algorithms

MDPI

Article

User Armor: An Extension for AppArmor

Mario Alviano *,† and Pierpaolo Sestito †

Department of Mathematics and Computer Science, University of Calabria, 87036 Rende, Italy; pierpaolo.sestito@outlook.it

* Correspondence: mario.alviano@unical.it

† These authors contributed equally to this work.

Abstract: AppArmor is a mandatory access control (MAC) system for Linux based on profiles. It focuses on protecting processes, without differentiating profiles based on the users running the processes themselves. Moreover, it does not implement inheritance mechanisms to simplify the management of profiles and avoid the duplication of rules. This work introduces UserArmor, an extension of AppArmor that overcomes the aforementioned limitations by allowing specific profiles to be associated with users and implementing an inheritance system to reduce complexity, improve reusability, and ensure consistency in security rules. An application to Answer Set Programming is discussed.

Keywords: mandatory access control (MAC); profile inheritance; user-based profiles

1. Introduction

AppArmor [1] is a security Mandatory Access Control (MAC) framework for Linux [2,3], based on security profiles that apply restriction on the resources available to processes [4,5]. In fact, these profiles define strict policies for accessing files, directories, networks and other critical resources [5,6]. Restrictions are applied at the kernel level, providing a mandatory control independent of application behavior [4]. Even if AppArmor improves multi-user security, it lacks flexibility in managing user-specific rules. Specifically, AppArmor allows profiles to be defined for applications or processes, but it lacks a mechanism to differentiate between different users running the same process. Suck a lack of user-level control makes it complex to ensure security in multi-user scenarios.

Example 1. *Let us consider an application that can be executed by two users, namely user1 and user2. The application writes to a file whose name depends on the user running the process. The most restrictive polices that can be implemented in AppArmor for such a scenario must include the following rules:*

```
/var/log/my_confined_app/user1.log rw,
/var/log/my_confined_app/user2.log rw,
```

It is not possible to further restrict permissions without breaking the application logic, i.e., each user must be able to access its own file via the application. This can lead to the following vulnerabilities: a bug in the application could allow user1 to read or write to the file associated with user2, compromising the confidentiality and integrity of such potentially sensible resources.

The proposed extension involves the use of different subprofiles for each user. For the scenario reported above, we would have the following subprofiles:

```
profile user1 {
    ...
```


Academic Editor: Francesc Pozo

Received: 14 February 2025
Revised: 3 March 2025
Accepted: 19 March 2025
Published: 24 March 2025

Citation: Alviano, M.; Sestito, P. User Armor: An Extension for AppArmor. *Algorithms* **2025**, *18*, 185. https://doi.org/10.3390/a18040185

```
    /var/log/my_confined_app/user1.log rw,
}
profile user2 {
    ...
    /var/log/my_confined_app/user2.log rw,
}
```

Therefore, when the application is executed by `user1`*, the subprofile* `user1` *is enforced, and in this way, access to the* `user2.log` *file is inhibited at kernel level, even in the case of bugs in the application.*

Another limitation of AppArmor is the absence of an inheritance mechanism for profile policies. Even if profiles can be nested, creating a seemingly hierarchical structure, AppArmor subprofiles are designed to define different security rules for subprocesses of an application. The way they are designed, subprofiles do not inherit the rules of the main profile, as it is possible that a subprocess needs a less restrictive security policy to work correctly. Technically, AppArmor implements a notion of abstractions to reuse common rules; such rules are stored in abstraction files and can be reused in other security policies via the `#include` directive. However, abstractions are an all or nothing option: it is not possible to select specific rules within an abstraction, for example, based on the user running the process. Due to the absence of inheritance mechanisms, policy management is cumbersome and prone to errors, especially in complex environments where configurations must be updated frequently or affect multiple users.

Example 2. *Consider a Bash Application* `my_confined_app` *that needs the* `cat` *command to read a configuration file and access the network. The application can be run by two users, both sudoers, only one of whom needs administrative capabilities. As in Example 1, users must have read and written access to their log files. The use of AppArmor that we propose would therefore need the following file:*

```
#include <tunables/global>
/usr/bin/my_confined_app {
    profile user1 {
        #include <abstractions/base>
        #include <abstractions/bash>
        /usr/bin/my_confined_app r,
        /etc/my_confined_app.conf r,
        /usr/bin/cat ix,
        network inet,
        /var/log/my_confined_app/user1.log rw,
    }

    profile user2 {
        #include <abstractions/base>
        #include <abstractions/bash>
        /usr/bin/my_confined_app r,
        /etc/my_confined_app.conf r,
        /usr/bin/cat ix,
        capability sys_admin,
        network inet,
        /var/log/my_confined_app/user2.log rw,
    }
}
```

Above, the included abstractions set common permissions for Bash scripts, including access to `.bash_profile`*,* `.bash_rc` *and* `.profile` *files. The file above has several duplicate rules. The*

extension proposed in this paper provides an inheritance system by which the described scenario can be modeled as follows:

```
#include <tunables/global>
/usr/bin/my_confined_app {
    #include <abstractions/base>
    #include <abstractions/bash>
    /usr/bin/my_confined_app r,
    /etc/my_confined_app.conf r,
    /usr/bin/cat ix,
    #@selectable{adm}  capability sys_admin,
    #@selectable{net}  network inet,

    profile user1 {
        #@select: adm net
        /var/log/my_confined_app/user1.log rw,
    }

    profile user2 {
        #@select: net
        /var/log/my_confined_app/user2.log rw,
    }
}
```

In the above policy file, the `#@selectable` *and* `#@select` *directives avoid the duplication of rules. Their usage is detailed in Section 3.*

This paper presents UserArmor, an extension of AppArmor, with the goal of achieving greater granularity in user-level security policy management, and introducing an inheritance system based on tags. The user-level granularity is achieved by structuring AppArmor security policies hierarchically. In the proposed structure, the general application profile serves as the base level, while every user executing the application can be associated with a (nested) subprofile. As shown in Figure 1, in UserArmor, each confined application is associated with a directory containing user subprofiles, each one stored in a separate file to favor the modularity and scalability of the approach. The profiles are then copied into a single file, namely `mappings`, that is included in the security policy via the `#include` directive. In this way, when the security policy of an application `confined_app` is processed, subprofiles are loaded into the kernel and associated with names like `confined_app//user_name`.

UserArmor automates the creation and management of the hierarchical structure of profiles and the activation of the profile associated with the user running an application. Moreover, its inheritance system is designed to eliminate the duplication of rules. The idea is to tag some rules as selectable, and specify in each subprofile which tags to select. Untagged rules, on the other hand, are essential and included in all subprofiles. Figure 2 illustrates the usage of UserArmor for the scenario described in Example 2. UserArmor command-line tools, namely `ua-generate`, `ua-enforce` and `ua-exec`, are described in Section 3.

The remainder of this article is structured as follows. Section 2 introduces the required background on AppArmor, and in particular, the notion of profile and the include directives. Section 3 defines the proposed tag system to reuse AppArmor rules, and the command-line tools introduced by UserArmor are as follows: `ua-generate` to produce skeleton files for user profiles; `ua-enforce` to process user profiles and obtain files understandable by AppArmor; `ua-exec` to execute confined application with the correct profile. Section 4 overviews related works in the literature. Section 5 presents an experiment aimed at measuring the overhead introduced by UserArmor with respect to AppArmor, that is, to exclude denial-of-service

(DoS) vulnerabilities due to excess resource consumption to enforce the user-based policies. The results of our experiment show that the overhead is minimal, and significantly less than the overhead introduced by sandboxing tools such as Bubblewrap. Section 6 reports an application of UserArmor to harden Answer Set Programming solvers [7–11], and in particular, the state-of-the-art solver CLINGO [12].

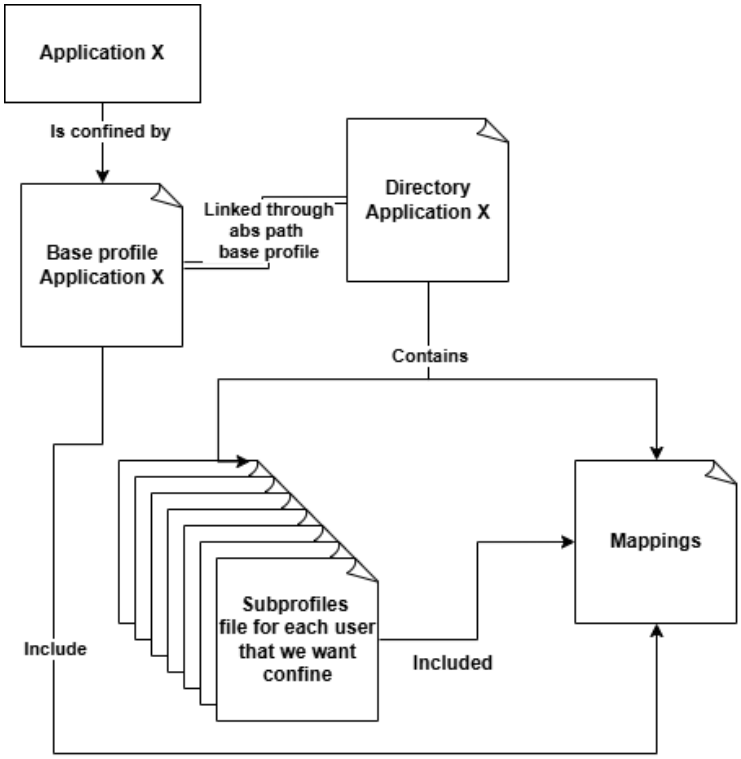

Figure 1. Hierarchical structure of UserArmor profiles.

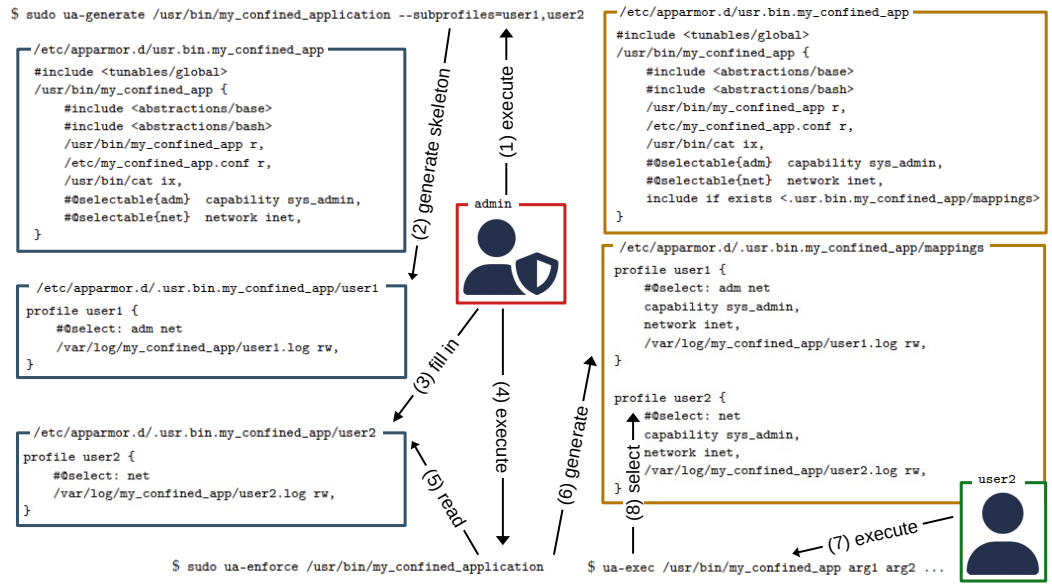

Figure 2. UserArmor usage for Example 2. Initially, the administrator executes `ua-generate` to produce skeleton files for user profiles and fills them in with the required permissions (blue boxes). The administrator can take advantage of the UserArmor tag system to reuse common rules. After that, the administrator executes `ua-enforce` to process the written profiles (blue boxes) and generate files that are understandable by AppArmor (orange boxes). Finally, `user2` executes the confined application via `ua-exec`, which selects the profile associated with the user (i.e., `/usr/bin/my_confined_app//user2`).

2. Background

In AppArmor, processes can be associated with *profiles* that enables access to system resources. Profile files are conventionally saved in the directory `/etc/apparmor.d/`, with filenames obtained from the absolute path of the executable files (replacing `/` with `.`). A profile file has the form

```
profile NAME /ABSOLUTE/PATH {
    RULES
    SUBPROFILES
}
```

where `RULES` is a list of rules (defined next), and `SUBPROFILES` is a possibly empty list of profiles; the keyword `profile` and the `NAME` can be omitted from the main profile. The rules may concern *files*, *capabilities* and *networking*. A rule about files has the form

```
/ABSOLUTE/PATH FLAGS,
```

where `FLAGS` includes one or more flags; flags relevant to this work are `r`, `w` and `x` for reading, writing and executing files; and `ix` for executing files maintaining the current profile. A rule on capabilities has the form

```
capability CAPABILITIES,
```

where `CAPABILITIES` is a capability in the Linux kernel (e.g., `setuid` or `setguid`). A rule on networking has the form

```
network TYPE,
```

where `TYPE` is the type of communication (e.g., `inet` for IPv4 or `inet6` for IPv6). Rules can be preceded by the keyword `deny` to block access (rather than allow access).

AppArmor has several tools for simplifying profile generation and management. The tools most relevant for this work are the following: `aa-genprof` generates a new profile by tracking the execution of an application and asking the user to confirm or deny the accesses requested by the application; `aa-logprof` analyzes the audit log files and helps improve an existing profile by suggesting new rules based on recorded events; `apparmor_parser` loads and reloads existing profiles into the kernel, verifies and compiles profiles transforming them into a kernel format; `aa-complain` activates the logging of a profile, without enforcing its restrictions; `aa-enforce` applies restrictions (and logging) of a profile; `aa-disable` disallows restrictions and logging of a profile.

Finally, AppArmor allows the reuse of common rules through abstraction files, which can be included with the directive

```
#include <file>
```

where `file` is the path of the abstraction file relative to the directory `/etc/apparmor.d/`. If the file does not exist, AppArmor raises an error. Alternatively, it is possible to use

```
include if exists <file>
```

to include the file under the condition of its existence.

3. User Armor

The tag system uses a comment-based syntax, described below. Rules (and blocks) are associated with aliases (i.e., identifying strings).

- Base rules. Untagged rules are considered essential and are automatically inherited by all subprofiles.
- Selectable rules. Rules starting with a tag

  ```
  #@selectable{ALIAS}
  ```

are not inherited automatically (and are not part of the main profile), but they can be included in the subprofiles via their `ALIAS`.

- Selectable blocks. A syntax similar to the previous one can be applied to rule blocks as follows:

```
#@selectable{ALIAS}
#     RULES
#@end
```

- Removable rules. Rules ending with a tag

```
#@removable{ALIAS}
```

are inherited in subprofiles unless explicitly removed by their `ALIAS`. Removable rules enable the possibility to have a more relaxed base policy (which can ease the gradual integration of UserArmor, but this is discouraged on a stable environment). Note that there is no notion of a removable block.

- Subprofile inheritance. Subprofiles can select rules and blocks using the following tag:

```
#@select: ALIASES
```

where `ALIASES` is a space-separated list of aliases. Selected rules and blocks are added to the basic rules. Untagged rules are added at the beginning of each subprofile, unless their alias is listed in the following tag:

```
#@remove: ALIASES
```

The tag system is applied to profile files within a directory associated with the confined application. The directory contains a profile file for each involved user and a `mappings` file that includes all the other files in the directory. The `mappings` file can then be included in the profile file in `/etc/apparmor.d`. In this way, AppArmor loads subprofiles into the kernel with name `confined_app//user_name`, and UserArmor can easily apply the correct profile by identifying the user who is running the application.

Example 3 (Continuing Example 2)**.** *The scenario depicted in the introduction can be modeled by the following profile file* `/etc/apparmor.d/usr.bin.my_confined_app`*:*

```
#include <tunables/global>
/usr/bin/my_confined_app {
    #include <abstractions/base>
    #include <abstractions/bash>
    /usr/bin/my_confined_app r,
    /etc/my_confined_app.conf r,
    /usr/bin/cat ix,
    #@selectable{adm}  capability sys_admin,
    #@selectable{net}  network inet,
    include if exists <.usr.bin.my_confined_app/mappings>
}
```

Additional files are stored in the directory `/etc/apparmor.d/.usr.bin.my_confined_app`*, as follows:*

- `user1`*, associated with the first user, with the following content:*

```
profile user1 {
    #@select: adm net
    /var/log/my_confined_app/user1.log rw,
}
```

- `user2`*, associated with the second user, with the following content:*

```
profile user2 {
```

```
    #@select: net
    /var/log/my_confined_app/user2.log rw,
}
```

- `mappings`, *including the above files with the expansion of the selected permissions, as follows:*

```
profile user1 {
    #@select: adm net
    capability sys_admin,
    network inet,
    /var/log/my_confined_app/user1.log rw,
}

profile user2 {
    #@select: adm
    network inet,
    /var/log/my_confined_app/user2.log rw,
}
```

UserArmor simplifies the creation of the above files by automatically adding the `include if exists <.usr.bin.my_confined_app/mappings>` *directive in the profile of the confined application and generating the content of the file* `/etc/apparmor.d/.usr.bin.my_confined_app/mappings`.

UserArmor comprises the following three command-line utilities: `ua-generate`, `ua-enforce`, and `ua-exec`. These tools simplify the process of defining and enforcing user-specific security policies, ensuring that applications run with the appropriate restrictions based on the user executing them. The `ua-generate` command is the starting point for setting up UserArmor. Given the absolute path of an executable and a comma-separated list of users, it automatically creates the necessary directory structure and generates a subprofile file for each specified user. If a subprofile already exists, it remains unchanged, preventing unintended overwrites. Since this command modifies AppArmor profiles, it must be executed with superuser privileges. Once the user-specific profiles are in place, `ua-enforce` takes over to integrate them into the main AppArmor profile of the executable. Given the absolute path of the executable, it ensures that the AppArmor profile contains the necessary `include if exists` directive, allowing the system to conditionally load the user-specific subprofiles. Such subprofiles are collected in the `mappings` file, which is also generated by `ua-enforce` by using the tag system defined in Section 3. Like `ua-generate`, this command requires root privileges, as it modifies the security policies of AppArmor. The final piece of the puzzle is `ua-exec`, which ensures that when an executable is run, it is confined under the correct UserArmor subprofile. Unlike the previous two utilities, 0|ua-exec| can be run by any user. It works by identifying the current user, selecting the corresponding subprofile, and executing the application via `aa-exec` to enforce the correct restrictions. For security reasons, `aa-exec` is expected to be executable only by the root and members of the `userarmor` group, preventing unauthorized users from bypassing confinement. This way, users in the `userarmor` group are correctly recognized and associated with their subprofile, while users that are not members of the `userarmor` group will be forced to use the base profile (i.e., the standard model adopted by AppArmor).

Example 4 (Continuing Example 3)**.** *The starting point is the file defining the base profile. Given the file* `/etc/apparmor.d/usr.bin.my_confined_app` *from Example 3, with or without the* `include if exists`*, UserArmor can generate the directory structure with the following command:*

```
$ sudo ua-generate /usr/bin/my_confined_application --subprofiles=user1,user2
```

The user profiles are initially empty (unless the files already exist), and the administrator can populate them with the content reported in Example 3. After that, the administrator can issue the `ua-enforce` *command to collect the subprofiles in the* `mappings` *file and to add the* `include if exists` *directive to the* `/etc/apparmor.d/usr.bin.my_confined_app` *file, if not already present, as follows:*

```
$ sudo ua-enforce /usr/bin/my_confined_application
```

With the above command, UserArmor also takes care of interacting with AppArmor in order to enforce the provided security policy. In order to execute `my_confined_app`, `user1` *issues the following command:*

```
$ ua-exec /usr/bin/my_confined_app arg1 arg2 ...
```

Note that bypassing `ua-exec` *with the command*

```
$ my_confined_app arg1 arg2 ...
```

would run the application with the base profile, hence, with no access to the `/var/log/my_confined_app/user1.log` *file.*

4. Literature Review

The closest work to our proposal is Paranoid Penguin [5], which introduces an extension of AppArmor primarily aimed at improving usability. The main contribution of Paranoid Penguin is the integration of a graphical user interface (GUI) to facilitate the creation, modification and management of AppArmor profiles. This approach makes AppArmor more accessible to users who may not be familiar with its command-line tools, lowering the barrier to entry for system administrators and security practitioners. The GUI provides an intuitive way to define and enforce security policies without requiring deep knowledge of the syntax used by AppArmor, reducing the risk of misconfigurations that could compromise security. While Paranoid Penguin enhances the usability of AppArmor, it does not extend or modify the underlying security enforcement model. That is, the access control mechanism itself remains unchanged, and security policies are still applied in the same way as in standard AppArmor. Stated differently, the framework does not introduce new functionalities for improving security beyond profile simplification.

In contrast, UserArmor goes beyond usability improvements by introducing two key enhancements to the core security model of AppArmor. First, UserArmor allows binding security profiles to specific users, ensuring fine-grained access control in multi-user environments. Traditional AppArmor applies profiles at the application level, meaning that all users executing the same process are subject to the same restrictions. This limitation can lead to security gaps, as different users may require different levels of access to resources. UserArmor addresses this by associating distinct subprofiles with individual users, ensuring that permissions are customized per user rather than being uniformly applied to all instances of an application. Second, UserArmor introduces an inheritance system for policy rules, which improves policy management and scalability. In standard AppArmor, profiles must explicitly define all required permissions, leading to redundancy when multiple profiles share common rules. While AppArmor provides a mechanism called abstractions to group reusable rules, these operate by an all-or-nothing inclusion, meaning administrators cannot selectively inherit only specific rules from an abstraction. UserArmor solves this problem by introducing a tag-based inheritance mechanism, where profiles can inherit only the necessary rules, reducing duplication and simplifying policy maintenance. By addressing these limitations, UserArmor extends the capabilities of AppArmor beyond usability enhancements and introduces structural improvements that make it more suitable for multi-user environments and dynamic security policies. Stated

differently, while Paranoid Penguin provides a more user-friendly interface for AppArmor, UserArmor fundamentally improves its security model by enabling user-specific policy enforcement and hierarchical policy inheritance, making it a more flexible and scalable solution for access control in modern Linux systems.

A broader area of research focuses on dynamic profiling to enhance security at runtime. These approaches monitor application behavior and adaptively adjust permissions, allowing for more flexible and responsive access control. Unlike traditional security policies, which are static and predefined, dynamic profiling mechanisms observe how applications interact with system resources and adjust security policies accordingly to mitigate risks. One branch of research explores cloud-based profile generation, as seen in [13,14], where profile management is offloaded to Cloud Services. This approach offers greater flexibility and scalability, as security policies can be updated dynamically based on centralized monitoring and threat detection. On the other hand, some solutions are designed to run directly on the operating system, ensuring real-time policy enforcement at the local level [15]. These techniques provide a more fine-grained and immediate reaction to potential security threats, without requiring cloud infrastructure.

Among the various dynamic profiling approaches, containerized environments, particularly Docker-based infrastructures, have received significant attention due to their unique security challenges. Since containers package applications with their dependencies, they are vulnerable to container breakout attacks and misconfigurations that may lead to privilege escalation. Research works such as [16,17] propose automated profile generation techniques for containers, leveraging Auditd and SystemTap to analyze the required permissions dynamically. These solutions have been further integrated into a unified framework [13], which combines the strengths of both methods, ensuring a more robust security model. Another significant advancement is given by [18], which introduces dynamic runtime updates for container security policies. This mechanism allows security rules to be adjusted on the fly as the application state evolves, reducing the need for manual intervention. With the widespread adoption of container orchestrators like Kubernetes, Ref. [19] extends these profiling techniques to automatically generate AppArmor profiles for distributed containers in Kubernetes clusters. This automation streamlines security management and reduces misconfiguration risks in large-scale cloud environments. These studies share the overarching goal of optimizing security enforcement—either by introducing new functionalities or enhancing existing mechanisms. UserArmor aligns with these objectives, but instead of focusing on runtime profiling, it introduces structural improvements to AppArmor by enabling user-specific profile enforcement and hierarchical policy inheritance. While dynamic profiling aims to enhance security by adapting to application behavior, UserArmor provides a systematic and scalable approach to user-aware security policies, ensuring multi-user protection and reducing policy redundancy. Furthermore, the ability of UserArmor to limit process resource usage (e.g., CPU, memory, open files) makes it a complementary solution to dynamic profiling methods, contributing to a more robust and flexible security ecosystem.

Finally, we acknowledge that Linux already provides Role-Based Access Control (RBAC) via SELinux. However, SELinux is significantly more complex than AppArmor and is typically used with predefined policies rather than custom configurations per user [6,20,21]. Organizations that deploy SELinux often rely on standard policies without distinguishing between different users running the same application, because customizing SELinux requires deep knowledge of the security contexts, domains, and type enforcement. UserArmor, on the other hand, complements the simpler AppArmor security mechanisms by introducing a finer-grained, user-specific confinement model, ensuring that each user receives a tailored security profile without modifying global policies.

5. Experiment

In order to assess UserArmor empirically, we designed a benchmark comprising the security policies in the examples given in the Introduction. Our benchmark comprises a very simple Bash script, as follows:

```
cat /etc/my_confined_app.conf  >> /var/log/my_confined_app/$USER.log
```

The above script needs to read access to the file `/etc/my_confined_app.conf`, and write access to the file `/var/log/my_confined_app/$USER.log`. Note that the name of the log file is determined using the environment variable `USER`, the value of which is usually the username of the logged-in user. Hence, in the expected usage, if `user1` executes the above Bash script, the log file is `/var/log/my_confined_app/user1.log`. On the other hand, this is a weak assumption, and `user1` can easily reassign the environment variable to corrupt the file associated with `user2`, i.e., `/var/log/my_confined_app/user2.log`. We measure the execution time over repeated executions of the script. The experiment is run on a 13th Gen Intel(R) Core(TM) i7-1360P @ 2.2 GHz CPU with 32 GB RAM, and its aim is to determine the overhead introduced by UserArmor.

The overall results are shown in Figure 3. We observe that AppArmor introduces no overhead, while Bubblewrap is more expensive. The performance of UserArmor is between the following two alternatives, as expected: it works on top of AppArmor, so it cannot be faster than AppArmor, but anyhow less expensive than Bubblewrap. We observe that the time measured for UserArmor are closer to those obtained by AppArmor than to the time needed by Bubblewrap, which is a positive result.

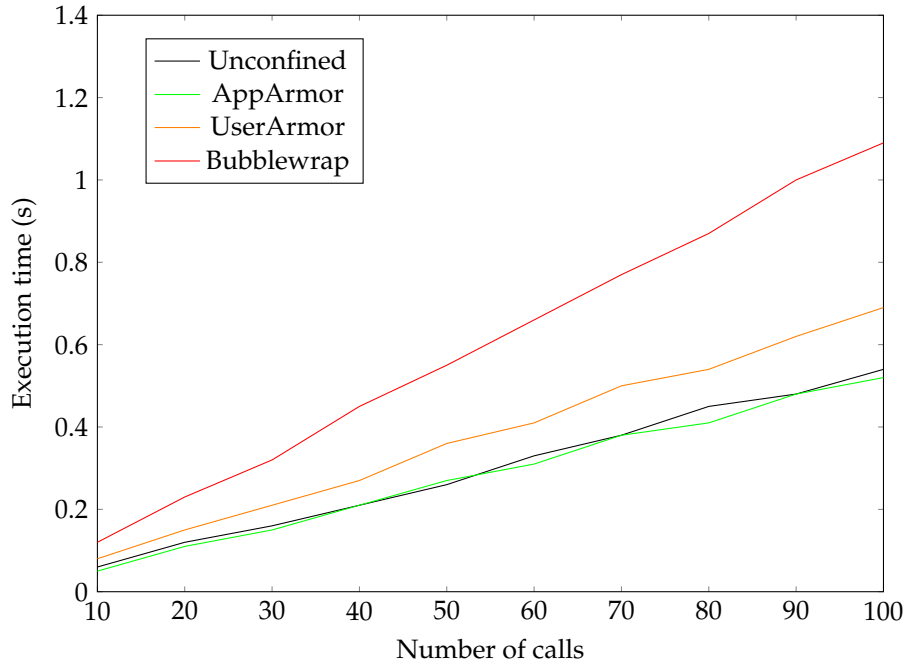

Figure 3. Execution time for running the Bash script in our benchmark under different security policies.

6. Application: Hardening Answer Set Programming Solvers

Answer Set Programming (ASP) is a declarative programming paradigm designed for solving complex combinatorial problems [22–24]. It is widely used in artificial intelligence, knowledge representation and automated reasoning [25–28]. Without going into much details, ASP programs comprise logic rules and are associated with zero or more stable models, that is, first-order models satisfying an additional stability condition designed to interpret default negation.

Example 5. *Let us consider the Hamiltonian Path problem:*

> *Given a directed graph G, a starting node s and a target node t, we need to find a path from s to t that visits all nodes of G exactly once.*

The graph shown in Figure 4 can be represented by the following ASP:

```
start("A"). target("G").
node("A").  link("A","B").  link("A","C").
node("B").  link("B","A").  link("B","C").  link("B","D").
node("C").  link("C","A").  link("C","D").
node("D").  link("D","B").  link("D","C").  link("D","E").
node("E").  link("E","D").  link("E","F").  link("E","H").
node("F").  link("F","E").  link("F","G").
node("H").  link("H","F").
node("G").  link("G","F").  link("G","H").
```

We couple the above facts with rules addressing the Hamiltonian Path problem as follows:

```
reach(X) :- start(X).
reach(Y) :- next(X,Y).

{next(X,Y) : link(X,Y)} = 1 :- reach(X), not target(X).
:- next(X,Y), next(Z,Y), X < Z.
:- node(X), not reach(X).
```

Notably, the first two rules above define the reached nodes (`reach/1`) from the starting node (`start/1`), following the selected links (`next/2`). The links are selected by the third rule, which is also called the choice rule; for every reached node not being the target node, exactly one of its outgoing link is selected. The last two rules above, which are also called constraints, ensure that no node has two selected incoming links and that every node is reached. Stable models of the above program represent the Hamiltonian Paths of the given graph; in this case, the unique stable model extends the input facts with the following facts:

```
reach("A").  reach("B").  reach("C").  reach("D").
reach("E").  reach("F").  reach("H").  reach("G").
next("A","B").  next("B","C").  next("C","D").  next("D","E").
next("E","H").  next("F","G").  next("H","F").
```

The example can be run in the browser using the following ASP Chef [29,30] recipe: https://asp-chef.alviano.net/s/hamiltonian-path@algorithm2025 (accessed on 26 February 2025). A graphical representation of the computed Hamiltonian Path is given in Figure 4.

The state-of-the-art solver CLINGO [12] is one of the most popular ASP solvers, offering a powerful inference engine that can be extended with Python v3.12 and Lua v5.4 scripts to enhance reasoning capabilities. Other frequently used ASP solvers are DLV [31], SMODELS [32], CMODELS [33], IDP [34] and WASP [35]. ASP can be used to develop web-based AI applications, and this is a common scenario for researchers aiming at showcasing their AI-powered tools that solve hard combinatorial tasks using ASP [36–38]. They provide a web app with a user-friendly interface (e.g., a textarea for input), where an ASP solver is executed in the backend server to process queries and return computed results.

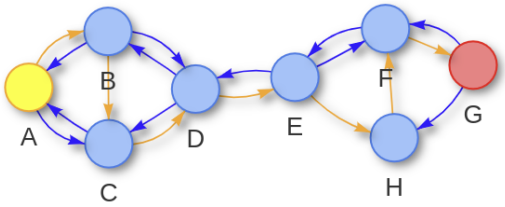

Figure 4. Graph used in Example 5. A Hamiltonian Path from A to G is shown using yellow arrows. Links not being part of the computed Hamiltonian Path are shown in blue.

While this approach is functional, it introduces serious security vulnerabilities. CLINGO supports @-terms, which allow interpreted functions to be evaluated dynamically. These functions can be defined in either Lua or Python, meaning that CLINGO can execute arbitrary Python or Lua code. This feature is useful for extending the solver, but it also exposes the system to remote code execution (RCE) vulnerabilities. Malicious users could input a crafted ASP program containing an @-term that executes system commands, leading to arbitrary code execution on the backend server. For example, a user could submit an ASP program like the following:

```
#script(python)
import~subprocess

def rce(cmd):
    return subprocess.check_output(["sh", "-c", cmd.string], text=True)

#end.

out(@rce("whoami")).
```

Or, in Lua, as follows:

```
#script(lua)

function rce(cmd)
    local f = assert(io.popen(cmd.string, 'r'))
    local output = f:read('*a')
    f:close()
    return output
end

#end.

out(@rce("whoami")).
```

The above programs can be easily adapted to delete critical files, exfiltrate data, or install malware on the backend server, completely compromising its security.

A robust solution is to contain the execution of CLINGO by restricting the permissions of the process running the web app. This is where UserArmor comes into play. By defining a UserArmor security profile for `www-data` (or the user running the web service), we can limit CLINGO permissions to the absolute minimum required. More specifically, we can enforce the following restrictions:

- No filesystem access: Prevents CLINGO from reading or modifying system files.
- No network access: Blocks CLINGO from making outbound requests, preventing data exfiltration or malware downloads.
- Minimal execution environment: Restricts process execution, preventing the use of external system tools.

A UserArmor profile for `www-data` could take the following form:

```
profile www-data {
    # Deny access to the entire filesystem except for necessary paths
    deny / rwx,
    /usr/lib/** rm,

    # Allow read access only to the directory where ASP encodings are stored
    /var/www/clingo_input/** r,

    # No network access
    deny network inet,
    deny network inet6,

    # Deny execution of system commands
    deny capability sys_admin,
    deny capability setuid,
    deny capability setgid,
}
```

With this policy, even if an attacker injects Python or Lua code, CLINGO cannot access critical files, execute system commands, or connect to the internet, effectively neutralizing the RCE attack.

Without UserArmor, the only alternative is to apply AppArmor restrictions to CLINGO globally. This means that every instance of CLINGO on the backend server would be restricted, including those run by legitimate users (e.g., via SSH). This is problematic because some users may need unrestricted access to CLINGO. For example, researchers or developers using CLINGO from the terminal might rely on @-terms for legitimate purposes, such as reading external files and making network requests to fetch data. By applying a global AppArmor profile to CLINGO, all users on the system would be restricted in the same way, even those who should have full access.

A possible alternative to UserArmor is to execute CLINGO inside a sandbox, such as Bubblewrap (`bwrap`). Bubblewrap is a lightweight user-space sandboxing tool that can create isolated environments for applications. For example, CLINGO could be executed inside a Bubblewrap sandbox as follows:

```
$ bwrap --unshare-net --ro-bind /usr/bin/clingo /usr/bin/clingo \
    --ro-bind /usr/lib /usr/lib --ro-bind /lib /lib --ro-bind /lib64 /lib64 \
    --ro-bind /var/www/clingo_input /input  /usr/bin/clingo /input/encoding.lp
```

This setup prevents network access (`--unshare-net`) and restricts filesystem access (`--ro-bind` for read-only mounts). However, sandboxing introduces more overhead than AppArmor/UserArmor (every CLINGO execution requires setting up a new isolated environment) and is more difficult to manage (running a sandbox requires an invasive change in how CLINGO is invoked, which may not be practical in existing systems).

We provide scripts showcasing the above scenarios as follows:

- Unconfined execution of CLINGO: This script configures a web app vulnerable to RCE and subject to system takeover.
- Confined execution of CLINGO via AppArmor: This script configures a web app that is not vulnerable to RCE, but SSH users are subject to the same restriction of the web app.
- Confined execution of CLINGO via UserArmor: This script configures a web app that is not vulnerable to RCE, and SSH users are free to use CLINGO as they would normally do.

- Sand-boxed execution of CLINGO via Bubblewrap: This script configure a web app that is not vulnerable to RCE but requires a new isolated for each execution of CLINGO.

The script files are available online (https://github.com/pierpaolosestito-dev/ASPArmor; accessed on 13 February 2025). We ran a second experiment using the aforementioned scripts on a 13th Gen Intel(R) Core(TM) i7-1360P @ 2.2 GHz CPU with 32 GB RAM. In this experiment, we repeatedly call CLINGO to produce 10 thousand integers with the program

```
number(1..10,000).
```

We opted for such a simple, deterministic program to obtain a stable performance of CLINGO. As shown in Figure 5, the results confirm the observations from the previous section as follows: UserArmor introduces a negligible overhead, and its performance is superior to Bubblewrap.

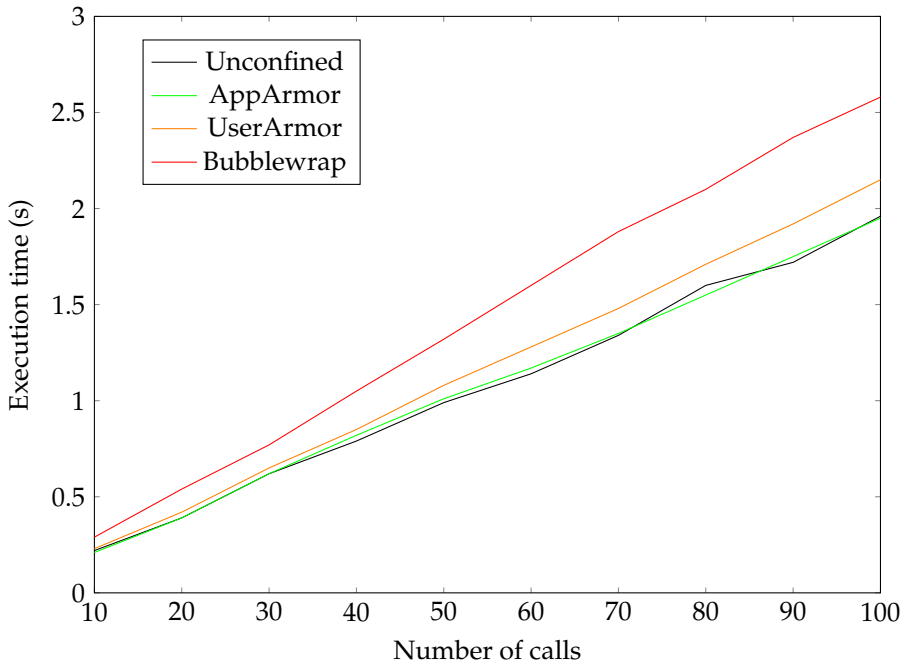

Figure 5. Execution time of repeated calls to CLINGO under different security policy mechanisms.

To sum up, hardening CLINGO in a web-based environment is crucial due to its support for @-terms, which can lead to RCE vulnerabilities. While sandboxing tools like Bubblewrap can provide isolation, they introduce overhead and require modifications to the execution workflow. UserArmor, on the other hand, offers a more lightweight and kernel-enforced approach, allowing administrators to restrict CLINGO permissions only for the web service user (`www-data`), without affecting other users who might need full access. For completeness, we also mention the possibility to run CLINGO (and similar engines like MiniZinc [39] and Nemo [40,41]) directly in the browser, by relying on their WebAssembly versions; this is the case, for example, of ASP Chef [29,30], which provides a low-code programming environment powered by ASP and integrating several languages and frameworks [42,43].

7. Conclusions

In this paper, we introduced UserArmor, an extension of AppArmor that enhances security policy management by incorporating user-level granularity and inheritance mechanisms. Our approach addresses the following two major limitations of AppArmor: the inability to differentiate permissions for different users running the same process and the lack of an efficient inheritance system for security policies. By structuring security profiles hierarchically and introducing selectable rules via tagging, UserArmor allows administra-

tors to enforce user-specific security constraints while minimizing policy duplication. This improves security, maintainability, and scalability, especially in multi-user environments where different users require distinct levels of access. Our implementation includes a set of CLI tools to facilitate profile management, ensuring that UserArmor is practical and easy to use. Performance evaluation shows that the additional overhead introduced is negligible. Future work on UserArmor will consider the definition of additional tools to support common operations with user accounts, like user renaming (profile files must be renamed accordingly) and deletion (the associated profile files can be deleted as well).

Author Contributions: Conceptualization, P.S.; methodology, M.A.; software, P.S.; validation, P.S.; formal analysis, M.A.; investigation, P.S.; resources, P.S.; data curation, P.S.; writing—original draft preparation, P.S.; writing—review and editing, M.A.; visualization, P.S.; supervision, M.A.; project administration, M.A.; funding acquisition, M.A. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Italian Ministry of University and Research (MUR) under PRIN project PRODE "Probabilistic declarative process mining", CUP H53D23003420006, under PNRR project FAIR "Future AI Research", CUP H23C22000860006, under PNRR project Tech4You "Technologies for climate change adaptation and quality of life improvement", CUP H23C22000370006, and under PNRR project SERICS "SEcurity and RIghts in the CyberSpace", CUP H73C22000880001; by the Italian Ministry of Health (MSAL) under POS projects CAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by the Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUP B29J23000430005); under PN RIC project ASVIN "Assistente Virtuale Intelligente di Negozio" (CUP B29J24000200005); and by the LAIA lab (part of the SILA labs). Mario Alviano is member of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

Data Availability Statement: Source code and data used in this article are available online at https://github.com/pierpaolosestito-dev/ASPArmor.

Conflicts of Interest: The author declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Cowan, C. Securing linux systems with apparmor. *DEF CON* **2007**, *15*, 15–26.
2. Jiang, Y.; Lin, C.; Yin, H.; Tan, Z. Security analysis of mandatory access control model. In Proceedings of the 2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No. 04CH37583), Hague, The Netherlands, 10–13 October 2004; Volume 6, pp. 5013–5018.
3. Osborn, S. Mandatory access control and role-based access control revisited. In Proceedings of the Second ACM Workshop on Role-Based Access Control, Fairfax, VA, USA, 6–7 November 1997; pp. 31–40.
4. Ecarot, T.; Dussault, S.; Souid, A.; Lavoie, L.; Ethier, J. AppArmor For Health Data Access Control: Assessing Risks and Benefits. In Proceedings of the 7th International Conference on Internet of Things: Systems, Management and Security, IOTSMS 2020, Virtual Event, France, 14–16 December 2020; Boubchir, L., Benkhelifa, E., Jararweh, Y., Saleh, I., Eds.; IEEE: Piscataway, NJ, USA, 2020; pp. 1–7. [CrossRef]
5. Bauer, M. Paranoid penguin: An introduction to Novell AppArmor. *Linux J.* **2006**, *2006*, 13.
6. Chen, H.; Li, N.; Mao, Z. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, CA, USA, 8–11 February 2009.
7. Alviano, M.; Dodaro, C.; Fiorentino, S.; Previti, A.; Ricca, F. Enumeration of Minimal Models and MUSes in WASP. In Proceedings of the Logic Programming and Nonmonotonic Reasoning— 16th International Conference, LPNMR 2022, Genova, Italy, 5–9 September 2022; Gottlob, G., Inclezan, D., Maratea, M., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2022; Volume 13416, pp. 29–42. [CrossRef]

8. Alviano, M.; Faber, W.; Leone, N.; Perri, S.; Pfeifer, G.; Terracina, G. The Disjunctive Datalog System DLV. In Proceedings of the Datalog Reloaded—First International Workshop, Datalog 2010, Oxford, UK, 16–19 March 2010; Revised Selected Papers; de Moor, O., Gottlob, G., Furche, T., Sellers, A.J., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2010; Volume 6702, pp. 282–301. [CrossRef]
9. Cat, B.D.; Bogaerts, B.; Bruynooghe, M.; Janssens, G.; Denecker, M. Predicate logic as a modeling language: The IDP system. In *Declarative Logic Programming: Theory, Systems, and Applications*; Kifer, M., Liu, Y.A., Eds.; ACM Books; ACM/Morgan & Claypool: San Rafael, CA, USA, 2018; Volume 20, pp. 279–323. [CrossRef]
10. Lierler, Y.; Maratea, M. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In Proceedings of the Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, 6–8 January 2004; Lifschitz, V., Niemelä, I., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2004; Volume 2923, pp. 346–350. [CrossRef]
11. Janhunen, T.; Niemelä, I. GNT—A Solver for Disjunctive Logic Programs. In Proceedings of the Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, 6–8 January 2004; Lifschitz, V., Niemelä, I., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2004; Volume 2923, pp. 331–335. [CrossRef]
12. Gebser, M.; Kaminski, R.; Schaub, T. Complex optimization in answer set programming. *Theory Pract. Log. Program.* **2011**, *11*, 821–839. [CrossRef]
13. Zhu, H.; Gehrmann, C. AppArmor Profile Generator as a Cloud Service. In Proceedings of the 11th International Conference on Cloud Computing and Services Science, CLOSER 2021, Online Streaming, 28–30 April 2021; Helfert, M., Ferguson, D., Pahl, C., Eds.; SCITEPRESS: Setúbal, Portugal, 2021; pp. 45–55. [CrossRef]
14. Zhu, H.; Gehrmann, C.; Roth, P. Access security policy generation for containers as a cloud service. *SN Comput. Sci.* **2023**, *4*, 748.
15. Li, Y.; Huang, C.; Yuan, L.; Ding, Y.; Cheng, H. ASPGen: An Automatic Security Policy Generating Framework for AppArmor. In Proceedings of the IEEE International Conference on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking, ISPA/BDCloud/SocialCom/SustainCom 2020, Exeter, UK, 17–19 December 2020; Hu, J., Min, G., Georgalas, N., Zhao, Z., Hao, F., Miao, W., Eds.; IEEE: Piscataway, NJ, USA, 2020; pp. 392–400. [CrossRef]
16. Mattetti, M.; Shulman-Peleg, A.; Allouche, Y.; Corradi, A.; Dolev, S.; Foschini, L. Securing the infrastructure and the workloads of linux containers. In Proceedings of the 2015 IEEE Conference on Communications and Network Security (CNS), Florence, Italy, 28–30 September 2015; pp. 559–567. [CrossRef]
17. Loukidis-Andreou, F.; Giannakopoulos, I.; Doka, K.; Koziris, N. Docker-sec: A fully automated container security enhancement mechanism. In Proceedings of the 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, 2–6 July 2018; pp. 1561–1564.
18. Huang, C.; Wang, K.; Li, Y.; Li, J.; Liao, Q. ASPGen-D: Automatically Generating Fine-grained Apparmor Policies for Docker. In Proceedings of the IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking, ISPA/BDCloud/SocialCom/SustainCom 2022, Melbourne, Australia, 17–19 December 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 822–829. [CrossRef]
19. Zhu, H.; Gehrmann, C. Kub-Sec, an automatic Kubernetes cluster AppArmor profile generation engine. In Proceedings of the 14th International Conference on COMmunication Systems & NETworkS, COMSNETS 2022, Bangalore, India, 4–8 January 2022; pp. 129–137. [CrossRef]
20. Schreuders, Z.C.; McGill, T.; Payne, C. Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor, and FBAC-LSM. *ACM Trans. Inf. Syst. Secur.* **2011**, *14*, 1–28. [CrossRef]
21. Shepherd, C.; Markantonakis, K. Operating System Controls. In *Trusted Execution Environments*; Springer: Berlin/Heidelberg, Germany, 2024; pp. 33–53.
22. Marek, V.; Truszczyński, M. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*; Springer: Berlin/Heidelberg, Germany, 1999; pp. 375–398. [CrossRef]
23. Niemelä, I. Logic programming with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.* **1999**, *25*, 241–273. [CrossRef]
24. Gelfond, M.; Lifschitz, V. Logic programs with classical negation. In *Logic Programming*; Warren, D.; Szeredi, P., Eds.; ACM: New York, NY, USA, 1990; pp. 579–597.
25. Cappanera, P.; Gavanelli, M.; Nonato, M.; Roma, M. Logic-Based Benders Decomposition in Answer Set Programming for Chronic Outpatients Scheduling. *Theory Pract. Log. Program.* **2023**, *23*, 848–864. [CrossRef]
26. Cardellini, M.; Nardi, P.D.; Dodaro, C.; Galatà, G.; Giardini, A.; Maratea, M.; Porro, I. Solving Rehabilitation Scheduling Problems via a Two-Phase ASP Approach. *Theory Pract. Log. Program.* **2024**, *24*, 344–367. [CrossRef]

27. Wotawa, F. On the Use of Answer Set Programming for Model-Based Diagnosis. In Proceedings of the Trends in Artificial Intelligence Theory and Applications. Artificial Intelligence Practices—33rd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2020, Kitakyushu, Japan, 22–25 September 2020; Fujita, H., Fournier-Viger, P., Ali, M., Sasaki, J., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2020; Volume 12144, pp. 518–529. [CrossRef]
28. Taupe, R.; Friedrich, G.; Schekotihin, K.; Weinzierl, A. Solving Configuration Problems with ASP and Declarative Domain Specific Heuristics. In Proceedings of the 23rd International Configuration Workshop (CWS/ConfWS 2021), Vienna, Austria, 16–17 September 2021; Aldanondo, M., Falkner, A.A., Felfernig, A., Stettinger, M., Eds.; CEUR Workshop Proceedings; CEUR-WS.org: Aachen, Germany, 2021; Volume 2945, pp. 13–20.
29. Alviano, M.; Cirimele, D.; Reiners, L.A.R. Introducing ASP recipes and ASP Chef. In Proceedings of the International Conference on Logic Programming 2023 Workshops Co-Located with the 39th International Conference on Logic Programming (ICLP 2023), London, UK, 9–10 July 2023; Arias, J., Batsakis, S., Faber, W., Gupta, G., Pacenza, F., Papadakis, E., Robaldo, L., Rückschloß, K., Salazar, E., Saribatur, Z.G., et al., Eds.; CEUR Workshop Proceedings; CEUR-WS.org: Aachen, Germany, 2023; Volume 3437.
30. Alviano, M.; Reiners, L.A.R. ASP Chef: Draw and Expand. In Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning, KR 2024, Hanoi, Vietnam, 2–8 November 2024; Marquis, P., Ortiz, M., Pagnucco, M., Eds.; IJCAI Organization: Darmstadt Germany, 2024. [CrossRef]
31. Calimeri, F.; Leone, N.; Melissari, G.; Pacenza, F.; Perri, S.; Reale, K.; Ricca, F.; Zangari, J. ASP-Based Declarative Reasoning in Data-Intensive Enterprise and IoT Applications. *Algorithms* **2023**, *16*, 159. [CrossRef]
32. Syrjänen, T.; Niemelä, I. The Smodels System. In Proceedings of the Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, 17–19 September 2001; Eiter, T., Faber, W., Truszczynski, M., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2001; Volume 2173, pp. 434–438. [CrossRef]
33. Giunchiglia, E.; Maratea, M. On the Relation Between Answer Set and SAT Procedures (or, Between cmodels and smodels). In Proceedings of the Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, 2–5 October 2005; Gabbrielli, M., Gupta, G., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3668, pp. 37–51. [CrossRef]
34. Bogaerts, B.; Jansen, J.; Cat, B.D.; Janssens, G.; Bruynooghe, M.; Denecker, M. Bootstrapping Inference in the IDP Knowledge Base System. *New Gener. Comput.* **2016**, *34*, 193–220. [CrossRef]
35. Alviano, M.; Amendola, G.; Dodaro, C.; Leone, N.; Maratea, M.; Ricca, F. Evaluation of Disjunctive Programs in WASP. In Proceedings of the Logic Programming and Nonmonotonic Reasoning—15th International Conference, LPNMR 2019, Philadelphia, PA, USA, 3–7 June 2019; Balduccini, M., Lierler, Y., Woltran, S., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2019; Volume 11481, pp. 241–255. [CrossRef]
36. Eiter, T.; Ianni, G.; Schindlauer, R.; Tompits, H. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. In Proceedings of the IJCAI-05, the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, 30 July–5 August 2005; Kaelbling, L.P., Saffiotti, A., Eds.; Professional Book Center: Bethesda, MD, USA 2005; pp. 90–96.
37. Calimeri, F.; Germano, S.; Palermiti, E.; Reale, K.; Ricca, F. Developing ASP Programs with ASPIDE and LoIDE. *Künstliche Intell.* **2018**, *32*, 185–186. [CrossRef]
38. Calimeri, F.; Fuscà, D.; Germano, S.; Perri, S.; Zangari, J. Fostering the Use of Declarative Formalisms for Real-World Applications: The EmbASP Framework. *New Gener. Comput.* **2019**, *37*, 29–65. [CrossRef]
39. Nethercote, N.; Stuckey, P.J.; Becket, R.; Brand, S.; Duck, G.J.; Tack, G. MiniZinc: Towards a Standard CP Modelling Language. In Proceedings of the CP 2007, Providence, RI, USA, 23–27 September 2007; Bessiere, C., Ed.; LNCS; Springer: Berlin/Heidelberg, Germany, 2007; Volume 4741, pp. 529–543. [CrossRef]
40. Ivliev, A.; Gerlach, L.; Meusel, S.; Steinberg, J.; Krötzsch, M. Nemo: A Scalable and Versatile Datalog Engine. In Proceedings of the 5th International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog-2.0 2024) Co-Located with the 17th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2024), Dallas, TX, USA, 11 October 2024; Alviano, M., Lanzinger, M., Eds.; CEUR; Workshop Proceedings; CEUR-WS.org: Aachen, Germany, 2024; Volume 3801, pp. 43–47.
41. Ivliev, A.; Gerlach, L.; Meusel, S.; Steinberg, J.; Krötzsch, M. Nemo: Your Friendly and Versatile Rule Reasoning Toolkit. In Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning, KR 2024, Hanoi, Vietnam, 2–8 November 2024; Marquis, P., Ortiz, M., Pagnucco, M., Eds.; IJCAI Organization: Darmstadt Germany, 2024. [CrossRef]

42. Alviano, M.; Reiners, L.A.R. Integrating MiniZinc with ASP Chef: Browser-Based Constraint Programming for Education and Prototyping. In Proceedings of the Logic Programming and Nonmonotonic Reasoning—17th International Conference, LPNMR 2024, Dallas, TX, USA, 11–14 October 2024; Dodaro, C., Gupta, G., Martinez, M.V., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2024; Volume 15245, pp. 174–186. [CrossRef]
43. Alviano, M.; Guarasci, P.; Reiners, L.A.R.; Vasile, I.R. Integrating Structured Declarative Language (SDL) into ASP Chef. In Proceedings of the Logic Programming and Nonmonotonic Reasoning—17th International Conference, LPNMR 2024, Dallas, TX, USA, 11–14 October 2024; Dodaro, C., Gupta, G., Martinez, M.V., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2024; Volume 15245, pp. 387–392. [CrossRef]