of ddos attacks. http://blog.radware.com/security/2012/02/4-massive-myths-of-ddos/
2. Availability overrides security concerns. http://www.hrfuture.net/performance-and-productivity/availability-over-rides-cloud-security-concerns.php?Itemid=169
3. CWE-400: Uncontrolled resource consumption. http://cwe.mitre.org/data/definitions/400.html
4. Dyninst API. http://www.dyninst.org/dyninst
5. Mobile users favor productivity over security. http://www.infoworld.com/article/2686762/security/mobile-users-favor-productivity-over-security-as-they-should.html
6. pthread livelock. http://www.paulbridger.com/livelock/
7. Sqlite livelock. http://www.mail-archive.com/sqlite-users@sqlite.org/msg54618.html
8. Systemtap. https://sourceware.org/systemtap/
9. Unixbench. https://github.com/kdlucas/byte-unixbench
10. Vectorized implementation of k-means++. https://github.com/michaelchughes/KMeansRex
11. 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: Proceedings of the 9th ACM Conference on Computer and Communications Security. pp. 48–58. CCS '02, ACM, New York, NY, USA (2002)
12. Antunes, J., Neves, N.F., Veríssimo, P.J.: Detection and prediction of resource-exhaustion vulnerabilities. In: Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on. pp. 87–96. IEEE (2008)
13. Arthur, D., Vassilvitskii, S.: k-means++: The advantages of careful seeding. In: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 1027–1035. Society for Industrial and Applied Mathematics (2007)
14. Bejerano, G., Yona, G.: Variations on probabilistic suffix trees: statistical modeling and prediction of protein families. Bioinformatics 17(1), 23–43 (2001)
15. Carlini, N., Wagner, D.: Rop is still dangerous: Breaking modern defenses. In: USENIX Security Symposium (2014)
16. Chang, R.M., Jiang, G., Ivancic, F., Sankaranarayanan, S., Shmatikov, V.: Inputs of coma: Static detection of denial-of-service vulnerabilities. In: Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009. pp. 186–199. IEEE Computer Society (2009)

17. Chee, W.O., Brennan, T.: Layer-7 ddos (2010)
18. Crosby, S., Wallach, D.: Algorithmic dos. In: Encyclopedia of Cryptography and Security, pp. 32–33. Springer (2011)
19. Dekel, O., Shalev-Shwartz, S., Singer, Y.: The power of selective memory: Self-bounded learning of prediction suffix trees. In: Advances in Neural Information Processing Systems. pp. 345–352 (2004)
20. Desnoyers, M.: Using the linux kernel tracepoints. https://www.kernel.org/doc/Documentation/trace/tracepoints.txt
21. Fu, S.: Performance metric selection for autonomic anomaly detection on cloud computing systems. In: Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE. pp. 1–5. IEEE (2011)
22. Ganai, M.K.: Dynamic livelock analysis of multi-threaded programs. In: Runtime Verification. pp. 3–18. Springer (2013)
23. Groza, B., Minea, M.: Formal modelling and automatic detection of resource exhaustion attacks. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. pp. 326–333. ACM (2011)
24. 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. Springer (2008)
25. Kayacik, H.G., et al.: Mimicry attacks demystified: What can attackers do to evade detection? In: Privacy, Security and Trust, 2008. PST'08. Sixth Annual Conference on. pp. 213–223. IEEE (2008)
26. Kostadinov, D.: Layer-7 ddos attacks: detection and mitigation - infosec institute. http://resources.infosecinstitute.com/layer-7-ddos-attacks-detection-mitigation/ (2013)
27. Lin, Y., Kulkarni, S.S.: Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 237–247. ACM (2014)
28. Mazeroff, G., Gregor, J., Thomason, M., Ford, R.: Probabilistic suffix models for API sequence analysis of Windows XP applications. Pattern Recogn. 41(1), 90–101 (Jan 2008)
29. Parampalli, C., Sekar, R., Johnson, R.: A practical mimicry attack against powerful system-call monitors. In: Proceedings of the 2008 ACM symposium on Information, computer and communications security. pp. 156–167. ACM (2008)
30. Ron, D., Singer, Y., Tishby, N.: The power of amnesia: Learning probabilistic automata with variable memory length. Mach. Learn. 25(2-3), 117–149 (Dec 1996)
31. Rutar, N., Hollingsworth, J.: Data centric techniques for mapping performance measurements. In: Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on. pp. 1274–1281 (May 2011)
32. Saltzer, J., Schroeder, M.: The protection of information in computer systems. Proceedings of the IEEE 63(9), 1278–1308 (Sept 1975)
33. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on. pp. 144–155. IEEE (2001)
34. Sekar, R., Venkatakrishnan, V., Basu, S., Bhatkar, S., DuVarney, D.C.: Model-carrying code: a practical approach for safe execution of untrusted applications. ACM SIGOPS Operating Systems Review 37(5), 15–28 (2003)
35. Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J., Keromytis, A.D.: Assure: automatic software self-healing using rescue points. ACM SIGARCH Computer Architecture News 37(1), 37–48 (2009)
36. Uh, G.R., Cohn, R., Yadavalli, B., Peri, R., Ayyagari, R.: Analyzing dynamic binary instrumentation overhead. In: Workshop on Binary Instrumentation and Application (2007)

37. Zargar, S.T., Joshi, J., Tipper, D.: A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks. Communications Surveys & Tutorials, IEEE 15(4), 2046–2069 (2013)
38. Zheng, L., Myers, A.C.: End-to-end availability policies and noninterference. In: Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop. pp. 272–286. IEEE (2005)
39. Zinke, J.: System call tracing overhead. In: The International Linux System Technology Conference (Linux Kongress) (2009)

## A PST Hyperparameters Grid

**Table 5.** Hyperparameter values used in training the PSTs.

| Param. | Possible values | Chosen (median) |
|---|---|---|
| $\gamma_{min}$ | $\{10^{-5}, 10^{-7}, 10^{-11}, 10^{-13}\}$ | $10^{-11}$ |
| $r_{min}$ | $\{1.05\}$ | 1.05 |
| $L$ | $\{30, 40, 50, 60\}$ | 40 |

## B Starvation and Livelock Snippets

**Listing 1.1.** `filelock`

```
void filelock() {
  fork()
  ...
  system("lockfile lockfile.lock");
  ...
  // do some work
  ...
  system("rm -f lockfile.lock");
}
```

**Listing 1.2.** `sqlite-lock`

```
void execute(char *s) {
  ...
  while (sqlite3_step(stmt) == SQLITE_BUSY)
    sleep(1);
  sqlite3_finalize(stmt);
}

void thread2() {
  open_db();
  execute("UPDATE foo SET ...");
  ...
}

void thread1() {
  open_db();
  ...
  sqlite3_prepare_v2("SELECT id FROM foo", ...);
  sqlite3_step(stmt);
  ...
  start_thread(thread2, ...);
  ...
  // livelock if interrupted thread2
  execute("INSERT INTO foo VALUES(100)");
  ...
}
```

**Listing 1.3.** twolocks

```
void thread1() {
  while (true) {
    ...
    lock lock_x(resource_x);
    ...
    try_lock lock_y(resource_y);
    if (!lock_y) continue;
    ...
  }
}

void thread2() {
  while (true) {
    ...
    lock lock_y(resource_y);
    ...
    try_lock lock_x(resource_x);
    if (!lock_x) continue;
    ...
  }
}
```