Inhaltsverzeichnis

1 Background **3**

2 Present situation **7**

3 Required steps **9**

4 Planned design **11**

4.1 Integration into Genode 11

4.1.1 Copy base-linux 11

4.1.2 Include base-linux 11

4.1.3 Extend base-linux 11

5 Implemented design **13**

6 Evaluation **15**

7 Conclusion **17**

1 Background

At the present time we are surrounded by devices that run complex software and operating systems. The number of these devices is still rising and their complexity is growing by the increasing amount computational resources, functionalities and interfaces. Also many of these devices are taking over important tasks such as smart locks or even cardiac pacemakers. Since most of these devices are connected to the public Internet they cannot only be accessed by their owners but anyone. These facts raise the question if we can trust these devices. Many cases in the present have shown that nearly all of these devices have security issues. Since the used systems are too complex to guarantee the absence of software errors the common practice is patching these devices when an error has been discovered. Unfortunately the approach of fixing errors has not proven reliable. Updates are often not or only for a short time available and often do not find their way to the end user. A difference to this practice are sectors where manufacturers are liable for the correctness of their products including software parts. This correctness is achieved by the separation of components and their individual evaluation and verification. The downside of this approach are the high costs, especially for a high amount of components. To reduce the costs multiple functionalities are often combined in one component. To securely separate the functionalities from each other separation kernels are used. Since those need to be secure they only consist of a low complexity and a few thousand lines of code. This reduction comes at the cost of flexibility at runtime which often lets vendors choose more complex but easy to use and configure systems. These often have a large trusted computing base (TCB) of multiple million lines of code. This leads to the question how this contradiction between flexibility and security can be resolved.

Genode aims to solve this problem through multiple ways. Instead of a monolithic kernel it uses microkernels. They provide a greater flexibility than separation kernels why keeping the size of the trusted computing base at a manageable level. This is accomplished for example by running device drivers as separate user space processes. User space applications in Genode are also managed by capabilities. They provide a way to allow and deny access to a certain number of system resources without the need for a global complex policy. Additionally to capabilities resources can be managed in a hierarchical way. Software components can be be partitioned into parts of different complexity. This helps to exclude complex parts from the trusted computing base and therefore reduce the amount of code that needs to be verified. The continuation of

this approach leads to an application specific trusted computing base that consists of the application itself and all the parts it depends on [1.1]. [1]

The Genode OS framework is the implementation of this architecture for highly secure special purpose operating systems. Mainly written in C++ it scales from embedded systems with only a few megabytes of memory to big systems with dynamic workloads. The architecture consists of a recursive sandbox structure. Those are controlled from their parents and separated from their siblings. But they can create additional sandboxes and subtrees of children. This creates hierarchical structures and reduces the attack surface of the system since a compromised component cannot access resources of other components. These components are small blocks that can create complex systems without unnecessarily increasing the trusted computing base. Beside being programs they can also provide other functionalities such as device drivers or protocol stacks.

Genode supports multiple CPU architectures such as Intel i386 and ARM. Multiple microkernels such as NOVA and seL4 can be used. Additionally Genode provides a custom kernel implementation that in comparison to other kernels further reduces the trusted computing base. The current framework also provides many usable driver and application components. Beside common periphery drivers such as USB and Intel wireless this also includes complex application frameworks such as Qt5. [1]

To debug components in the development process Genode can also be started on the current running Linux kernel. This is a good approach for applications but is also limited. Some resources cannot be used since the Linux kernel doesn't provide them to the user space, for example memory mapped IO or direct memory access. To develop and debug native Genode device drivers the kernel needs to act as a microkernel and provide access to these resources in Genode. Microkernelizing Linux is the goal of this thesis and consists of two parts. The functional part is to provide the system resources such as direct memory access, IO ports or memory mapped IO. To use this on different platforms Genode should also boot directly on a Linux kernel as init process. When the functional part is done Linux should be stripped down to reduce the trusted computing base.

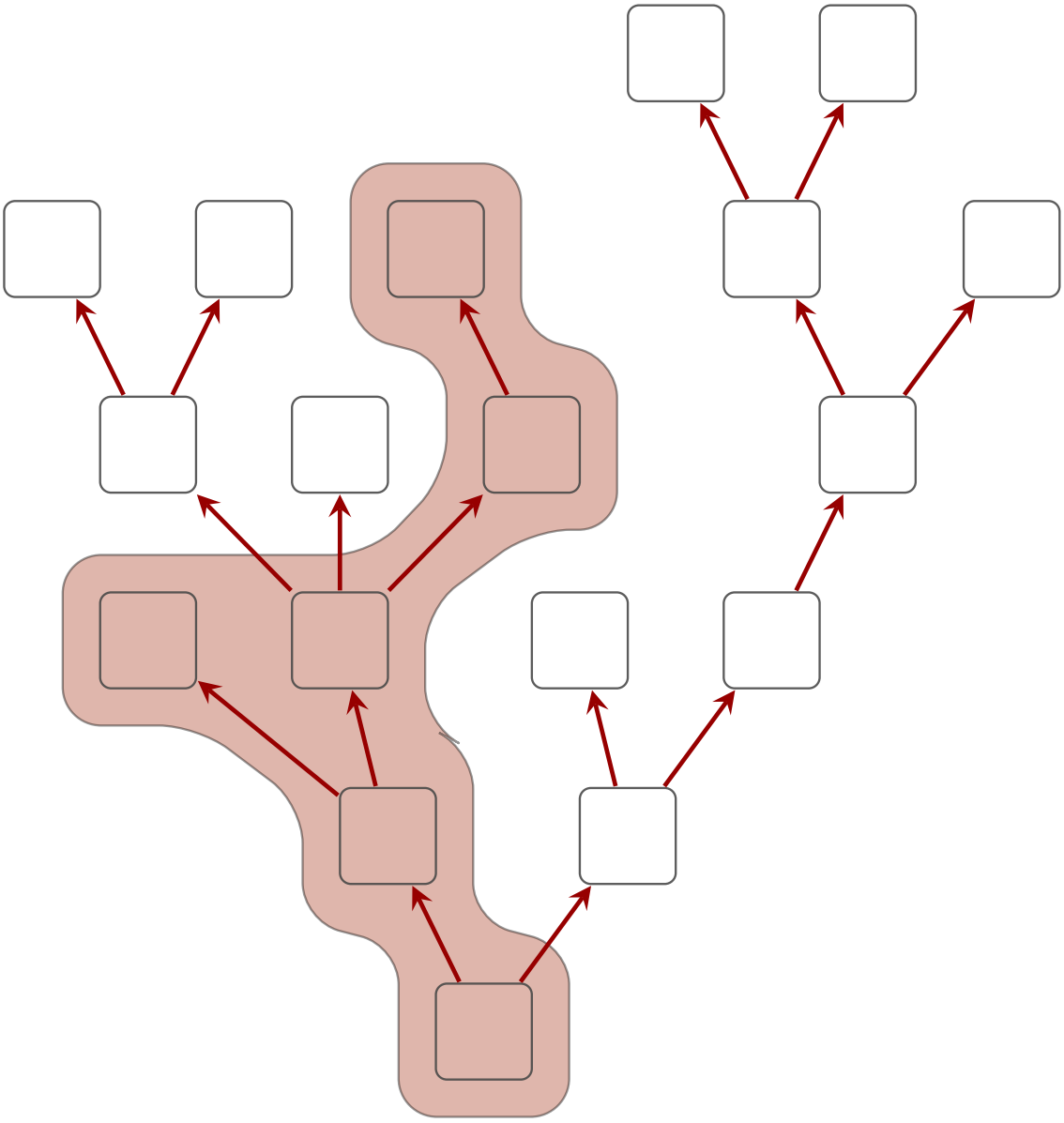

Abbildung 1.1: Application based TCB [1]

2 Present situation

3 Required steps

4 Planned design

4.1 Integration into Genode

The integration into Genode is done by creating a base directory. This is used to build a core binary which then is executed by the kernel. Since there is already a core that is able to use the Linux kernel it would be redundant to rewrite these parts. Reusing the already available code can be done in three different ways.

4.1.1 Copy base-linux

The simplest way to build upon base-linux is creating a copy it in a new base directory. The advantage of this approach lies in the fact that there is, beside including the new directory, no need to modify upstream code. On the other hand this leads to code duplication and any updates of base-linux probably need to be ported to its modified copy.

4.1.2 Include base-linux

Creating a new base directory and including the needed contents from base-linux via Makefile fragments. While this would include the advantages of the copy based approach there is less to no code duplication. The disadvantage here is the complexity due to the build system that does not provide a standard way to include parts of other base directories.

4.1.3 Extend base-linux

To prevent both code duplication and complex includes the already existing base directory could be used and extended. While this approach seems to be easier it has two caveats. Since it requires modifications of the original base-linux core the current functionality needs to be maintained. This can be done either by creating a hybrid core that is able to run like the current one and on a bare Linux kernel or a switch needs to be added that chooses the target on build time. Additionally this code needs

to be brought upstream as it cannot be plugged into the source tree like a separate base directory.

5 Implemented design

6 Evaluation

7 Conclusion

Literaturverzeichnis

[1] Norman Feske. Genode Operating System Framework 17.05 Foundations. Technical report, May 2017.