Gapfruit Trustworthy Execution Platform

Sid Hussmann
Gapfruit Technologies
Switzerland

July 14, 2021

Abstract

In a competitive environment where vendors have to bring products to market quickly, engineers are often forced to design products that serve an overly broad spectrum of customers. To satisfy their needs, they must focus on their core expertise - often neglecting security. Integrating technologies and applications from many different vendors into one product is difficult and time-consuming. Under ever-growing threats, this complexity leads to increased security and safety problems and results in enormous expenses for manufacturers, integrators, and customers. The antiquated operating system concepts upon which today's IT is built are not appropriate for the 21st century.

Gapfruit Trustworthy Execution Platform (TEP) is a modern microkernel platform with capability-based security. Gapfruit TEP scales from specialized embedded systems, over carrier-grade appliances to high-performance cloud servers. The platform architecture gives complete control over all software stacks and allows to form rational arguments as to why the platform is considered trustworthy. It eliminates typical attack vectors and reduces the impact of software flaws significantly. Gapfruit TEP allows running mixed critical applications side by side while guaranteeing strong separation of concerns. Additionally, Gapfruit provides several low-risk migration paths enabling any unmodified application to run in strongly isolated environments.

A particular use-case of Gapfruit TEP is being able to design custom Trusted Execution Environments with attestation, described in section 5.

Contents

1 Introduction

Today's operating systems are built in a way that the kernel, the most critical part of the system, consists of millions of lines of code. In these cases, the kernel forms the first abstraction of the hardware. Software that runs on these operating systems can access a vast number of system calls provided by the kernel. These system calls are used so that applications can access system resources such as file-systems, networking, devices, etc. Code, running in kernel mode, has complete access to all of the hardware and can execute any instruction the machine is capable of executing. Important but complex software like device drivers and protocol stacks run in privileged mode. An exploit of a likely bug in any of these components results in the complete takeover of the computer (see Figure 1).

Instead of talking about secure computing, the term *trusted computing* is preferred by the security research community. The heart of every trusted system should be a minimal *Trusted Code Base (TCB)*. It is undisputed to keep the critical components in a system as simple as possible. The kernel is the most trusted part of the operating system. Microkernels are used in areas where vendors are held responsible for the correctness of their products and where physical separation is not feasible. These systems are kept in service for decades and an error may have disastrous consequences.

Many areas in IT and OT suffer from cyber threats such as data breaches, DDoS, ransomware, etc.; threats, that would significantly be reduced by strong separation. So why are microkernels not applied in these other areas? For one part, developing products with microkernels is extremely difficult and the systems are mostly static and hard to maintain. On the other hand, there used to be a price on performance compared to monolithic kernels.

With Gapfruit Trustworthy Execution Platform (TEP) these concerns are obsolete. Its groundbreaking SLICE architecture with capability-based security enables one to design systems where functionality from different sources come together without sacrificing availability, integrity, and confidentiality - combined with great performance.

2 Context

Competitive markets put product manufacturers under great pressure to implement their ideas and release new products as fast as possible. Functionality is what counts. That's what customers are willing to pay for. Security is something to be considered *after* implementing the main features - if thought of at all. There are way too many aspects in software security that need to be considered, to provide a secure solution. Thus, vendors trying to bring new high-quality products to the market often either miss deadlines and budget goals or deploy products with severe security vulnerabilities.

2.1 Complexity Hides Dependencies

In markets where vendors are held liable for the correctness of their products, physical separation provides the highest assurance for the independence and protection of different functionalities from each other. For example, aircraft contain dozens of electronic control units (ECU) that can be individually evaluated and certified. However, factors such as cost, power, or weight considerations call for the consolidation of multiple features into a single ECU. At this point, microkernels are applied to partition the hardware resources into isolated compartments. As the isolation is only as strong as the correctness of the kernel, such kernels must undergo a thorough evaluation. In the face of being liable, an oversight during the evaluation phase may have disastrous consequences for the vendor and people exposed to the system. Each line of code to be evaluated is an expense. Hence, microkernels are minimized to the lowest possible complexity – up to only a few thousand lines of code [2].

On the other hand, kernels of today's general-purpose operating systems such as Windows, Linux, *BSD or OSX have millions of lines of code (LOC). Linux currently counts over 30M LOC and the Windows kernel is estimated to consist of 65M LOC [1]. All of which evolved over decades and their initial focus was on usability, performance, and portability. Different security mechanisms were considered after the architecture was set in stone. For instance, the features of the kernel are configured via multiple types of configuration files, which access is granted to specific users via access control lists (ACL) of the file-system. Additionally, each process can invoke over 300 system calls, which provide access to features the kernel provides. Figure 1 illustrates the attack surface on monolithic kernels.

The formula $Risk = Likelihood * Impact$ is often used [3] to argue about the risk a system is posing when in use. It is a simplified formula, which states that risk is a product of likelihood and impact. In the case of monolithic kernels, the likelihood of an error is high because of the vast complexity. The impact of an error is also very high since every line of code that runs in the kernel is critical and an exploit results in the complete takeover of the whole system. Thus, using monolithic operating systems for critical infrastructure is risky. The perception of what should be considered critical in an infrastructure is shifting towards a common understanding that we need to trust the interconnected things around us.

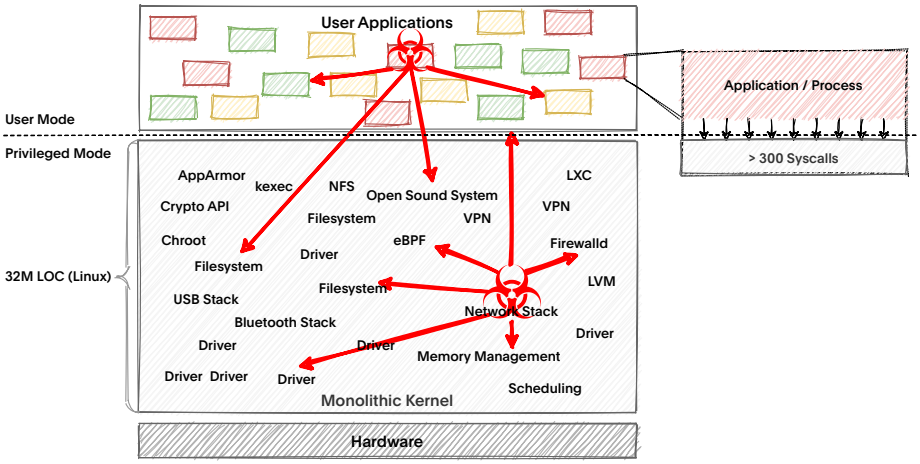

Figure 1: Attack surface of monolithic operating systems

2.2 Product Development and Maintenance

Writing secure software is hard. Additionally, developing products on top of monolithic operating systems, with their large attack surface, requires a constant re-evaluation of security issues.

Products are often in use long after the vendor has ceased support. Especially in markets such as transportation, telecommunication, banking, industrial/building automation, automotive, critical infrastructure, or the military, devices are in use for decades. To ensure the quality of their products over the whole product-life-cycle, manufacturers need to spend a fortune on maintenance, including products sold years ago. Due to the growing complexity, security vulnerabilities are increasing exponentially. For manufacturers need to track *Common Vulnerabilities and Exposures* (CVE) [4] of each library or 3rd party software component and respond by updating their products for the whole lifecycle of the product.

Again, because of the complexity of those systems, maintaining monolithic kernel features is very time-consuming. As kernel-internal APIs and ABIs are not stable, third-party or self-developed drivers may not work with future versions of the kernel. This results in not being able to benefit from security patches introduced in new kernels and leading to the divergence of code-bases. Also, bug fixes often take a long time to get integrated [5, 6] or compete against performance requirements [7]. Even with Long Time Support (LTS) kernels, many CVEs remain unfixed [8].

Additionally, deploying upgrades often have unforeseen consequences to other parts of the system. Thus, the whole product needs to be rigorously tested before a new system image is sent to the device. It is also very common that even this upgrade mechanism fails, leaving the system in a non-operational state.

Maintaining products is therefore very expensive for manufacturers and integrators, resulting in unpatched and unmaintained products waiting to be exploited and putting the vendor's reputation at risk.

Containerization is a ramification of architectural limitations in commodity operating systems, programming languages, and frameworks. The main objec-

tive is that different teams deploy their code independently, the impact of errors is limited and new features are released rapidly. Such issues can be addressed using container technologies like Docker [9]. However, these "light-weight virtualization" techniques raise new security issues because they all use the same kernel with weak separation. Containerization doesn't reduce the size of the trusted code base which the container isolation and security safeguards built upon.

Furthermore, containers are black boxes. If the system runs as expected, it's easy to neglect which software and which specific version is deployed. A container may look like it's performing perfectly from the operational point of view even when it's running components that are contaminated by a critical security flaw. This flaw has been fixed long ago upstream, but not in the local image.

3 Vision

Even though Linux is open source, verification of its correctness is far from feasible due to its enormous complexity. So the true question rises:

How can we prove the trustworthiness of a product?

This is where Gapfruit Trustworthy Execution Platform comes in. Gapfruit TEP is built with the core security principles, *Isolation, Containment, and Simplicity* [11], from the ground up. The primary objective of its architecture is to enable absolute control over all software stacks while delivering a platform for securely separated building blocks in a robust and scalable way. Gapfruit provides a novel solution to design secure systems using modern operating system concepts. **Each component only has the smallest possible set of privileges that are required to fulfill its functionality.** This greatly reduces the trusted code base of any software feature. Resources such as RAM or CPU are distributed in a recursive parent-child relationship, while access to services such as networking or file-systems is controlled by a client-server service topology (Figure 2). Device drivers and protocol stacks are moved from the privileged mode to userspace. So even an untrusted driver or network stack only has an impact on the services it provides and the resources (their own) it received from its parent.

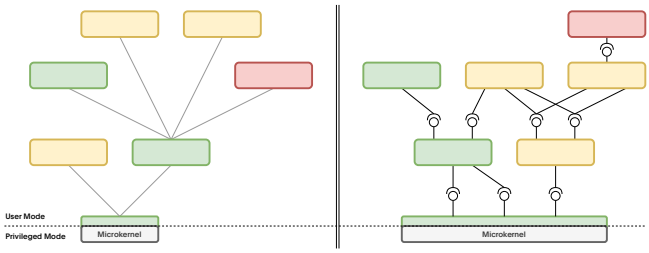

Figure 2: Control over the trusted code base in terms of resource distribution (left) and service topology (right)

To improve the situation regarding software updates on devices, Gapfruit uses a sophisticated package management system supporting reliable and reproducible upgrades (Chapter 4.5). The upgrade mechanism is not only used for user applications but also system components such as device drivers, protocol stacks, software libraries, and even the boot image. This enables rapid rollouts of upgrades and new features.

Gapfruit combines the lessons learned from the following disruptive technologies:

- Trusted Execution Environments.
- Micro-Services.
- Software-Defined Networking.
- Network Function Virtualization.
- Compartmentalization.

The solution is a rock-solid operating system that is inherently secure and future-proof. Gapfruit TEP is therefore suited for many different use-cases such as edge computing, hardware security modules, secure endpoints, IoT gateways, medical devices, banking infrastructure, transportation, automotive, or industrial and building automation systems [10]. With Gapfruit TEP, device manufacturers can focus on their core expertise and will leap ahead of their competitors in the vital aspect of security. Thus, customer satisfaction and trust are increased, whereas the Total Cost of Ownership (TCO) is significantly reduced.

Additionally, with Gapfruit TEP, device manufacturers can prove the trustworthiness of their product through attestation and by making audits and code walkthroughs feasible.

4 Core Concepts of Gapfruit TEP

Gapfruit introduces a new level of security to computer systems. The Trustworthy Execution Platform is built with the Genode Framework [12] that supports several microkernels and runs on x86-64, ARM32/64, and RISC-V chips. The building blocks for Gapfruit TEP are called SLICEs. A SLICE is a *Secure and Light Instance of Contained Enclave* which has strong isolation guarantees so that application and data are protected at runtime. Furthermore, potentially malicious code is contained from breaking out. An analogy would be the objective of enclaves combined with what virtual machines or sandboxes try to achieve (Figure 3). So the isolation of SLICEs is guaranteed from outside-in as well as from inside-out.

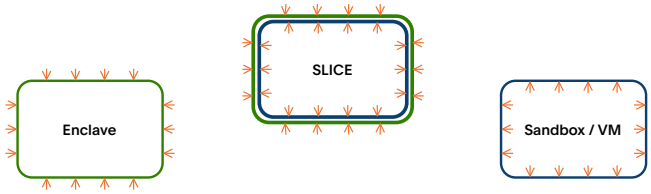

Figure 3: The isolation of a SLICE in contrast to enclaves and sandboxes

Each SLICE may provide and may consume services. SLICEs are connected using a service-oriented communication mechanism (Figure 5). A SLICE may consist of a single component, a group of components (Figure 11), or even a virtual machine monitor (VMM) to host a complete multi-purpose operating system such as Linux, Microsoft Windows, or *BSD (Figure 7).

Another core concept of Gapfruit TEP is how dependencies are handled. There are three types of dependencies which are shown in Figures 4 and 5.

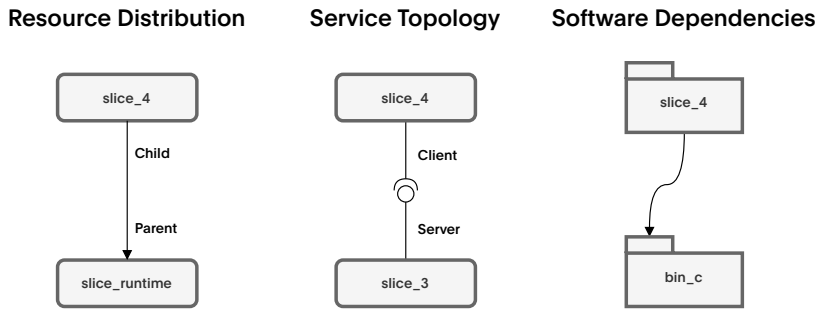

Figure 4: Types of dependencies

The first type of dependency is *Resource Distribution*. A child component depends on its parent. Parent components are critical to the system, thus they have a very small code base, so they can easily be verified for correctness. A parent component is called *init* which can be configured to spawn child components and provide them with resources such as RAM and CPU. *init* may also establish service connections to other components. These connections form the second type of dependency: *Service Topology*. Gapfruit TEP has a service-oriented

architecture where a client depends on a server. Note that even though the server is more critical in regards of availability to the client, confidentiality and integrity is still guaranteed. The third type of dependency controls the supply chain of *Software Dependencies*. Where a SLICE depends on binaries, libraries, or other artifacts that are part of distributable packages. Figure 5 shows these three types of dependencies in three views of the same system.

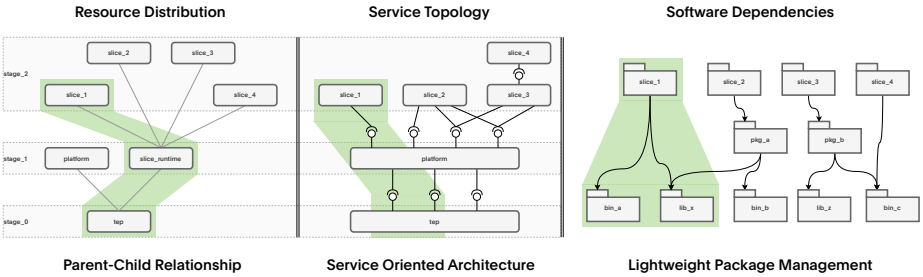

Figure 5: Topologies of dependencies

A device running Gapfruit TEP is bootstrapped in three stages. *Stage 0* is the static system, containing the microkernel, *core*, *init*, and the minimal IO drivers to get the system started. *Stage 1* brings the device drivers and multiplexers that set the service interface for SLICEs to connect to. Figure 5 shows on the left the *slice_runtime*, the parent component of all SLICEs, and which is responsible for resource distribution and service routing between the SLICEs and the platform SLICE. *Stage 0* and *stage 1* are typically built by board support package (BSP) developers enabling Gapfruit TEP to a given hardware. Once the platform is enabled, SLICEs can be deployed to *stage 2*.

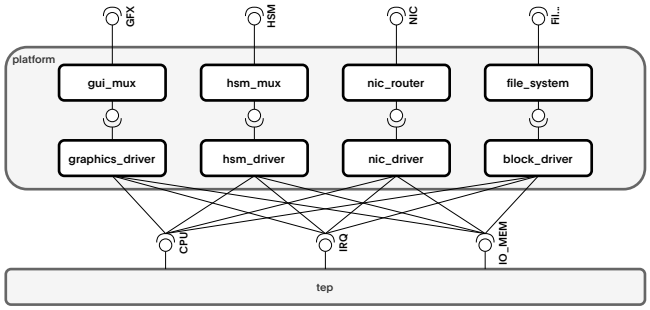

Figure 6: Gapfruit TEP device drivers and service multiplexers

The platform SLICE in *stage 1* abstracts the hardware and contains all low-level drivers that are needed on that particular device. Device drivers encapsulate singleton resources and multiplexers isolate the communication to different components. Multiplexers are simple components with a very small trusted code base so that they can easily be verified for correctness.

4.1 Recursive System Composition

Since every component only has access to the resources and services it really needs, defining a flat configuration for a full system with all processes is not feasible. A SLICE encapsulates the configuration of its internal components and specifies the services it requires. This way, SLICEs can be deployed to every platform that provides the required services. Figures 6, 7, and 11 show examples of such encapsulations.

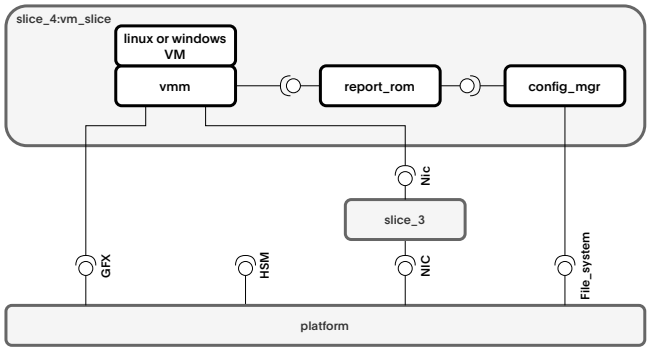

Figure 7: Example SLICE and its internals

4.2 Trusted Computing

Gapfruit TEP builds on the following modern OS security concepts:

4.2.1 Security Model

Commodity operating systems have huge APIs all processes have access to. All *doors* are open. There are some security features available for mitigation, limiting access rights on file-systems or more complicated tools that close certain *doors* such as SELinux [13], AppArmor [14], or seccomp [15].

With the capability-based architecture of Gapfruit TEP on the other hand, there are no doors. Just like with a blueprint for a building, paths are specifically defined and opened for each component if it is required to fulfill its functionality. Unlike with a building, however, this blueprint is highly dynamic. It forms a mandatory access control system that is inherently defined in the SLICE topology. This technique limits the attack surface of a specific component to the bare minimum.

For a component to communicate, it requires a capability. A capability is an access control token created and managed by the microkernel. The creation of capabilities is controlled by explicit policies that specify what component may communicate with which other component(s) or use which resource(s).

4.2.2 Small Trusted Code Base

As software complexity correlates with the likelihood for bugs, having security-sensitive functionality depending on high-complexity software is risky [2]. The term *Trusted Code Base (TCB)* denotes the amount of code that cannot be compromised to uphold the integrity and availability of an application.

Gapfruit TEP limits the complexity of all of its critical components. These components are small, open-source, and verifiable by design. This reduces the certification and especially the re-certification costs for e.g. safety-critical products significantly. Figure 5 shows how straight forward it is to identify the critical components and their resulting TCB (green). Non-critical or even untrusted components can be deployed to the same platform without sacrificing e.g. the functional safety certification of the critical part(s) of the system.

4.2.3 IPC Addressing

The principle of least privilege states that all code should have only the privileges needed to provide the required functionality. For maximum performance Gapfruit TEP processes communicate directly with each other. However, this communication channel has to be established by a mutually trusted parent component. The parent is made specifically to enforce policies. A component is otherwise completely isolated from the rest of the system. Instead of performing access control based on a client identification in the server (like it's done in e.g. QNX [16]), access control for IPC is solely performed by the microkernel on the invocation of capabilities.

By using nested page tables, the microkernel allows components to securely share memory areas, that afterward can be re-used to exchange data efficiently without compromising confidentiality.

4.2.4 Secure Virtualization

For running full-blown guest operating systems inside SLICEs, Gapfruit TEP supports several existing virtual machine monitors (VMM) that contain large code-bases [17, 18, 19]. However, it does not rely on its separation technique. Every guest OS has its own VMM. When a virtual machine gets compromised, due to a likely bug in the VMM and the malicious code can escape the virtual machine, the rest of the system is protected by the strong separation of Gapfruit TEP and the hardware virtualization features provided by the CPU.

4.2.5 Userland Drivers

In a typical operating system like Linux, Windows, or *BSD, the core networking code, including network card drivers and protocol stacks, all run in kernel mode with privileged rights. Any bug in a driver or a protocol stack that gets exploited in an attack would result in a full system compromise [20].

Userland drivers mitigate many problems concerning kernel space drivers. Generally, they are simpler and more flexible and can profit from all the tools that are normally available for software development. Performance can be significantly better when combined with disabled interrupt handling since there are far fewer context switches. Thus, packets sent and received do not have to be copied between user- and kernel space [21]. Examples of such drivers are DPDK [22] or Snabb [23].

To minimize the attack surface, in Gapfruit TEP, device drivers run in userland and are separated by IOMMU protection.

4.3 Low-Risk Migration Paths and Compatibility

Today, the security benefits of microkernels are solely used in niche markets. There are countless software solutions in productive service. The learning curve to migrate existing applications to a capability-based microkernel system such as e.g. CAmkES [24] is extremely steep. There is no real migration path. This is where Gapfruit comes in. Gapfruit TEP is based on a capability-based microkernel architecture that provides runtime environments to run all existing software solutions: e.g. POSIX compliant C/C++ runtime, ADA/SPARK [25], JVM [26], Python [27], various unikernels [28], Wasm [29], VirtualBox [17], etc. For drivers, Gapfruit TEP features Linux and BSD device driver environments. Gapfruit provides support for several microkernels such as seL4 [30], nova [31], or base-hw [12]. As an intermediate step for existing Linux-based appliances, Gapfruit TEP may also be used on top of the Linux kernel while having capability-based security and the isolation mechanism enforced by seccomp [32].

4.4 Scalability and Flexibility

Network Function Virtualization (NFV) introduced a new level of scalability and cost savings for telecom companies by running entire classes of network node functions in virtual environments, that previously used to be specific hardware appliances. These virtual building blocks may be connected or chained together to create communication services. One of the many benefits of NFV is that these virtual environments can be orchestrated to commercial off-the-shelf products or to the cloud.

To this scalability, *Software Defined Networking (SDN)* adds the flexibility of configuring the network topology. SLICE topologies are defined in a desired state management pattern and these configurations are pushed to the device on the management plane [33, 34, 35].

Gapfruit takes this idea a step further. Not only is it possible to run these NFV building blocks efficiently on the platform, but it also does so in an unprecedented secure way. Furthermore, Gapfruit TEP is not limited to running virtual network functions. With its SLICE building blocks, functionalities such as disk encryption, file analysis, protocol gateways, etc. can easily be added to the system.

4.5 Package Management and Deployment

Gapfruit applies the lessons learned from managing micro-services in cloud environments. Its lightweight packet distribution system solves the trade-off between deploying subsystems independently and sharing common libraries in a secure way. Once a SLICE package is built, it can be deployed everywhere.

The package management system is inspired by the one of NixOS [36] and has the following characteristics:

- *Integrity protected:* Each package is individually signed. Critical packages can have multiple signatures making Gapfruit TEP work well where TUF/Uptane [37] is required.
- *Transactional upgrades:* During a package upgrade, the system remains in a consistent state. If a SLICE launches at any point in time, it's either the old or the new version. But not something in between.
- *Independent deployment:* Installing and upgrading packages will not break other packages.
- *Rollbacks:* Upgrades don't overwrite the old packages. If a new SLICE doesn't work, it's always possible to revert to the previous state. This makes the rollout of upgrades substantially less jeopardous.
- *Reproducibility:* Gapfruit TEP downloads the dependencies in the exact versions required by a SLICE.
- *Lightweight:* If a library is required by two separate packages and the version matches, it is only downloaded once.
- *Transparent:* Auditing which packages are in use is trivial and can be done during build, deployment, and run-time.

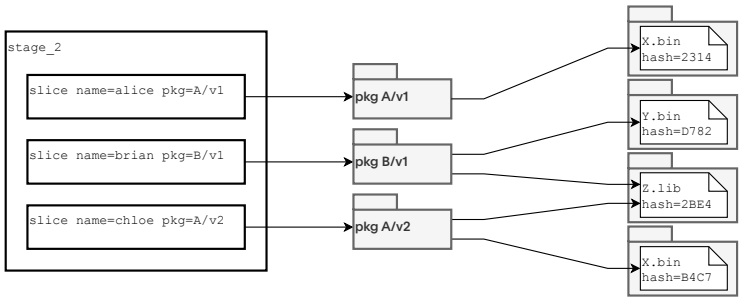

Figure 8: Package dependencies

Gapfruit SLICE packages only store the dependency to archives needed for a SLICE to run. If an archive does not exist, it is downloaded and the integrity is verified. Figure 8 shows how SLICEs map to packages and different versions of these running in parallel. In this example, three SLICEs depend on two packages *A* and *B* of which *A* exists in version *v1* and *v2*. Package *A/v1* only depends on binary *X.bin*. However, new features were added to package *A (A/v2)* which resulted in a new version of *X.bin* and a dependency on *Z.lib*. Package *B/v1* depends on *Y.bin* and the same *Z.lib* that is used by *A/v2*. Thus, the archive which contains library *Z.lib* is only stored once on the file-system.

4.6 Availability and Resilience

Availability is similarly handled as in Kubernetes [38]. The desired state of the SLICE topology is defined using a RESTful configuration interface. SLICEs can be started and stopped individually.

Gapfruit TEP supports analyzing the health of each SLICE during run-time and depending on the criteria, restarting SLICEs when required. Figure 6 shows that even device drivers are designed in a way that they can be restarted while keeping the impact on the overall system to a bare minimum.

Via the system configuration, it is possible to pin a SLICE to one or more specific CPU cores. This can be used to prevent interruption of a critical SLICE so that its functionality is deterministic and its availability is guaranteed.

Each SLICE can be restricted by memory usage. The kernel enforces this limit and stops (and may restart) components that exceed the configured memory limit. This prevents the system from becoming unstable due to exceeding memory consumption of individual SLICEs.

4.7 On-Chip Security Features

Gapfruit runs on x86-64, ARM32/64, and RISC-V chips and supports hardware-provided security features such as Intel VT-d/VT-x, AMD-V, AMD IOMMU, TPM 2.0, and ARM TrustZone.

5 Trusted Remote Execution with Attestation

Having complete control over the Trusted Code Base has many use-cases. One of which is the Trusted Execution Environment (TEE) with attestation.

Trusted Execution Environments aim to protect the integrity and confidentiality of computation over sensitive data and/or sensitive code during runtime. TEEs are becoming a requirement across various industries and use-cases such as banking, GDPR compliant machine learning, or blockchain applications such as smart-contracts or signing cryptocurrency transactions.

Trusted Execution Environments should provide at least the following:

- Physically and logically secure the execution of the application code from any interference (Isolation Property).
- Attest to a verifier the runtime integrity of an application, including the whole TCB.
- Creating an immutable record of the execution of the application code, detailing input, output, time, and device state (Audit Property).
- Restrict access to start an execution to authorized entities (Authorization Property).

This entails having a mechanism to securely load code, protect code from alteration, and extend to protecting the processed data and its output. A TEE on Gapfruit TEP can prove that a certain output was generated from a specific input, executed at a specific time with specific code. So it works like a notary that proves its trustworthiness through attestation.

With Gapfruit, TEP device manufacturers, platform providers, and the enclave application developers can customize TEEs to their requirements and threat models. Gapfruit TEP allows any application to run inside a TEE.

Additionally, Gapfruit provides an easy-to-use API [39], which eliminates dealing with the complex handling of crypto serialization systems such as ASN.1, PKCS#11, etc.

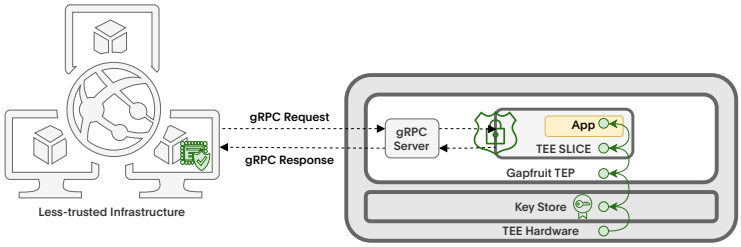

Figure 9: Remote trusted execution

Figure 9 shows a high-level use-case where a TEE client connects to a *TEE SLICE* running on Gapfruit TEP via gRPC [40]. A more detailed diagram of what it means to have customizable Trusted Execution Environments can be seen in figure 10. This example shows a multi-tenant situation where *Domain Orange* belongs to one tenant, while *Domain Green* belongs to another. The *gRPC Server* on the left is connected to a *NIC* session (Network Interface Card). This way it can be reached over an external network from a TEE client, seen in Figure 9. The *gRPC Server* contains complex third party code. Multiple threads are running, multiple TCP connections and TLS sessions are handled, as well as HTTP and Protobuf [41] is being parsed. The system is designed in a way that the *gRPC Server* is outside of the TCB and can be restarted at any time without having an impact on the health of the system. The integrity is protected within the signed requests and the signed responses.

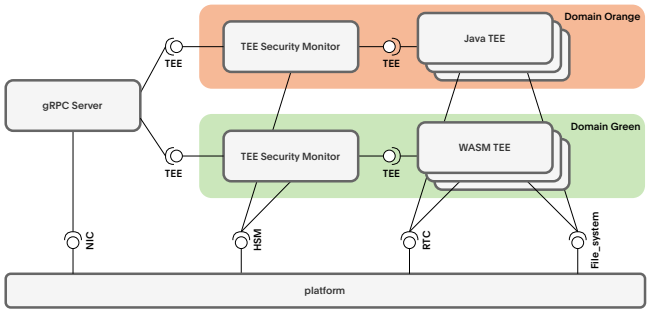

Figure 10: Customizable Trusted Execution Environments

The *gRPC Server* forwards requests to the *Security Monitor* of the addressed *Domain*. The *Security Monitor* has two main responsibilities: Enforcing access control for TEE requests and providing attestation information to responses. After verifying the client-signature, the *Security Monitor* forwards the request to the *TEE SLICE*. The *TEE SLICE* loads the executable, calculates its hash,

and executes it in the runtime given by the application type. The SLICE then provides input from the request to the executable. After execution, the SLICE creates a response containing the output from the executable and sends it to the *Security Monitor*. Here, the response is enriched with information about the system, such as the platform state in form of a Merkle tree, timestamp, and secure counter value for non-repudiation. The response is then signed with an attestation key and sent back to the TEE client via the *gRPC Server*.

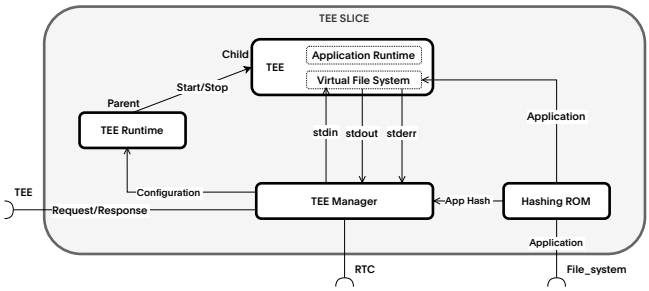

Figure 11: Trusted Execution Environment Internals

Figure 11 shows the internals of a *TEE SLICE*, in this example the *Java TEE* shown in Figure 10. The *TEE Manager* receives requests and sends back responses from and to the *Security Monitor* accordingly. The *TEE Manager* controls the *TEE Runtime* and establishes communication channels to *stdin*, *stderr*, and *stdout* of the application. The *TEE Runtime* is a parent component that spawns the JVM. The *Hashing ROM* component loads the application image from the file-system, calculates its hash, and provides said hash value to the *TEE Manager*. Accordingly, the *TEE Manager* includes said hash value into the execution record of the respective application code. The *Hashing ROM* then provides the application code as read-only memory to the JVM to execute. The *Hashing ROM* and its relationship to the JVM are built in a way so that time-of-check to time-of-use (TOC/TOU) attacks are prevented. After successful execution, the execution record and the output on *stdout* as well as *stderr* are then added to the response which is sent back to the *Security Monitor*. The same mechanism can be applied to all runtimes [4.3] supported by Gapfruit TEP.

5.1 Attestation

The chain of trust used in software attestation is rooted at a signing key owned by the hardware manufacturer of an HSM, TPM [42], or any other form of a secure element such as OpenTitan [43]. The architecture in Figure 10 shows that only the *Security Monitor* has access to the *HSM* service. This is enforced by the capability-based security of Gapfruit TEP. The platform makes sure that only the *Security Monitor* of a specific domain has access to the key store of the respective domain. Every gRPC response is signed with a domain-specific attestation key. The certificate of the attestation key is signed by a device-specific key, which in turn is signed with a root key of the device manufacturer.

Since domain-specific keys are linked to device keys, any response that is signed by the respective *Security Monitor* can be cryptographically linked to a specific device via the certificate chain of the attestation key. A gRPC client may request a quote from a domain. The quote can be used to provision a TEE.

The quote holds information about the environment such as:

- Certificate chains.
- Fingerprints of allowed client certificates.
- Information about the specific hardware.
- Timestamp.
- Secure monotonic counter.
- Environment settings.
- The Trusted Code Base (TCB) of the domain.
- etc.

The TCB of the domain holds a view of the complete trusted code base in form of a Merkle tree [44]. The Merkle tree represents the TCB of the whole subsystem of the device which is involved to execute code inside of a *TEE SLICE*. It unifies the TCB in terms of *Resource Distribution*, *Service Topology,* and *Software Dependencies* as shown in Figure 5. While up to *Stage 0* the integrity of the system is verified with the static root of trust measurement (SRTM), the Merkle tree represents the dynamic root of trust (DRTM), which in turn is linked to *Stage 0*.

5.2 Real-World Example

Figure 12 shows a real-world scenario, similar to how it is deployed in a banking environment. In this example a large business transaction needs to be authorized. The architecture has several entities:

- *Core Banking Infrastructure*: This is where all the business logic of a bank is computed.
- *HSM*: The Hardware Security Module that signs large transactions, interbank-clearance, smart contracts, etc.
- *Alice and Bob*: Bank employees that need to approve certain transactions.
- *TEE*: Trusted Execution Environment as physical device with similar properties against physical attacks as the HSM. Executes the TEE app and attests the security properties of the device.
- *TEE app*: The application that decides if it approves the business transaction based on the same information provided to Alice and Bob. The TEE app is developed by either the bank or a 3rd party, and is verified and deployed to the device by the Security Officer.
- *ExecuteRequest and ExecuteResponse*: Protobuf messages defined in the API [39].

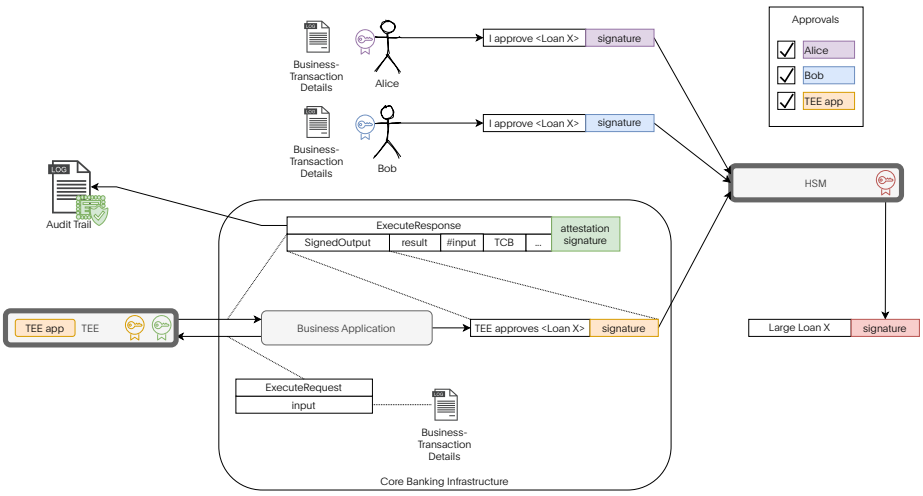

Figure 12: Multi-Authorization Use-Case

The HSM appliance on the right has a multi-authorization feature that signs a large business transaction, when three parties approve: Alice, Bob, and a TEE application. For this, a security officer has to onboard each of the approver's public key. On the left, there is an appliance shown that runs Gapfruit TEP with the TEE functionality. During provisioning, an *attestation key* and an *output signing key* is generated. Also, the TEE application is loaded. In this case, the TEE application is a compliance filter that goes through a series of

checks that make sure the business transaction satisfies the bank's interests and regulations. Details about the to-be-checked business transactions are part of the *input* of the *ExecuteRequest*. The *Business Application* within the *Core Banking Infrastructure* sends such an *ExecuteRequest* the TEE, which provides the input to the TEE application, which in turn executes the compliance checks over said input. If the checks pass, the TEE application writes the output which then gets signed with the *output signing key*. This signed output is then sent back as part of the *ExecuteResponse*. Exactly this signed output is then sent to the HSM as the final approval for the business transaction. Additionally, the *ExecuteResponse* is signed with the *attestation key* and then added to an audit database so it can be verified by internal or governmental regulatory agencies. The *ExecuteResponse* attests that a certain output was generated from a specific input, executed at a specific time with specific code.

6 Comparison with Other Platforms

The problems with monolithic operating systems stated in 2.1 have widely been acknowledged [1, 45, 46]. This section describes platforms that address some parts of the broad scope of what Gapfruit TEP solves fundamentally. They can be grouped in Hypervisors, Microkernel Platforms, and Trusted Execution Environments. Some of these solutions were developed because a trustworthy system such as Gapfruit TEP did not exist.

6.1 Hypervisors

Hypervisors manage virtual machines (VM). In some products such as Wind River Helix [47] these hypervisors support pinning specific CPU cores to certain VMs. This enables one to create products where one VM holds a static real-time operating system (RTOS) while the other VM runs a full-blown Linux distribution. The problem with this approach is that now the manufacturer has to maintain the hypervisor, the RTOS, and the Linux VM(s) individually, while the maintenance effort of each of these subsystems is substantial. Even though Gapfruit TEP supports virtualization as well, it is encouraged to run code in one of the many supported runtimes (Section 4.3) and benefit from the powerful yet lightweight package manager (Section 4.5).

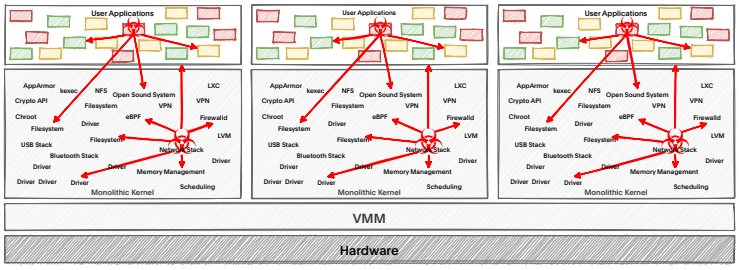

Figure 13: Separation with virtualization

6.2 Microkernel Platforms

Especially in safety-critical products, there are different embedded microkernel platforms on the market, such as QNX, Integrity, or VxWorks. Conceptionally, these products are not that different from each other as they all lack capability-based security and update-ability. So in this comparison, the focus is on QNX and CAmkES.

6.2.1 QNX

QNX [16] is a microkernel operating system *without* capability-based security. That means that the dependencies are not formally defined and enforced as it is done with Gapfruit TEP. The IPC policies are defined in the servers. There is no package management system, so updates are done by distributing the whole system image that includes applications, drivers, network-stacks, and the OS in one binary.

6.2.2 CAmkES

CAmkES [24] is a software development and runtime framework for building systems on the seL4 microkernel. It has a static service-oriented architecture with capability-based security. While it provides rudimentary virtualization support, it lacks any sort of package management system. The system is built and distributed as a single static boot image. In contrast to Gapfruit TEP, CAmkES-based systems are defined at design time and remain fixed at runtime.

6.3 Trusted Execution Environments

There are two popular Trusted Execution Environments currently in the market: ARM TrustZone and Intel SGX. They have been developed because these vendors acknowledged that current operating systems cannot be made secure and in their threat model are considered compromised. These solutions have been added to the CPU by hardware, microcode, or a combination of them. These technologies can be combined with Gapfruit TEP, if the threat model of a particular use-case calls for it. However, it must be noted that the features of both these technologies are either impossible or hard to update.

6.3.1 ARM TrustZone

ARM TrustZone [48] provides two areas on the same processor: The *secure world* and the *normal world*. The *secure world* is protected from the *normal world*, even from code running in the kernel. The *secure world* also has access to more parts of the SoC than code running in *normal world*. This means that code, that runs in the *secure world* is even more critical than code running in the kernel of the *normal world*. Hence, unlike Gapfruit TEP as described in section 4, ARM TrustZone fails to prevent applications from having unrestricted access to resources of the computing environment outside the trusted execution environment. Additionally, ARM TrustZone has no mechanism to attest that code running within the *secure world* was in fact executed on the particular hardware as TEEs on Gapfruit TEP described in section 5.1.

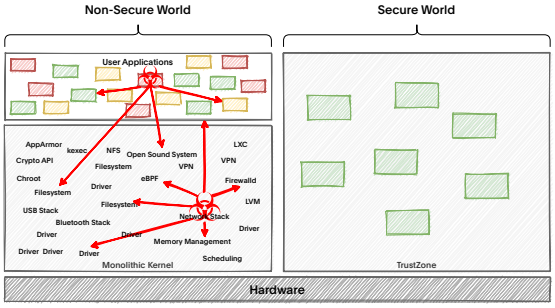

Figure 14: Separation with ARM TrustZone

6.3.2 Intel SGX

Intel SGX [49] is a TEE technology for Intel x86 CPUs that is entirely implemented in microcode, except for the memory encryption engine [46]. The threat model assumes that code running in kernel and in user mode is compromised. Intel claims that SGX reduces the TCB of the critical code substantially. However, this is not quite true, since the complexity is being pushed to where it is not fixable nor auditable.

While some concepts of Intel SGX are sound, the intuition behind SGX's memory access protections can be built by considering what it would take to implement the same protections in a trusted operating system or hypervisor, solely by using the page tables that direct the CPU's address translation feature [46], section 6.2.1. This is exactly what Gapfruit TEE SLICEs provide while developing software that runs within a Gapfruit TEE is easy - unlike developing software that runs within SGX.

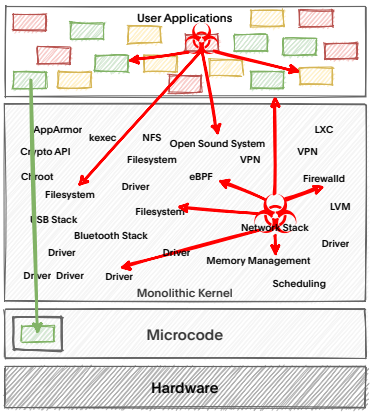

Figure 15: Separation with Intel SGX

7 Business Benefits

With Gapfruit, businesses that build products for mixed critical use-cases can create scalable and inherently secure solutions that provide high levels of return on investment. These businesses now have the ability to prove the trustworthiness through attestation, rather than just by claims. The secure platform architecture allows developers to focus on innovation and new functionality, rather than spending valuable resources on security issues. Different development teams can build and deploy only the code they really need without interfering with each other.

- Provable trustworthiness.
- Auditable.
- Secure out of the box.
- Reduced time-to-market.
- Fast certification.
- Risk-free maintenance of devices that are already in productive environments.
- Increased integrity by the secure update of parts of the system.
- Increased availability by the risk-free update of the system.
- Rapid deployments.
- Automated CVE Monitoring.

7.1 Business Benefits TEE

Gapfruit Trusted Execution Environments as described in section 5, offer one or a combination of the following benefits:

- Provide traceability for audits.
- Usage as a compliance filter for decision making.
- Computation of confidential data.
- Executing confidential code.

8 Conclusion

Rather than approaching today's security challenges effectively, enterprises apply all kinds of solutions to their infrastructure. If one technology is insufficient, keep adding more. This patchwork security does not only make us all less safe - it proves the fact that the most fundamental technology that we build our IT on is a big pile of sand: The operating system. As Biggs, Lee, and Heiser emphasize:

> "The conclusion is inevitable: From the security point of view, the monolithic OS design is flawed and a root cause of the majority of compromises. It is time for the world to move to an OS appropriate for 21st-century security requirements." [1]

Today's and tomorrow's computing challenges are addressed by Gapfruit TEP, a unique operating system designed for security. Its modern architecture eliminates typical attack vectors and minimizes the impact of software flaws. Gapfruit TEP enables mixed critical applications run side by side while guaranteeing strong separation of concerns. Gapfruit provides several low-risk migration paths enabling any unmodified application to run in strongly isolated environments.

Gapfruit's architecture is future-proof and allows reliable and rapid rollout of upgrades and new services. It is therefore suited for many different use-cases such as transportation, edge computing, secure endpoints, IoT gateways, medical devices, automotive, industrial and building automation systems.

Product manufacturers can differentiate themselves with the provable trustworthiness capabilities provided by Gapfruit TEP.

About Gapfruit AG

Gapfruit AG is a deep-tech company based in Switzerland with a proven track record in systems security, software engineering, and innovation. The founding team developed a military-grade operating system fulfilling the requirements set by national governments and security agencies across the world for ironclad security. With this expertise, Gapfruit provides the Gapfruit Trustworthy Execution Platform for today's and future challenges. The developers at Gapfruit have been contributing to the Genode Framework [12] for many years.

If you want to deliver trustworthy products yet focus on your core expertise, get in contact with us today.

info@gapfruit.com

References

[1] S. Biggs, D. Lee, G. Heiser, "Monolithic OS Design is Flawed"
http://ts.data61.csiro.au/publications/csiro_full_text/Biggs_LH_18.pdf

[2] Dr. Norman Feske, "Genode Foundations"
https://genode.org/documentation/genode-foundations-20-05.pdf

[3] www.cio.com, "Risk = Likelihood * Impact"
https://www.cio.com/article/3111304/risk-likelihood-x-impact.html

[4] Common Vulnerabilities and Exposures
https://cve.mitre.org/

[5] Teardown of a Failed Linux LTS Spectre Fix
https://grsecurity.net/teardown_of_a_failed_linux_lts_spectre_fix

[6] The Life of a Bad Security Fix
https://grsecurity.net/the_life_of_a_bad_security_fix

[7] Tightening security: not for the impatient
https://lwn.net/Articles/503660/

[8] 10 Years of Linux Security
https://grsecurity.net/10_years_of_linux_security.pdf

[9] Docker
https://www.docker.com/

[10] X. Wang, "Enhanced Security of Building Automation Systems"
http://www.arguslab.org/documents/WangCCNCPS17.pdf

[11] Core Security Principles
https://www.flashpoint-intel.com/blog/emerging-threats/black-hat-parisa-tabriz

[12] Genode Framework
https://www.genode.org

[13] SELinux
https://selinuxproject.org

[14] AppArmor
https://www.apparmor.net/

[15] seccomp
https://man7.org/linux/man-pages/man2/seccomp.2.html

[16] IPC on QNX
http://www.qnx.com/developers/docs/7.0.0

[17] VirtualBox
https://www.virtualbox.org/

[18] Seoul VMM
https://genodians.org/alex-ab/2019-05-09-seoul-vmm

[19] VMM for ARM
https://genodians.org/skalk/2020-04-09-arm-vmm

[20] BlueBorne
https://www.armis.com/blueborne

[21] P. Emmerich, M. Pudelko, S. Bauer, G. Carle, "User Space Network Drivers"
https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/ixy_paper_draft2.pdf

[22] Data Plane Development Kit
https://core.dpdk.org

[23] Snabb: a simple and fast packet networking toolkit
https://lwn.net/Articles/713918

[24] CAmkES (component architecture for microkernel-based embedded systems)
https://docs.sel4.systems/CAmkES

[25] ADA/SPARK
https://www.adacore.com

[26] JVM Languages
https://en.wikipedia.org/wiki/List_of_JVM_languages

[27] Python
https://www.python.org/

[28] Solo5
https://github.com/Solo5/solo5

[29] Wasm
https://webassembly.org/

[30] seL4
https://sel4.systems/

[31] NOVA Microhypervisor
http://hypervisor.org/

[32] Capability-based Security on Linux
https://genode.org/documentation/release-notes/20.05#Capability-based_security_using_seccomp_on_Linux

[33] YANG
https://tools.ietf.org/html/rfc7950

[34] NETCONF
https://tools.ietf.org/html/rfc6241

[35] RESTCONF
https://tools.ietf.org/html/rfc8040

[36] The Nix Package Manager
https://nixos.org/

[37] Securing Software Updates for Automobiles
https://ssl.engineering.nyu.edu/papers/kuppusamy_escar_16.pdf

[38] Kubernetes
https://kubernetes.io/

[39] Gapfruit TEE API
https://docs.gapfruit.com/api/tee_api.html

[40] gRPC, a high performance, open-source universal RPC framework
https://www.grpc.io/docs

[41] Protocol Buffers
https://developers.google.com/protocol-buffers/

[42] Trusted Platform Module
https://trustedcomputinggroup.org/resource/
tpm-library-specification/

[43] OpenTitan
https://opentitan.org/

[44] Merkle Tree
https://en.wikipedia.org/wiki/Merkle_tree

[45] Nizza Secure-System Architecture. Hermann Härtig
https://os.inf.tu-dresden.de/papers_ps/nizza.pdf

[46] Intel SGX Explained
https://eprint.iacr.org/2016/086.pdf

[47] Wind River Helix
https://www.windriver.com/products/operating-systems/
virtualization/

[48] ARM TrustZone
https://developer.arm.com/ip-products/security-ip/trustzone

[49] Intel SGX
https://software.intel.com/content/www/us/en/develop/topics/
software-guard-extensions.html