Survivor: A Fine-Grained Intrusion Response and Recovery Approach for Commodity Operating Systems

Ronny Chevalier, David Plaquin, Chris Dalton, Guillaume Hiet

To cite this version:

Ronny Chevalier, David Plaquin, Chris Dalton, Guillaume Hiet. Survivor: A Fine-Grained Intrusion Response and Recovery Approach for Commodity Operating Systems. ACSAC 2019 - 35th Annual Computer Security Applications Conference, Dec 2019, San Juan, Puerto Rico. 10.1145/3359789.3359792 . hal-02289315v2

HAL Id: hal-02289315

https://hal.inria.fr/hal-02289315v2

Submitted on 26 Sep 2019

Survivor: A Fine-Grained Intrusion Response and Recovery Approach for Commodity Operating Systems

Ronny Chevalier
HP Labs
CentraleSupélec, Inria, CNRS, IRISA
ronny.chevalier@hp.com

David Plaquin
HP Labs
david.plaquin@hp.com

Chris Dalton
HP Labs
cid@hp.com

Guillaume Hiet
CentraleSupélec, Inria, CNRS, IRISA
guillaume.hiet@centralesupelec.fr

ABSTRACT

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**.

KEYWORDS

Intrusion Survivability, Intrusion Response, Intrusion Recovery

ACM Reference Format:
Ronny Chevalier, David Plaquin, Chris Dalton, and Guillaume Hiet. 2019. Survivor: A Fine-Grained Intrusion Response and Recovery Approach for Commodity Operating Systems. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3359789.3359792

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 [39] (e.g., do the benefits of an intrusion for criminals outweigh their costs?).

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 [23, 27, 34] (e.g., they force a system shutdown) or that they neither stop intrusions from reoccurring nor withstand reinfections [23, 27, 34, 71, 74]. 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, on the other hand, apply responses [20] to stop an intrusion or limit its impact on the system. However, existing approaches apply coarse-grained responses that affect the whole system and not just the compromised services [20] (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 [20, 62] 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 that 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

ACSAC '19, December 9–13, 2019, San Juan, PR, USA

service into a *degraded mode*, because they remove some privileges normally needed by the service.

The degraded mode is 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 it stops the attackers from achieving their goals. Second, the degraded mode 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 maximize 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 (section 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 [65], CRIU [12], Linux audit [29], and snapper [63] (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 document is structured as follows. First, in section 2, 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 which we then evaluate in section 7. In section 8, we discuss some limitations of our work. Finally, we conclude and give the next steps regarding our work in section 9.

2 RELATED WORK

Our work is based on the concept of survivability [37], and specifically intrusion survivability, since our approach focuses on withstanding ongoing intrusions. We make a trade-off between the availability of the functionalities of a vulnerable service and the security risk associated to maintaining them. In this section, we review existing work on intrusion recovery and response systems.

2.1 Intrusion Recovery Systems

Intrusion recovery systems [23, 27, 34, 71, 74] focus on system integrity by recovering legitimate persistent data. Except SHELF [74] and CRIU-MR [71], 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. Furthermore, except for CRIU-MR, they log all system events to later replay legitimate operations [23, 34, 74] or rollback illegitimate ones [27], 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 [74] and CRIU-MR [71]. SHELF recovers the state of processes and identify 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 Intrusion Detection System (IDS) notifies CRIU-MR that a container is infected, it checkpoints the container's state (using CRIU [12]), 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.

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.

2.2 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 [5, 20, 62] 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 [5], on attack graphs [20], or attack defense trees [62]. Shameli-Sendi et al. [62] 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, on the other hand, combines the ability to restore and to apply cost-sensitive per-service responses.

Some of these approaches [20, 62] 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. [28] 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 that 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.

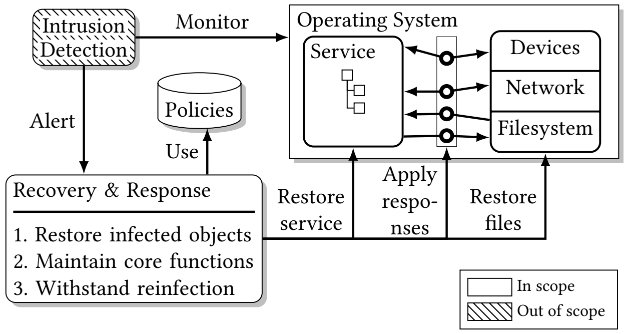


Figure 1: High-level overview of our intrusion survivability approach

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 filesystem 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 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., data theft) 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 trade-off between the availability of a function (that requires specific privileges) in a service and the intrusion risk. We designed a process to select cost-sensitive responses (see section 5) based on such policies. Administrators, developers, or maintainers 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 reasonable in recent firmware using a hardware-protected root of trust [26, 58] at boot time and protection of firmware runtime services [9, 75, 76]. For the OS kernel, one can use UEFI Secure Boot [69] at boot time, and rely on e.g., security invariants [64] or a hardware-based integrity monitor [4] 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,

[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.

- 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 (e.g., side-channel attacks [35, 38, 41, 60]),
- 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.

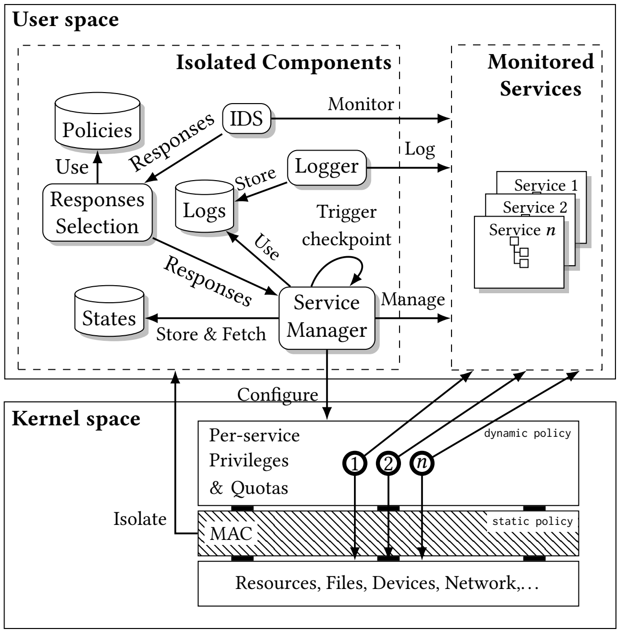


Figure 2: Overview of the architecture

4.1 Overview

During the normal operation of the OS, the *service manager* creates periodic checkpoints of the services and snapshots of the filesystem. 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 *responses 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 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. To select the last known safe state, we rely on the IDS to identify the first alert related to the intrusion. Then, we consider that the first state prior to this alert is safe.

4.2 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 [55], to isolate the components we used in our approach. 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 killed.[3] 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 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 [14] is SELinux-aware [70]).

4.3 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 [49] from Malware Attribute Enumeration and Characterization (MAEC) [36]), 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) [6] and MAEC [36, 51].

4.4 Service Manager

Commodity OSs rely on a user space service manager (e.g., the Service Control Manager [59] for Windows, or systemd [65] for Linux distributions) to launch and manage services. In our architecture, we rely on it, since it provides the appropriate level of abstraction

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

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 that 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 [24] in Linux or job objects [46] 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 framework to help select the cost-sensitive responses that minimize such a cost and maintain 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 inaccuracies, do not require such data, and are easier to understand [72]. In addition, we would like to limit the input from the user so that it improves the framework usability and its likelihood to be adopted in production.

In the rest of this section, we first describe the models that our framework relies on. Then, we detail how our framework selects cost-sensitive responses using such models.

5.1 Malicious Behaviors and Responses

Intrusions may exhibit multiple malicious behaviors that need to be stopped or mitigated differently. Here we work at the level of a malicious behavior and we select a response for each malicious behavior.

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 [49] from the project MAEC [36] of MITRE is a suitable candidate for such a hierarchy.[4] 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 $m \in M^i \implies m' \in M^i$.

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. We provide an example of this response hierarchy in Figure 4 of Appendix A.

5.2 Cost Models

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 response cost (in terms of quality of service lost) and malicious behavior cost that an administrator needs to set.

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 $rcost : \mathbf{S} \times \mathbf{R} \to \mathbf{C_r}$, 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.

For example, the policy of a web server could express that the ability to listen on ports 80 and 443 is critical for its core functions. However, if the web server would lose write access to the filesystem, the cost would be high and not critical, since it can still provide access to websites for many use cases.

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 $mbcost : \mathbf{S} \times \mathbf{M} \to \mathbf{C_{mb}}$, 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 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.

Following the same example, the policy could express that intrusions that compromise data availability (e.g., ransomware) have

[4] Another project that can help is the MITRE ATT&CK knowledge base [50], but it does not provide a hierarchy.

a high impact for the web server, since it would not provide access to the websites anymore. While on the other hand, it could express that an intrusion that only consumes system resources (e.g., a cryptocurrency mining malware) has a moderate cost.

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 than one that handles transactions.

5.3 Response Performance

While responses have varying costs on the quality of service, they also differ in performance against a malicious behavior. Hence, in our framework, we consider the performance as a criterion to select a response, among others.[5]

The space of qualitative response performances is denoted $\mathbf{P_r} \subseteq \mathbf{Q}$. We define $rperf: \mathbf{R} \times \mathbf{M} \to \mathbf{P_r}$, that 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 and proposed responses with their respective performance. Such information comes from threat intelligence sources that are shared, for example, using STIX. For example, STIX has a property called "efficacy" in its "course-of-action" object that represent responses.

5.4 Risk Matrix

We rely on the definition of a risk matrix that satisfies the axioms proposed (i.e., weak consistency, betweenness, and consistent coloring) to provide consistent risk assessments [1]. 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 4 of Appendix A is one instantiation of such a matrix.

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).

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

5.5 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, *mbcost* and *rcost* are defined for each service depending on its context, and *rperf* is constant and can be applied for any system. 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 response do not render a service unusable (i.e., by disabling a core functionality).

5.6 Optimal Response Selection

We now discuss how we use our 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 MOO methods [43] to select the most cost-effective response, as does other work on response selection [52, 62].

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}.

5.6.1 *Overview.*

When the IDS triggers an alert, it provides the confidence $i_{cf} \in \mathbf{I_{cf}}$ of the intrusion $i \in \mathbf{I}$ and the set of malicious behaviors $M^i \subseteq \mathbf{M}$. Before selecting an optimal response, we filter out any response that have 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:

$$\hat{R}^m = \{ r \in R^m \mid p_r < \text{critical} \}$$

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.

5.6.2 *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.

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 (or Pareto front). 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:

$$[p_r > p_{r'} \land c_r \le c_{r'}] \lor [p_r \ge p_{r'} \land c_r < c_{r'}]$$

MOO methods rely on preferences to choose solutions among the Pareto-optimal set (e.g., should we put the priority on the performance of the response or on reducing the cost?) [43]. They rely on a scalarization that converts a MOO problem into a SOO problem. One common scalarization approach is the weighted sum method that assigns a weight to each objective and compute the sum of the product of their respective objective. However, this method is not guaranteed to always give solutions in the Pareto-optimal set [43].

Shameli-Sendi et al. [62] decided to apply the weighted sum method on the Pareto-optimal set instead of on the whole solution

[5] The most effective response would be to stop the service. While our model allows it, in this paper we only mention responses that aim at maintaining the availability.

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:

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

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

We rely on a function l that maps the linguistic constants to a numerical value[6] between 0 and 1. In our case, we convert the constants critical, very high, high, moderate, low, very low, and none, to respectively the value 1, 0.9, 0.8, 0.5, 0.3, 0.1, and 0.

For the weights, we use the risk $k = risk(c_{mb}, i_{cf})$ as a weight for the performance of the response $w_p = l(k)$ which also gives us the weight for the cost of a response $w_c = 1 - w_p$. It 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:

$$\underset{r \in O}{\arg\max}\, w_p l(p_r) + w_c(1 - l(c_r))$$

6 IMPLEMENTATION

We implemented a Linux-based prototype by modifying several existing projects. While our implementation relies on Linux features such as namespaces [31], seccomp [10], or cgroups [24], our approach does not depend on OS-specific paradigms. For example, on Windows, one could use Integrity Mechanism [45], Restricted Tokens [48], and Job Objects [46]. 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 1 the different projects we modified where we added in total nearly 3600 lines of C code.

Table 1: Projects modified for our implementation

Project	From version	Code added
CRIU	3.9	383 lines of C
systemd	239	2639 lines of C
audit		
user space	2.8.3	79 lines of C
Linux kernel	4.17.5	460 lines of C
Total		3561 lines of C

At the time of writing, the most common service manager on Linux-based systems is systemd [65]. We modified it to checkpoint and to restore services using CRIU [12] and snapper [63], and to apply responses at the end of the restoration.

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 [57]). 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 that 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 that 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 [3, 29] 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

[6] An alternative would be to use fuzzy logic to reflect the uncertainty regarding the risk assessment from experts when using linguistic constants [15].

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 in order 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 our solution incurs on the system resources?
(5) Do services continue to function (i.e., no crash) when they are restored with less privileges that 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 (see Appendix B.1 for more details). While malware could use anti-virtualization techniques [8, 56], to the best of our knowledge, none of our samples used such techniques.[7] 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, such as web servers (nginx [53], Apache [2]), database (mariadb [42]), work queue (beanstalkd [7]), message queue (mosquitto [19]), or git hosting services (gitea [21]). It shows that our approach is applicable to a diverse set of services.

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 Appendix B.2 for the hashes of the malware samples):

Linux.BitCoinMiner Cryptocurrency mining malware that connects to a mining pool using attackers-controlled credentials [68].

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 [17].

Hakai Malware that receives instructions from a Command and Control (C&C) server to launch DDoS attacks, and to infect other systems by brute forcing credentials or exploiting vulnerabilities in routers [18, 54].

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 that the administrator can see the ransom note [16].

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` [25].

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, with different 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.

Table 2: 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

The results we obtained are summarized in Table 2. 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

[7] This is consistent with the study of Cozzi et al. [11] 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.

self-hosted Git-repository hosting service (an open source clone of the services provided by GitHub [22]), as a use case for a service. We chose Gitea, because it requires a diverse set of privileges and it shows how our approach can be applied to a complex real-world service.

In our use case, we configured Gitea with the principle of least privileges. It means that restrictions which corresponds 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 and our approach handles such cases.

We consider an intrusion that compromised our Gitea service with the `Linux.Encoder.1` ransomware.[8] When executed, it encrypts all the git repositories and the database used by Gitea. Hence, we previously configured the policy to set the cost of such a malicious behavior to high,[9] since it would render the site almost unusable: *mbcost*("gitea", "compromise-data-availability") = "high".

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 [32].

In practice, however, 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 3, we display a set of responses for the ransomware that we devised based on existing strategies to mitigate ransomware, such as CryptoDrop lockdown [13] or Windows controlled folder access [47]. None of them could have been applied proactively by the developer, because they degrade the quality of service. We set their respective cost for the service and the estimated performance.

Table 3: Responses to withstand ransomware reinfection with their associated cost and performance for Gitea

#	Response	Cost	Performance
1	Disable open system call	Very High	Very High
2	Read-only filesystem 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 filesystem	Critical	Very High

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 filesystem as read-only protecting all information stored by Gitea (git repositories and its database). The only exception is the folder used by Gitea to store sessions since having this folder read-only would render the site unusable, thus it is a core function (see the cost critical in Table 3). Gitea is restored with all the encrypted files. The selected response prevents the attacker to reinfect the service since the exploit require 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 previously mentioned 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 unlikely, but it is in fact a false positive. The risk computed is *risk*("high", "very unlikely") = low. It is similar to the previous case where the response selected is response 5 due to a low risk. However, in this case, there is no actual ransomware. In terms of quality of service, users have access to most functions (e.g., login, clone all repositories, or add issues), they just cannot push modifications to the protected repositories. 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 make the service leave the degraded mode.

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 four common 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. In average the time to checkpoint was always below 300 ms. Table 6 of Appendix B.3 gives more detailed timings.

The 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 for a small period of time (maximum 300 ms), we consider such a cost acceptable. In comparison, SHELF [74]

[8] In our experiments, we used an exploit for the version 1.4.0 [66].

[9] One would have to assign a cost for other malicious behaviors, but for the sake of conciseness we only show the relevant ones.

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 used in section 7.3.1. In average, it took less than 325 ms. Table 7 of Appendix B.3 gives more detailed timings.

In contrast to the checkpoint, the restore procedure loses all connections due to the kill operation. The experiment, however, show that the time to restore a service is small (less than 325 ms). For example, in comparison, CRIU-MR [71] took 2.8 s in average to complete their restoration procedure.

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 (i.e., even for the non-monitored services). 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 both overhead by running synthetic and real-world workload benchmarks, from the Phoronix test suite [40], 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).

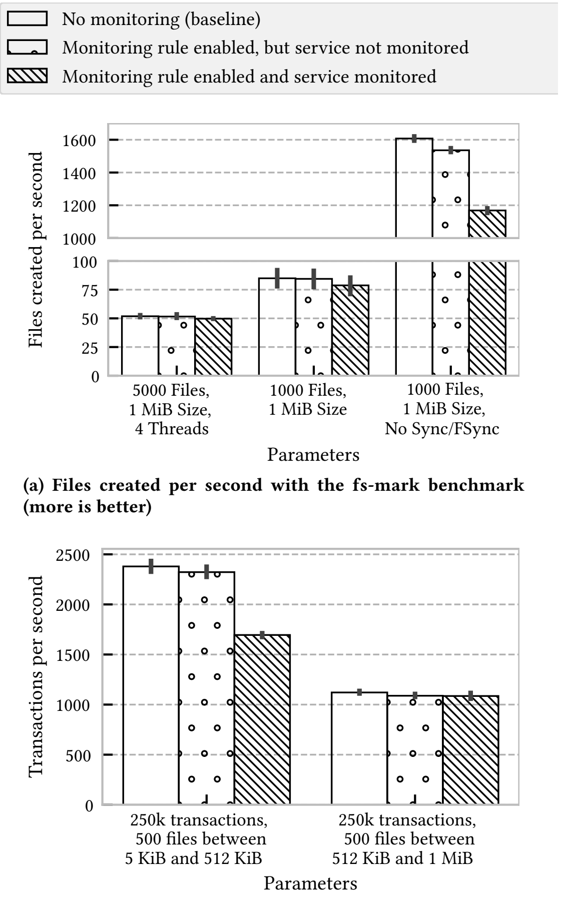
7.4.1 Synthetic Benchmarks. We ran synthetic I/O benchmarks that stress the system by performing many open, read, and write system calls: `compilebench` [44], `fs-mark` [73], and `Postmark` [30]. `compilebench` emulates disk I/O operations related to the compilation of a kernel tree, reading the tree, or its creation. `fs-mark` creates files and directories, at a given rate and size, either synchronously or asynchronously. `Postmark` emulates an email server by performing a given number of transactions that create, read, append to, and delete files of varying sizes.

The results of the `read compiled tree` test of the `compilebench` benchmark confirmed that the overhead is only due to `open` system calls with write access mode. This test only reads files and we did 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, illustrated in Figure 3. In both experiments, we notice a small overhead when the service is not monitored (between 0.6 % and 4.5 %). With `fs-mark` (Figure 3a), when writing 1000 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 3b), we observe a high overhead (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` measures the time to compile the Linux kernel[10] and `unpack-linux` measures the time to extract the archive of the Linux kernel source code.



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

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

Figure 3: Results of synthetic benchmarks to measure the overhead of the monitoring

When the service is monitored, the overhead is only significant with `unpack-linux` 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`, 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 [74] 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.

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

7.5 Storage Space Overhead

Checkpointing services requires 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 web servers (nginx, Apache), databases (mariadb), work queues (beanstalkd), message queues (mosquitto), and git hosting services (gitea). None of the services crashed when restored with a policy that removed privileged that they required (i.e., 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 when this is not the case.

8 DISCUSSION

Let us now discuss non-exhaustively limitations or areas that would need further work.

False Positives Our approach relies on an IDS, hence we inherit its limitations. In the case of false positives, the recovery and response procedures would impact the service availability, despite thwarting no threat. Our approach, however, minimizes this risk by considering the likelihood of the intrusion for the selection of cost-sensitive responses and by ensuring that we maintain core functions.

CRIU Limitations At the moment, CRIU cannot support all 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 its state may be impossible (except for virtual devices not corresponding to any physical devices). 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.

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 [33, 62, 67].

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 [6] and malicious behaviors [36, 49, 51] 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 our experiments, we extracted information about malicious behaviors from textual descriptions [16–18, 54, 68] and reused the existing standards to describe such malicious behaviors [36, 49, 51]. Likewise, we extracted information about responses to counter such malicious behaviors from textual descriptions [13, 16–18, 47, 54, 68].

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 can automatically consider generic responses. For example, with ransomware we know that responses that either render the filesystem read-only or only specific directories will work. We would not know that, for instance, blocking a specific system call would have stopped the malware, but we know what all ransomware need, and we can respond accordingly. Such generic responses might help mitigate the lack of precise information.

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 framework to select cost-sensitive responses that do 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.

ACKNOWLEDGMENTS

The authors would like to thank Pierre Belgarric and Maugan Villatel for their helpful comments, feedback, and proofing of earlier versions of this paper. We would also like to thank Stuart Lees, Daniel Ellam, and Jonathan Griffin, for their help in setting up and running some of the experiments using their isolated and virtualized environment. In addition, we would like to thank the anonymous reviewers for their feedback. Finally, we would like to thank Hybrid Analysis for providing access to malware samples used in our experiments.

A EXAMPLES OF MODELS

Figure 4 is an example of a non-exhaustive per-service response hierarchy that can be used for the hierarchy defined in section 5.1. 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. It means that if an administrator only specified the cost of putting `/var` read-only, but a response, among the set of possible responses, sets `/var/www` read-only, our response selection framework uses the cost of its parent (i.e., `/var`).

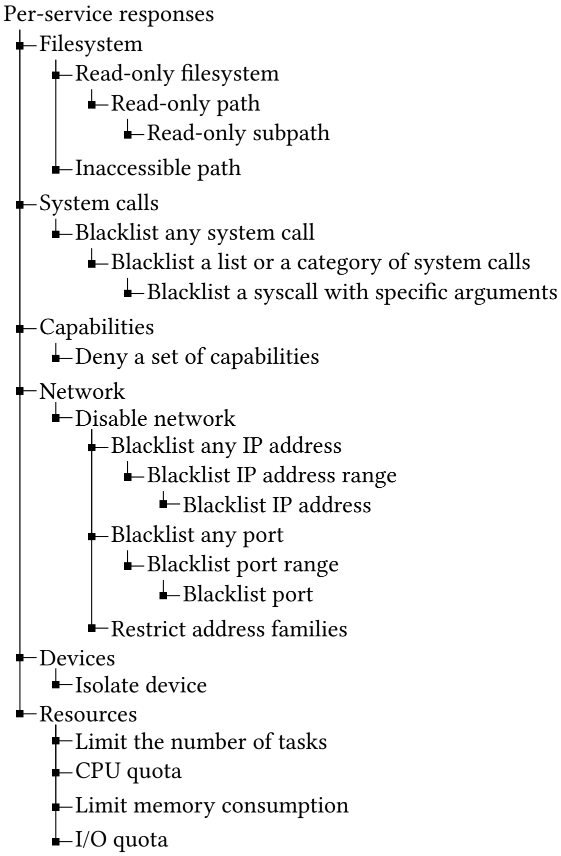


Figure 4: Example of a non-exhaustive per-service response hierarchy

Table 4 is an example of a risk matrix that can be used. This matrix can vary depending on the risk attitude (risk averse, risk neutral, or risk seeking).

Table 4: Example of a 5 × 5 risk matrix that follows the requirements for our risk assessment

	Malicious Behavior Cost				
Confidence (Likelihood)	Very low	Low	Moderate	High	Very high
Very likely	L	M	H	H	H
Likely	L	M	M	H	H
Probable	L	L	M	M	H
Unlikely	L	L	L	M	M
Very unlikely	L	L	L	L	L

B EVALUATION DETAILS

B.1 Setup of the Virtualized Environment

We ran the experiments regarding the effectiveness of our responses in a virtualized environment. It helped us control malware propagation and their behavior in general.

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 VM connected to an isolated VLAN with access to the Internet. We connected the second VM, that 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.

B.2 Malware Samples

Table 5: 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`

In Table 5, we list the malware samples used in our experiments alongside their respective SHA-256 hash.

B.3 Checkpoint and Restore Operations

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 [61]. 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. As mentioned previously, all requests were successful (i.e., no errors or timeouts) and the maximum latency during a checkpoint was 286 ms.

In Table 6, we show the time measured to perform the different operations executed during a checkpoint: initialize (i.e., to initialize structures, to create or open directories, and to freeze processes),

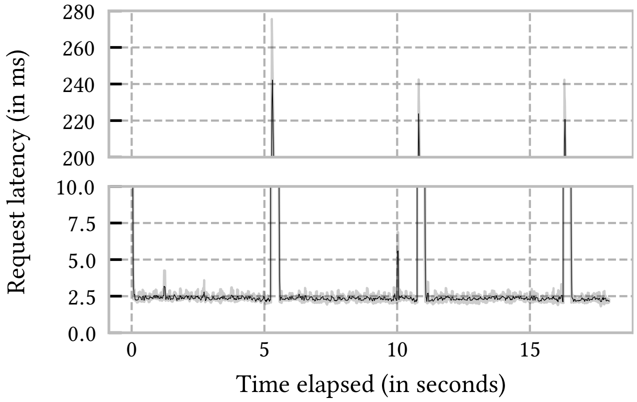


Figure 5: Impact of checkpoints on the latency of HTTP requests made to an nginx server (less is better)

snapshot of the filesystem, serialization of the service's metadata, and checkpointing of the processes using CRIU. The time to perform this last operation varies depending on the service (e.g., the number of processes, memory used, or files opened).

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 filesystem	(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		

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		

REFERENCES

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

[2] apache 2019. *Apache HTTP Server*. Retrieved September 20, 2019 from https://httpd.apache.org/

[3] audit 2019. *The Linux Audit Project*. Retrieved September 20, 2019 from https://github.com/linux-audit/

[4] 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 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, 90–102. https://doi.org/10.1145/2660267.2660350

[5] Ivan Balepin, Sergei Maltsev, Jeff Rowe, and Karl Levitt. 2003. Using Specification-Based Intrusion Detection for Automated Response. In *Recent Advances in Intrusion Detection*. 136–154. https://doi.org/10.1007/978-3-540-45248-5_8

[6] Sean Barnum. 2014. Standardizing cyber threat intelligence information with the Structured Threat Information eXpression (STIX).

[7] beanstalkd 2019. *beanstalkd*. Retrieved September 20, 2019 from https://kr.github.io/beanstalkd/

[8] 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 *38th IEEE/IFIP International Conference On Dependable Systems and Networks*. 177–186. https://doi.org/10.1109/DSN.2008.4630086

[9] 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 Annual Computer Security Applications Conference (ACSAC'17)*. ACM, 399–411. https://doi.org/10.1145/3134600.3134622

[10] Jonathan Corbet. 2009. Seccomp and sandboxing. *LWN* (13 May 2009). https://lwn.net/Articles/332974/

[11] Emanuele Cozzi, Mariano Graziano, Yannick Fratantonio, and Davide Balzarotti. 2018. Understanding Linux Malware. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP '18)*. 161–175. https://doi.org/10.1109/SP.2018.00054

[12] CRIU 2018. *CRIU*. Retrieved September 20, 2019 from https://criu.org/

[13] CryptoDrop, LLC. 2019. *CryptoDrop*. Retrieved September 20, 2019 from https://www.cryptodrop.org/

[14] dbus 2019. *D-Bus*. Retrieved September 20, 2019 from https://www.freedesktop.org/wiki/Software/dbus/

[15] Yong Deng, Rehan Sadiq, Wen Jiang, and Solomon Tesfamariam. 2011. Risk analysis in a linguistic environment: a fuzzy evidential reasoning-based approach. *Expert Systems with Applications* 38, 12 (2011), 15438–15446. https://doi.org/10.1016/j.eswa.2011.06.018

[16] Dr. Web. 2015. *Linux.Encoder.1*. Retrieved September 20, 2019 from https://vms.drweb.com/virus/?i=7703983

[17] Dr. Web. 2016. *Linux.Rex.1*. Retrieved September 20, 2019 from https://vms.drweb.com/virus/?i=8436299

[18] Dr. Web. 2018. *Linux.BackDoor.Fgt.1430*. Retrieved September 20, 2019 from https://vms.drweb.com/virus/?i=17573534

[19] Eclipse Foundation. 2019. *Mosquitto*. Retrieved September 20, 2019 from https://mosquitto.org/

[20] 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. https://doi.org/10.1109/DSN.2005.17

[21] gitea 2019. *Gitea*. Retrieved September 20, 2019 from https://gitea.io/

[22] GitHub, Inc. 2019. *GitHub*. Retrieved September 20, 2019 from https://github.com/

[23] 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. https://doi.org/10.1145/1095810.1095826

[24] Tejun Heo. 2015. *Control Group v2*. Retrieved September 20, 2019 from https://www.kernel.org/doc/Documentation/cgroup-v2.txt

[25] Daniel Hodson. 2017. *Remote LD_PRELOAD Exploitation*. Retrieved September 20, 2019 from https://www.elttam.com.au/blog/goahead/

[26] HP Inc. 2019. *HP Sure Start: Automatic Firmware Intrusion Detection and Repair.* Technical Report. HP Inc. http://h10032.www1.hp.com/ctg/Manual/c06216928

[27] 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 Annual Computer Security Applications Conference (ACSAC '06)*. 257–268. https://doi.org/10.1109/ACSAC.2006.16

[28] Zhen Huang, Mariana DAngelo, Dhaval Miyani, and David Lie. 2016. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*. 618–635. https://doi.org/10.1109/SP.2016.43

[29] 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*. Chapter 6, 185–204.

[30] Jeffrey Katcher. 1997. *Postmark: A new file system benchmark.* Technical Report 3022. Network Appliance.

[31] Michael Kerrisk. 2013. Namespaces in operation, part 1: namespaces overview. *LWN* (4 Jan. 2013). https://lwn.net/Articles/531114/

[32] 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 *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 3–24. https://doi.org/10.1007/978-3-319-20550-2_1

[33] Nizar Kheir, Hervé Debar, Nora Cuppens-Boulahia, Frédéric Cuppens, and Jouni Viinikka. 2009. Cost Evaluation for Intrusion Response Using Dependency Graphs. In *International Conference on Network and Service Security.*

[34] 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.

[35] 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 *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA '14)*. IEEE Press, 361–372. https://doi.org/10.1109/ISCA.2014.6853210

[36] Ivan Kirillov, Desiree Beck, Penny Chase, and Robert Martin. 2011. Malware Attribute Enumeration and Characterization.

[37] 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. https://doi.org/10.1109/DISCEX.2003.1194874

[38] 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 *40th IEEE Symposium on Security and Privacy (S&P'19)*.

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

[40] Michael Larabel and Matthew Tippett. 2019. *Phoronix Test Suite.* Retrieved September 20, 2019 from https://www.phoronix-test-suite.com/

[41] 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 *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990.

[42] mariadb 2019. *mariadb.* Retrieved September 20, 2019 from https://mariadb.org/

[43] R. Timothy Marler and Jasbir S. Arora. 2004. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization* 26, 6 (April 2004), 369–395. https://doi.org/10.1007/s00158-003-0368-6

[44] Chris Mason. 2008. *Compilebench.* Retrieved September 20, 2019 from https://oss.oracle.com/~mason/compilebench/

[45] Microsoft. 2017. *Windows Integrity Mechanism Design.* Retrieved September 20, 2019 from https://msdn.microsoft.com/en-us/library/bb625963.aspx

[46] Microsoft. 2018. *Job Objects.* Retrieved September 20, 2019 from https://msdn.microsoft.com/en-us/library/windows/desktop/ms684161(v=vs.85).aspx MSDN.

[47] Microsoft. 2018. *Protect important folders with controlled folder access.* Retrieved September 20, 2019 from https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-exploit-guard/controlled-folders-exploit-guard?ocid=cx-blog-mmpc

[48] Microsoft. 2018. *Restricted Tokens.* Retrieved September 20, 2019 from https://msdn.microsoft.com/en-us/library/windows/desktop/aa379316(v=vs.85).aspx

[49] MITRE. 2014. *Malware Capabilities.* Retrieved September 20, 2019 from https://github.com/MAECProject/schemas/wiki/Malware-Capabilities

[50] MITRE. 2019. *ATT&CK.* Retrieved September 20, 2019 from https://attack.mitre.org/

[51] MITRE. 2019. *Encyclopedia of Malware Attributes.* Retrieved September 20, 2019 from https://collaborate.mitre.org/ema/

[52] 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 Journal on Information Security* (2017), 12. https://doi.org/10.1186/s13635-017-0063-6

[53] nginx 2019. *nginx.* Retrieved September 20, 2019 from https://nginx.org/

[54] Ruchna Nigam. 2018. *Unit 42 Finds New Mirai and Gafgyt IoT/Linux Botnet Campaigns.* Retrieved September 20, 2019 from https://researchcenter.paloaltonetworks.com/2018/07/unit42-finds-new-mirai-gafgyt-iotlinux-botnet-campaigns/

[55] NSA and Red Hat. 2019. *SELinux.* Retrieved September 20, 2019 from https://selinuxproject.org/

[56] 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.

[57] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree Filesystem. *ACM Transactions on Storage (TOS)* 9, 3 (2013), 9. https://doi.org/10.1145/2501620.2501623

[58] 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, Chapter 6, 143–163. https://doi.org/10.1007/978-1-4302-6572-6_6

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

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

[61] Tomás Senart. 2019. *Vegeta.* Retrieved September 20, 2019 from https://github.com/tsenart/vegeta

[62] Alireza Shameli-Sendi, Habib Louafi, Wenbo He, and Mohamed Cheriet. 2018. Dynamic Optimal Countermeasure Selection for Intrusion Response System. *IEEE Transactions on Dependable and Secure Computing* 15, 5 (2018), 755–770. https://doi.org/10.1109/TDSC.2016.2615622

[63] snapper 2018. *snapper.* Retrieved September 20, 2019 from http://snapper.io/

[64] 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 2016 Annual Network and Distributed System Security Symposium (NDSS '16)*. San Diego, CA. https://doi.org/10.14722/ndss.2016.23218

[65] systemd 2019. *systemd System and Service Manager.* Retrieved September 20, 2019 from https://www.freedesktop.org/wiki/Software/systemd/

[66] Kacper Szurek. 2018. *Gitea 1.4.0 Unauthenticated Remote Code Execution.* Retrieved September 20, 2019 from https://security.szurek.pl/gitea-1-4-0-unauthenticated-rce.html

[67] Thomas Toth and Christopher Kruegel. 2002. Evaluating the Impact of Automated Intrusion Response Mechanisms. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC '02)*. IEEE Computer Society. https://doi.org/10.1109/CSAC.2002.1176302

[68] Trend Micro Cyber Safety Solutions Team. 2018. *Cryptocurrency Miner Distributed via PHP Weathermap Vulnerability, Targets Linux Servers.* Retrieved September 20, 2019 from https://blog.trendmicro.com/trendlabs-security-intelligence/cryptocurrency-miner-distributed-via-php-weathermap-vulnerability-targets-linux-servers/

[69] UEFI Forum. 2019. *Unified Extensible Firmware Interface Specification.* https://uefi.org/sites/default/files/resources/UEFI_Spec_2_8_final.pdf Version 2.8.

[70] Sven Vermeulen. 2014. Handling SELinux-aware Applications. In *SELinux Cookbook.* Packt Publishing, Chapter 10.

[71] 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, Baltimore, MD, 1199–1211.

[72] Evan Wheeler. 2011. Risky Business. In *Security risk management: Building an information security risk management program from the Ground Up* (1st ed.). Syngress Publishing, Chapter 2, 37–40.

[73] Ric Wheeler. 2016. *fs-mark.* Retrieved September 20, 2019 from https://sourceforge.net/projects/fsmark/

[74] Xi Xiong, Xiaoqi Jia, and Peng Liu. 2009. SHELF: Preserving Business Continuity and Availability in an Intrusion Recovery System. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC '09)*. IEEE Computer Society, 484–493. https://doi.org/10.1109/ACSAC.2009.52

[75] 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. https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Supporting_SMM_Resource_Monitor_using_the_EFI_Developer_Kit_II.pdf

[76] Jiewen Yao and Vincent J. Zimmer. 2017. *A Tour Beyond BIOS - Memory Protection in UEFI BIOS.* Technical Report. Intel. https://edk2-docs.gitbooks.io/a-tour-beyond-bios-memory-protection-in-uefi-bios/content/