

Sechduler: A Security-Aware Kernel Scheduler

Saman Zonouz, Rui Han
Electrical and Computer Engineering
University of Miami
s.zonouz@miami.edu, r.han@umiami.edu

Parisa Haghani
Electrical and Computer Engineering
University of Illinois
haghani1@illinois.edu

Abstract—Trustworthy operation of safety-critical infrastructures necessitates efficient solutions that satisfy both realtimeness and security requirements simultaneously. In this paper, we present Sechduler, a formally verifiable security-aware operating system scheduler that dynamically makes sure that system computational resources are allocated to individual waiting tasks in an optimal order such that, if feasible, neither realtime nor security requirements of the system are violated. Additionally, if not both of the requirements can be satisfied simultaneously, Sechduler makes use of easy-to-define linear temporal logic-based policies as well as automatically generated Büchi automaton-based monitors, compiled as loadable kernel modules, to enforce which requirements should get the priority. Our experimental results show that Sechduler can adaptively enforce the system-wide logic-based temporal policies within the kernel and with minimal performance overhead of 3% on average to guarantee high level of combined security and realtimeness simultaneously¹.

I. INTRODUCTION

Maintenance of the safety-critical infrastructures, e.g., nuclear power plants and avionics systems, is extremely crucial because a failure to meet a single requirement may lead to a catastrophic consequence such as an explosion or an accident leading to loss of life. In particular, the realtime scheduling of tasks in those infrastructures such that individual timing requirements are met reliably is often a challenging endeavor. Furthermore, to guarantee core functionalities, those systems need to be secure and intrusion resilient as they operate in possibly adversarial environments. Currently many commercial and open-source security solutions are available that can monitor different security aspects of the systems. Clearly, the most comprehensive security level will be achieved by running a set of those security sensors in parallel; however, this would result in computationally intensive security analyses and hence over-consumption of the system's limited resources. Therefore, the system's core realtime functionality requirements could be violated as the system's critical tasks are deprived of the resources. This signifies the fact that to ensure timely accomplishment of the core system functionalities, the deployed security solutions need to be resource aware and satisfy the system-wide realtime requirements, i.e., *realtime security*. The same rationale justifies an urgent need for solutions to guarantee the *secure realtimeness* property provided by realtime solutions, e.g., realtime schedulers, that are aware of the system security requirements according to the high-level organizational objectives.

Previous efforts in designing realtime and security solutions have fallen short in several major aspects. There have been many theoretical as well as heuristic scheduling algorithms such as the Linux kernel 3.X Completely-Fair Scheduler [19], RTLinux [24] attempt to allocate the system CPU(s) to individual waiting tasks such that the likelihood of task

starvations and deadline misses are minimized. Although the abovementioned solutions can be employed to ensure timely accomplishments of safety-critical and realtime applications, none of them take into account the existence possibility of malicious activities, e.g., an adversarial unfinished task waiting for execution. Security and privacy researchers have proposed numerous host-based intrusion prevention and detection solutions, e.g., Samhain [29], as well as forensics and root-cause analysis algorithms and tools, such as Backtracker [13], and FloGuard [37], in order to detect and terminate ongoing malicious misbehaviors with minimum amount of performance overhead on the target system. Even though the abovementioned security solutions attempt to minimize the overhead as a best effort to terminate attacks before it gets too late, e.g., confidential data is sent out to network, there is currently no generic and theoretically sound solution that considers the system's overall realtime requirements and guarantees timely reaction against the ongoing intrusions.

In this paper, we present Sechduler, a formally verifiable security-aware operating system scheduler that guarantees simultaneous satisfaction of the system-wide realtimeness as well as security requirements. In particular, Sechduler accomplishes its objectives through three major steps. First, during a one-time offline phase, system security policies are defined that determine how the security vs. realtimeness tradeoffs should be resolved. These policies can be designed following whitelisting (deny by default), blacklisting (allow by default) or other more generalized paradigms. Second, during an online phase while the system is operating, Sechduler selects the appropriate subset of policies, given the current security state of the system, and generates the corresponding single logic-based conjunctive policy predicate. Sechduler then converts the policy to an extended finite state machine-based monitor automatically. Finally, Sechduler enhances the kernel scheduler with the generated monitor dynamically for runtime monitoring and verification of the system computational resource allocations. Consequently, Sechduler modifies the kernel's resource allocation schedule actively if it is about to violate any of the predefined system-wide security policies.

More specifically, Sechduler makes use of an easy to understand formal language, namely *three-valued linear temporal logic* that facilitates formulation of comprehensive temporal system-wide security policies for the system administrators. Needless to mention, the designed policies can be reused across systems (analogous to the SE-Linux access control policies). The employed three-value logic, i.e., *true*: policy-compliant, *false*: policy violation, and *inconclusive*: insufficient information, enables Sechduler to use the designed policies for accurate verification and reconfiguration of the kernel task scheduling dynamically based on the observed scheduling trace, i.e., the past and current (to be) scheduled tasks. Sechduler considers the trace formally as a finite prefix

¹This material is based upon work supported by the Office of Naval Research under Award Number N00014-12-1-0462.

of the (potentially) infinite task scheduling sequence in the future. For the kernel to understand and enforce the policies, Sechduler converts the logic-based policies automatically to an extended finite state machine, so-called Büchi automaton, working with the three-value logic. The Büchi automaton monitors the kernel scheduling trace and determines whether the policy is about to be violated. If so, the scheduler is reconfigured and the system CPU is allocated to the next non-policy-violating waiting task with an urgent need for execution. It is noteworthy that the automated conversion algorithm in Sechduler results in an automaton with provably minimum number of states ensuring that the performance overhead of the runtime monitoring and verification is minimized. Consequently, using a realtime and security-aware scheduling algorithm through continuous optimization for timely resource allocations and discrete logic-based monitoring for security verifications, Sechduler makes sure to provide both realtimeness and security guarantees simultaneously if feasible depending on the available time and resources.

In summary, the contributions of this paper are as follows:

- 1) We propose an easy-to-understand logical formulation formalism to declare the system security requirements for different system security states;
- 2) We introduce a three-value logic-based automated algorithm to construct security formal monitors dynamically for runtime verification and temporal policy enforcement;
- 3) We propose a hybrid operating system task scheduling algorithm using continuous task ranking optimization and discrete logic-based formal verification techniques;
- 4) We validate the Sechduler framework on a real-world host system through implementation and deployment of a working prototype of the proposed algorithms. It is also important to mention what Sechduler does not contribute to. In particular, Sechduler does not present a new intrusion detection sensor and automatic logic-based policy generation algorithm. Instead, Sechduler makes use of those solutions to provide the runtime verification capability to maintain the system security and realtime requirements and avoid potential violations of the previously defined temporal policies.

We implemented a working prototype of the Sechduler framework for the Linux kernel 3.4.4. In particular, once a host-based intrusion detection system, e.g., the Samhain file integrity checker, identifies a malicious behavior while the system is operating, Sechduler selects the relevant subset of the security policies based on the current security state of the system, determined by the triggered intrusion detection system alerts, and creates a conjunctive logical predicate. We modified the kernel scheduler such that, after the automated predicate-to-automaton conversion, the generated automaton can be compiled dynamically and inserted as loadable kernel module into the running kernel. The modified scheduler and the loaded module makes sure that the individual tasks are completed in a timely manner as long as the temporal security policies are not violated.

II. MOTIVATION AND SOLUTION OVERVIEW

In this section, we first explain several major practical scenarios where Sechduler can help to resolve the involved challenging issues effectively. Then, we present an overview of the Sechduler architecture and its individual components.

We are interested in the following challenging questions. How can we make sure that the system satisfies its security and realtimeness requirements simultaneously? If this is infeasible due to the limited resources, which one should be sacrificed for the other? The answer to the latter question is subjective,

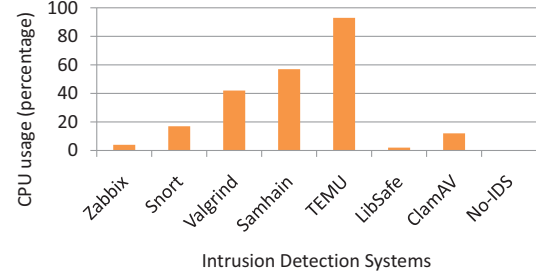


Fig. 1. CPU Overhead of Intrusion Detection Sensors

and hence cannot be automated completely. This is because resolution of the realtimeness vs. security tradeoff depends heavily on the overall mission objective that are not often documented clearly. For instance, realtime requirements in a nuclear power plant are often much more strict than the requirements regarding its privacy concerns whereas in a financial institute it is often the other way around. How should this problem (resolution of the realtimeness vs. security tradeoff) be formulated such that it is easy to define by the security administrators for different mission objectives? How can those human-readable policies be interpreted dynamically and enforced system-wide while the system is in its operational mode? and finally, how can all of these be accomplished with minimum overhead so that the solution can be employed in real-world large-scale infrastructures?

Motivating Scenarios. 1) *Partial System Pause for Root-Cause Analysis:* Currently, there are many available security solutions such as host-based intrusion detection systems that can be deployed permanently to monitor almost every system aspect to terminate potentially malicious activities. However, only few sensors are often deployed in real-world systems because of the sensors' resource requirements and possibly high performance overheads. For instance, based on our experiments, the TEMU information flow tracking and root-cause analysis engine [25] slows down the system's overall throughput 20X on average. Figure 1 shows the average CPU overhead for some of the well-known intrusion detectors. Therefore, several reactive root cause analysis and sensor selection solutions such as BackTracker [14] and FloGuard [35] have been proposed that deploy and turn on sensors and analysis engines on demand as a reaction to a detected malicious activity. As a case in point, BackTracker starts the analysis of the system's `syslog` file once a lightweight file integrity checker, e.g., Samhain [29], reports a suspicious confidentiality-sensitive file read by a process P_i . The BackTracker engine completes a computationally intensive data flow-based backtracking analysis to determine the main source of the detected anomalous sensitive file read, such as a network socket, and whether the source was malicious, e.g., a known malicious website. To prevent the possibility of sensitive data disclosure, e.g., through a socket send out command by the suspicious process which accessed the file, BackTracker needs to pause the *whole* system until the end of the tedious backtracking analysis with a proof that the source is not malicious. Alternatively, Sechduler makes sure that the system continues its normal operation completely except when the following temporal security policy is about to be violated:

```

No [Process  $P_i$  execution] UNTIL [[BackTracker Completes]
AND [Source is legitimate]]

```

Fig. 3. A Sample Temporal Security Policy

In the case of policy violation, Sechduler will enforce the policy actively by pausing the system partially, i.e. the process

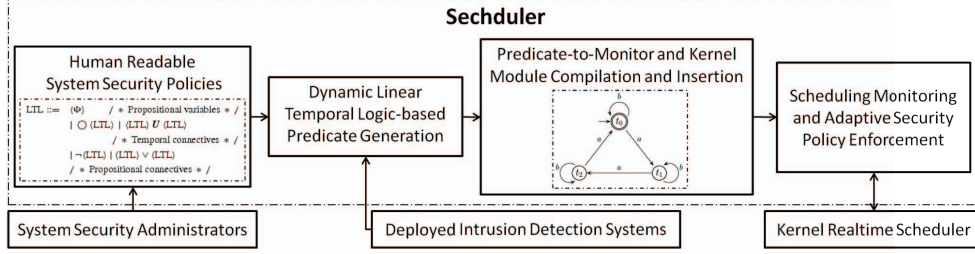


Fig. 2. Sechduler's High-Level Architecture and Interconnected Components

P_i will not be given the CPU access for execution.

2) *Security-Aware Task Scheduling*: Generally, although simultaneous consideration of realtime and security requirements occurs frequently in real-world settings, handling it efficiently and accurately is a challenging endeavor. For instance, let us consider an automatic generation control² process P_j with (*urgent*) realtime deadline constraints asks for the system CPU while its security level is marked *low* due to a potentially malicious buffer overflow detected by the Memcheck [17] dynamic memory access monitoring suite. Whether it should be given the CPU access must be determined after a security risk analysis of the possible outcomes of either letting the (malicious) task execute and send corrupted control commands with potentially catastrophic consequences such as generator explosions [18] or denying its CPU access request and hence avoiding a possibly corrective power generation adjustment when the load has increased significantly. Operating system schedulers are one of the very few places in a system that the security and realtimeness requirements meet and may contradict each other, and hence the security vs. realtimeness tradeoff can be resolved. Currently, no operating system scheduler handles both criteria simultaneously. Sechduler offers to resolve such security-realtimeness dilemmas through enforcement of predefined temporal logic-based policies dynamically. For instance, in the abovementioned scenario, given the following policy rule, Sechduler would enforce the policy and

```
Globally [[Security( $P_j$ ) > Medium] AND
[Realtimeness( $P_j$ ) > Unnecessary]]  $\rightarrow$  [Process  $P_j$ 
execution]
```

Fig. 4. A Security-aware Scheduling Policy Rule

let the process P_j get access to the system CPU and continue execution. It is noteworthy that the above rule somewhat sacrifices the security for the system's realtimeness in a power-grid critical infrastructure. The same policy may be defined completely differently in a financial enterprise environment, where the monetary transaction security is more important than their timely completion. The policy rule in Figure 4 has been defined specifically for the process P_j ; however, the policies in Sechduler are defined parametrically for the system and application processes generically to minimize the required amount of effort to complete the policy definition phase. The parameters are later replaced with concrete process IDs dynamically based on the triggered security sensor alerts.

Sechduler Overview. In this section, we present an overview of how the proposed framework achieves the abovementioned objectives in systems with both realtimeness and security requirements. Figure 2 illustrates Sechduler's different components and how individual components are interconnected logically. Initially, the security administrators write

²In a power-grid critical infrastructure, automatic generation control is a controller software for adjusting the power output of multiple generators at different power plants, in response to changes in the load [4].

system security temporal policies using the easy-to-understand formalism in Sechduler. This phase is very similar to writing access control policies for firewalls or host-based SE-Linux systems; however, in Sechduler, administrators concentrate on timing- and scheduling-related security concerns instead. Briefly, each policy determines the scheduling constraints that need to be held at a system security state by the operating system to guarantee that the system-wide security is maintained. Although, in this paper, we assume that the policy writing is completed as a one-time manual effort, Sechduler could be extended and make use of the recent (semi-)automated policy writing algorithms and tools [22]. We consider those directions outside the scope of this paper and currently investigate them as future work. Furthermore, like SE-Linux policy rulesets, Sechduler's temporal policies may also be reused across different systems to reduce or completely eliminate the need for the policy writing phase.

During the system's operational mode, we assume that appropriate host-based intrusion detection systems are deployed and are monitoring important aspects of the target system, such as the filesystem integrity using, for instance, periodic hash function-based scans [29]. In case a malicious activity is identified, Sechduler receives the triggered intrusion detection system alerts that cooperatively report the system's current security state. Sechduler goes through its policy repository dynamically and selects the relevant (possibly empty) subset of policy rules that correspond to the system's current state. Sechduler then constructs a single system-wide temporal logic-based predicate using the collected policy rules, and converts the predicate into a Büchi automaton-based monitor automatically. The automaton is compiled as a loadable kernel module and inserted into the running operating system kernel. The modified kernel scheduler notices the inserted module, and from then on verifies its individual task scheduling decisions using the loaded monitor. Additionally, if needed, it enforces the policies by reconfiguring the system's resource allocations, i.e., scheduling decisions, adaptively.

III. SYSTEM-WIDE REQUIREMENT DESCRIPTION

To formulate the system scheduling security policies, Sechduler makes use of an extended three-valued linear temporal logic formalism [2], [20]. Let us define A to be a finite set of atomic logical propositions $\{b_1, b_2, \dots, b_{|A|}\}$ and $\Sigma = 2^A$ a finite alphabet composed of the abovementioned propositions. Every element of the alphabet is a possibly empty set of propositions from A , and is denoted by a_i , e.g., $a_i = b_1, b_4, b_9$. Additionally, as Sechduler deals with runtime verification of the past and current system traces³ of the scheduled tasks, we define Σ^* to be all of the possible finite traces over Σ , e.g., $(a_0 a_1 a_2)$, where two subsequent events a_i and a_j are

³We define *trace* as a sequence of scheduled tasks in the target system.

represented by symbolic concatenation $a_i a_j$. Similarly, Σ^ω is defined to be the set of infinite system traces.

The set of linear temporal logic-based security policies is inductively defined by the grammar

$$\varphi ::= \text{true} \mid b \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X} \varphi, \quad (1)$$

where φ is a logical predicate; \mathbf{U} and \mathbf{X} represent the temporal *until* and *next* operators, respectively. For policy description simplicity, like in the related past literature, Sechduler also makes use of the following three redundant notations: $\varphi \wedge \psi$ instead of $\neg(\neg\varphi \vee \neg\psi)$, $\varphi \rightarrow \psi$ instead of $\neg\varphi \vee \psi$, $\mathbf{F} \varphi$ (eventually) instead of $\text{true} \mathbf{U} \varphi$, and $\mathbf{G} \varphi$ (globally) instead of $\neg(\text{true} \mathbf{U} \neg\varphi)$.

Furthermore, the semantics of the scheduling security policy description formalism is defined inductively as follows

$$\begin{aligned} \omega, i &\models \text{true} \\ \omega, i &\models \neg\varphi & \text{iff} & \quad \omega, i \not\models \varphi \\ \omega, i &\models b & \text{iff} & \quad b \in a_i \\ \omega, i &\models \varphi_1 \vee \varphi_2 & \text{iff} & \quad \omega, i \models \varphi_1 \text{ or } \omega, i \models \varphi_2 \\ \omega, i &\models \varphi_1 \mathbf{U} \varphi_2 & \text{iff} & \quad \exists k \geq i \text{ s.t. } \omega, k \models \varphi_2 \\ & & & \text{and } \forall i \leq l < k \text{ s.t. } \omega, k \models \varphi_1 \\ \omega, i &\models \mathbf{X} \varphi & \text{iff} & \quad \omega, i+1 \models \varphi \\ \omega, i &\models \varphi & \text{iff} & \quad \omega, 0 \models \varphi \end{aligned} \quad (2)$$

where $\omega = a_0 a_1, \dots, a_i, \dots \in \Sigma^\omega$ is an infinite system scheduling trace (a sequence of scheduled system tasks by the kernel's scheduler), and i represents the index of a particular task in the sequence. As discussed above, a_i denotes an element of the alphabet Σ and consists of a possibly empty set of atomic propositions. For a given temporal security policy predicate φ , we define the language $L(\varphi) = \{\omega \in \Sigma^\omega \mid \omega \models \varphi\}$ to be the set of all infinite-length system scheduling traces that comply the predicate φ .

Original temporal logic formulation [20] works with the abovementioned Boolean semantics that determines whether $\omega \models \varphi$ for a particular predicate φ and an *infinite* word ω holds or not, i.e., *true* or *false*. However, in practice, Sechduler completes the runtime system scheduling security verification dynamically given a finite prefix v of ω . Such a finite prefix of system scheduling activities may not include sufficient information to determine whether or not the system-wide security predicate holds, because two different infinite suffix traces ω'_b , $b \in \{0, 1\}$ of the prefix v may occur in the future, i.e., $\omega_0 = v\omega'_0$ or $\omega_1 = v\omega'_1$, that result in different Boolean values, e.g., $\omega_0 \models \varphi$ and $\omega_1 \not\models \varphi$. Therefore, Sechduler makes use of an extended three-valued semantics [2] that assign each finite prefix v a value of *true*, *false*, or *inconclusive* depending on the sufficiency of the information within v . Sechduler determines $v \models \varphi$ (or $v \not\models \varphi$) if $\omega_i \models \varphi$ (or $\omega_i \not\models \varphi$) holds for any possible infinite suffix system scheduling trace ω'_i . The assessment value for the remaining cases depends on the suffix trace values, and hence Sechduler's evaluation results in *inconclusive* and Sechduler waits for the future system scheduling events to make up its definite mind. In particular the semantics is defined as follows:

$$[v \models \varphi] = \begin{cases} \text{true} & \text{if } \forall \omega'_i \in \Sigma^\omega : v\omega'_i \models \varphi, \\ \text{false} & \text{if } \forall \omega'_i \in \Sigma^\omega : v\omega'_i \not\models \varphi, \\ \text{inconclusive} & \text{otherwise.} \end{cases} \quad (3)$$

It is noteworthy that because Sechduler's objective is to prevent a particular task execution if its access to CPU

violates a particular policy, during the system-wide scheduling security verification with a single finite system scheduling trace, Sechduler has to detect the policy violation *as soon as* the finite prefix gives that sufficient information so that Sechduler can deny the corresponding CPU access. Formally, Sechduler needs to be able to recognize the minimum-length *informative* prefixes.

IV. ONLINE SECURITY PREDICATE GENERATION

Detector-Capability Matrix. During its runtime operation, Sechduler makes use of a security knowledge-base, so-called *detector-capability matrix*, that encodes all the domain knowledge about the security incidents of interest as well as the available detection mechanisms and their incident detection capabilities. The matrix is designed once and can be reused across different systems. The incidents in the detector-capability matrix include all possible types of security incidents (events) $e_i \in E$ of interest that could potentially occur in the target system. To improve the scalability and ease of design, each incident type in the database represents a generic class, e.g. *system memory over-usage by a process*, that encompasses all target systems, without mentioning the specific context like in the following described event *The apache process reads the /etc/shadow file*. Sechduler also stores the set of available host-based intrusion detection systems $d_i \in D$ that may be deployed to detect particular security incidents within the target system. For each detection mechanism, Sechduler requires relative accuracy measure values associated with the detector. The accuracy values for each detector d_i are its corresponding false positive $F_p(d_i, e_j)$ and false negative $F_n(d_i, e_j)$ rates that encode its capability in detecting a particular security incident e_j .

Given the abovementioned information, Sechduler creates the detector-capability matrix, which indicates the ability of a given intrusion detection system to detect various incident types. In particular, the matrix is defined over the Cartesian product of the incident type set and the set of detectors, and shows how likely it is that each detection system could detect the occurrence of a specific incident type. In our implementations, we have used relative qualitative measure values. C (or N) means that the detection technique can always (or never) detect the incident, while L/M/H means that it can detect the instances of a incident occurrence with low/medium/high probability. These values are later translated to their corresponding numeric values by Sechduler as follows: C(1), H(0.75), M(0.5), L(0.25), N(0).

Policy Selection. The temporal security policies $p_i \in P$ in Sechduler are written in a specific format such that Sechduler can parse and process the different segments properly, and are stored in the policy repository. In particular, the i -th policy is represented by an XML element with the following attributes: 1) *Precondition*: The precondition attribute encodes the triggering security incident $e(p_i)$ that, if occurred, necessitates the enforcement of this scheduler temporal security policy; 2) *Enforcement threshold*: To deal with the uncertainty of the intrusion detection alerts, the enforcement threshold $\tau(p_i)$ stores the probability value that if the precondition's likelihood in the target system exceeds, Sechduler has to enforce the policy; and 3) *Predicate*: The predicate attribute of the policy stores the main linear temporal logic-based scheduler security predicate $\varphi(p_i)$ that Sechduler needs to load and enforce if the precondition incident occurs.

During the system's operation, Sechduler starts the event-driven policy selection procedure once any intrusion detection

alert (observable) $o_{d_i}^{e_j} \in O_{d_i}$, which reports occurrence of the j -th incident by the i -th intrusion detection system, is triggered. Sechduler looks up the probability that the alert's corresponding security incident has occurred $Pr(e_j)$ in the detector-capability matrix by calculating

$$Pr(e_j|o_i) = \frac{[(1 - F_n(e_j, o_i)) \cdot Pr(e_j)]}{(1 - F_n(e_j, o_i)) \cdot Pr(e_j) + F_p(e_j, o_i) \cdot (Pr(\bar{e}_j))}, \quad (4)$$

where, for deployment simplicity, the alerts from different intrusion detection systems regarding the same security incident are assumed to be independent. However, Sechduler can employ the corresponding cross-detector joint uncertainty distribution if available.

Consequently, given the calculated incident probability value $Pr(e_j|o_i)$, Sechduler investigates each policy p_i within the system policy repository and selects the policy for enforcement if the precondition matches the incident e_j , i.e., $e_j = e(p_i)$ and $Pr(e_j|o_i)$ is larger than the policy's enforcement threshold $\tau(p_i) < Pr(e_j|o_i)$. Needless to mention that Sechduler does not pick the policy if it is already selected and is being enforced currently. Finally, Sechduler constructs a conjunctive linear temporal logic-based predicate using the selected subset of policies P : $\phi = \bigwedge_{p_i \in P} \phi(p_i)$. It is noteworthy that Sechduler does not need to know about the set of running processes in the system a priori. The processes' names/IDs are reported by IDSes and then used to update the corresponding fields in the policies before the kernel modules are generated/compiled. Consequently, within the kernel, the modified scheduler knows the exact target processes' names/IDs. As discussed in the next section, Sechduler dynamically enforces the generated conjunctive predicate system-wide until it is satisfied completely.

V. STATE-BASED SCHEDULER MONITOR

Kernel Scheduler Security Monitor. Given the selected predicate, Sechduler converts it to a monitor that is later loaded on the kernel to verify the system scheduling activities and detect the scheduling decisions that violate the system-wide temporal security policies. In particular, Sechduler makes use of a state-based monitor using the Büchi automaton formalism, such that the system is in a particular state at each time instant. Every scheduling activity is modeled as an event and causes a state transition on the model. Formally, a Büchi automaton is defined as a tuple $(\Sigma, Q, Q_0, \delta, F)$, where Σ is a finite alphabet; Q is a finite non-empty set of system security states; $Q_0 \subseteq Q$ is a set of initial security states of the system; $\delta: Q \times \Sigma \rightarrow 2^Q$ is the security state transition function⁴ that determines the state updates according to the most recent system scheduling events; and $F \subseteq Q$ is a set of accepting security states that represent the situations that the predefined temporal system security policies are completely satisfied.

An infinite series of operating system scheduling decisions, formulated as a word $\omega = a_0a_1a_2... \in \Sigma^\omega$, is modeled as a sequence of states q_i and transitions a_i in the Büchi automaton $\eta = q_0a_0q_1a_1q_2...$ such that $q_0 \in Q_0$ and $q_{i+1} \in \delta(q_i, a_i)$. The Büchi automaton accepts such an infinite run η iff intersection of F and the set of states which are visited infinitely often during η is not empty. As discussed, Büchi automata are defined for infinite input traces; however, Sechduler takes, at each time instant, a finite input trace of the system's scheduling

activities related to the past and current incidents, i.e., the future task scheduling are not predictable. Sechduler makes use of a modified version of the Büchi [2] that can deal with finite system traces such as $\omega = a_0a_1a_2...a_c \in \Sigma^*$, where a_c represents the current scheduling incident. We call the corresponding state-transition sequence $\eta = q_0a_0q_1a_1q_2...a_cq_{c+1}$ accepting iff $q_{c+1} \in F$. Furthermore, to improve the runtime system monitoring performance, Sechduler first converts the abovementioned non-deterministic automaton to a deterministic finite automaton (DFA) model with a single initial state $|Q_0 = 1|$, where the transition function results in a single destination state given a source and action, i.e., $|\delta(q, a)| = 1$.

Automated Security Predicate-to-Monitor Conversion.

As discussed above, Sechduler needs to translate the selected temporal system-wide security policies ϕ into a finite state machine monitor $M(\phi)$ that will be used later for dynamic monitoring and verification of the system's scheduler security $v \models \phi$ given a finite sequence of the past and current scheduling decisions v . Sechduler starts with converting security requirement predicate to its corresponding Büchi automaton automatically. In particular, like in [8], Sechduler implements a recursive depth first search algorithm to construct the corresponding generalized Büchi automaton [6] that goes through a simple *degeneralization* transformation [28] to result in a classical Büchi automaton. Consequently, Sechduler translates the generated Büchi automaton into a deterministic state machine using the well-known power set construction algorithm [1], [2]. Finally, to minimize the size of the generated automaton, and hence the performance overhead of the Sechduler framework on the target system, Sechduler also implement a minimization algorithm [10] to produce the automaton with provably minimum number of states for the initial given system security predicate.

Specifically, Sechduler generates two separate Büchi automata $B(\phi)$ and $B(\neg\phi)$ that accept all models of ϕ and $\neg\phi$ (that falsify ϕ), respectively. Let us define $F(\phi)$ to be the set of states q_i in the automaton $B(\phi)$ for which the language of the automaton starting in state q_i is not empty, i.e., $L(B_{Q_0=q_i}(\phi)) \neq \emptyset$. Employing the Tarjan's algorithm [27] to obtain $F(\cdot)$, Sechduler, at each time instant, evaluates the current finite prefix of the system scheduling activities as follows:

$$[v \models \phi] = \begin{cases} \text{true} & \text{if } v \notin L(B^F(\neg\phi)), \\ \text{false} & \text{if } v \notin L(B^F(\phi)), \\ \text{inconclusive} & \text{if } v \in (L(B^F(\neg\phi)) \cap L(B^F(\phi))), \end{cases} \quad (5)$$

where $B^F(\phi) = (\Sigma, Q, Q_0, \delta, F(\phi))$ and $B^F(\neg\phi) = (\Sigma, Q, Q_0, \delta, F(\neg\phi))$. Putting it in words, the first condition assigns the value *true* to those system activities trace prefixes v that cannot satisfy the automaton $B^F(\neg\phi)$ under any possible future trace continuation (suffix), and therefore will always satisfy $B^F(\phi)$. The second line targets the prefixes that result in the value *false*. The remaining prefixes are assigned the value *inconclusive* due to the lack of sufficient information so far.

Given the produced non-deterministic automata $B^F(\phi)$ and $B^F(\neg\phi)$, Sechduler implements the power set construction procedure to construct the corresponding deterministic models $B_d^F(\phi)$ and $B_d^F(\neg\phi)$ that accept provably exact same languages. Finally, Sechduler generates the scheduler finite state machine monitor as the product of the two automata $M(\phi) = (\Sigma, Q, Q_0, \delta, \lambda)$, where $Q = Q_{B^F(\phi)} \times Q_{B^F(\neg\phi)}$; $Q_0 =$

⁴Similarly, δ' is defined for Σ^* by $\delta'(Q', \epsilon) = Q'$ and $\delta'(Q', ab) = \bigcup_{q' \in \delta(Q', a)} \delta(q', b)$ $Q' \subseteq Q, a \in \Sigma^*$.

$(Q_{0_{B^F(\varphi)}}, Q_{0_{B^F(\neg\varphi)}}); \delta((q, q'), a) = (\delta_{B^F(\varphi)(q, a)}, \delta_{B^F(\neg\varphi)(q, a)})$, and λ is defined as

$$[\lambda((q, q'))] = \begin{cases} \text{true} & \text{if } q' \notin F_d(\neg\varphi), \\ \text{false} & \text{if } q \notin F_d(\varphi), \\ \text{inconclusive} & \text{if } q \in F_d(\varphi) \\ & \text{and } q' \in F_d(\neg\varphi). \end{cases} \quad (6)$$

Consequently, as discussed in the next section, Sechduler enhances the operating system kernel using the generated state-based scheduler security monitor $M(\varphi)$ to guarantee runtime system-wide security according to the predefined temporal policies.

VI. RUNTIME POLICY ENFORCEMENT

Dynamic Verification. Once the monitor is constructed, Sechduler compiles it as a Linux kernel module and loads it on the running kernel dynamically. Therefore, at any time instant, there is a possibly empty conjunctive linear temporal logic-based monitor loaded on the kernel that Sechduler must enforce by tracing, and if necessary intercepting and modifying, the kernel's task scheduling decisions. In particular, Sechduler puts a checking point between a) the kernel scheduler's individual decisions upon which task should be given the CPU access to execute and b) the actual CPU allocation. Before the actual CPU allocation, Sechduler assumes that the task is given the CPU access and updates the temporal logic-based monitor's current state accordingly. Consequently, Sechduler checks the monitor's current state value and allows the kernel to proceed with the CPU allocation only if the monitor returns either *true* or *inconclusive*. Otherwise, i.e. if it returns *false*, Sechduler denies the CPU access request, undoes the monitor state update, and selects the next most demanding waiting task following the original scheduling algorithm.

Policy Revocation. Using a three-valued formal logic, dynamic assessment of each monitor results in one of the *true*, *false*, or *inconclusive* values during the Sechduler's runtime verification and enforcement. Sechduler makes use of these values to decide upon the necessary monitor revocations, i.e., when the monitor is not needed anymore. In particular, a monitor is not needed if its current state value is *true* that means the policy is already satisfied according to the existing scheduling activity prefix, i.e., sequence of the scheduled tasks since the monitor's loading time, regardless of the prefix's future continuation (suffix). When Sechduler chooses to revoke a particular security monitor, it unloads its corresponding module from the kernel dynamically and the system continues its operation with the remaining loaded monitors.

Complexity Analysis. We provide the theoretical time complexity analysis of different steps in Sechduler. Negation of ϕ is clearly a linear operation in the size of ϕ . Implementation of the Büchi automaton generation requires three lower level steps that is 1) to generate a non-deterministic Büchi automaton resulting in an exponential time complexity in the theoretical worst case; 2) to construct finite automaton that does not change the size of the original automaton; and 3) to generate deterministic finite automaton using the powerset construction method that results in an exponential complexity under worst case scenario. As discussed above, the monitor construction algorithm cannot be completed in a practically feasible time limit in the worst case. However, as shown by other researchers [2] and in our evaluations for scheduling security verification, the growth of the generated monitor sizes is completely feasible and easy to handle in practical scenarios

to enforce temporal security requirements on a real-world test-bed system.

VII. EVALUATIONS

We deployed Sechduler in a testbed environment and evaluated various aspects of its operation. In particular, we designed a set of experiments to empirically answer the following questions: How efficiently does Sechduler process the given temporal security policies and generate the corresponding automata dynamically? How accurately can Sechduler enforce the loaded policies within the kernel? How many task schedulings will Sechduler have to monitor in a real-world setting, and how well does Sechduler cope with the scalability problem? How much performance overhead does Sechduler introduce to the system's overall throughput? How does Sechduler work in details through a complete real-world case study scenario? We start by describing the experimentation setup, and then proceed to examine the abovementioned questions.

Implementations. Although Sechduler provides a unified temporal scheduler security monitoring solution, it makes use of several tools to accomplish its various subtasks. To this end, Sechduler uses the `LTL2BA` translator that converts linear temporal logic security formula into corresponding Büchi automata. Furthermore, Sechduler employs the `LTL3tools` monitor generator⁵ that is a collection of functions that constructs the final monitor, i.e., a Moore-type finite-state machine. For visualization purposes, Sechduler employs the `dot` utility from the `Graphviz` toolset to generate graphical representations of the generated state-based monitors dynamically. Finally, to reduce the framework's runtime overhead, Sechduler deploys the `AT&T FSM Library` to minimize the size of the generated automaton-based monitor.

We developed a kernel module that Sechduler can load on the running kernel dynamically and use it for runtime policy insertions within the kernel. The module has two interfaces that facilitates two-way dynamic communication between the kernel's internal functions and Sechduler's monitor generation engine. In particular, the module can receive the temporal policy monitors from Sechduler's monitor generation component, and consequently let the running kernel know about the updated policy that should be enforced. To make the kernel capable of temporal security policy enforcement, we extended the kernel's scheduler portion and added several functionalities to it. In particular, we modified the scheduling logic for how different types of kernel tasks, e.g., realtime task `rq_rt` and ordinary task `rq_fair` run-queues, were scheduled for execution. Since the version 3.0, the Linux kernel implements the *completely fair scheduler* algorithm for fair kernel task scheduling that deploys a red-black tree data structure, where each tree node represents a `task_struct` structure. Every tree node maintains a *virtual time* `vruntime` variable that indicates the amount of time its corresponding task has had the CPU access for execution in the past. Once a task schedule is needed, through the `__schedule` function call, the kernel prepares the leftmost node of the tree, which by definition has spent the least amount of time on CPU, for execution. Briefly, our implementations intercept each individual task selection step and double-check whether the task complies with the most updated loaded temporal security policy. A different policy enforcement mechanism was implemented for the kernel's for the realtime tasks as those tasks make use a multi-queue task

⁵ Available at <http://ltl3tools.sourceforge.net>.

```

never {
T0_init:
if
:: (!bash_pid4187_execution) -> goto T0_init
:: (backTracker_pid5829_completes &&
clamAV_pid3275_completes) -> goto accept_all
fi;
accept_all:
skip
}

```

Fig. 5. The Büchi Automaton Generation Output

selection algorithm, instead the completely-fair scheduler implementation, for tasks with different realtime priority levels. We do not present our implementation details for the realtime task scheduler here due to the space limitations.

As Sechduler requires kernel code modifications, new vulnerabilities in the system may be introduced if the code is not implemented carefully. To address this concern, we have minimized our intra-kernel modifications and the kernel-user space interface (< 500 LOC) and kept most of the code outside of the kernel space.

To further facilitate the interaction among various components of Sechduler within and outside of the kernel, we implemented two system calls, namely `int getsec(struct task_struct*)` and `int setsec(struct task_struct*, int)`, that allow Sechduler’s detection engine to (get) modify the running processes’ security levels such that Sechduler’s intra-kernel scheduler can obtain such information whenever necessary in order to make correct decisions upon whether a particular task should be given the CPU access for execution. For instance, Sechduler’s detection engine may set a particular process’s security level to `low` and spawn a system-wide forensics tools once the process is identified to abnormally write to a system sensitive file. Assume that an already loaded policy within the kernel mandates the scheduler to avoid any process execution with a security level lower than `medium`. Therefore, the abovementioned process cannot resume its execution until Sechduler reverts the process’s security level to `high` upon receipt of the mission completion and system-is-secure confirmation message from the forensics tool.

The operating system kernel within an ordinary desktop computer schedules more than 3K tasks per second on average. Therefore, runtime verification of each individual task scheduling could cause high performance overhead on the system if it is not designed properly. To this end, we developed a cache buffer for the allowed and denied set of tasks to accelerate the runtime system security verification procedure. In particular, once a task is checked by the currently loaded policy, Sechduler puts it in the cache with the time stamp and the verification result. The next time when the task needs verification, Sechduler checks the cache for the result, and will redo the policy-based checking if either the task is not found in the cache or its time stamp is older than a age threshold, i.e., 1 seconds in our implementations. To accelerate the cache search procedure, we have employed a hash table structure to store the analyzed tasks. Furthermore, Sechduler periodically goes over the cache, every 5 seconds in our implementation, and disposes the tasks which are not active anymore, i.e., they are not found on the any of the scheduler’s `rq` run-queues. For overall performance improvement and due to the finite number of the tasks within the kernel, i.e., often < 300, instead of dynamic cache size allocation, we implemented a fixed size cache, i.e., 100 entries (linked lists) which the recently analyzed tasks are added to.

```

G ((receive_request ∧ ¬send_response ∧ F send_response) →
(sensitive_file_access → (¬send_response U (security_check ∧
¬send_response)))) U send_response

```

Fig. 11. A Sample Temporal Security Policy

Experimental Results. We evaluated the accuracy and performance of the various components of Sechduler through an extensive set of experiments. An Ubuntu 11.10 computer system with Intel® Core™ i7 3.4 GHz Processor and 4 GB of memory was used for the experiments. Sechduler translates the human-readable policy rules into monitors that can be interpreted by the machine dynamically. As discussed, the first step is to construct the Büchi automata $B(\phi)$ and $B(\neg\phi)$. We used an updated version of the linear temporal logic formula illustrated in Figure 3 to construct the automata. Figure 5 shows the $B(\phi)$ automaton in a *never claim* format in Promela [12]⁶ Figures 7 and 8, respectively, illustrate the generated $B(\phi)$ and $B(\neg\phi)$ automata for the given scheduler logic-based policy rule.

We tested Sechduler for more complex temporal logic predicates such as workflow-based security policy rules for server systems where individual processes are spawned to carry out a particular task, such as *receive request* and *send out response*. In particular, an example policy may require a security check completion after any sensitive data access before the response is sent out to the network. The generated automaton is illustrated in Figure 10 which includes 8 states and 20 transitions and the original policy is shown in Figure 11. Furthermore, for a generic evaluation of the automata size range that Sechduler has to process in real-world settings, we measured the size of generated automata for typical and frequently used linear temporal logic-based software specification formula⁷ introduced in [7]. Table I illustrates few of the individual temporal security policies that Sechduler selected, combined, and used in our experiments⁸.

As Sechduler verifies whether each scheduled tasks should be given CPU access, we collected statistics of the kernel-level scheduled tasks during a normal host computer usage session. Figure 6(b) shows the number of scheduled tasks for each second during the session. In particular, the session included a Web browser launch followed by an Office document editor application spawn. As demonstrated, the number of scheduled tasks can go up to 18K per second during a normal computer usage session. We measured the time requirements for the policy-to-automata conversion for the typical linear temporal logic-based system specification policies [7]. Figure 6(c) shows the results for individual temporal security policies. Sechduler completed the conversion for individual temporal requirements in approximately 0.58 seconds on average. This suggests that Sechduler can scale well for real-world settings where many requirements may be involved in the final logic-based predicate.

Case Study: Sensitive File Modification. In this section, we show how Sechduler protects a target host system once the system is hit by a sensitive file modification attack. Samhain was deployed as the attack conse-

⁶This result can be fed into the Spin model checker to verify the system security properties in an offline manner; however, in this paper, we only focus on the runtime verification of the system during its execution time.

⁷<http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml> (Figure 6(a)).

⁸The weak until operator W is related to the strong until operator U by the following equivalences: $pWq = ([p])(pUq) = \neg([p] \rightarrow \neg(pUq)) = pU(q \vee \neg p)$.

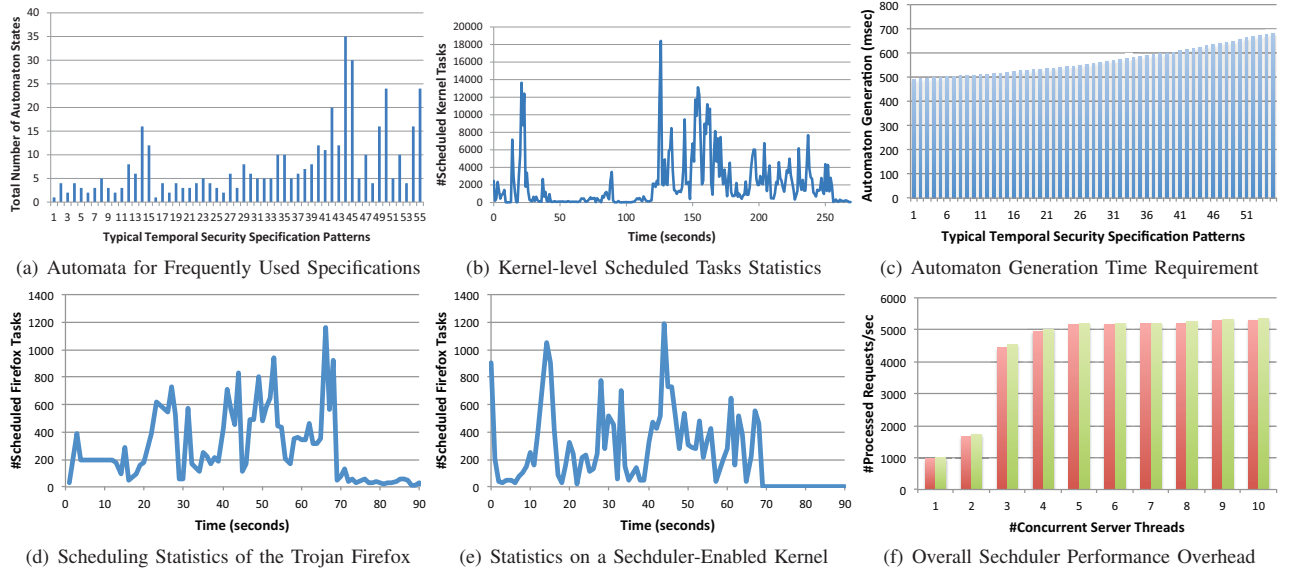


Fig. 6. Sechduler Evaluation Results

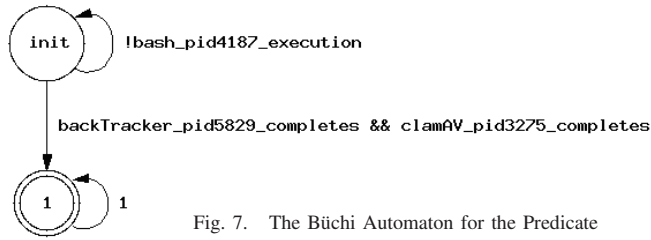


Fig. 7. The Büchi Automaton for the Predicate

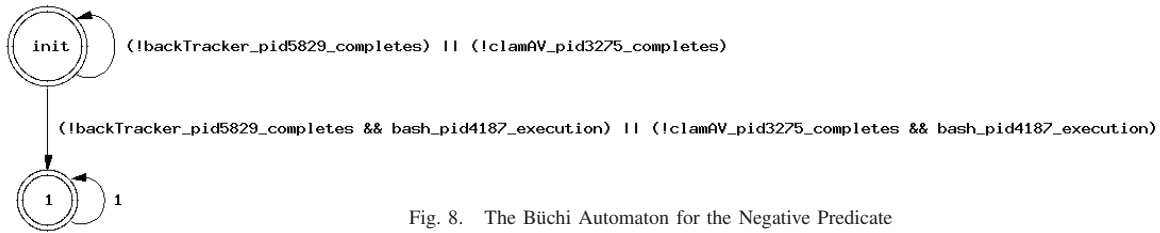


Fig. 8. The Büchi Automaton for the Negative Predicate

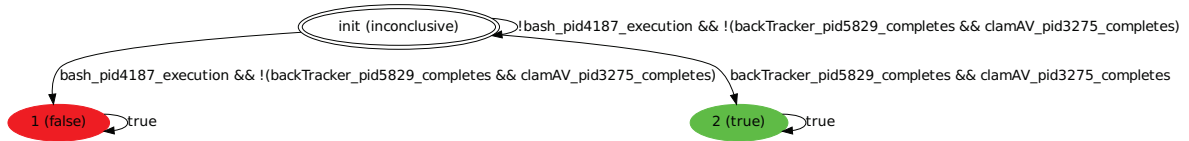


Fig. 9. The Generated Scheduler Monitor to be Loaded on the Kernel

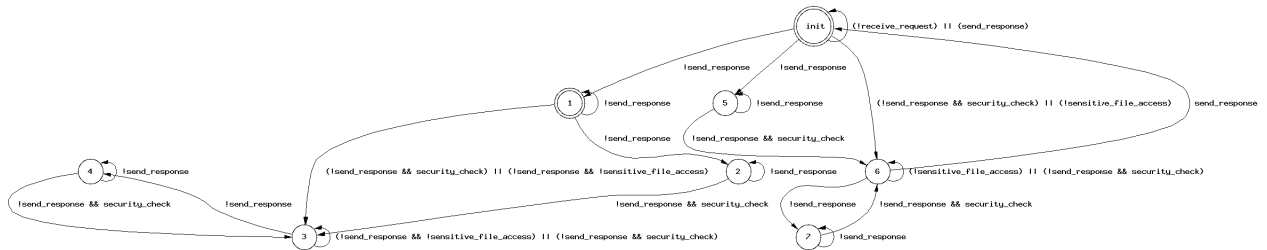


Fig. 10. The Workflow-Based Automaton to Prevent Potential Data Disclosures

TABLE I
SOME OF THE TYPICAL TEMPORAL SECURITY POLICIES USED IN THE EXPERIMENTS

Description	Policy
Process P is suspended (does not execute)	$\llbracket (!P)$
P always executes on core 2 after Q until R	$\llbracket (Q \& !R \rightarrow (PWR))$
Process P executes at some point between Q and R	$\llbracket (Q \& !R \rightarrow (!RW(P \& !R)))$
Processes S and T without Z execute after P	$\llbracket (P \rightarrow \langle \rangle (S \& !Z \& o(!ZUT)))$
Process P causes execution of S and T before R	$\langle \rangle R \rightarrow ((!(S \& (!R) \& o(!RU(T \& !R))))U(R P))$

quence detection system. Specifically, we modified its configuration, i.e., `/etc/samhain/samhainrc`, to monitor the files and directories in which we are interested, and configured it to report events with at least `crit` severity level. Before the experiments, we created its initial database, i.e., `/var/state/samhain/samhain_file`, using `samhain -t init`, and its database was updated, using `-t update`. During the normal operational mode of the system, Samhain was configured to check the marked sensitive files and directories against its database and fire an alert upon identifying a modification or access (depending on the policy defined in the configuration file).

To simulate an attack, we implemented a trojan Firefox that modified sensitive user files that had been marked to be monitored by Samhain. Figure 6(d) shows the malware’s scheduling activity statistics within a non-Sechduler aware kernel. Consequently, Samhain fired the alert illustrated by Figure 12. Upon the receipt of the alert, Sechduler performed three tasks. It 1) called the `setsec` system call and lowers the Firefox’s security level variable within the kernel; 2) spawned a comprehensive ClamAV virus scan on the Firefox’s executable; and 3) compiled the triggered alert’s corresponding policy module and loaded it on the kernel dynamically.

Enforcing the loaded policy, Sechduler manipulated the task selection procedure within the kernel scheduler to ensure that (from its point of view) the potentially malicious Firefox process did not get CPU access and waited for the ClamAV’s green light. However, in our experiments, ClamAV triggered an alert denoting that the executable contains malicious content. Consequently, the suspended Firefox process was terminated by Sechduler and its executable was removed. Figure 6(e) a different run of the trojan Firefox on a Sechduler-enabled Linux kernel. As shown on the graph, Sechduler denies its requests for execution since the 69-th seconds and finally terminates the process. We implemented the process termination as a single countermeasure action; however, more complicated actions can be defined by policies and implemented. We consider that direction outside the scope of this paper and will investigate different possibilities as future work.

It is important that Sechduler performs the runtime system security verification efficiently such that the system’s overall throughput is not affected significantly. We measured the Sechduler’s overall performance overhead on our testbed system’s overall throughput. In particular, we employed the `ab` Apache Webserver benchmarking toolset to measure the system throughput. To make the webpage processing more CPU-intensive, we designed a very simple HTML webpage. For our server system, we define the overall performance measure as the number of requests that can be processed per second. Figure 6(f) shows how the system’s throughput is affected by the runtime verification of individual task scheduling decisions. We believe that the overall performance overhead of the Sechduler solution can be further reduced by optimizing our code. For instance, several data structures that are searched frequently, with $O(n)$ complexity, can be

redesigned for logarithmic search, and overall system performance improvement.

VIII. RELATED WORK

There has been several past work on resource-aware intrusion detection and forensics analyses. Mukkamala et al. [16] introduce a light-weight intrusion detection algorithm based on artificial neural networks to discover sources of information breaches. Carrier [5] presents an on-demand smart filesystem-based detection techniques to determine the source of security breaches by investigating their effects on the file-objects, e.g., file modification dates. Taser [9] is an operator-assisted post-intrusion forensics system based on pessimistic taint tracking; hence, it may result in a large set of possible attack sources. BackTracker [13] and Panorama [31] aid off-line forensic analysis by producing taint-traces of file and process communications that led to a detected security breach. Additionally, few frameworks has recently been proposed for semi-realtime system security. For instance, RRE [36] and EliMet [34] present game-theoretic intrusion response and recovery capabilities that are indeed best-effort solutions to select and carry out optimal response actions, i.e., security services and not the system’s core functionalities, as quickly as possible. Because none of these security tools are aware of the mission-specific realtime requirements, they focus only on enhancement and/or maintenance of the target system’s security, and hence do not necessarily guarantee realtime task accomplishments.

Conventional realtime scheduling algorithms such as Rate Monotonic (RM) algorithm [15], Earliest Deadline First (EDF) [26], and Spring scheduling algorithm [21], [32] have been successfully applied in realtime systems. Previous work has been done to facilitate realtime computing in heterogeneous systems. Huh et al. proposed an approach to dynamically managing resources in realtime systems [11]. Santos et al. developed a probabilistic model for a client/server multimedia system [23]. However, most of existing realtime scheduling algorithms perform poorly for realtime and security-critical applications due to the oversight and ignorance of security requirements imposed by the applications. Xie et al. [30] extends the EDF algorithm to takes into account performance overhead of various security solutions before their execution. The proposed model adds up the given numeric values that represent the individual tasks’ overheads, realtime priorities, and the amount of their benefits to the overall system security. The model consequently selects the task with the maximum aggregated value of the total benefit for execution. Despite its theoretical contributions, the solution require a lot of numeric input values and has not been validated empirically.

The temporal dimension in access control solutions has also attracted a great amount of interest recently. Bertino et al. [3] introduce temporal role-based access control that supports periodic role enabling and disabling possibly with individual exceptions for particular users and temporal dependencies among such actions, expressed by means of role triggers. Zhu

```
CRIT: [2012-11-05T21:53:01-0500] msg= POLICY [IgnoreNone], path=/home/user/sensitive.db, inode_old=824046,
inode_new=789469, dev_old=8,1, dev_new=8,1, size_old=123, size_new=127, atime_old=[2012-11-06T02:43:54],
atime_new=[2012-11-06T02:50:58], mtime_old=[2012-11-06T02:43:45], mtime_new=[2012-11-06T02:49:44],
chksum_old=87C08...50EB7, chksum_new=F38...3E92
```

Fig. 12. A Sample Alert by the Samhain File Integrity Checker

et al. [33] propose a temporal attribute-based access control for the cloud environments in which each outsourced resource is associated with an access policy on a set of temporal attributes, e.g., period-of-validity, opening hours, or hours of service. The main drawback of the abovementioned solutions is that they do not take into account the low system-level realtimeness requirements.

Currently, to the best of our knowledge, none of the past solutions considered both logic-based security policies and realtimeness criteria simultaneously and efficiently proved by a complete real-world evaluation. Furthermore, in the past work, decisions upon the system security and the system realtimeness were made at different semantic levels within the target system. More specifically, there are many host-based security solutions that monitor, analyze, detect and respond to intrusions in application levels, whereas the system realtimeness is usually resolved in the kernel's scheduler; therefore, there is no common point where the tradeoff can be resolved optimally and efficiently. For instance, the intrusion detection engine, due to lack of information, may choose to launch a complete and tedious system-wide filesystem scan when the realtime requirements are barely satisfied because of extremely tight hard task deadlines.

IX. CONCLUSIONS

In this paper, we presented Sechduler, a security-aware kernel scheduler that takes into account not only the realtime requirements of the waiting tasks but also the predefined set of temporal system-wide security policies. As empirically shown, Sechduler can efficiently process and enforce the predefined temporal security policies. We believe that Sechduler opens a new research direction to formally consider realtime and security requirements of a target system simultaneously and adaptively within the operating system kernel. As future work, we currently investigate the possibility of offline formal and combined realtimeness-security verification of a given system.

REFERENCES

- [1] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing ltl semantics for runtime verification. *J. Log. and Comput.*, 20(3):651–674, June 2010.
- [2] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011.
- [3] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. Trbac: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, August 2001.
- [4] H. Bevrani and T. Hiyama. *Intelligent Automatic Generation Control*. CRC Press, 2011.
- [5] Brian Carrier. *File System Forensic Analysis*. Addison-Wesley Professional, 2005.
- [6] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Form. Methods Syst. Des.*, 1(2-3):275–288, October 1992.
- [7] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering, ICSE '99*, pages 411–420, New York, NY, USA, 1999. ACM.
- [8] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, 1996. Chapman & Hall, Ltd.
- [9] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The taser intrusion recovery system. In *SOSP*, pages 163–76, 2005.
- [10] John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.
- [11] Eui-Nam Huh, Lonnie R. Welch, Behrooz A. Shirazi, and Charles D. Cavanaugh. Heterogeneous resource management for dynamic real-time systems. In *Heterogeneous Computing Workshop*, 2000.
- [12] Ke Jiang and Bengt Jonsson. Using spin to model check concurrent algorithms, using a translation from c to promela. In *Proc. 2nd Swedish Workshop on Multi-Core Computing*, pages 67–69. Department of Information Technology, Uppsala University, 2009.
- [13] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the Nineteenth ACM symposium on Operating systems principles*, volume 37, pages 223–236, 2003.
- [14] S.T. King and P.M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems (TOCS)*, 23(1):51–76, 2005.
- [15] C. L. Liu and James W. Layland. Scheduling algorithms for multi-programming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.
- [16] Srinivas Mulkamala and Andrew H. Sung. Identifying significant features for network forensic analysis using artificial intelligent techniques. *International Journal of Digital Evidence*, 1:1–17, 2005.
- [17] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44–66, 2003.
- [18] Department of Homeland Security, 2007. Staged cyber attack reveals vulnerability in power grid, Available at <http://www.cnn.com/2007/US/09/26/power.at.risk/index.html>.
- [19] Chandandeep Singh Pabla. Completely fair scheduler. *Linux J.*, 2009(184), August 2009.
- [20] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.
- [21] K. Ramamritham and J. A. Stankovic. Dynamic task scheduling in hard real-time distributed systems. *IEEE Softw.*, 1(3):65–75, July 1984.
- [22] Ramani Routray, Rui Zhang, David Eysers, Douglas Willcocks, Peter Pietzuch, and Prasenjit Sarkar. Policy generation framework for large-scale storage infrastructures. In *IEEE Symposium on Policies for Distributed Systems and Networks*, pages 65–72, 2010.
- [23] R. M. Santos, J. Santos, and J. Orozco. Scheduling heterogeneous multimedia servers: different qos for hard, soft and non real-time clients. In *Euromicro conference on Real-time systems*, pages 247–253, 2000.
- [24] H. Sato and T. Yakoh. A real-time communication mechanism for rtlinux. In *Annual Conference of the IEEE Industrial Electronics Society*, volume 4, pages 2437–2442 vol.4, 2000.
- [25] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the International Conference on Information System Security (ICISS)*, pages 1–25, 2008.
- [26] John A. Stankovic, Krithi Ramamritham, and Marco Spuri. *Deadline Scheduling for Real-Time Systems: Edf and Related Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.
- [27] Robert Tarjan. Depth-first search and linear graph algorithms. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 114–121, oct. 1971.
- [28] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *VIII Banff Higher order workshop conference on Logics for concurrency*, pages 238–266, 1996.
- [29] Brian Wotrny, Bruce Potter, Marcus Ranum, and Rainer Wichmann. *Host Integrity Monitoring Using Osiris and Samhain*. Syngress Publishing, 2005.
- [30] Tao Xie, Xiao Qin, and Andrew Sung. SAREC: A Security-Aware Scheduling Strategy for Real-Time Applications on Clusters. In *International Conference on Parallel Processing*, pages 5–12, 2005.
- [31] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 116–127. ACM, 2007.
- [32] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Trans. Comput.*, 36(8):949–960, August 1987.
- [33] Yan Zhu, Hongxin Hu, Gail-Joon Ahn, Dijiang Huang, and Shan-Biao Wang. Towards temporal access control in cloud computing. In Albert G. Greenberg and Kazem Sohraby, editors, *INFOCOM*, pages 2576–2580. IEEE, 2012.
- [34] S. Zonouz, A. Houmansadr, and P. Haghani. EliMet: Security metric elicitation in power grid critical infrastructures by observing system administrators' responsive behavior. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, 2012.
- [35] Saman A. Zonouz, Kaustubh R. Joshi, and William H. Sanders. Flo-guard: Cost-aware systemwide intrusion defense via online forensics and on-demand ids deployment. In *SAFECOMP; Lecture Notes in Computer Science*, pages 338–354, 2011.
- [36] Saman A. Zonouz, Himanshu Khurana, William H. Sanders, and Tim M. Yardeley. RRE: A game-theoretic intrusion response and recovery engine. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 439–448, 2009.
- [37] Saman Aliari Zonouz, Kaustubh R. Joshi, and William H. Sanders. Flo-guard: cost-aware systemwide intrusion defense via online forensics and on-demand ids deployment. In *International conference on Computer safety, reliability, and security*, pages 338–354, 2011.