Intrusion Survivability for Commodity Operating Systems

Ronny Chevalier, David Plaquin, Chris Dalton, Guillaume Hiet

Intrusion Survivability for Commodity Operating Systems

RONNY CHEVALIER, HP Labs and CentraleSupélec, Inria, CNRS, IRISA
DAVID PLAQUIN and CHRIS DALTON, HP Labs
GUILLAUME HIET, CentraleSupélec, Inria, CNRS, IRISA

Despite the deployment of preventive security mechanisms to protect the assets and computing platforms of users, intrusions eventually occur. We propose a novel intrusion survivability approach to withstand ongoing intrusions. Our approach relies on an orchestration of fine-grained recovery and per-service responses (e.g., privileges removal). Such an approach may put the system into a degraded mode. This degraded mode prevents attackers to reinfect the system or to achieve their goals if they managed to reinfect it. It maintains the availability of core functions while waiting for patches to be deployed. We devised a cost-sensitive response selection process to ensure that while the service is in a degraded mode, its core functions are still operating. We built a Linux-based prototype and evaluated the effectiveness of our approach against different types of intrusions. The results show that our solution removes the effects of the intrusions, that it can select appropriate responses, and that it allows services to survive when reinfected. In terms of performance overhead, in most cases, we observed a small overhead, except in the rare case of services that write many small files asynchronously in a burst, where we observed a higher but acceptable overhead.

CCS Concepts: • **Security and privacy → Operating systems security**; **Malware and its mitigation**;

Additional Key Words and Phrases: Intrusion survivability, intrusion response, intrusion recovery

ACM Reference format:
Ronny Chevalier, David Plaquin, Chris Dalton, and Guillaume Hiet. 2020. Intrusion Survivability for Commodity Operating Systems. *Digit. Threat.: Res. Pract.* 1, 4, Article 21 (December 2020), 30 pages.
https://doi.org/10.1145/3419471

1 INTRODUCTION

Despite progress in preventive security mechanisms such as cryptography, secure coding practices, or network security, given time, an intrusion will eventually occur. Such a case may happen due to technical reasons (e.g., a misconfiguration, a system not updated, or an unknown vulnerability) and economic reasons [49] (e.g., do the benefits of an intrusion for criminals outweigh their costs?). It means that we should not only build systems to prevent intrusions, but also to detect and survive them.

Authors' addresses: R. Chevalier, HP Labs, CentraleSupélec, Inria, CNRS, IRISA; email: ronny.chevalier@hp.com; D. Plaquin, HP Labs; email: david.plaquin@hp.com; C. Dalton, HP Labs; email: cid@hp.com; G. Hiet, CentraleSupélec, Inria, CNRS, IRISA; email: guillaume.hiet@centralesupelec.fr.

2576-5337/2020/12-ART21
https://doi.org/10.1145/3419471

The idea of Intrusion Detection Systems (IDSs) dates back to the 1980s [1, 21]. Since then, more intrusion detection approaches were introduced, refined, and transferred from academia to industry. Most of today's commodity Operating Systems (OSs) can be deployed with some kind of Intrusion Detection System (IDS). However, as the name suggests, IDSs only focus on the detection and do not provide the ability to survive or withstand an intrusion once it has been detected.

To limit the damage done by security incidents, intrusion recovery systems help administrators restore a compromised system into a sane state. Common limitations are that they do not preserve availability [31, 35, 44] (e.g., they force a system shutdown) or that they neither stop intrusions from reoccurring nor withstand reinfections [31, 35, 44, 84, 87]. If the recovery mechanism restores the system to a sane state, the system continues to run with the same vulnerabilities and nothing stops attackers from reinfecting it. Thus, the system could enter a loop of infections and recoveries.

Existing intrusion response systems, however, apply responses [27] to stop an intrusion or limit its impact on the system; but existing approaches apply coarse-grained responses that affect the whole system and not just the compromised services [27] (e.g., blocking port 80 for the whole system, because a single compromised service uses this port maliciously). They also rely on a strong assumption of having complete knowledge of the vulnerabilities present and used by the attacker [27, 73] to select responses.

These limitations mean that they cannot respond to intrusions without affecting the availability of the system or of some services. Whether it is due to business continuity, safety reasons, or the user experience, the availability of services is an important aspect of a computing platform. For example, while web sites, code repositories, or databases are not safety-critical, they can be important for a company or for the workflow of a user. Therefore, the problem that we address is the following: How to design an Operating System (OS) so its services can survive ongoing intrusions while maintaining availability?

Our approach distinguishes itself from prior work on three fronts. First, we *combine* the restoration of files and processes of a service with the ability to apply responses after the restoration to withstand a reinfection. Second, we apply *per-service* responses that affect the compromised services instead of the whole system (e.g., only one service views the file system as read-only). Third, after recovering a compromised service, the responses we apply can put the recovered service into a *degraded mode*, because they remove some privileges normally needed by the service.

The degraded mode is introduced on purpose. When the intrusion is detected, we do not have precise information about the vulnerabilities exploited to patch them or we do not have a patch available. The degraded mode allows the system to survive the intrusion for two reasons. First, after the recovery, the degraded mode either stops the attackers from reinfecting the service or from achieving their malicious goals. Second, it keeps as many functions of the service available as possible, thus maintaining availability while waiting for a patch.

We maintain the availability by ensuring that *core functions* of services are still operating, while non-essential functions might not be working due to some responses. For example, a web server could have "provide read access to the website" as core function and "log accesses" as non-essential. Thus, if we remove the write access to the file system it would degrade the service's state (i.e., it cannot log anymore), but we would still maintain its core function. We developed a cost-sensitive response selection where administrators describe a policy consisting of cost models for responses and malicious behaviors. Our solution then selects a response that maximizes the effectiveness while minimizing its impact on the service based on the policy.

This approach gives time for administrators to plan an update to fix the vulnerabilities (e.g., wait for a vendor to release a patch). Finally, once they patched the system, we can remove the responses, and the system can leave the degraded mode.

Contributions. Our main contributions are the following:

- We propose a novel intrusion survivability approach to withstand ongoing intrusions and maintain the availability of core functions of services (Sections 3.1 and 4).

- We introduce a cost-sensitive response selection process to help select optimal responses (Section 5).
- We develop a Linux-based prototype implementation by modifying the Linux kernel, systemd [77], CRIU [17], Linux audit [38], and snapper [75] (Section 6).
- We evaluate our prototype by measuring the effectiveness of the responses applied, the ability to select appropriate responses, the availability cost of a checkpoint and a restore, the overhead of our solution, and the stability of the degraded services (Section 7).

Outline. The rest of this article is structured as follows: First, in Section 2, we mention related concepts about our work, and we review the state-of-the-art on intrusion recovery and response systems. In Section 3, we give an overview of our approach, and we define the scope of our work. In Section 4, we specify the requirements and architecture of our approach. In Section 5, we describe how we select cost-sensitive responses and maintain core functions. In Section 6, we describe a prototype implementation that we then evaluate in Section 7. In Section 8, we discuss some limitations of our work, and we give a summary of the comparison with the related work. We conclude and give the next steps regarding our work in Section 9.

2 RELATED WORK

In this section, we first discuss related concepts close to our work such as dependability, resiliency, or survivability. Then, we review existing work on intrusion recovery and response systems.

2.1 Related Concepts

Our contributions and the idea behind this work in general are related to concepts such as security, dependability, resiliency, and survivability. As a reference, we recommend the work of Avizienis et al. [6] that defined, compared, and summarized most of these concepts. Here, we give an overview of one core concept related to our work: survivability. For more details, the reader can consult the aforementioned references.

Survivability. Ellison et al. defined survivability as "the capability of a system to fulfill its mission, in a timely manner, in the presence of attacks, failures, or accidents" [26]. Avizienis et al. [6] suggested that—based on this definition—dependability and survivability were similar concepts, where dependability is "the ability of a system to avoid service failures that are more frequent or more severe than is acceptable" [6]. Knight et al. [47] weighed that survivability distance itself from dependability, since it should encompass the notion of degraded service and a tradeoff between the availability of some functions and the cost to maintain and provide them. Intuitively, Knight et al. defined survivability as the ability "to provide one or more alternate services (different, less dependable, or degraded) in a given operating environment" [47] essentially providing "a tradeoff between functionality and resources" [47].

In the rest of this article, we assume that survivability refers to such a degradation and tradeoff. More specifically, since we care about *intrusion survivability*, we consider the tradeoff to be between the availability of the different functionalities of a vulnerable service and the security risk associated to maintaining them.

Moreover, the definition of survivability has always been applied to networked systems or critical information systems. In our case, however, we apply the concept of intrusion survivability to commodity OSs (e.g., Linux-based distributions or Windows).

2.2 Intrusion Recovery Systems

Intrusion recovery systems [31, 35, 44, 84, 87] focus on system integrity by recovering legitimate persistent data. Except SHELF [87] and CRIU-MR [84], the previous approaches do not preserve availability, since their restore procedure forces a system shutdown, or they do not record the state of the processes (e.g., their memory and the value of their CPU registers). Furthermore, except for CRIU-MR, they log all system events to later replay legitimate operations [31, 44, 87] or rollback illegitimate ones [35], thus providing a fine-grained recovery. Such an approach, however, generates gigabytes of logs per day, inducing a high storage cost.

Most related to our work are SHELF [87] and CRIU-MR [84]. SHELF recovers the state of processes and identifies infected files using a log of system events. During recovery, it quarantines infected objects by freezing processes or forbidding access to files. SHELF, however, removes this quarantined state as soon as it restores the system, allowing an attacker to reinfect the system. In comparison, our approach applies responses after a restoration to prevent reinfection or the reinfected service to cause damages to the system.

CRIU-MR restores infected systems running within a Linux container. When an IDS notifies CRIU-MR that a container is infected, it checkpoints the container's state (using CRIU [17]) and identifies malicious objects (e.g., files) using a set of rules. Then, it restores the container while omitting the restoration of the malicious objects. CRIU-MR differs from other approaches, including ours, because it uses a checkpoint immediately followed by a restore, only to remove malicious objects (as opposed to checkpointing when there is no suspected intrusion and restoring later when there is an intrusion).

The main limitation affecting prior work, including SHELF and CRIU-MR, is that they neither prevent the attacker from reinfecting the system nor allow the system to withstand a reinfection, since vulnerabilities are still present.

2.3 Intrusion Response Systems

Having discussed systems that recover from intrusions and showed that it is not enough to withstand them, we now discuss intrusion response systems [8, 27, 73] that focus on applying responses to limit the impact of an intrusion.

One area of focus of prior work is on how to model intrusion damages, or response costs, to select responses. Previous approaches either rely on directed graphs about system resources and cost models [8], on attack graphs [27], or attack defense trees [73]. Shameli-Sendi et al. [73] use Multi-Objective Optimization (MOO) methods to select an optimal response based on such models.

A main limitation, and difference with our work, is that these approaches apply system-wide or coarse-grained responses that affect every application in the OS. Our approach is more fine-grained, since we select and apply per-service responses that only affect the compromised service and not the rest of the system. Moreover, these approaches cannot restore a service to a sane state. Our approach, however, combines the ability to restore and to apply cost-sensitive per-service responses.

Some of these approaches [27, 73] also rely on the knowledge of vulnerabilities present on the system and assume that the attacker can only exploit these vulnerabilities. Our approach does not rely on such prior knowledge, but relies on the knowledge that an intrusion occurred and the malicious behaviors exhibited by this intrusion.

Huang et al. [36] proposed a closely related approach that mitigates the impact of waiting for patches when a vulnerability is discovered. However, their system is not triggered by an IDS but only by the discovery of a vulnerability. They instrument or patch vulnerable applications so they do not execute their vulnerable code, thus losing some functionality (similar to a degraded state). They generate workarounds that minimize the cost of losing a functionality by reusing error-handling code already present. In our work, however, we do not assume any knowledge about the vulnerabilities, and we do not patch or modify applications.

3 PROBLEM SCOPE

This section first gives an overview of our approach (illustrated in Figure 1), then it describes our threat model and some assumptions that narrow the attack scope.

3.1 Approach Overview

Since we focus our research on intrusion survivability, our work starts when an IDS detects an intrusion in a service.

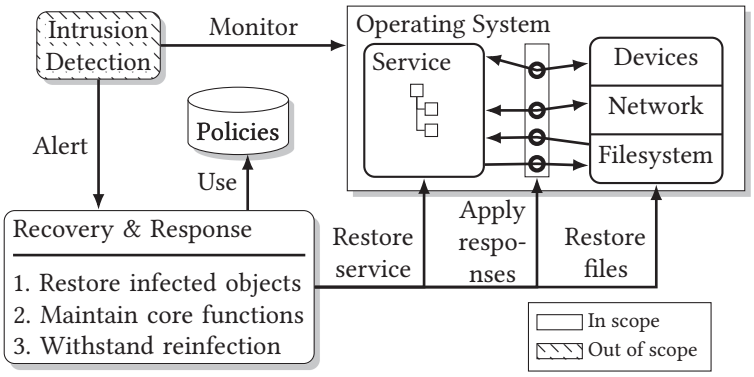


Fig. 1. High-level overview of our intrusion survivability approach.

When the IDS detects an intrusion, we trigger a set of responses. The procedure must meet the following goals: restore infected objects (e.g., files and processes), maintain core functions, and withstand a potential reinfection. We achieve these goals using recoveries, responses, and policies.

Recovery. Recovery actions restore the state of a service (i.e., the state of its processes and metadata describing the service) and associated files to a previous safe state. To perform recovery actions, we create periodic snapshots of the file system and the services during the normal operation of the OS. We also log all the files modified by the monitored services. Hence, when restoring services, we only restore the files they modified. This limits the restoration time and it avoids the loss of known legitimate and non-infected data.[1] To perform recovery actions, we do not require for the system to be rebooted, and we limit the availability impact on the service.

Response. A response action removes privileges, isolates components of the system from the service, or reduces resource quotas (e.g., CPU or RAM) of one service. Hence, it does not directly affect any other component of the system (including other services).[2] Its goal is to either prevent an attacker to reinfect the service or to withstand a reinfection by stopping attackers from achieving their goals (e.g., compromising data availability or integrity) after the recovery. Such a response may put the service into a degraded mode, because some functions might not have the required privileges anymore (or limited access to some resources).

Policies. We apply appropriate responses that do not disable core functions (e.g., the ability to listen on port 80 for a web server). To refine the notion of core functions, we rely on policies. Their goal is to provide a tradeoff between the availability of a function (that requires specific privileges) in a service and the intrusion risk. We designed a cost-sensitive response selection process (see Section 5) based on such policies. Administrators, maintainers, and developers of the services can provide the policies by specifying the cost of losing specific privileges (i.e., if we apply a response) and the cost of a malicious behavior (exhibited by an intrusion).

3.2 Threat Model and Assumptions

We make assumptions regarding the platform's firmware (e.g., BIOS or UEFI-compliant firmware) and the OS kernel where we execute the services. If attackers compromise such components at boot time or runtime, they could compromise the OS including our mechanisms. Hence, we assume their integrity. Such assumptions are

[1]Other files that the service depends on can be modified by another service; we handle such a case with dependencies information between services.
[2]However, if components depend on a degraded service, they can be affected indirectly.

reasonable in recent firmware using a hardware-protected root of trust [34, 68] at boot time and protection of firmware runtime services [14, 88, 89]. For the OS kernel, one can use UEFI Secure Boot [81] at boot time, and rely on, e.g., security invariants [76] or a hardware-based integrity monitor [7] at runtime. The main threat that we address is the compromise of services inside an OS.

We make no assumptions regarding the privileges that were initially granted to the services. Some of them can restrict their privileges to the minimum. On the contrary, other services can be less effective in adopting the principle of least privilege. The specificity of our approach is that we deliberately remove privileges that could not have been removed initially, since the service needs them for a function it provides. Finally, we assume that the attacker cannot compromise the mechanisms we use to checkpoint, restore, and apply responses (Section 4 details how we protect such mechanisms).

We model an attacker with the following capabilities:

- Can find and exploit a vulnerability in a service,
- Can execute arbitrary code in the same context as the compromised service,
- Can perform some malicious behaviors even if the service had initially the minimum amount of privileges to accomplish its functions,
- Can compromise a privileged service or elevate the privileges of a compromised service to superuser,
- Cannot exploit software-triggered hardware vulnerabilities[3] (e.g., side-channel attacks [45, 48, 51, 70]),
- Do not have physical access to the platform.

4 ARCHITECTURE AND REQUIREMENTS

Our approach relies on four components. In this section, we first give an overview of how each component works and interacts with the others, as illustrated in Figure 2. Then, we detail requirements about our architecture.

4.1 Overview

During the normal operation of the OS, the *service manager* creates periodic checkpoints of the services and snapshots of the file system. In addition, a *logging facility* logs the path of all the files modified by the monitored services since their last checkpoint. The logs are later used to filter the files that need to be restored.

The *IDS* notifies the *response selection* component when it detects an intrusion and specifies information about possible responses to withstand it. The selected responses are then given to the service manager. The service manager restores the infected service[4] to the last known safe state including all the files modified by the infected service. Then, it configures kernel-enforced per-service privilege restrictions and quotas based on the selected responses.

Finally, we rely on a static Mandatory Access Control (MAC) policy to isolate our components. This policy is also enforced by the kernel, but in comparison to the per-service responses, it does not change over time.

4.2 Last Known Safe State

To select the last known safe state, we rely on the IDS to identify the first alert related to the intrusion and to provide us with a timestamp. Then, we consider that the first state prior to the timestamp of this alert is safe. In practice, however, the IDS might not be aware of the intrusion until a certain point in time. For example, if an intrusion happens at time t, but the IDS is only aware of this intrusion at time $t + 2$, we would select the state taken at $t + 1$ as the last known safe state. Therefore, we would restore the state of the service to an infected state.

[3]While the exploitation of such vulnerabilities could be detected and our solution could help in surviving intrusions relying on them, they could also be used to bypass the protections of our mechanisms. Therefore, we exclude them from our threat model at the moment.
[4]We could also take a snapshot just before restoring the infected service to a previous state. This snapshot would be helpful to perform further offline forensic analyses. Such an approach, however, was not implemented in our prototype.

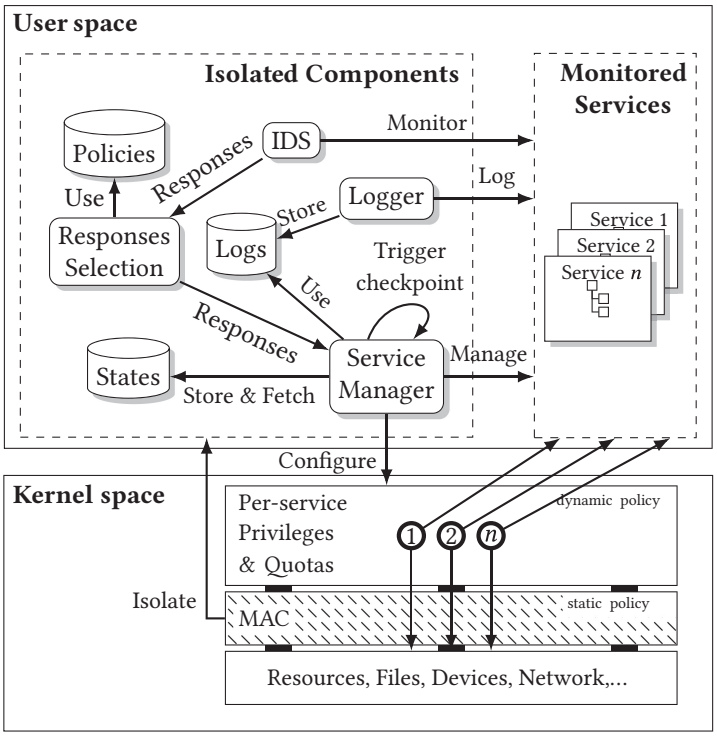


Fig. 2. Overview of the architecture.

Nevertheless, even if the restored state is infected because the IDS was not aware of the intrusion at the time, we apply responses. The per-service privilege restrictions and quotas that we apply stop the exploitation of a vulnerability (i.e., the attacker is not able to reinfect the service) or withstand a reinfection (i.e., attackers successfully reinfected the service, but they are restricted).

4.3 Isolation of the Components

For our approach to be able to withstand an attacker trying to impede the detection and recovery procedures, the integrity and availability of each component is crucial. Different solutions (e.g., a hardware isolated execution environment or a hosted hypervisor) could be used. In our case, we rely on a kernel-based Mandatory Access Control (MAC) mechanism, such as SELinux [65], to isolate the components we used. Such a mechanism is available in commodity OSs, can express our isolation requirements, and does not modify the applications. We now give guidelines on how to build a MAC policy to protect our components.

First, the MAC policy must ensure that none of our components can be stopped.[5] Otherwise, e.g., if the responses selection component is not alive, no responses will be applied.

Second, the MAC policy must ensure that only our components have access to their isolated storage (e.g., to store the logs or checkpoints). Otherwise, attackers might e.g., erase an entry to avoid restoring a compromised file.

Third, the MAC policy must restrict the communication between the different components, and it must only allow a specific program to advertise itself as one of the components. Otherwise, an attacker might impersonate

[5]One can also use a watchdog to ensure that the components are alive.

a component or stop the communication between two components. In our case, we assume a Remote Procedure Call (RPC) or an Inter-Process Communication (IPC) mechanism that can implement MAC policies (e.g., D-Bus [19] is SELinux-aware [83]).

4.4 Intrusion Detection System

Our approach requires an IDS to detect an intrusion in a monitored service. We do not require a specific type of IDS. It can be external to the system or not. It can be misuse-based or anomaly-based. We only have two requirements.

First, the IDS should be able to pinpoint the intrusion to a specific service to apply per-service responses. For example, if the IDS analyzes event logs to detect intrusions, they should include the service that triggered the event.

Second, the IDS should have information about the intrusion. It should map the intrusion to a set of malicious behaviors (e.g., the malware capabilities [59] from Malware Attribute Enumeration and Characterization (MAEC) [46]), and it should provide a set of responses that can stop or withstand them. Both types of information can either be part of the alert from the IDS or be generated from threat intelligence based on the alert. Generic responses can also be inferred due to the type of intrusion if the IDS lacks precise information about the intrusion. For example, a generic response for ransomware consists in setting the filesystem hierarchy as read-only. Information about the alert, the responses, or malicious behaviors, can be shared using standards such as Structured Threat Information eXpression (STIX) [9] and Malware Attribute Enumeration and Characterization (MAEC) [46, 61].

4.5 Service Manager

Commodity OSs rely on a user space service manager (e.g., the Service Control Manager [69] for Windows, or systemd [77] for Linux distributions) to launch and manage services. In our architecture, we rely on such a service, since it provides the appropriate level of abstraction to manage services and it has the notion of dependencies between services. Using such information, we can restore services in a coherent state. If a service depends on other services (e.g., if one service writes to a file and another one reads it), we checkpoint and restore them together.

We extend the service manager to checkpoint and restore the state of services. Furthermore, we modify the service manager so it applies responses before it starts a recovered service. Since such responses are per-service, the service manager must have access to OS features to configure per-service privileges and resource quotas.

The service manager must be able to kill a service (i.e., all alive processes created by the service) if it is compromised and needs to be restored. Therefore, we bound processes to the service that created them, and they must not be able to break the bound. For example, we can use cgroups [32] in Linux or job objects [56] in Windows.

Finally, the MAC policy must ensure that only the service manager manages the collections of processes (e.g., `/sys/fs/cgroup` in Linux). Otherwise, if an attacker breaks the bound of a compromised service, it would be difficult to kill the escaped processes. Likewise, the MAC policy must protect configuration files used by the service manager.

5 COST-SENSITIVE RESPONSE SELECTION

For a given intrusion, multiple responses might be appropriate, and each one incurs an availability cost. We devised a cost-sensitive response selection process that minimizes such a cost and maintains the core functions of the service.

We use a qualitative approach using linguistic constants (e.g., low or high) instead of a quantitative one (e.g., monetary values). Quantitative approaches require an accurate value of assets and historical data of previous intrusions to be effective, which we assume missing. Qualitative approaches, while prone to biases and

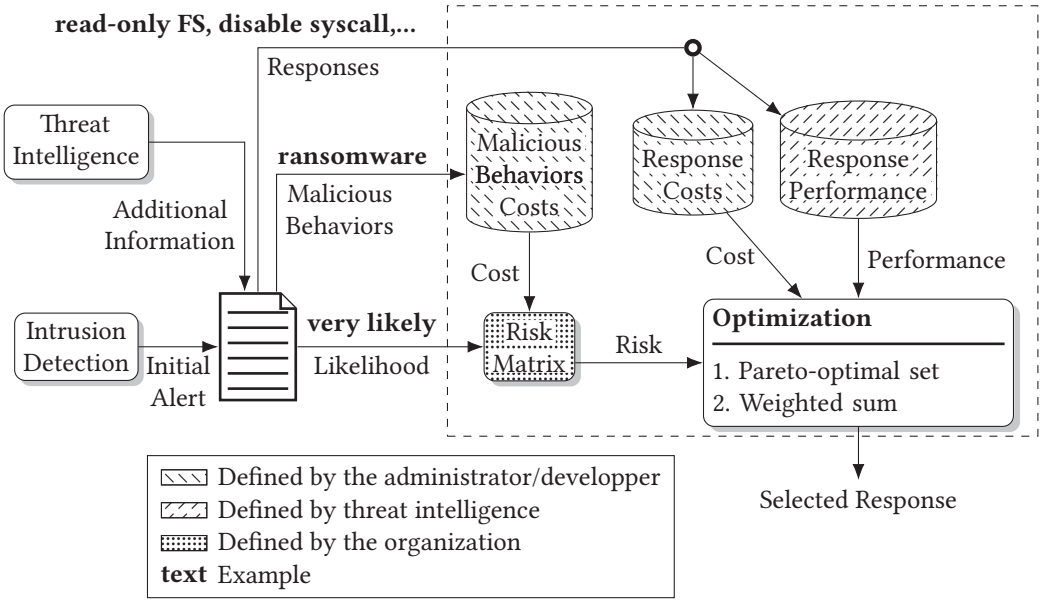


Fig. 3. High-level overview of the response selection process.

inaccuracies, do not require such data and are easier to understand [85]. In addition, we would like to limit the input from the user so it improves the approach's usability and its applicability to practical systems.

A high-level overview of our approach is illustrated in Figure 3. In the rest of this section, we first describe the models that we rely on. Then, we detail how we select cost-sensitive responses using such models.

5.1 Models

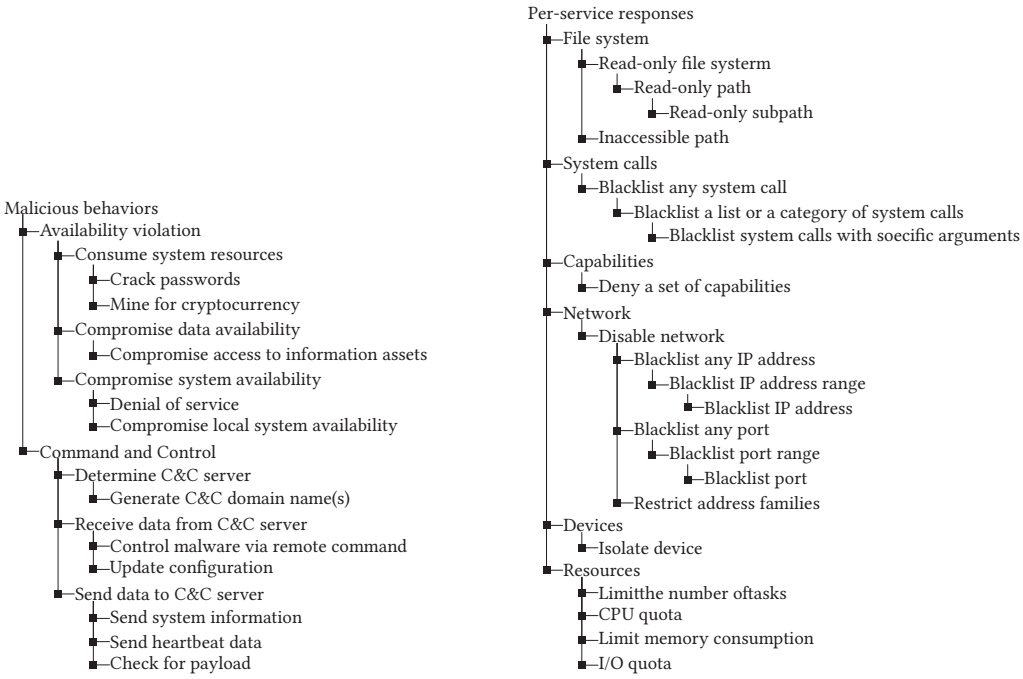
In comparison to previous approaches, we do not introduce a model and a response selection based on vulnerability graphs, attack graphs, or similar [27, 42, 72, 73]. We consider that we do not know how the attacker managed to compromise the service at the moment. We also assume that we cannot predict the next steps of the attacker. Instead, we use a different paradigm where we consider that we have information about the characteristics of the intrusions or the behaviors exhibited by the attacker.

5.1.1 Malicious Behaviors and Responses. Intrusions may exhibit multiple malicious behaviors that need to be stopped or mitigated differently. In our approach, we work at the level of a malicious behavior, and we select a response for each malicious behavior of an intrusion.

Our models rely on a hierarchy of malicious behaviors where the first levels describe high-level behaviors (e.g., compromise data availability), while lower levels describe more precise behaviors (e.g., encrypt files). The malware capabilities hierarchy [59] from the project MAEC [46] of MITRE is a suitable candidate for such a hierarchy.[6] Figure 4(a) illustrates an example of a non-exhaustive malicious behavior hierarchy with behaviors that relates to availability violations and to Command and Control (C&C).

We model this hierarchy as a partially ordered set $(\mathbf{M}, \prec_{\mathbf{M}})$ with $\prec_{\mathbf{M}}$ a binary relation over the set of malicious behaviors $\mathbf{M}$. The relation $m \prec_{\mathbf{M}} m'$ means that m is a more precise behavior than m'. Let $\mathbf{I}$ be the space of intrusions reported by the IDS. We assume that for each intrusions $i \in \mathbf{I}$, we can map the set of malicious behaviors $M^i \subseteq \mathbf{M}$ exhibited by i. By construct, we have the following property: If $m \prec_{\mathbf{M}} m'$, then $m \in M^i \implies m' \in M^i$.

[6] Another project that can help is the MITRE ATT&CK knowledge base [60], but it does not provide a hierarchy.



(a) Example of a non-exhaustive malicious behavior hierarchy

(b) Example of a non-exhaustive per-service response hierarchy

Fig. 4. Examples of hierarchies used in our models.

We also rely on a hierarchy of responses where the first levels describe coarse-grained responses (e.g., block the network), while lower levels describe more fine-grained responses (e.g., block port 80). We define the hierarchy as a partially ordered set $(\mathbf{R}, \prec_{\mathbf{R}})$ with $\prec_{\mathbf{R}}$ a binary relation over the set of responses $\mathbf{R}$ ($r \prec_{\mathbf{R}} r'$ means that r is a more fine-grained response than r'). Let $R^m \subseteq \mathbf{R}$ be the set of responses that can stop a malicious behavior m. By construct, we have the following property: If $r \prec_{\mathbf{R}} r'$, then $r \in R^m \implies r' \in R^m$. Such responses are based on the OS-features available to restrict privileges and quotas on the system.

Figure 4(b) is an example of this response hierarchy. Note that for each response with arguments (e.g., read-only paths or blacklisting IP addresses), the hierarchy provides a sub-response with a subset of the arguments. For example, if there is a response that puts `/var` read-only, there is also the responses that puts `/var/www` read-only.

5.1.2 Malicious Behavior Cost and Response Cost. Let the space of services be denoted $\mathbf{S}$ and let the space of qualitative linguistic constants be a totally ordered set, denoted $\mathbf{Q}$ composed as follows: none < very low < low < moderate < high < very high < critical. We extend each service configuration file with the notion of malicious behavior costs and response costs (in terms of quality of service lost) that an administrator needs to set.

A malicious behavior cost $c_{mb} \in \mathbf{C_{mb}} \subseteq \mathbf{Q}$ is the qualitative impact of a malicious behavior $m \in \mathbf{M}$. We define $mbcost : \mathbf{S} \times \mathbf{M} \rightarrow \mathbf{C_{mb}}$, the function that takes a service, a malicious behavior, and returns the associated cost.

We require for each service that a malicious behavior cost is set for at least the first level of the malicious behaviors hierarchy (e.g., there are only 20 elements on the first level of the hierarchy from MAEC). We do not require it for other levels, but if more costs are set, then the response selection will be more accurate. The

mbcost function associates a cost for each malicious behavior m. The cost, however, could be undefined. In such a case, we take the cost of m' such that $mbcost(s, m')$ is defined, $m \prec_{\mathbf{M}} m'$, and $\nexists m''$ such that $m < m'' < m'$ with $mbcost(s, m'')$ defined.

For example, the policy of a web server w could express that any malicious behavior that violate availability has a high cost in general:

$$mbcost(w, \text{“availability-violation”}) = \text{“high”}$$

It means that malicious behaviors that are children of this node in the hierarchy also have their cost automatically defined, such as:

$$mbcost(w, \text{“compromise-data-availability”}) = \text{“high”}$$

We could consider that the cost of this malicious behavior does not need to be refined, since an intrusion that compromises data availability (e.g., ransomware) has a high cost, since the web server would not provide access to the websites anymore. However, if not all availability violations have the same cost, they could be refined. For instance, the policy could express that an intrusion that only consumes system resources (e.g., a cryptocurrency mining malware) has a moderate cost, since it would only slow down the system or the service. It would be expressed by overriding the cost for a more specific malicious behavior:

$$mbcost(w, \text{“consume-system-resources”}) = \text{“moderate”}$$

A response cost $c_r \in \mathbf{C_r} \subseteq \mathbf{Q}$ is the qualitative impact of applying a response $r \in \mathbf{R}$ on a service to stop a malicious behavior. We define $rcost : \mathbf{S} \times \mathbf{R} \rightarrow \mathbf{C_r}$, the function that takes a service, a response, and returns the associated response cost.

Response costs allow an administrator or developer of a service to specify how a response, if applied, would impact the overall quality of service. The impact can be assessed based on the number of functions that would be unavailable and their importance for the service. More importantly, with the value critical, we consider that a response would disable a core function of a service and thus should never be applied. Similarly, the cost could be undefined, hence, we take the cost of r' such that $rcost(s, r')$ is defined, $r \prec_{\mathbf{R}} r'$, and $\nexists r''$ such that $r < r'' < r'$ with $rcost(s, r'')$ defined.

Following the same example, the policy could express that the network, and especially the ability to listen on ports 80 and 443 is critical for the core functions of the web server, while removing the ability to listen on other ports does not have any impact on the service. First, we express that the network is critical:

$$rcost(w, \text{“network”}) = \text{“critical”}$$

Then, we state that removing the ability to listen on ports in general does not have an impact:

$$rcost(w, \text{“blacklist-any-port”}) = \text{“none”}$$

Except on port 80 and 443:

$$rcost(w, \text{“blacklist-port-80”}) = \text{“critical”}$$
$$rcost(w, \text{“blacklist-port-443”}) = \text{“critical”}$$

Similarly, having access to the file system is critical, but if the web server would lose write access to the file system, the cost could be high and not critical, since it can still provide access to websites for many use cases:

$$rcost(w, \text{“filesystem”}) = \text{“critical”}$$
$$rcost(w, \text{“read-only-filesystem”}) = \text{“high”}$$

Both costs need to be configured depending on the *context* of the service. For example, a web server that provides static content does not have the same context, hence the same costs as one that handles transactions.

Table 1. Example of a 5 × 5 Risk Matrix that Follows the Requirements for Our Risk Assessment

Confidence (Likelihood)	Malicious Behavior Cost				
	Very low 0–0.2	Low 0.2–0.4	Moderate 0.4–0.6	High 0.6–0.8	Very high 0.8–1
Very likely 0.8–1	L	M	H	H	H
Likely 0.6–0.8	L	M	M	H	H
Probable 0.4–0.6	L	L	M	M	H
Unlikely 0.2–0.4	L	L	L	M	M
Very unlikely 0–0.2	L	L	L	L	L

5.1.3 Response Performance. While responses have varying costs on the quality of service, they also differ in performance against a malicious behavior (i.e., their efficiency at stopping a specific malicious behavior). For example, making a subset of the file system read-only is less efficient to stop some ransomware than making the whole file system read-only. Hence, we consider the performance as a criterion to select a response, among others. The most effective response in terms of performance would be to stop the infected service. While our model allows it, in this work we only mention fine-grained responses that aim at maintaining the availability. Other work [8, 27, 73] already studied the use of such coarse-grained responses.

The space of qualitative response performances is denoted $\mathbf{P_r} \subseteq \mathbf{Q}$. We define $rperf : \mathbf{R} \times \mathbf{M} \rightarrow \mathbf{P_r}$, which takes a response, a malicious behavior, and returns the associated performance.

In contrast to the cost models previously defined that are specific to a system and its context (and need to be set, e.g., by an administrator of the system), such a value only depends on the malicious behavior and is provided by security experts that analyzed similar intrusions. This response performance comes from threat intelligence sources that are shared, for example, using Structured Threat Information eXpression (STIX) [9].[7] For example, STIX has a property called "efficacy" in its "course-of-action" object that represents responses.

5.1.4 Risk Matrix. We rely on the definition of a risk matrix that satisfies the axioms proposed by Anthony Tony Cox [3] to provide consistent risk assessments: weak consistency, betweenness, and consistent coloring. While he defines these axioms more formally, we summarize them as follows:

Weak consistency. Each risk qualified as high should have a quantitative risk higher than all risk qualified as low.

Betweenness. A small increase in confidence or in cost should not change the risk rating from low to high (there should always be an intermediate).

Consistent coloring. Equal quantitative risks should have the same qualitative risk rating if either one of them is rated as high or low.

The risk matrix needs to be defined ahead of time by the administrator depending on the risk attitude of the organization: whether the organization is risk averse, risk neutral, or risk seeking. The 5 × 5 risk matrix shown in Table 1 is one instantiation of such a matrix.

[7]Organizations such as the Information Technology - Information Sharing and Analysis Center (IT-ISAC) [37] or national Computer Emergency Response Teams (CERTs) [82] provide threat intelligence feeds to their members using STIX.

The risk matrix outputs a qualitative malicious behavior risk $k \in \mathbf{K} \subseteq \mathbf{Q}$. The risk matrix depends on a malicious behavior cost (impact) and on the confidence level $i_{cf} \in \mathbf{I_{cf}} \subseteq \mathbf{Q}$ that the IDS has on the intrusion (likelihood).[8]

We define $risk : \mathbf{C_{mb}} \times \mathbf{I_{cf}} \rightarrow \mathbf{K}$, the function representing the risk matrix that takes a malicious behavior cost, an intrusion confidence, and returns the associated risk.

5.1.5 Policy Definition and Inputs. Having discussed the various models we rely on, we can define the policy as a tuple of four functions:

$$\langle rcost, rperf, mbcost, risk \rangle.$$

The *risk* function is defined at the organization level. It needs to be defined once, it is the same for each service, and unless the risk tolerance of the organization evolves it should not change. Moreover, $rperf$ is constant and can be applied for any system. However, *mbcost* and *rcost* are defined for each service, depending on its context. Hence, the most time-consuming parameters to set are *mbcost* and *rcost*.

The function *mbcost* can be defined by someone that understands the impact of malicious behaviors based on the service's context (e.g., an administrator). *rcost* can be defined by an expert, a developer of the service, or a maintainer of the OS where the service is used, since they understand the impact of removing certain privileges to the service. For example, some Linux distributions provide the security policies (e.g., SELinux or AppArmor) of their services and applications. Much like SELinux policies, *rcost* could be provided this way, since the maintainers would need to test that the responses do not render a service unusable (i.e., by disabling a core functionality).

5.2 Optimal Response Selection

We now discuss how we use this policy to select cost-sensitive responses. Our goal is to maximize the performance of the response while minimizing the cost to the service. We rely on known Multi-Objective Optimization (MOO) methods [53] to select the most cost-effective response, as does other work on response selection [62, 73].

For conciseness, since we are selecting a response for a malicious behavior $m \in \mathbf{M}$ and a service $s \in \mathbf{S}$, we now denote $rperf(r, m)$ as p_r, $rcost(s, r)$ as c_r, and $mbcost(s, m)$ as c_{mb}.

5.2.1 Overview. When the IDS triggers an alert, we assume that it provides the confidence $i_{cf} \in \mathbf{I_{cf}}$ of the intrusion $i \in \mathbf{I}$ and the set of malicious behaviors $M^i \subseteq \mathbf{M}$ corresponding to this intrusion.[9] Before selecting an optimal response, we filter out any response that has a critical response cost from R^m (the space of responses that can stop a malicious behavior m). Otherwise, such responses would impact a core function of the service. We denote $\hat{R}^m \subseteq R^m$ the resulting set:

$$\hat{R}^m = \{\, r \in R^m \mid c_r < \text{critical} \,\}.$$

For each malicious behavior $m \in M^i$, we compute the Pareto-optimal set from $\hat{R}^m$, where we select an optimal response from. We now describe these last steps.

5.2.2 Pareto-Optimal Set. In contrast to a Single-Objective Optimization (SOO) problem, a MOO problem does not generally have a single global solution. For instance, in our case, we might not have a response that provides both the maximum performance and the minimum cost, because they are conflicting, but rather a set of solutions that are defined as optimum. A common concept to describe such solutions is Pareto optimality.

[8] If the IDS does not provide confidence metrics, the organization or an administrator could define one by default that would need to be defined once (such as "very likely").

[9] The set of malicious behaviors can either be provided by the IDS in the alert or inferred using threat intelligence sources as long as we have information (e.g., an identifier or a generic description) from the IDS that allows us to match it with such sources.

A solution is Pareto-optimal (non-dominated) if it is not possible to find other solutions that improve one objective without weakening another one. The set of all Pareto-optimal solutions is called a Pareto-optimal set.[10] More formally, in our context, we say that a response is Pareto-optimal if it is non-dominated. A response $r \in R^m$ dominates a response $r' \in R^m$, denoted $r \succ r'$, if the following is satisfied:

$$[p_r > p_{r'} \wedge c_r \leq c_{r'}] \vee [p_r \geq p_{r'} \wedge c_r < c_{r'}].$$

MOO methods help to choose solutions among the Pareto-optimal set using preferences (e.g., should we put the priority on the performance of the response or on reducing the cost?) [53]. They rely on a scalarization that converts a MOO problem into a Single-Objective Optimization (SOO) problem. One common scalarization approach is the weighted sum method, which assigns a weight to each objective and computes the sum of the product of their respective objective. However, this method is not guaranteed to always give solutions in the Pareto-optimal set [53].

Shameli-Sendi et al. [73] decided to apply the weighted sum method on the Pareto-optimal set instead of on the whole solution space to guarantee to have a solution in the Pareto-optimal set. We apply the same reasoning, so we reduce our set to all non-dominated responses. We denote the resulting Pareto-optimal set O:

$$O = \{ r_i \in \hat{R}^m \mid \nexists r_j \in \hat{R}^m, r_j \succ r_i \}.$$

5.2.3 Response Selection. Before selecting a response from the Pareto-optimal set using the weighted sum method, we need to set weights and to convert the linguistic constants into numerical values.

We rely on a function l that maps the linguistic constants to a numerical value[11] between 0 and 1. In our case, we convert the linguistic constants critical, very high, high, moderate, low, very low, and none, to, respectively, the values 1, 0.9, 0.7, 0.5, 0.3, 0.1, and 0.

We use the risk matrix to set the weight of the response performance:

$$k = risk(c_{mb}, i_{cf}),$$
$$w_p = l(k).$$

Then, we set the weight of the response cost as follows:

$$w_c = 1 - w_p.$$

This means that we prioritize the performance if the risk is high, while we prioritize the cost if the risk is low.

We obtain the final optimal response by applying the weighted sum method:

$$\arg\max_{r \in O} w_p l(p_r) + w_c(1 - l(c_r)).$$

6 IMPLEMENTATION

We implemented a Linux-based prototype by modifying several existing projects. While our implementation relies on Linux features such as namespaces [40], seccomp [15], or cgroups [32], our approach does not depend on OS-specific paradigms. For example, on Windows, one could use Integrity Mechanism [55], Restricted Tokens [58], and Job Objects [56]. In the rest of this section, we describe the projects we modified, why we rely on them, and the different modifications we made to implement our prototype. You can see in Table 2 the different projects we modified where we added in total nearly 3,600 lines of C code.

At the time of writing, the most common service manager on Linux-based systems is systemd [77]. We modified it to checkpoint and to restore services using CRIU [17] and snapper [75] and to apply responses at the end of the restoration.

[10] Also known as a Pareto front.

[11] An alternative would be to use fuzzy logic to reflect the uncertainty regarding the risk assessment from experts when using linguistic constants [20].

Table 2. Projects Modified for Our Implementation

Project	From version	Code added
CRIU	3.9	383 lines of C
systemd	239	2,639 lines of C
audit		
user space	2.8.3	79 lines of C
Linux kernel	4.17.5	460 lines of C
Total		3,561 lines of C

6.1 Checkpoint and Restore

CRIU is a checkpoint and restore project implemented in user space for Linux. It can checkpoint the state of an application by fetching information about it from different kernel APIs and then store this information inside an image. CRIU reuses this image and other kernel APIs to restore the application. We chose CRIU because it allows us to perform transparent checkpointing and restoring (i.e., without modification or recompilation) of the services.

Snapper provides an abstraction for snapshotting filesystems and handles multiple Linux filesystems (e.g., BTRFS [67]). It can create a comparison between a snapshot and another one (or the current state of the filesystem). In our implementation, we chose BTRFS due its Copy-On-Write (COW) snapshot and comparison features, allowing a fast snapshotting and comparison process.

When checkpointing a service, we first freeze its cgroup (i.e., we remove the processes from the scheduling queue) to avoid inconsistencies. Thus, it cannot interact with other processes nor with the filesystem. Second, we take a snapshot of the filesystem and a snapshot of the metadata of the service kept by systemd (e.g., status information). Third, we checkpoint the processes of the service using CRIU. Finally, we unfreeze the service.

When restoring a service, we first kill all the processes belonging to its cgroup. Second, we restore the metadata of the service and ask snapper to create a read-only snapshot of the current state of the filesystem. Then, we ask snapper to perform a comparison between this snapshot and the snapshot taken during the checkpointing of the service. It gives us information about which files were modified and how. Since we want to only recover the files modified by the monitored service, we filter the result based on our log of files modified by this specific service (see Section 6.3 for more details) and restore the final list of files. Finally, we restore the processes using CRIU. Before unfreezing the restored service, CRIU calls back our function that applies the responses. We apply the responses at the end to avoid interfering with CRIU, which requires certain privileges to restore processes.

6.2 Responses

Our implementation relies on Linux features such as namespaces, seccomp, or cgroups to apply responses. Here is a non-exhaustive list of responses that we support: filesystem constraints (e.g., put all or any part of the filesystem read-only), system call filters (e.g., blacklisting a list or a category of system calls), network socket filters (e.g., deny access to a specific IP address), or resource constraints (e.g., CPU quotas or limit memory consumption).

We modified systemd to apply most of these responses just before unfreezing the restored service, except for system call filters. Seccomp only allows processes to set up their own filters and prevent them to modify the filters of other processes. Therefore, we modified systemd so when CRIU restores a process, it injects and executes code inside the address space of the restored process to set up our filters.

6.3 Monitoring Modified Files

The Linux auditing system [5, 38] is a standard way to trigger events from the kernel to user space based on a set of rules. Linux audit can trigger events when a process performs write accesses on the filesystem. However, it cannot filter these events for a set of processes corresponding to a given service (i.e., a cgroup). Hence, we

modified the kernel side of Linux audit to perform such filtering to only log files modified by the monitored services. Then, we specified a monitoring rule that relies on such filtering.

We developed a userland daemon that listens to an audit netlink socket and processes the events generated by our monitoring rules. Then, by parsing them, our daemon can log which files a monitored service modified. To that end, we create a file hierarchy under a per-service private directory. For example, if the service `abc.service` modified the file `/a/b/c/test`, we create an empty file `/private/abc.service/a/b/c/test`. This solution allows us to log modified files without keeping a data structure in memory.

7 EVALUATION

We performed an experimental evaluation of our approach to answer the following questions:

(1) How effective are our responses at stopping malicious behaviors in case a service is compromised?
(2) How effective is our approach at selecting cost-sensitive responses that withstand an intrusion?
(3) What is the impact of our solution on the availability or responsiveness of the services?
(4) How much overhead does our solution incur on the system resources?
(5) Do services continue to function (i.e., not crash) when they are restored with less privileges than they initially needed?

For the experiments, we installed Fedora Server 28 with the Linux kernel 4.17.5, and we compiled the programs with GCC 8.1.1. We ran the experiments that used live malware in a virtualized environment to control malware propagation.

The setup consisted of an isolated network connected to the Internet with multiple Virtual Local Area Networks (VLANs), two Virtual Machines (VMs), and a workstation. We executed the infected service on a Virtual Machine (VM) connected to an isolated Virtual Local Area Network (VLAN) with access to the Internet. We connected the second VM, which executes the network sniffing tool (tcpdump) to another VLAN with port mirroring from the first VLAN. Finally, the workstation, connected to another isolated VLAN, had access to the server managing the VMs, the VM with the infected service, and the network traces.

While malware could use anti-virtualization techniques [13, 66], to the best of our knowledge, none of our samples used such techniques.[12] We executed the rest of the experiments on bare metal on a computer with an AMD PRO A12-8830B R7 at 2.5 GHz, 12 GiB of RAM, and a 128 GB Intel SSD 600p Series.

Throughout the experiments, we tested our implementation on different types of services: web servers (nginx [63] and Apache [4]), database (mariadb [52]), work queue (beanstalkd [10]), message queue (mosquitto [25]), and git hosting services (gitea [29]).

7.1 Responses Effectiveness

Our first experiments focus on how effective our responses against distinct types of intrusions are. We are not interested, per se, in the vulnerabilities that attackers can exploit, but on how to stop attackers from performing malicious actions after they have infected a service. Here, we do not focus on response selection, which is discussed in Section 7.2.

The following list describes the malware and attacks used (see Table 3 for the hashes of the malware samples):

Linux.BitCoinMiner. Cryptocurrency mining malware that connects to a mining pool using attacker-controlled credentials [80].

Linux.Rex.1. Malware that joins a Peer-to-peer (P2P) botnet to receive instructions to scan systems for vulnerabilities to replicate itself, elevate privileges by scanning for credentials on the machine, participate in a Distributed Denial-of-Service (DDoS) attack or send spam [23].

[12] This is consistent with the study of Cozzi et al. [16] that showed that in the 10, 548 Linux malware they studied, only 0.24 % of them tried to detect if they were in a virtualized environment.

Table 3. Malware Used in Our Experiments with the SHA-256 Hash of the Samples

Malware	SHA-256
Linux.BitCoinMiner	`690aea53dae908c9afa933d60f467a17ec5f72463988eb5af5956c6cb301455b`
Linux.Rex.1	`762a4f2bf5ea4ff72fce674da1adf29f0b9357be18de4cd992d79198c56bb514`
Linux.Encoder.1	`18884936d002839833a537921eb7ebdb073fa8a153bfeba587457b07b74fb3b2`
Hakai	`58a5197e1c438ca43ffc3739160fd147c445012ba14b3358caac1dc8ffff8c9f`

Table 4. Summary of the Experiments that Evaluate the Effectiveness of the Responses against Various Malicious Behaviors

Attack Scenario	Malicious Behavior	Per-service Response Policy
Linux.BitCoinMiner	Mine for cryptocurrency	Ban mining pool IPs
Linux.BitCoinMiner	Mine for cryptocurrency	Reduce CPU quota
Linux.Rex.1	Determine C&C server	Ban bootstrapping IPs
Hakai	Receive data from C&C	Ban C&C servers' IPs
Linux.Encoder.1	Encrypt data	Read-only filesystem
GoAhead exploit	Exfiltrate via network	Forbid connect syscall
GoAhead exploit	Data theft	Render paths inaccessible

Hakai. Malware that receives instructions from a Command and Control (C&C) server to launch Distributed Denial-of-Service (DDoS) attacks and to infect other systems by brute forcing credentials or exploiting vulnerabilities in routers [24, 64].

Linux.Encoder.1. Encryption ransomware that encrypts files commonly found on Linux servers (e.g., configuration files or HTML files) and other media-related files (e.g., JPG or MP3), while ensuring that the system can boot so the administrator can see the ransom note [22].

GoAhead exploit. Exploit that gives remote code execution to an attacker on all versions of the GoAhead embedded web server prior to `3.6.5` [33].

Our work does not focus on detecting intrusions but on how to recover from and withstand them. Hence, we selected a diverse set of malware and attacks that covered various malicious behaviors.

For each experiment, we start a vulnerable service, we checkpoint its state, we infect it, and we wait for the payload to execute (e.g., encrypt files). Then, we apply our responses and we evaluate their effectiveness. We consider the restoration successful if the service is still functioning and its state corresponds to the one that has been checkpointed. Finally, we consider the responses effective if we cannot reinfect the service or if the payload cannot achieve its goals anymore.

The results we obtained are summarized in Table 4. In each experiment, as expected, our solution successfully restored the service after the intrusion to a previous safe state. In addition, as expected, each response was able to withstand a reinfection for its associated malicious behavior and only impacted the specific service and not the rest of the system.

7.2 Cost-Sensitive Response Selection

Our second set of experiments focus on how effective is our approach at selecting cost-sensitive responses. We chose Gitea, a self-hosted Git-repository hosting service[13] as a use case for a service, because it requires a diverse set of privileges and it shows how our approach can be applied to a complex real-world service.

[13]Gitea is considered as an open source clone of the services provided by GitHub [30].

Table 5. Responses to Withstand Ransomware Reinfection with Their Associated Cost and Performance for Gitea

#	Response	Cost (c_r)	Performance (p_r)
1	Disable open system call	Very High	Very High
2	Read-only file system except sessions folder	High	Very High
3	Paths of git repositories inaccessible	High	Moderate
4	Read-only paths of all git repositories	Moderate	Moderate
5	Read-only paths of important git repositories	Low	Low
6	Read-only file system	Critical	Very High

In our use case, we configured Gitea with the principle of least privileges. It means that restrictions that correspond to responses with a cost assigned to none are initially applied to the service (e.g., Gitea can only listen on port 80 and 443 or Gitea have only access to the directories and files it needs). Even if the service follows the best practices and is properly protected, an intrusion can still do damages (e.g., compromise the integrity of the repositories or the database) and our approach handles such cases.

The main goal of Gitea is to provide access to repositories with read access (e.g., cloning a repository or accessing some commits) and write access (e.g., pushing new commits to a branch). However, one could consider that the core function of Gitea is to at least provide read access to the repositories. Hence, we configured the policy (similar to what an administrator would do) to set the cost of an intrusion that compromises data availability to high, since it would render the site almost unusable:

$$mbcost(\text{“gitea”}, \text{“compromise-data-availability”}) = \text{“high”}$$

Other malicious behaviors would not necessarily have such a high cost. For example, intrusions that would have malicious behaviors related to consuming system resources would have a lesser impact on Gitea (it would just be slower), so we configured the policy to set the cost of such behaviors[14] as moderate:

$$mbcost(\text{“gitea”}, \text{“consume-system-resources”}) = \text{“moderate”}$$

We now consider an intrusion that compromised our Gitea service with the `Linux.Encoder.1` ransomware.[15] When executed, it encrypts all the git repositories and the database used by Gitea.

Since our focus is not on intrusion detection, we assume that the IDS detected the ransomware. This assumption is reasonable with, for example, several techniques to detect ransomware such as API call monitoring, file system activity monitoring, or the use of decoy resources [41].

However, in practice, an IDS can generate false positives, or it can provide non-accurate values for the likelihood of the intrusion leading to a less adequate response. Hence, we consider three cases to evaluate the response selection: the IDS detected the intrusion and accurately set the likelihood, the IDS detected the intrusion but not with an accurate likelihood, and the IDS generated a false positive.

In Table 5, we display a set of responses for the ransomware. Such a set of responses might be provided automatically, e.g., by the IDS, an antivirus, or threat intelligence sources. In our case, since we were not able to find open sources providing such information (see Section 8.1), we devised this set based on existing strategies to mitigate ransomware, such as CryptoDrop lockdown [18] or Windows controlled folder access [57]. None of these responses could have been applied proactively by the developer of Gitea, because it degrades the quality of service. This table also shows the respective cost for the service (defined previously by an administrator) and the estimated performance (in general, those are defined by security experts and provided with the responses—here,

[14]One would have to assign a cost for other malicious behaviors, but for the sake of conciseness, we only show a sample of the policy.
[15]In our experiments, we used an exploit for Gitea version 1.4.0 [78].

we made an educated guess based on the sources we used). Finally, as a risk matrix, we use the one previously described in Table 1.

Now let us consider the three cases previously mentioned. In the first case, the IDS detected the intrusion and considered the intrusion very likely. After computing the Pareto-optimal set, we have three possible responses left (2, 4, and 5). The risk computed is *risk*("high", "very likely") = high. The response selection then prioritizes performance and selects the response 2 that sets the file system as read-only except the session folder. This protects all information stored by Gitea (git repositories and its database). The session folder remains writable, since having this folder read-only would render the site unusable, thus it is a core function (see the cost critical in Table 5). Gitea is restored with all the encrypted files. The selected response prevents the attacker to reinfect the service, since the exploit requires write accesses. In terms of quality of service, users can connect to the service and clone repositories, but due to the response a new user cannot register and users cannot push to repositories. Hence, this response is adequate, since the service cannot get reinfected, core functions are maintained, and other functions (e.g., users can log in) are available.

In the second case, the IDS detected the intrusion but considered the intrusion very unlikely while the attacker managed to infect the service. The risk computed is *risk*("high", "very unlikely") = low. The response selection then prioritizes cost and selects the response 5 that sets a subset of git repositories (the most important ones for the organization) as read-only. With this response, the attacker managed to reinfect the service and the ransomware encrypted many repositories, but not the most important ones. In terms of quality of service, users can still access the protected repositories, but due to the intrusion users cannot login anymore and they cannot clone the encrypted repositories (i.e., Gitea shows an error to the user). Hence, the response is less adequate when the IDS provides an incorrect value for the likelihood of the intrusion, since the malware managed to encrypt many repositories, but the core functions of Gitea are maintained.

In the third case, the IDS detected an intrusion with the likelihood being very likely, but it is in fact a false positive. The risk computed is *risk*("high", "very likely") = high. It is similar to the first case where the response selected is response 2 due to a high risk. However, in this case, there is no actual ransomware. In terms of quality of service, users still have access to the service, since they can access the site, they can log in, and they can clone all repositories. However, they cannot register, they cannot push modifications to the repositories, and they cannot add issues. It shows that even with false positives, our approach minimizes the impact on the quality of service. Once an analyst classified the alert as a false positive, the administrator can configure the service to leave the degraded mode by deactivating the response.[16]

7.3 Availability Cost

In this subsection, we detail the experiments that evaluate the availability cost for the checkpoint and restore procedures.

7.3.1 Checkpoint. Each time we checkpoint a service, we freeze its processes. As a result, a user might notice a slower responsiveness from the service. Hence, we measured the time to checkpoint different services: Apache HTTP server (v2.4.33), nginx (v1.12.1), mariadb (v10.2.16), and beanstalkd (v1.10). We repeated the experiment 10 times for each service. On average, the time to checkpoint was always below 300 ms. In Table 6, we show more detailed timings about the different operations executed during a checkpoint: initialize (i.e., to initialize structures, to create or open directories, and to freeze processes), snapshot the file system, serialize the service's metadata, and checkpoint the processes using CRIU. We see that the time to perform this last operation varies,

[16]In this case, technically speaking, we would deactivate the response by remounting the directories as writable in the namespace of the service. However, other responses might be more difficult to deactivate. For instance, for some responses on our Linux-based prototype, such as system call filters, it would be difficult to deactivate them without doing a checkpoint directly followed by a restore. System call filters cannot be disabled once set, so we would need to recreate the current state of the service while removing some system call filters in the image of the processes.

Table 6. Time to Perform the Checkpoint Operations of a Service

Checkpoint Operation		Mean	Standard deviation	Standard error of the mean
Service-independent operations				
Initialize	(µs)	643.20	90.75	14.35
Checkpoint service metadata	(µs)	51.47	8.45	1.33
Snapshot file system	(ms)	98.95	1.38	2.19
Checkpoint processes (CRIU)				
httpd	(ms)	199.24	11.05	3.49
nginx	(ms)	51.59	3.99	1.26
mariadb	(ms)	171.77	8.52	2.69
beanstalkd	(ms)	16.25	1.37	0.43
Total				
httpd	(ms)	298.88		
nginx	(ms)	151.24		
mariadb	(ms)	271.41		
beanstalkd	(ms)	115.89		

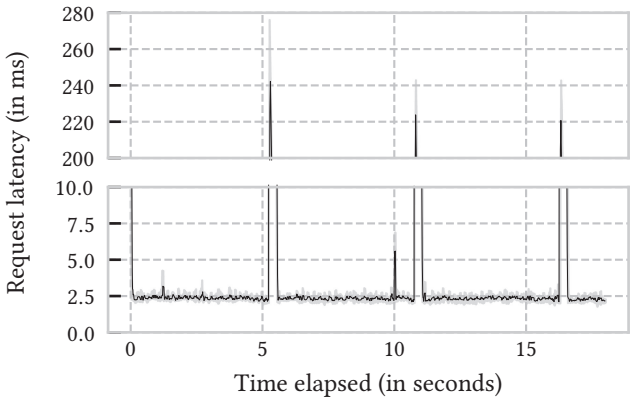


Fig. 5. Impact of checkpoints on the latency of HTTP requests made to an nginx server (less is better).

depending on the service while other operations are not dependent on the service. It is related to the resources used by the service (e.g., the number of processes, memory used, or files opened): More resources used means more time spent by CRIU to checkpoint them.

In Figure 5, we illustrate the results of the availability cost that users could perceive by measuring the latencies of HTTP requests made to an nginx server. We generated 100 requests per second for 20 seconds with the HTTP load testing tool Vegeta [71]. During this time, we checkpointed nginx at approximately 5, 11, and 16 seconds. We repeated the experiment three times. The output gave us the latency of each request, and we applied a moving average filter with a window size of 5. All requests were successful (i.e., no errors or timeouts) and the maximum latency during a checkpoint was 286 ms.

Both results show that our checkpointing has a small, but acceptable, availability cost. We do not lose any connection, we only increase the requests' latency when the service is frozen. Since the latency increases only

Table 7. Time to Perform the Restore Operations of a Service

Restore Operation		Mean	Standard deviation	Standard error of the mean
Kill processes				
httpd	(ms)	16.39	2.52	1.13
nginx	(ms)	19.24	3.69	1.65
mariadb	(ms)	28.48	2.16	0.97
beanstalkd	(ms)	10.85	1.19	0.53
Service-independent operations				
Initialize	(µs)	209.40	32.07	7.17
Compare Snapshots	(ms)	148.23	32.01	7.16
Restore service metadata	(µs)	212.75	36.23	8.10
Restore processes (CRIU)				
httpd	(ms)	132.42	6.09	2.72
nginx	(ms)	59.88	4.88	2.18
mariadb	(ms)	147.07	2.59	1.16
beanstalkd	(ms)	36.63	2.87	1.28
Total				
httpd	(ms)	299.29		
nginx	(ms)	227.79		
mariadb	(ms)	324.22		
beanstalkd	(ms)	196.16		

for a small period of time (maximum 300 ms), we consider such a cost acceptable. In comparison, SHELF [87] incurs a 7.6 % latency overhead for Apache during the whole execution of the system.

7.3.2 Restore. We also evaluated the time to restore the same services. On average, it took less than 325 ms. In Table 7, we show more detailed timings about the different operations executed during a restore: initialize (i.e., to initialize structures and to open directories), kill the processes, compare the snapshots of the file system, restore the service's metadata, and restore the processes using CRIU. When restoring, the time to kill the processes is service-dependent due to the different processes used and their number. We also see that the operation related to the comparison of the snapshots prior to restoring infected files takes a significant portion of the time.[17]

In contrast to the checkpoint, the restore procedure loses all network connections, since we kill the processes before restoring them.[18] The experiments, however, show that the time to restore a service is small (less than 325 ms). For example, in comparison, CRIU-MR [84] took 2.8 s on average to complete their restoration process.

7.4 Monitoring Cost

As detailed in Section 6.3, our solution logs the path of any file modified by a monitored service. This monitoring, however, incurs an overhead for every process executing on the system—even for the non-monitored services.[19]

[17]Note that in this experiment, we performed a checkpoint; then, directly afterward, we restored the service. It means the service did not modify files during the procedure. Hence, we only evaluated the time to compare snapshots and not the time it takes to effectively restore files.

[18]It could be possible to implement a method that retains packets in a buffer during the operation to avoid losing them, as implemented by CRIU-MR [84].

[19]For example, checking whether a process is part of the monitored services adds additional operations in the kernel.

There is also an additional overhead for monitored services that perform write accesses due to the audit event generated by the kernel and then processed by our daemon.

Therefore, we evaluated the runtime overhead due to the monitoring by running synthetic and real-world workload benchmarks, from the Phoronix test suite [50], for three different cases:

(1) no monitoring is present (**baseline**),
(2) monitoring rule enabled, but the service running the benchmarks **is not monitored** (no audit events are triggered),
(3) monitoring rule enabled and the service **is monitored** (audit events are triggered).

7.4.1 Synthetic Benchmarks. We ran synthetic I/O benchmarks that stress the system by performing many open, read, and write system calls:

compilebench. It emulates disk I/O operations related to the compilation of a kernel tree, reading the tree, or its creation [54].
fs-mark. It creates files and directories, at a given rate and size, either synchronously or asynchronously [86].
Postmark. It emulates an email server by performing a given number of transactions that create, read, append to, and delete files of varying sizes [39].

The results of the `read compiled tree` test of the `compilebench` benchmark (Figure 6(c)) confirmed that the overhead is only due to `open` system calls with write access mode. This test only reads files, and we do not observe any noticeable overhead (less than 1%, within the margin of error).

We now focus on the results of the `fs-mark` and `Postmark` benchmarks, respectively, illustrated in Figure 6(a) and Figure 6(b). In both experiments, we notice a small overhead when the service is not monitored (between 0.6% and 4.5%). With `fs-mark` (Figure 6(a)), when writing 1,000 files synchronously, we observe a 7.3% overhead. In comparison, when the files are written asynchronously, there is a 27.3% overhead. With `Postmark` (Figure 6(b)), we observe that the overhead is quite important (28.7%) when it writes many small files (between 5 KiB and 512 KiB) but remains low (3.1%) with bigger files (between 512 KiB and 1 MiB).

In summary, these synthetic benchmarks show that the worst case for our monitoring is when a monitored service writes many small files asynchronously in burst.

7.4.2 Real-world Workload Benchmarks. To have a different perspective than the synthetic benchmarks, we chose two benchmarks that use real-world workloads:

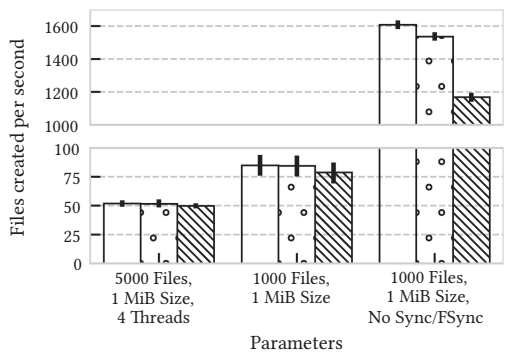
build-linux-kernel. It measures the time to compile the Linux kernel.[20]
unpack-linux. It measures the time to extract the archive of the Linux kernel source code.

We illustrate the results in Figure 7. When the service is monitored, the overhead is only significant with `unpack-linux` (Figure 7(a)) where we observe a 23.7% overhead. It concurs with our results from the synthetic benchmarks: Writing many small files asynchronously incurs a significant overhead when the service is monitored (the time to decompress a file in this benchmark is negligible). With `build-linux-kernel` (Figure 7(b)), we observe a small overhead (1.1%) even when the service is monitored (the time to compile the source code masks the overhead of the monitoring).

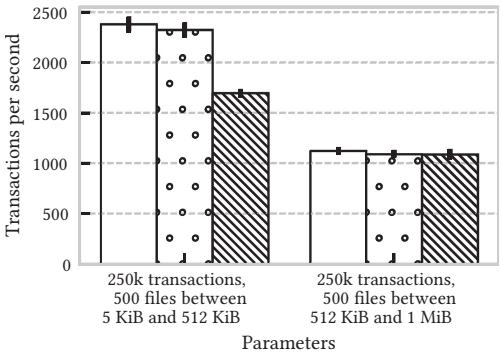
In comparison, SHELF [87] has a 65% overhead when extracting the archive of the Linux kernel source code and an 8% overhead when building this kernel.

In conclusion, both the synthetic and non-synthetic benchmarks show that our solution is more suitable for workloads that do not write many small files asynchronously in burst. For instance, our approach would be best suited to protect services such as web, databases, or video encoding services.

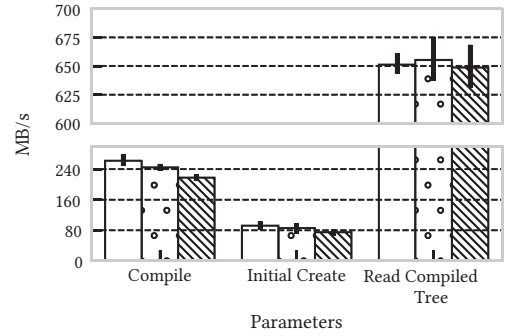
[20] While `build-linux-kernel` is CPU bound, it also performs many system calls, such as opening files to store the output of the compilation.



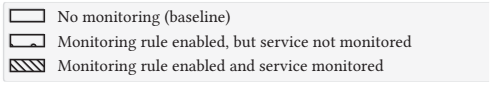
(a) Files created per second with the fs-mark benchmark (more is better)



(b) Transactions per second with the Postmark benchmark (more is better)

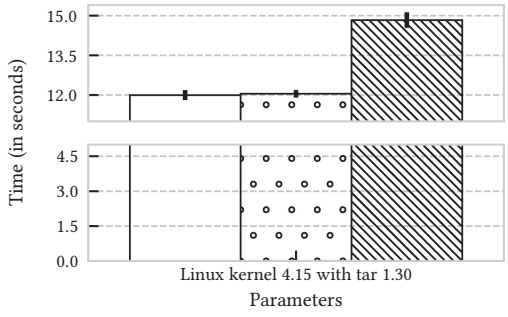


(c) MB/s score with the Compilebench benchmark (more is better)

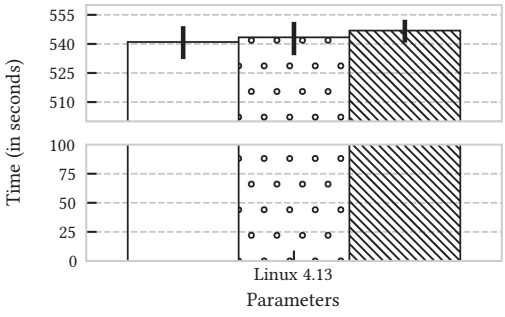


Legend

Fig. 6. Results of synthetic benchmarks to measure the overhead of the monitoring.



(a) Time (in seconds) to extract the archive (.tar.gz) of the Linex kernel source code (less is better)



(b) Time (in seconds) to build the Linux kernel (less is better)

Fig. 7. Results of real-world workload benchmarks to measure the overhead of the monitoring.

7.5 Storage Space Overhead

Checkpointing services require storage space to save the checkpoints. To evaluate the disk usage overhead, we checkpointed the same four services used in Section 7.3.1. Each checkpoint took, respectively, 26.2 MiB, 7.5 MiB, 136.0 MiB, and 130.1 KiB of storage space. The memory pages dumps took at least 95.3% of the size of their checkpoint. Hence, if a service uses more memory under load (e.g., Apache), its checkpoint would take more storage space.

7.6 Stability of Degraded Services

We tested our solution on diverse types of services: web servers (nginx, Apache), databases (mariadb), work queues (beanstalkd), message queues (mosquitto), or git hosting services (gitea). In terms of restoration, none of the services crashed when restored with a policy that removed privileges that they required (i.e., when they are in a degraded mode). The reason is twofold.

First, we provided a policy that specified the responses with a critical cost. Therefore, our solution never selected a response that removes a privilege needed by a core function. Second, the services checked for errors when performing various operations. For example, if a service needed a privilege that we removed, it tried to perform the operation and failed, but only logged an error and did not crash. If we generalize our results, it means our solution will not make other services (that we did not test) crash if they properly check for error cases. This practice is common, and it is often highlighted by the compiler or static analysis tools when this is not the case.[21]

8 DISCUSSION

Having demonstrated the effectiveness of the responses and showed the performance impact of our prototype, we now discuss the limitations of our approach, and we give a summary of the comparison of our approach against the related work.

8.1 Limitations

We discuss non-exhaustively limitations, areas that would need further work, or alternative choices for our solution.

False Positives. Since our approach relies on an IDS, we also inherit the limitations of this IDS. It is possible that we start the recovery and response procedures due to a false positive from the IDS. In this case, it will negatively impact the service's availability and its functions, while preventing no intrusion. Our approach, however, minimizes this risk by considering the likelihood of the intrusion for the selection of cost-sensitive responses and by ensuring that core functions are maintained.

CRIU Limitations. At the moment, CRIU cannot support all types of applications, since it has issues when handling external resources or graphical applications. For example, if a process has opened a device to have direct access to some hardware, checkpointing the state of the process may not be possible (except if it uses virtual or pseudo devices not corresponding to any physical devices). This technical limitation is because CRIU cannot be sure that when it restores a process the physical device has the same state as when CRIU checkpointed the process. Since our implementation relies on CRIU, we inherit its limitations. Therefore, at the moment, our implementation is better suited for system services that do not have a graphical part and do not require direct access to some hardware.

[21]For example, the CERT C Coding Standard recommends not to ignore values returned by functions, and it provides a list of tools that automatically detect any violation of this recommendation [12]. It also recommends "implement[ing] a consistent and comprehensive error-handling policy" [11] with an example of a tool to help detect when this is not the case.

Alternative checkpointing. An alternative to our transparent checkpointing with CRIU would be to implement a cooperative checkpointing approach where the service itself saves its state and uses it to restore itself after the IDS detected an intrusion. On one hand, such an approach would require changes in each service by adding a procedure to dump their state. For example, Android applications have such methods [2]. On the other hand, we would be able to apply more fine-grained mitigations, because the restoration procedure would have semantic information and knowledge of the data structures used by each service. For example, as a more fine-grained mitigation, we could change the implementation of a protocol.

Service Dependencies. In our work, at the moment, we only use the service dependency graph provided by the service manager (e.g., systemd) to recover and checkpoint dependent services together. We could also use this same graph to provide more precise response selection by taking into account the dependency between services, their relative importance, and to propagate the impact a malicious behavior can have. It could be used as a weight (in addition to the risk) to select optimal responses. Similar, but network-based, approaches have been heavily studied in the past [43, 73, 79].[22]

Models Input. For our cost-sensitive response selection, we first need to associate an intrusion to a set of malicious behaviors and the course of action to stop these behaviors. While standards exist to share threat information [9] and malicious behaviors [46, 59, 61] exhibited by malware, or attackers in general, we were not able to find open sources that provided them directly for the samples we used. This issue might be related to the fact that, to the best of our knowledge, no industry solution would exploit such information in an automated fashion. In our experiments, we extracted information about malicious behaviors from textual descriptions [22–24, 64, 80] and reused the existing standards to describe such malicious behaviors [46, 59, 61]. Likewise, we extracted information about responses to counter such malicious behaviors from textual descriptions [18, 22–24, 57, 64, 80]. One may, nonetheless, assume that if approaches similar to ours, or intrusion response systems in general, are becoming more prevalent and used in production, threat intelligence sources will eventually provide such data.

State Inconsistencies. Let us consider a service that does not follow properly the principle of least privilege with unnecessary write access to various files on the system. This service gets compromised and an attacker compromises files on the system that other services depend on. If we restore the files after the intrusion is detected, it could result in state inconsistencies in the services that depend on these files. Indeed, since we do not have information about which services use or depend on these files, we cannot restore their processes as well. It is the result of the tradeoff we initially made where we do not monitor every event on the system to limit the performance overhead. More work would be needed to maintain the low overhead while improving the reliability in such cases. For example, we could use the information available during the installation of a service[23] to know a subset of the files it depends on, or we could ask maintainers to list the directories or files it relies on.

Generic Responses. If we do not have precise information about the intrusion, but only a generic behavior or category associated to it (one of the top elements in the malicious behaviors hierarchy), we could automatically consider generic responses. For example, with ransomware, we know that responses that either render the file system read-only or only specific directories will work. Such generic responses might help mitigate the lack of precise information.

[22]The host-based approach from Balepin et al. [8] requires administrators to give dependency information between applications and the resources they use on the system. While it may be more precise than service dependencies, it is a tedious, error-prone process, and it introduces a maintenance burden.

[23]For example, Linux distributions rely on packages to install services. Each package contains the files to install on the system. In addition, it may contain instructions to create directories or files that the service requires. We could use both information to improve the reliability of the recovery in the case we discussed.

Table 8. Summary of the Comparison between Our Intrusion Survivability Approach and the Related Work

Characteristics	Approach										
	Our approach	SHELF [87]	CRIU-MR [84]	Taser [31]	Retro [44]	Shan et al. [74]	Foo et al. [27]	Balepin et al. [8]	Shameli-Sendi et al. [73]	Gehani and Kedem [28]	Huang et al. [36]
Recover from intrusions	●	●	●	●	●	●	○	◒	○	○	○
Withstand (re)infections	●	○	○	○	○	○	●	●	●	●	●
Maintain availability	●	●	●	○	○	○	◒	◒	◒	●	●
Host-based approach	●	●	●	●	●	●	○	●	○	●	●
Transparent	●	●	●	●	●	●	●	●	●	○	○
Fine-grained responses	●	○	○	○	○	○	○	○	○	●	●
Use service dependency graphs	◒	○	○	○	○	○	○	●	●	○	○
Quantitative risk assessment	○	○	○	○	○	○	○	●	●	●	○
Qualitative risk assessment	●	○	○	○	○	○	○	○	○	○	○
Legend											
● Has the property											
○ Does not have the property											
◒ Partially											

8.2 Comparison with Related Work

Our approach addresses various issues that we identified from the literature. We summarize in Table 8 a comparison of our intrusion survivability approach against previous related work.

To the best of our knowledge, our approach is the first to combine the ability to recover from intrusions with the ability to withstand potential reinfections after the system has been restored. Balepin et al. [8] had the ability to recover a file from a backup as a response, but they assume that the IDS knows precisely which file has been compromised (and did not describe how they were taking a backup of the files nor the performance impact). In practice, we might not have an accurate view of the illegitimate actions that have been carried out by the attacker—we can overestimate them or miss some of them. In our approach, we make a tradeoff between precision and performance by restoring all the files modified by a compromised service.[24] In addition, Balepin et al. [8] do not maintain the consistency between the state of the processes and the files they depend on. In our approach, to reduce the possibility of inconsistencies, we first freeze the processes. Then, we take simultaneously a checkpoint of both the file system and the state of the processes.

The fact that we can maintain the availability of the services and their core functions despite the presence of an adversary means that the OS can survive intrusions. Previous work on intrusion response [8, 27, 73] considered the cost of a response on the availability only in their models. Some of their responses, however, would significantly impact the availability without significant benefit for the security. For example, Shameli-Sendi et al.

[24]We assume that we might not know all the illegitimate actions that were done.

[73] have responses that reboot completely a system or restart a compromised service. In our approach, in addition to taking into account the cost of a response, our ability to restore the state of a service to a previous safe state minimizes the availability impact.[25] Finally, in comparison to previous work, since we apply per-service responses, we minimize the impact of a response on the rest of the system.

9 CONCLUSION AND FUTURE WORK

This work provides an intrusion survivability approach for commodity OSs. In contrast to other intrusion recovery approaches, our solution is not limited to restoring files or processes, but it also applies responses to withstand a potential reinfection. Such responses enforce per-service privilege restrictions and resource quotas to ensure that the rest of the system is not directly impacted. In addition, we only restore the files modified by the infected service to limit the restoration time. We devised a way to select cost-sensitive responses that does not disable core functions of services. We specified the requirements for our approach and proposed an architecture satisfying its requirements. Finally, we developed and evaluated a prototype for Linux-based systems by modifying systemd, Linux audit, CRIU, and the Linux kernel. Our results show that our prototype withstands known Linux attacks. Our prototype only induces a small overhead, except with I/O-intensive services that create many small files asynchronously in burst.

In the future, we would like to investigate how we could automatically adapt the system to gradually remove the responses that we applied to withstand a reinfection. Such a process involves being able to automatically fix the vulnerabilities or to render them non-exploitable.

REFERENCES

[1] James P. Anderson. 1980. *Computer Security Threat Monitoring and Surveillance.* Technical Report. James P. Anderson Co., Fort Washington, PA. Retrieved from http://seclab.cs.ucdavis.edu/projects/history/papers/ande80.pdf.

[2] Android Developers. 2019. Understand the Activity Lifecycle. Retrieved from https://developer.android.com/guide/components/activities/activity-lifecycle.

[3] Louis Anthony Tony Cox. 2008. What's wrong with risk matrices? *Risk Anal.* 28, 2 (2008), 497–512. DOI: https://doi.org/10.1111/j.1539-6924.2008.01030.x

[4] Apache. 2019. Apache HTTP Server. Retrieved from https://httpd.apache.org/.

[5] Audit. 2019. The Linux Audit Project. Retrieved from https://github.com/linux-audit/.

[6] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (Jan. 2004), 11–33. DOI: https://doi.org/10.1109/TDSC.2004.2

[7] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the ARM trustzone secure world. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'14).* ACM, 90–102. DOI: https://doi.org/10.1145/2660267.2660350

[8] Ivan Balepin, Sergei Maltsev, Jeff Rowe, and Karl Levitt. 2003. Using specification-based intrusion detection for automated response. In *Recent Advances in Intrusion Detection.* Springer Berlin Heidelberg, 136–154. DOI: https://doi.org/10.1007/978-3-540-45248-5_8

[9] Sean Barnum. 2014. Standardizing cyber threat intelligence information with the Structured Threat Information eXpression (STIX). MITRE. Retrieved from https://stixproject.github.io/about/STIX_Whitepaper_v1.1.pdf.

[10] Beanstalkd. 2019. beanstalkd. Retrieved from https://kr.github.io/beanstalkd/.

[11] CERT C Coding Standard. 2019. ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy. Retrieved from https://wiki.sei.cmu.edu/confluence/display/c/ERR00-C.+Adopt+and+implement+a+consistent+and+comprehensive+error-handling+policy.

[12] CERT C Coding Standard. 2019. EXP12-C. Do not ignore values returned by functions. Retrieved from https://wiki.sei.cmu.edu/confluence/display/c/EXP12-C.+Do+not+ignore+values+returned+by+functions.

[13] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks.* 177–186. DOI: https://doi.org/10.1109/DSN.2008.4630086

[25] Restoring to a previous state also allows us to implement responses that would be more difficult otherwise (e.g., additional system call filters).

[14] Ronny Chevalier, Maugan Villatel, David Plaquin, and Guillaume Hiet. 2017. Co-processor-based behavior monitoring: Application to the detection of attacks against the system management mode. In *Proceedings of the 33rd Computer Security Applications Conference (ACSAC'17)*. ACM, 399–411. DOI: https://doi.org/10.1145/3134600.3134622
[15] Jonathan Corbet. 2009. Seccomp and sandboxing. *LWN* (13 May 2009). Retrieved from https://lwn.net/Articles/332974/.
[16] Emanuele Cozzi, Mariano Graziano, Yannick Fratantonio, and Davide Balzarotti. 2018. Understanding Linux malware. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'18)*. 161–175. DOI: https://doi.org/10.1109/SP.2018.00054
[17] CRIU. 2018. CRIU. Retrieved from https://criu.org/.
[18] CryptoDrop, LLC. 2019. CryptoDrop. Retrieved from https://www.cryptodrop.org/.
[19] Dbus 2019. D-Bus. Retrieved from https://www.freedesktop.org/wiki/Software/dbus/.
[20] Yong Deng, Rehan Sadiq, Wen Jiang, and Solomon Tesfamariam. 2011. Risk analysis in a linguistic environment: A fuzzy evidential reasoning-based approach. *Exp. Syst. Applic.* 38, 12 (2011), 15438–15446. DOI: https://doi.org/10.1016/j.eswa.2011.06.018
[21] Dorothy E. Denning. 1986. An intrusion-detection model. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 118–131. DOI: https://doi.org/10.1109/SP.1986.10010
[22] Dr. Web. 2015. Linux.Encoder.1. Retrieved from https://vms.drweb.com/virus/?i=7703983.
[23] Dr. Web. 2016. Linux.Rex.1. Retrieved from https://vms.drweb.com/virus/?i=8436299.
[24] Dr. Web. 2018. Linux.BackDoor.Fgt.1430. Retrieved from https://vms.drweb.com/virus/?i=17573534.
[25] Eclipse Foundation. 2019. Mosquitto. Retrieved from https://mosquitto.org/.
[26] Robert J. Ellison, David A. Fisher, Richard C. Linger, Howard F. Lipson, and Thomas Longstaff. 1997. *Survivable Network Systems: An Emerging Discipline*. Technical Report. Software Engineering Institute, Carnegie Mellon University. Retrieved from https://apps.dtic.mil/dtic/tr/fulltext/u2/a341963.pdf.
[27] Bingrui Foo, Yu-Sung Wu, Yu-Chun Mao, Saurabh Bagchi, and Eugene H. Spafford. 2005. ADEPTS: Adaptive intrusion response using attack graphs in an E-commerce environment. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'05)*. 508–517. DOI: https://doi.org/10.1109/DSN.2005.17
[28] Ashish Gehani and Gershon Kedem. 2004. RheoStat: Real-time risk management. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID'04)*. 296–314. DOI: https://doi.org/10.1007/978-3-540-30143-1_16
[29] Gitea 2019. Gitea. Retrieved from https://gitea.io/.
[30] GitHub, Inc. 2019. GitHub. Retrieved from https://github.com/.
[31] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. 2005. The taser intrusion recovery system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. 163–176. DOI: https://doi.org/10.1145/1095810.1095826
[32] Tejun Heo. 2015. Control Group v2. Retrieved from https://www.kernel.org/doc/Documentation/cgroup-v2.txt.
[33] Daniel Hodson. 2017. Remote LD_PRELOAD Exploitation. Retrieved from https://www.elttam.com.au/blog/goahead/.
[34] HP Inc. 2019. *HP Sure Start: Automatic Firmware Intrusion Detection and Repair*. Technical Report. HP Inc. Retrieved from http://h10032.www1.hp.com/ctg/Manual/c06216928.
[35] Francis Hsu, Hao Chen, Thomas Ristenpart, Jason Li, and Zhendong Su. 2006. Back to the future: A framework for automatic malware removal and system repair. In *Proceedings of the 22nd Computer Security Applications Conference (ACSAC'06)*. 257–268. DOI: https://doi.org/10.1109/ACSAC.2006.16
[36] Zhen Huang, Mariana DAngelo, Dhaval Miyani, and David Lie. 2016. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *Proceedings of the IEEE Symposium on Security and Privacy*. 618–635. DOI: https://doi.org/10.1109/SP.2016.43
[37] IT-ISAC. 2019. FAQ. Retrieved from https://www.it-isac.org/faq.
[38] Mirek Jahoda, Ioanna Gkioka, Robert Krátký, Martin Prpič, Tomáš Čapek, Stephen Wadeley, Yoana Ruseva, and Miroslav Svoboda. 2017. System auditing. In *Red Hat Enterprise Linux 7 Security Guide*. 185–204.
[39] Jeffrey Katcher. 1997. *Postmark: A New File System Benchmark*. Technical Report 3022. Network Appliance.
[40] Michael Kerrisk. 2013. Namespaces in operation, part 1: Namespaces overview. *LWN* (4 Jan. 2013). Retrieved from https://lwn.net/Articles/531114/.
[41] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. 2015. Cutting the Gordian knot: A look under the hood of ransomware attacks. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 3–24. DOI: https://doi.org/10.1007/978-3-319-20550-2_1
[42] Nizar Kheir, Nora Cuppens-Boulahia, Frédéric Cuppens, and Hervé Debar. 2010. A service dependency model for cost-sensitive intrusion response. In *Proceedings of the 15th European Conference on Research in Computer Security (ESORICS'10)*. 626–642. DOI: https://doi.org/10.1007/978-3-642-15497-3_38
[43] Nizar Kheir, Hervé Debar, Nora Cuppens-Boulahia, Frédéric Cuppens, and Jouni Viinikka. 2009. Cost evaluation for intrusion response using dependency graphs. In *Proceedings of the International Conference on Network and Service Security*.
[44] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2010. Intrusion recovery using selective re-execution. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, 89–104.
[45] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA'14)*. IEEE Press, 361–372. DOI: https://doi.org/10.1109/ISCA.2014.6853210

[46] Ivan Kirillov, Desiree Beck, Penny Chase, and Robert Martin. 2011. Malware Attribute Enumeration and Characterization. MITRE. Retrieved from https://www.researchgate.net/profile/Robert_Martin10/publication/267691330_Malware_Attribute_Enumeration_and_Characterization/links/54bd188e0cf218d4a169ee0c/Malware-Attribute-Enumeration-and-Characterization.pdf.

[47] John C. Knight, Elisabeth A. Strunk, and Kevin J. Sullivan. 2003. Towards a rigorous definition of information system survivability. In *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition*, Vol. 1. IEEE, 78–89. DOI: https://doi.org/10.1109/DISCEX.2003.1194874

[48] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*.

[49] Nir Kshetri. 2006. The simple economics of cybercrimes. *IEEE Sec. Priv.* 4, 1 (2006), 33–39. DOI: https://doi.org/10.1109/MSP.2006.27

[50] Michael Larabel and Matthew Tippett. 2019. Phoronix Test Suite. Retrieved from https://www.phoronix-test-suite.com/.

[51] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. USENIX Association, 973–990.

[52] Mariadb 2019. mariadb. Retrieved from https://mariadb.org/.

[53] R. Timothy Marler and Jasbir S. Arora. 2004. Survey of multi-objective optimization methods for engineering. *Struct. Multidisc. Optim.* 26, 6 (Apr. 2004), 369–395. DOI: https://doi.org/10.1007/s00158-003-0368-6

[54] Chris Mason. 2008. Compilebench. Retrieved from https://oss.oracle.com/ mason/compilebench/.

[55] Microsoft. 2017. Windows Integrity Mechanism Design. Retrieved from https://msdn.microsoft.com/en-us/library/bb625963.aspx.

[56] Microsoft. 2018. Job Objects. Retrieved from https://msdn.microsoft.com/en-us/library/windows/desktop/ms684161(v=vs.85).aspx

[57] Microsoft. 2018. Protect important folders with controlled folder access. Retrieved from https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-exploit-guard/controlled-folders-exploit-guard?ocid=cx-blog-mmpc.

[58] Microsoft. 2018. Restricted Tokens. Retrieved from https://msdn.microsoft.com/en-us/library/windows/desktop/aa379316(v=vs.85).aspx.

[59] MITRE. 2014. Malware Capabilities. Retrieved from https://github.com/MAECProject/schemas/wiki/Malware-Capabilities.

[60] MITRE. 2019. ATT&CK. Retrieved from https://attack.mitre.org/.

[61] MITRE. 2019. Encyclopedia of Malware Attributes. Retrieved from https://collaborate.mitre.org/ema/.

[62] Alexander Motzek, Gustavo Gonzalez-Granadillo, Hervé Debar, Joaquin Garcia-Alfaro, and Ralf Möller. 2017. Selection of Pareto-efficient response plans based on financial and operational assessments. *EURASIP J. Inf. Secur.* (2017), 12. DOI: https://doi.org/10.1186/s13635-017-0063-6

[63] Nginx 2019. nginx. Retrieved from https://nginx.org/.

[64] Ruchna Nigam. 2018. Unit 42 Finds New Mirai and Gafgyt IoT/Linux Botnet Campaigns. Retrieved from https://researchcenter.paloaltonetworks.com/2018/07/unit42-finds-new-mirai-gafgyt-iotlinux-botnet-campaigns/.

[65] NSA and Red Hat. 2019. SELinux. Retrieved from https://selinuxproject.org/.

[66] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies (WOOT'09)*. USENIX Association, 7.

[67] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Trans. Stor.* 9, 3 (2013), 9. DOI: https://doi.org/10.1145/2501620.2501623

[68] Xiaoyu Ruan. 2014. Boot with integrity, or don't boot. In *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, 143–163. DOI: https://doi.org/10.1007/978-1-4302-6572-6_6

[69] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. 2012. *Windows Internals, Part 2* (6th ed.). Microsoft Press.

[70] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. Retrieved from https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[71] Tomás Senart. 2019. Vegeta. Retrieved from https://github.com/tsenart/vegeta.

[72] Alireza Shameli-Sendi, Mohamed Cheriet, and Abdelwahab Hamou-Lhadj. 2014. Taxonomy of intrusion risk assessment and response system. 45 (Sept. 2014), 1–16. DOI: https://doi.org/10.1016/j.cose.2014.04.009

[73] Alireza Shameli-Sendi, Habib Louafi, Wenbo He, and Mohamed Cheriet. 2018. Dynamic optimal countermeasure selection for intrusion response system. *IEEE Trans. Depend. Sec. Comput.* 15, 5 (2018), 755–770. DOI: https://doi.org/10.1109/TDSC.2016.2615622

[74] Zhiyong Shan, Xin Wang, and Tzi-cker Chiueh. 2013. Malware clearance for secure commitment of OS-level virtual machines. 10, 2 (Mar. 2013), 70–83. DOI: https://doi.org/10.1109/TDSC.2012.88

[75] Snapper 2018. snapper. Retrieved from http://snapper.io/.

[76] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. 2016. Enforcing kernel security invariants with data flow integrity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'16)*. DOI: https://doi.org/10.14722/ndss.2016.23218

[77] Systemd 2019. systemd System and Service Manager. Retrieved from https://www.freedesktop.org/wiki/Software/systemd/.

[78] Kacper Szurek. 2018. Gitea 1.4.0 Unauthenticated Remote Code Execution. Retrieved from https://security.szurek.pl/gitea-1-4-0-unauthenticated-rce.html.
[79] Thomas Toth and Christopher Kruegel. 2002. Evaluating the impact of automated intrusion response mechanisms. In *Proceedings of the 18th Computer Security Applications Conference (ACSAC'02)*. IEEE Computer Society. DOI: https://doi.org/10.1109/CSAC.2002.1176302
[80] Trend Micro Cyber Safety Solutions Team. 2018. Cryptocurrency Miner Distributed via PHP Weathermap Vulnerability, Targets Linux Servers. Retrieved from https://blog.trendmicro.com/trendlabs-security-intelligence/cryptocurrency-miner-distributed-via-php-weathermap-vulnerability-targets-linux-servers/.
[81] UEFI Forum. 2019. *Unified Extensible Firmware Interface Specification*. Retrieved from https://uefi.org/sites/default/files/resources/UEFI_Spec_2_8_final.pdf Version 2.8.
[82] United States Computer Emergency Readiness Team. 2013. Automated Indicator Sharing (AIS). Retrieved from https://www.us-cert.gov/sites/default/files/ais_files/AIS_fact_sheet.pdf.
[83] Sven Vermeulen. 2014. Handling SELinux-aware applications. In *SELinux Cookbook*. Packt Publishing.
[84] Ashton Webster, Ryan Eckenrod, and James Purtilo. 2018. Fast and service-preserving recovery from malware infections using CRIU. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, 1199–1211.
[85] Evan Wheeler. 2011. Risky business. In *Security Risk management: Building an Information Security Risk Management Program from the Ground up* (1st ed.). Syngress Publishing, 37–40.
[86] Ric Wheeler. 2016. fs-mark. Retrieved from https://sourceforge.net/projects/fsmark/.
[87] Xi Xiong, Xiaoqi Jia, and Peng Liu. 2009. SHELF: Preserving business continuity and availability in an intrusion recovery system. In *Proceedings of the 25th Computer Security Applications Conference (ACSAC'09)*. IEEE Computer Society, 484–493. DOI: https://doi.org/10.1109/ACSAC.2009.52
[88] Jiewen Yao and Vincent J. Zimmer. 2015. *A Tour Beyond BIOS Supporting an SMM Resource Monitor Using the EFI Developer Kit II*. Technical Report. Intel. Retrieved from https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Supporting_SMM_Resource_Monitor_using_the_EFI_Developer_Kit_II.pdf.
[89] Jiewen Yao and Vincent J. Zimmer. 2017. *A Tour beyond BIOS—Memory Protection in UEFI BIOS*. Technical Report. Intel. Retrieved from https://edk2-docs.gitbooks.io/a-tour-beyond-bios-memory-protection-in-uefi-bios/content/.

Received March 2020; revised August 2020; accepted August 2020