CHAPTER 1 *Introduction*

This dissertation describes the architecture of a new class of operating systems: systems that are structured from the ground up to use a type-safe instruction set as their sole protection mechanism. In a type-safe instruction set instructions (operations) are applied to typed operands that refer to typed data entities. Strict rules describe what types of operands can refer to what types of data entities and what operations can be applied to them.

Using a type-safe instruction set instead of the traditional address-based protection has a number of advantages. The *robustness* of the system is improved because many kinds of programming errors can be detected at an early stage in the development by using the type system. An improved robustness has a positive effect on the *security* and *reliability* of the system, although I will show in this thesis that additional mechanisms are needed to maintain a secure state. The type-safe instruction set is a very fine-grained protection mechanism that allows to create *fine-grained protection* domains and enables an incremental *extensibility* of the system. The system design contains a communication mechanism that can be used for highly-efficient but unprotected communication inside a protection domain and for slower communication across protection boundaries. Because the programmer can use the same abstraction for intra-domain and inter-domain communication program module boundaries become independent from protection domain boundaries. The system can be *configured according to its intended use* on a scale that is bounded by the two extremes of placing each module in its own protection domain or placing all modules in the same domain. Systems that are dedicated to a very specific task, such as file-server appliances, benefit from the performance advantages of running all code in the same protection domain. Systems that run untrusted and potentially malicious code, such as an agent execution platform, benefit from the ability to spawn fine-grained protection domains. These systems must be able to completely isolate the untrusted code, which means *restricting access to information* that is available on the host as well as *restricting access to resources*. All these features must be realized without incurring excessive performance overhead compared to traditional operating system architectures.

1 Motivation

All hardware components of a computer have become more powerful in recent years: the access time of disks decreased while the throughput and capacity increased, CPU speed increased, memory access time decreased and memory capacity increased, accompanied by a steady reduction in hardware prices. This development enabled the ubiquitous use of computers for dedicated purposes, such as mobile devices, and lead to improved capabilities of general-purpose systems. While exploiting these capabilities software became more complex. To manage this complexity new software engineering techniques and methodologies have been developed. Examples for these techniques and

methodologies are object-oriented design and implementation, component-based systems, and design patterns. But these developments had little impact on operating systems. Although there are object-oriented operating systems [38][80] and component-based systems [71], system software is still complex, difficult to comprehend, and expensive to change and extend. Different technologies for application development and kernel development make the interaction with the kernel cumbersome and inefficient. Reuse of designs and code between kernel and application is difficult if not impossible and usually requires two different interfaces [56]. This leads to several complications when the boundary between the application and the operating system becomes smaller or disappears. Examples for such systems are application-specific operating systems for databases, such as Oracle's "Raw Iron" database system, or operating systems for special- purpose hardware, such as routers or mobile consumer devices.

1.1 Robustness, security, and reliability

Complexity is the worst enemy of robustness, security, and reliability. Although current systems have reached a considerable level of complexity we assume that systems will even become more complex in the future. The most effective medicine for complexity is the principle of divide-and-conquer. A large system, such as an operating system, must be split into smaller, comprehensible parts.

1.2 Configurability, reusability, and dedicated systems

A general-purpose operating system is optimized for a specific workload. Using it for a different purpose leads to a phenomenon called *mapping dilemma* [104]. A mapping dilemma arises, when abstractions (modules, systems, services) are used in a way not anticipated by the designer of the abstraction. The repeatedly cited example is a windowing system where a heavy-weight window is used to implement one cell of a spread sheet program. While this leads to a short and concise spread sheet program, the performance and resource requirements render the program unusable. There are many examples for mapping dilemmas in current operating systems:

Database. High performance databases do not use the file system abstraction of current server operating systems, because the on-disk file layout does not match the file-usage pattern of database systems. Stonebraker et al. [169] describe the performance problems when running the INGRES database on top of a UNIX file system. They solve the problems by using a database-specific buffer cache and disk layout.

Web proxy. Gabber and Shriver [70] describe a similar situation for caching web proxies. Using a specialized file system library they were able to achieve a 6-10 times performance improvement compared to a web proxy that uses a general-purpose file system.

NFS. It is difficult to use a standard UNIX system as a high performance NFS server because of the general-purpose nature of the network stack and the file system. To achieve highest performance, the network code, the virtual memory code, and the file system code must be integrated. Examples are Auspex's functional multiprocessing system NS5000 [92], [94], Network Appliance NetApp filers [93] [184], and IBM's HA-NFS [21].

Multimedia. Multimedia applications need access to resources in short burst and with minimal jitter. This kind of service is not be provided by a general-purpose scheduler [137].

All of the above systems are dedicated to a specific purpose. A reusability framework is a critical requirement for the cost-effective construction of these dedicated systems. Different execution environments and implicit assumptions lead to bad reuse inside the operating system and between operating system code and user level code.

1.3 Extensibility and fine-grained protection

Besides being able to configure dedicated systems it should also be possible to extend the system at run time. An extensible OS is not completely created at built time but contains mechanisms that allow to add or replace system functionality at run time.

A special form of extensibility are *hot pluggability* and *hot swappability.* Hot pluggability means the extension, hot swappability means the exchange of a system component during normal operation and without the need to restart the system. Hot swappability allows to update a system component with a new version. System busses for embedded systems, such as Compact peripheral component interconnect (CompactPCI), allows to exchange hardware during normal system operation. The exchange of hardware components usually must be followed by an exchange of driver software, which must be hot swappable. Hot swappable OS components used, for example, in telecom switches to allow system updates without shutting down the switch. Several research projects investigated hot swappability. Examples include the Synthetix [179] system that uses hot-swappability to replace a general component with a specialized component, and the K42 OS [96].

Most operating systems today are extensible using loadable kernel modules but this completely lacks protection. The combination of extensibility with protection is one of the major advantages of a microkernel-based system that adders to a multi-server approach. Spreading the OS functionality among many servers means that a part of the functionality can be replaced by replacing a server.

2 Design objectives

To provide the described properties the system design has the following objectives:

Support for context-independent code. A flexible definition of protection and trust boundaries must be possible. This allows reuse of code between different protection domains. Even OS-level code, such as device drivers, file systems, and network protocol handlers, should be reusable in an application context.

Separation of concerns. Code that implements a certain functionality should be separated from code that manages resources or enforces a certain security policy. This is an essential requirement for reusability.

Shared-nothing. As hardware, especially CPU and main memory, constantly becomes cheaper and more capable it becomes more cost-effective to run applications on one host that formerly required separate hosts. In order to keep the illusion of separate hosts the operating system must allow to completely isolate such applications. They should not content for resources and should not share any re-

sources. Typical examples for contention in traditional systems are the file system's buffer cache and the working set of a process.

Application-specific resource management. There is no single resource management strategy that is optimal for all applications. Thus applications should be able to implement their own resource management.

3 Contributions

The assumption that underlies this dissertation is that the root of the described problems is the address-based protection mechanism. I believe that this mechanism is not able to cope with current problems and that it should be replaced with a more flexible protection mechanism: type safety. The contribution of this dissertation is an operating system architecture that solely relies on a type-safe instruction set for protection. This architecture has been implemented and the prototype is able to run non-trivial applications, such as a file system, a network file system, a relational database system, a web server, and a window manager. The performance of these applications indicates that the system is usable even with today's hardware that is optimized for address-based protection.

This dissertation contains the first design, implementation, and performance evaluation of a multi-user timesharing operating system that relies on type-based protection[1].

4 Dissertation outline

The next chapter gives an overview of the research in operating system architectures. The discussion focuses on protection mechanisms, extensibility, and flexibility. Chapter 3 explains the overall architecture of JX. The following chapters describe the JX operating system in detail. Each chapter concentrates on one aspect, describes the design and implementation and evaluates the specific part of the system using micro benchmarks. They describe protection domains (Chapter 4), the communication system (Chapter 5), the component architecture (Chapter 6), CPU scheduling (Chapter 7), and memory management (Chapter 8). As these concepts are tightly interrelated it is sometimes necessary to use a detail that is explained in one of the following chapters. A forward reference is provided in such a circumstance. Chapter 9 describes the architectural support for device drivers. The security implications of the system design are studied in detail in Chapter 10. Chapter 11 surveys related work in Java-based operating systems and resource management. Chapter 12 evaluates the functionality and performance of the system. The final chapter gives a conclusion and perspectives for future work.

1. It is not possible to state such a claim in this generality. Thus we state that we were not able to find the description of such a system in any of the major operating system journals, conference proceedings, or workshop proceedings.