2019 NDIA GROUND VEHICLE SYSTEMS ENGINEERING AND TECHNOLOGY SYMPOSIUM
CYBER/VEA TECHNICAL SESSION
AUGUST 13-15, 2019 - NOVI, MICHIGAN

The seL4 Microkernel – A Robust, Resilient, and Open-Source Foundation for Ground Vehicle Electronics Architecture

Robbie VanVossen[1], Jesse Millwood[1], Chris Guikema[1], Leonard Elliott[2], Jarvis Roach[1]

[1]DornerWorks, Grand Rapids, MI
[2]GVSC, Warren, MI

ABSTRACT

There has been a lot of interest in the secure embedded L4 (seL4) microkernel in recent years as the basis of a cyber-security platform because it has been formally proven to be correct and free of common defects. However, while the seL4 microkernel has a formal proof of correctness, it does so at the cost of deferring functionality to the user space that most developers and system integrators would deem necessary for real life products and solutions, and use of formal proofs for user space can be prohibitively expensive. DornerWorks took an approach to bypass the need for native seL4 user space applications to develop a representative real-world system for GVSC VEA based on seL4 by enabling its virtual machine monitor functionality for ARMv8 platforms, allowing feature rich software stacks to be run in isolation guaranteed by the seL4 formal proofs. This paper describes that system and the efforts undertaken to achieve real world functionality.

Citation: R. VanVossen, J. Millwood, C. Guikema, L. Elliott, J. Roach, "The seL4 Microkernel – A Robust, Resilient, and Open-Source Foundation for Ground Vehicle Electronics Architecture", In *Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium* (GVSETS), NDIA, Novi, MI, Aug. 13-15, 2019.

1. INTRODUCTION

Safety and security concerns over computer systems within military ground vehicles bear many similarities to those that have been recently reported for commercial automotive systems, however there are several important distinctions. Military ground vehicles often host powerful weapons, are connected to larger military communication networks, and are responsible for critical functions during tactical operations. For these reasons, the electronics and computers within these vehicles need to function with a high-level of assurance. Historically, a lower level of rigor was observed during ground vehicle system development. This is due to many factors, including the lack of a need for the level of rigor associated with airworthiness certification and because, originally the electronics within these systems were standalone. The essence and severity of this deficiency was recently highlighted in a Government Accountability Office report titled “Weapon Systems Cybersecurity: DOD Just Beginning to Grapple with Scale of Vulnerabilities” [1] which presented an investigation revealing that major military systems including aircraft, ships, combat vehicles, and satellites could easily be compromised. Clearly more work needs to be done to protect military ground vehicle electronics. To accomplish this, these systems will need to meet more rigorous security certification standards.

“Software certification is expensive” is a truism in the ever-increasing number of industries that require it. All safety and security certification standards, such as Common Criteria for security, DO-178C for aviation, IEC 62304 for medical, IEC 61508 and its derivatives like ISO-26262 for automotive, share the same goal of providing a guarantee, or an assurance, that software developed to such a standard does what it is intended to do, and at higher criticality levels, does not do what it was not intended to do. To the uninitiated, this seemingly simple goal requires a surprising amount of effort above and beyond what is required to “just get the software working”. Defining and documenting what exactly is meant by “working”, in the form of defining a specification, is a common task or objective in software process standards. This is true, even when certifying to the least critical levels, and it takes time and effort to do correctly and unambiguously. Correct and complete requirements allow a team to develop tests and other means to verify that the software meets the specification. Even teams that are in the practice of testing their software may find the additional effort to write tests sufficient to exercise all of the requirements surprising. A natural consequence of better testing is a higher level of defects found in software and requirements. This tends to further drive development cost up, even if the project is disciplined enough and has the luxury of avoiding churn caused by a change in the requirements. These additional costs are borne even when projects attempt the easiest levels of certification, where the goal is only proving the software does what it is intended to do. The effort to provide assurance that the software does not do what it is not intended to do, necessary for the higher level of criticality, requires additional reviews and analyses that substantially add to the final bill.

Data shows that developing software to the lowest level criticality in the DO-178 standard, design assurance level (DAL-E), adds 0.11 to 0.44 hours, depending on the development team’s experience, per line of source code (SLOC) to generate the requirements, design documentation, source code, verification, analyses, and review evidence, collectively known as certification artifacts [2]. The most critical level, DAL-A, takes between 0.67 and 2.68 hours per SLOC. By these metrics, a microkernel on the order of 15,000 SLOC would require between 10,050 and 40,200 hours to develop to DAL-A. At the other end of the spectrum, the Linux kernel grew by 225,000 SLOC from 2017 to 2018 [3] to a grand total of around 25 million SLOC. Based on the metrics given above, which may be on the low side for such a large and complex code base, Linux would require 2.75 million to 11 million hours of development just to achieve DAL-E, and while Linux does not need to be entirely developed from scratch, testing alone can account for as much as 90% of the total labor effort [4] in development.

Formal methods, or mathematically based techniques, can be used for the specification, development, and verification of software to reduce these costs in the long run. The very act of capturing requirements using formal notations forces a more explicit, and hence more unambiguous, set of requirements, removing a source of errors very early in the software development lifecycle. Also, formally defined requirements can be used to carry out formal analyses to verify useful properties of all formally defined artifacts, such as consistency of and between requirements, design, and software;

freedom from deadlock or livelock; and satisfaction of high-level requirements and correctness of a proposed design [5]; properties which otherwise would have to be demonstrated through various tests and rigorous reviews of requirements, designs, and source code. Not only do formal methods prove correctness in a system, but they can also provide the means to detect all routes of execution, which is more thorough than traditional testing, As an additional benefit, formal notation enables these verification steps to be highly automated through the use of tools, drastically reducing the amount of time to re-verify and re-validate when changes have to be made.

The importance of formal methods in a certification context become apparent when one reads through the requirements of more stringent certification documents. In the context of international certification requirements, the Common Criteria comes to mind, where the requirement of formal methods is explicitly called out in the higher Evaluation of Assurance Level (EAL). The general approach to performing formal methods on a system is to represent the system in some abstract, high-level model where behavior can be reasoned about across subsystems. From there, a series of conversions are performed to lower level representations in order to prove that the behavior of the high-level model still holds. The Common Criteria specifically calls this out with the requirement of high-level models and low-level models. Once the conversion of models has approached the level of a low-level language representation, one has something deployable that one can be confident behaves exactly as modeled.

The issue here is that those steps from modeling and converting to lower and lower models is very time intensive and requires a lot of effort, including ramping up on new tools and ideas. It is especially burdensome with systems such as kernels that make up the base of one's system. There have been 20+ man years spent on modeling and formalizing the seL4 kernel from the high level model to binary to a point where they can prove that as long as one configures it properly and that the proof assumptions hold, no unintended behavior in the operation and API handling of the kernel and memory regions are isolated. This provides the basis for a high assurance system. It does not mean any system utilizing seL4 will automatically be security certified, because the behavior of applications are not proven and could introduce bugs, but at least the work needed only covers the user space and the API it uses. The Common Criteria outlines the need to modularize applications to make the proof easier to understand. Since seL4 is a microkernel it means that applications should be broken up for modularity and one can prove out subsystems easier this way. Once it is broken up, the only way the subsystems can communicate is through proven mechanisms provided by the kernel. Building up systems on the foundation that seL4 provides gives one an environment to create modularized secure applications that get one closer to achieving the requirements put forth by most high assurance certifications.

Incorporating formally verified components, such as the seL4 microkernel, is one way to harden military ground vehicle systems and bridge the gap to higher security certifications.

2. SEL4 BACKGROUND

2.1. *History*

The seL4 kernel was born out of a partnership with General Dynamics C4 Systems, National Communications Technology Australia Ltd (NICTA), and Open Kernel Labs (OKLabs, a spinoff of NICTA, later acquired by GD). NICTA later merged with Commonwealth Scientific and Industrial Research Organization (CSIRO), forming the group Data61 that operates within CSIRO and who continue to develop and maintain the seL4 source code and proofs.

seL4 builds on concepts derived from operating system research. Liedtke's L4 microkernel forms the basis of a family of microkernels that are composed of several foundational mechanisms. L4 microkernels generally are responsible for controlling address spaces, threads, scheduling, and inter-process communication (IPC). Most L4 microkernels are only responsible for the bare minimum to provide isolated process execution, communication, and scheduling; pushing the higher-level responsibilities such as user or file management and drivers to the user space to run as “servers”. Most L4 kernels do not share common APIs but they do share common concepts and principles, and for this reason are considered part of the same family of microkernels.

seL4 was written from scratch, adhering to the foundations of the L4 microkernel but focusing on using formal methods from the beginning. The seL4 microkernel began its life as an abstract executable model in Haskell. The formal proof of correctness was completed in 2009. These proofs showed that the implementation was correct against the Haskell specification and that the kernel was free of dead-locks, live-locks, buffer overflows, arithmetic exceptions, and uninitialized variable usage [6]. Throughout this paper, seL4 is said to be formally proven to be correct. This means that the implementation is accompanied by a set of mathematical proofs that show that it is consistent with the chain of specifications and refinement proofs leading to an implementation that is free from programmer-induced implementation defects [7]. In 2014 NICTA and General Dynamics announced the open-sourcing of seL4 and the accompanying proofs. As of the publication of this paper, seL4 is still the only general-purpose kernel with a full functional correctness proof [8]. This means that the compiled binary implements the behaviors specified in the high-level specifications and nothing more.

2.2. *Proof and Development of the seL4 Kernel*

Development of seL4 generally happens in two phases: formal methods and manual, high-performance C development. From the Haskell prototype/specification a number of intermediate representations, or layers, are produced where the final product is a binary and the functionality can be formally traced back to the original specification. This is not necessarily a linear process, Figure 1 shows a slightly simplified seL4 development and proof process as further explained in [7]. The key focus of the suite of specifications and proofs is to show that the seL4 kernel is functionally correct and free from programmer induced errors. It also is extended to prove additional properties such as functionally correct IPC, integrity and authority confinement, strong isolation, and a properly initialized user space that is consistent with the previously mentioned properties [7]. The strong isolation proof is handled by the information flow specification and related refinement proofs. Integrity and authority confinement are proven with the correct access control enforcement proof, which is based on capability access control. These access control properties form the basis for seL4’s strong security properties claim.

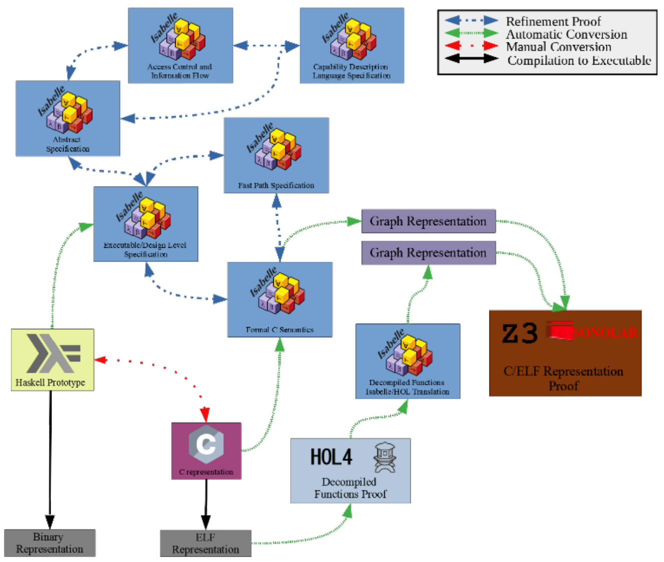

Figure 1: Development and Proof Process for seL4

The Haskell implementation can actually be compiled to a binary that can run on hardware, however the performance impacts caused by the Haskell garbage collected runtime and the presence of non-optimized code do not meet the high-performance standards required by embedded systems.

The Haskell implementation of the kernel is manually converted to a subset of C for which formal semantics exists. It is then hand optimized and automatically translated into an Isabelle/HOL representation as an Executable/Design specification. The design specification is responsible for defining how the kernel carries out behaviors that are defined in the abstract specification. This design specification serves as the basis for many of the other proofs that abstract from it and prove other aspects of seL4. Between the different Isabelle/HOL specifications, refinement proofs are used in order to prove that there are no other behaviors injected into the specifications that are not traced to related specifications. This forms a chain of formal specifications that explain different aspects of the system in a formal, mathematical, language. The ELF and C representations are proven to implement the same behaviors through the use of several more tools in addition to Isabelle/HOL. The optimized C representation is converted to an Isabelle/HOL representation in order to encode the C semantics used. A refinement proof shows that this formal representation of the C source code matches the design specification defined earlier. Isabelle/HOL is then used to convert this representation into an intermediate graph representation that other tools can reason about. The ELF binary that is produced from normal compilation of the C source is decompiled with the HOL4 tools and converted to Isabelle/HOL representations so that it can also be represented in an intermediate graph representation. The two graph representations are then proven to implement the same behaviors with the use of two SMT solving tools, Z3 and SONOLAR. It can be seen that through refinement proofs and automatic translations, the C source compiles to an executable ELF binary that is proven to behave according to the formal specifications written in the Isabelle/HOL language.

This graph of development shows how features and changes cannot just be introduced to the C implementation as the proofs and other source code directly rely on each other.

2.3. *Capabilities*

Capability access control forms the basis for the isolation guarantees provided by seL4. The concept of capabilities has been around and used by various projects (KeyKOS, PonyLang, TahoeFS, etc) since their introduction by Dennis and Van Horn [9] in 1966. Controlling resource access via capabilities differs from the well-known access control lists (ACL) mainly in how access is granted. Capabilities can be thought of as a controlled token. This token is controlled by a supervisor process, such as a kernel, and is passed along via system calls to prove that the caller has the correct permissions to do what they intend. This form of access control is much more fine-grained than access control lists, where access is granted to programs based on the group, or list, to which they belong. Another difference is that ACL systems use forgeable references, such as file names, where access is generally tied to the caller, in contrast to capability systems which use the capability objects to also refer to the resource, thus combining designation and authority.

It is important to not conflate capability access control with POSIX capabilities as they are different. POSIX capabilities are not associated with any object, where capabilities in the context of access control systems are.

In seL4, capabilities are managed by the kernel and are un-forgeable. The structure that is used for book keeping capabilities is a directed graph structure called a Cspace. A Cspace is made of Cnodes, which are each a table of Cslots. When a thread is created, a premade Cspace is passed to the thread initialization, giving that thread access to only the resources it is assigned. The only way for that thread to gain any more access is if another thread transfers a capability to it via inter-process communication (IPC). This transfer is also handled by the kernel.

In a capability-based system, there are capabilities for almost every action on every resource, and in seL4's case, even CPU time. Generating all of these capabilities and assigning them can be a very cumbersome and error prone task for a developer. Data61 has developed the CapDL (Capability Description Language) [10] for describing capabilities in the development phase so that the initial bootstrap environment of a static system can assign the proper capabilities to the proper threads. This tool is integrated into the Component architecture for microkernel-based embedded systems (CAmkES), aiding in the architecture development and compilation of non-trivial seL4 based applications and systems.

3. CURRENT & UPCOMING SEL4 EFFORTS

3.1. *Department of Defense Efforts*

Department of Defense (DoD) system developers recognize the important security benefits of deploying formally verified software and also that using open-source software in an appropriate manner can decrease lifecycle costs, enable reuse within DoD agencies, and support safety and security certifications since source code is readily available for analysis. seL4 is a formally verified, open-source microkernel and DoD agencies are extending and applying it in a variety of ways.

The Defense Advanced Research Projects Agency (DARPA) has provided foundational support in developing awareness of issues in the cyber-physical system problem space and the role that seL4 plays in creating a trusted computing base for military applications. DARPA currently spearheads a number of initiatives including the creation of the seL4 Center of Excellence (CoE) which aims to enrich the seL4 ecosystem by expanding the available features and capabilities available for seL4-based systems while also maintaining vetted configurations which are suitable for use in government projects. DARPA also provided support for development of a GDB debugger for seL4 and is currently supporting the development of a formally verified Internet Protocol (IP) network stack which is important due to the growing prevalence of Ethernet communication in embedded systems.

The U.S. Air Force Research Lab (AFRL) has had a role in supporting the seL4 CoE and explored the use of seL4 to harden Air Force weapon systems through their Agile and Resilient Embedded Systems (ARES) program [11]. The ARES program uses the seL4 microkernel as the foundation for a layered software environment that:

- Fully isolates and controls applications
- Protects and monitors processes, memory, interfaces, and communication
- Provides formally verifiable access control
- Secures communication via dynamic encryption

To date, the ARES program has been able to deliver utility infrastructure modules such as common library and driver development, secure inter-process communication, and process/memory introspection that has provided them with a significant increase in the ability to detect, isolate, and recover from high priority cyber effects. Recent developments in the ARES program include the integration of industrial flight management and control system on unmanned system flight module and this is scheduled to undergo flight and cyber assessment presently. The Air Force has also recognized that a typical trusted computing base for military systems consists of both verified (e.g. seL4) and unverified software (e.g. Linux). The use of unverified software in the trusted computing base significantly degrades assurance. AFRL is currently investigating formally verifiable information flow control mechanisms which can be used to reduce the size of the trusted computing base and help extend these mechanisms to secure user space software components.

The U.S. Army has sought to leverage high-assurance software to harden weapon systems and enable new approaches for integration of mixed-criticality systems in military ground vehicles. Combat Capabilities Development Command – Ground Vehicle System Center (CCDC-GVSC) has supported efforts to create high-assurance prototypes and integrate them into vehicle networks to increase the robustness of vehicular systems [12]. CCDC-GVSC has worked with DornerWorks to create prototype systems and at the same time contributed important features back into the seL4 ecosystem. Features that have been completed include [13]:

- Support for multiple virtual machines (VM)
- Porting of seL4 to NXP i.MX8 hardware
- Porting of seL4 to Xilinx ZCU102
- ARM GIC500 driver

The appropriate portions of these seL4 features have been cleared for public release and pushed back to the seL4 open-source repository so that they can be tested, maintained, and formally verified (a specialized process that the U.S. Army has relatively little organic capability to perform). Current efforts aim to further extend seL4 to support Intel Xeon platform, multicore, and 64-bit guests. Future work CCDC-GVSC is evaluating includes the use of seL4 to support the Robotic Operating System – Military (ROS-M) for use in semi-autonomous and autonomous systems, an area which the U.S. Army is giving high-priority as part of the Optionally Manned Fighting Vehicle (OMFV) program. CCDC-GVSC is also considering developing common seL4 features/capabilities separately from the specific programs and applications they will be used so that those common features can be more easily maintained, reused within the DoD, and open-sourced, if appropriate. Examples of common seL4 features and capabilities include:

- Cache coloring
- Message filtering frameworks
- Improved middleware support
- Improvement in configuration and debugging tools

Further Army efforts include the topic A18-133: “Open Source High Assurance System” [14]. The Phase I of this Small Business Innovation Research (SBIR) aims to gauge the feasibility of using open software and open hardware for high assurance aviation systems. The effort is motivated by the fact that proprietary hardware and/or software is often closed source and hide how each piece functions, which can make security certifications and assessments next to impossible. A full system stack that is open will alleviate this by being completely transparent. The Phase II will apply the findings in the Phase I to develop a secure, open prototype system. While the topic does not explicitly state that seL4 must be used, it is referenced and used as an example of a secure, open-source software solution. DornerWorks has received an award for this topic that will focus on improving the support for seL4 running on RISC-V platforms, by porting tools, such as CAmkES and writing example applications.

3.2. *Commercial Efforts*

In addition to the various efforts with Department of Defense agencies, seL4 has great potential to be, or is already being, used in a variety of commercial applications. With its provable guarantees regarding Worst Case Execution Time (WCET) and correct operation, seL4 is well suited for products requiring guaranteed real-time performance, guaranteed correct operation, security, or any combination of the three. The fact that it is open-source also makes it an ideal candidate to use in academic projects or as a low-cost enabler for start-up companies.

Using seL4 in commercial autonomous vehicle software seems like a natural progression from DoD autonomous vehicles, as Boeing is potentially doing for unmanned air vehicles (UAV) [16]. An aerospace product where seL4 has been used recently was in the QB50 CubeSats to conduct atmospheric research by the Universities of Sydney and Adelaide [17]. seL4 could easily find its way in other real-time markets with safety concerns, such as medical, or markets with high security, such as finance or voting products. seL4 especially will shine where those two markets intersect, for example in nuclear power plants. Consumer products may prove to be a viable market for seL4 as well. Considering that a customized version of L4 is used to run the Secure Enclave [18] providing security to Apple's mobile device operating system, iOS, it would not be too far of a stretch to consider using seL4 instead for formally proven correctness. Similarly, in the Android mobile device market space where ARM processors are heavily utilized, seL4 would make a great choice to run as the Trusted Execution Environment in the ARM TrustZone [19] to provide secure and trusted operations.

4. CHALLENGES TOWARDS UTILIZING SEL4

While seL4 provides the foundation to construct an incredibly secure and robust system, it was not designed around pre-existing APIs which makes application development and porting more difficult. There is less common code that can be utilized, forcing application developers to either implement translation interfaces, rewrite core components, or rely on virtualization. Each comes with its own trade-offs. Application development can ideally build off of the following middle layers on top of seL4:

- Virtual Machine
    - Have access to many drivers
    - No concept of capabilities within VM
    - Can only count on separation of VMs
- seL4 application
    - Very few drivers available
    - Can use capabilities but at the cost of manual setup
    - Separation applies to threads now
- CAmkES
    - Very few drivers available
    - Can use capabilities which are setup by the initial environment
    - Separation applies to components

Software middle layers shrink the amount of code required to get a system up and running but requires reliance on software stacks of varying complexity and provenance. These software middle layers can be used to encapsulate some complex code bases that do not have direct ports to run on seL4 based systems such as drivers.

4.1. *Lack of Drivers*

Drivers are some of the most important pieces of software that most developers do not need to think about. Most embedded Linux developers can take advantage of the fact that the drivers are present and just work. seL4's unique system API and microkernel architecture make it harder for drivers from one operating system to work as drop in replacements for an seL4 based system.

Developers are not completely out of luck though. The solution with the most drivers already, as mentioned earlier, is to use virtual machines. This comes with the cost of the size and resource overhead required to run and manage a virtual machine. It should be noted that if an application and driver are running in a virtual machine that is managed by seL4, any bugs or memory access violations within the VM will not be caught by seL4 since the concepts of capabilities and virtual memory separation do not extend into the virtual machines. However, due to the separation properties provided by seL4, the bug will only affect the VM in which it is found.

Another option exists, where NetBSD rump kernels are used to isolate just enough UNIX resources to run the required driver [20]. This rump kernel and driver is then isolated inside of an seL4 thread. Interaction with this driver now only requires the developer to develop an interface that works with seL4 semantics. Again, seL4 mechanisms cannot do anything about violations that happen within this rump kernel but in this case the driver and application are separated. These rump kernel drivers may not offer the full environment that one gets in a full virtual machine, but that may not always be necessary.

Other middle layer projects exist with various levels of licensing, usability, and completeness such as:

- Robogalia [21]
    - Capability OS with a Rust user space on top of seL4
- Genode [22]
    - Operating system framework which offers seL4 as a kernel option, among others
- UXRT [23]
    - QNX-like and Linux-compatible microkernel-based OS

If the developer wishes to still stay close to the metal and the interface of seL4 they can embark on developing drivers from scratch or porting drivers from other microkernel or RTOS projects to run in one or more seL4 threads. In this scenario they must determine how much of the architecture of the driver they are willing to rewrite and how far they would like to integrate seL4 conventions into the existing code base. This would make more components of the driver separate and increase the security of the system but the developer also loses more and more reliability from chunks of code that are tried and true.

4.2. *Unverified User Space*

While the foundation of an seL4 system provides proven separation and communication mechanisms it does not make the system un-hackable or impenetrable. By design, the seL4 kernel, by itself, cannot do very much. It was always intended to be a minimal kernel and rely on user space servers and drivers to interact with the outside world, while other user space applications orchestrate how the system achieves its tasks. The kernel was specified to work in a particular way and these specifications end at the boundary between user space and kernel space. The proofs applied to the seL4 kernel do not prevent a poorly implemented user space application from crashing, it is up to the developer to apply rigor and best practices to their user space application. In a similar vein, it is up to the system architect to make use of the separation and microkernel architecture to provide mechanisms such as monitoring services that can step in and kill/start misbehaving applications in user space.

Even though the seL4 proofs do not extend to user space code, user space application developers still have some options to ensure their systems meet their target level of assurance. seL4 applications are commonly written in C, and therefore developers have many common source code analysis tools and techniques at their disposal. Since each application or server is compiled into separate object code files, common analysis methods can be applied in a focused manner to each application. This can be very useful in applying more rigor to critical pieces of the system while other low priority components are kept separate.

Also, some methods exist that automate code generation of lower level layers for application code at the kernel/user space interface that can make these things easier. CAmkES is one tool from Data61 that brings with it a partially verified features such as capability generation through the CapDL language.

Additionally, one could explore the following to shrink the area of code that must be rigorously developed by hand:

- Verified libraries
    - Are not freely available but could be used to add to the stability of user space applications and enable reuse of verified code
- Cogent

  - This tool provides a high-level language for developers to write applications or drivers that supports generation of both C-code and Isabelle specifications in parallel

If developers would like to build off of the high assurance that is offered by the seL4 kernel they can extend some of that assurance into user space. Developers can start with an informal argument, based on a formal property expressed in the seL4 kernel proofs. They can use methods to show how their code implements the property of interest. This would show how the user space application meets the same security property as the kernel, extending that property to that user space application. There is not a well-known method for extending such properties to the user space in this way and is very dependent on the particular application.

It is in the developers' best interest to examine and plan out where high assurance is required and where less assurance is acceptable. Applying modeling and analysis tools to user space code running on top of seL4 makes for a robust platform.

4.3. *Licensing*[*1]

When seL4 was open-sourced it was done so with the GPLv2 license attached to the kernel code as well as the proofs. In many commercial and defense circles, the mention of GPLv2 marks the end of any evaluations. In the case of seL4 though this does not need to be the case. The points of the GPLv2 that many run away from are generally attributed to the following requirements [24]:

- The distributor of the binary must make the source code for that binary available as well.
- Additional restrictions may not be placed on the code.
- You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the program or any part thereof, to be licensed as a whole at no charge to all third parties.

Fortunately, seL4 is deployed much the same way as an embedded Linux system would be, and thus it is generally accepted to think of that as precedence. In this way seL4 is compiled into a separate binary. This binary is not linked nor does it make calls to anything in the user space of the system. Data is only shared between user space and kernel space via shared memory regions and control is transferred only by execution level hardware system changes via interaction with the hardware registers and system calls. This means that a more permissively licensed user space is not contained within the GPLv2 kernel nor does it share data structures or function calls. Just as in the case of an embedded Linux system, the kernel and user space can be considered separate pieces of software. A distributed seL4 based system generally is compiled of the following binaries:

- elfloader (GPLv2)
- seL4 kernel (GPLv2)
- Device Tree
- user space application

Data61 does provide many libraries for bootstrapping and managing a non-trivial seL4 compatible user space under the BSD license. However, if the developer includes any further software that is licensed as GPLv2 into the user space, it is up to them to ensure proper compliance.

1 The suggestions in this section are not made with counsel from any legal professionals. Any decisions based on information in the licensing section of this paper should be verified with the proper legal counsel. The authors and publisher do not take any responsibility for decisions to use or not use seL4 based on the licensing discussion in this paper.

Developers that do incorporate the seL4 kernel into their system must be aware that any changes to the kernel must be made obvious and available to anyone with a copy of the binary upon request. This is one reason where it helps to contribute these changes back to the original repository, so that the distributor that needed such a change is not on the hook for the future to make the modified source available.

One possible concern may be that changes to the kernel and announcement of them would signal to competitors or enemies' information about underlying hardware or architectural changes. To get around this issue, the seL4 kernel supports device tree binaries so these details can be kept as secret and separate binaries that do not depend on nor directly share data structures with the GPLv2 licensed kernel. These details generally include information about the system, which devices are available, and their location and sizes. Then seL4 can read that information at runtime to know which drivers and platform code to use.

4.4. *New Features Needed in Kernel*

As indicated in section 2.2, the proven aspects of the seL4 kernel rely on many separate moving parts. When one does come across the need for a feature in an seL4 based system, great care should be taken to try and implement it in user space first because altering the kernel will break the chain of proofs that ensure seL4 is a stable and secure kernel. As of this writing, Data61 uses three categories of verification compliance that classify different known hardware platforms using seL4. At the time of this writing, there is only one "Verified" platform. Two platforms are deemed functionally correct. It is important to note that the categorization of these platforms also rely on specific hardware configurations. Some hardware features, such as DMA, are not covered by the proof under any configuration. As new hardware platforms emerge, it is important to realize that the proof always lags behind the C implementation and available hardware features. This is why it is important to contribute to the shared base as well as funding training to contribute to moving the proofs forward.

Just because the hardware configuration that one may intend to use is not verified yet does not mean that seL4 is useless. The kernel still comes from a pedigree of high assurance and robustness. However, for any changes that are made, it may be important to exercise run-time test methods on those areas of the kernel or interact with unverified features of a hardware platform.

All of this is not to say that adding or modifying features of the seL4 kernel should never be done as sometimes there are very compelling reasons to modify the kernel. Some changes that DornerWorks needed to contribute were:

- Hardware Resource Management Features
  - ARM GIC 500
  - ARM SMMU
  - Hardware Virtualization
    - Virtual CPU handling
    - Memory Translation Tables
- Altering memory mapping algorithms that were not verified in the first place to allow mapping in smaller memory regions

It may look out of place that some drivers were added to the microkernel itself, however some devices must be handled by the seL4 microkernel since they have so much control over the entire system.

These additions were made to the C code in communication with the Data61 team to ease their eventual inclusion into the upstream code base. These changes have not been incorporated into the proofs at the time this paper was written. However, following the code conventions and working with the maintainers ensures a higher probability that the code is correct and will work well within the seL4 kernel.

4.5. *DOD funding and Open-Source SW*

One interesting issue is trying to effectively use any open-source software for government funded projects. Open-source software is very appealing to the government for the same reasons that it is appealing to everyone else. First, it is generally free to use as it does not come with expensive license costs that similar proprietary solutions do. Second, since it is openly available; a company can get started with it right away with no lead time. Third, if it is actively being utilized by a community, there is generally a fair amount of bug checking, maintenance, and feature development upon which everyone can capitalize. Fourth, obsolescence can be reduced by open source software. If it is being utilized by a large user base, out in the open, then it is less likely for the project to lose all of its maintainers.

However, there are some difficulties that arise in using open-source software. Since the user is not paying for a license, there is not an obvious support option. This generally results in more labor for integration and bug fixes or finding a company that does provide that support a la the Red Hat Linux model. Another complication is with open-source licensing. Some open-source licenses are very permissive and allow one to do most anything with the software, but others are more restrictive, such as with GPLv2 as described above. Figuring out what one can and cannot do with a particular license will result in more overhead costs or lawyer fees. Development teams should make sure to take extra care to track and analyze dependence on code of varying licenses.

As mentioned above, one of the main benefits of open-source software is providing one’s changes back to the community so that a larger group of developers and users can utilize it, test it, bug fix it, and help maintain it. However, this can be very difficult with government funded projects. There are various possible restrictions for any given project, including International Traffic in Arms Regulations (ITAR), export control, Controlled Unclassified Information (CUI), and classifications like Secret and Top Secret. Each one of these has different restrictions on who can see the code, how they can see it, and the steps one needs to go through to be able to release it back to the community, if even possible. This results in a process of figuring out which pieces are general enough to be useful for public release, if they can be released, and then going through the release process for that section of the government. This can be a fairly lengthy process that impedes the ability to mainline the release since the code is usually on a newer version once the release is approved. It is understandable that this process needs to exist so that the DoD can make sure that only the right pieces are being released and that anything that should be classified or secret remains that way, however the process needs to be streamlined so that useful code can be integrated with less effort and more quickly.

5. VEA SEL4 USE-CASE

The Vehicles Electronics and Architectures (VEA) group at the CCDC-GVSC was investigating new platform enablers that could provide strong isolation guarantees to meet more rigorous certification processes. They had the following requirements:

1. The platform software shall run on a modern 64-bit ARM platform.
2. The platform software shall provide robust space and time partitioning.
3. The platform software shall allow legacy software applications to run.
4. The platform software shall be able to meet a security certification process, such as Common Criteria (CC).
5. The platform software shall be able to run in a multicore configuration.

The fourth requirement is usually the most expensive one to meet, so the best option was one that could significantly reduce that effort without just buying a costly certification package.

DornerWorks and GVSC identified seL4 as the best solution for this use-case. The security certification effort is made much easier by the formal proof of seL4, since it means that the security properties of integrity and confidentiality can be relied upon when architecting the rest of the system. These guarantees can only be found in a proven system and meet the needs of robust space and time partitioning.

seL4 has the ability to run as a Virtual Machine Monitor (VMM) to allow guest operating systems to run and be strongly isolated from one another. This means that previously utilized Operating Systems, such as Linux, can run as a guest which can then run legacy applications. seL4 already supported a few 64-bit ARM (ARMv8) platforms, however, the codebase did not support VMM mode and ARMv8 together. GVSC contracted DornerWorks to update seL4 to meet all of the listed requirements and develop a representative demo of the functioning system.

5.1. *ARMv8 Virtualization*

DornerWorks identified the Zynq UltraScale+ MPSoC (ZUS+) from Xilinx as an excellent development platform for ARMv8 Virtualization. During a previous SBIR from US DARPA, DornerWorks ported seL4 to run on the ZUS+ and open-sourced the work back to the seL4 community. Furthermore, the ZUS+ runs quad-core Cortex-A53 processors that support virtualization extensions. Finally, DornerWorks had experience with Xen, another open-source hypervisor, running on the ZUS+. All of these factors lead to DornerWorks choosing the ZUS+ as the development platform for ARMv8 virtualization with seL4.

DornerWorks needed to modify seL4 in order to meet the GVSC requirements. The largest modification was updating seL4 to support virtual machines on 64-bit ARMv8 platforms. seL4 takes advantage of ARM virtualization hardware support to run different pieces of software at different exception levels (EL) as shown in Figure 2. EL0 runs user-level applications. EL1 runs privileged applications and is suitable for running Operation Systems like Linux. EL2 provides extensions for software to control multiple Operating Systems [25]. seL4 needed to run at EL2 in order to control the VMM and guest Operating Systems. Previously, seL4 would run at EL1 controlling software at EL0. The best solution was running seL4 at EL2 controlling generic user threads at EL0 and a special VMM thread at EL1. This VMM thread would manage VMs on behalf of seL4.

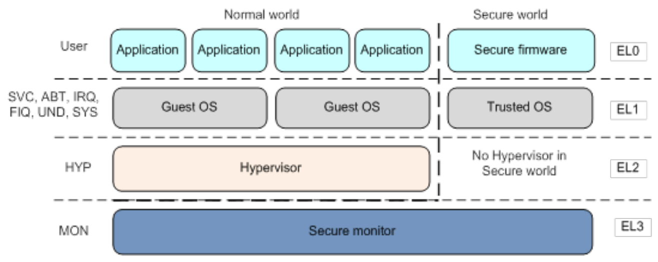

Figure 2: ARMv8 Exception Levels

seL4 provides a tool called the elfloader to do a few things. First, the elfloader needs to load the kernel image and initial user space image (called the rootserver) into memory. Second, the elfloader needs to jump into seL4. One of seL4's requirements is starting execution with the Memory Management Unit (MMU) enabled. This allows seL4 to manage hardware resources and isolate its memory from user space. The ARMv8 MMU uses a different paging scheme for EL2 vs EL1. In order to start running seL4 at EL2, DornerWorks had to modify the elfloader to map the kernel and rootserver images in this single address space, and initialize the required hardware registers such that the elfloader would jump into the kernel on an exception return instruction.

Running seL4 at EL2 required numerous kernel changes, both for the kernel and user space to run properly. seL4 provides a test-suite called sel4test which tests a large portion of the functionality of seL4, including system calls, error handling, and feature testing. DornerWorks determined that running sel4test with seL4 running at EL2 would be the best first step towards determining whether the kernel changes broke any features that seL4 provided. Some kernel changes were quite simple. seL4 needs to trap specific exceptions and handle them on behalf of user space. This allows the user space application to call into the kernel for IPC or other events. Therefore, specific EL2 handlers were added for these events. However, the largest seL4 change was the virtual address space (Vspace). Hypervisors, running at EL2, provide specific translations for Physical Address (PA) to Intermediate Physical Address (IPA), known as stage 2 translations. Operating Systems, running at EL1, provide Physical Address to Virtual Address (VA), known as stage 1 translations. When seL4 runs at EL1, it provides stage 1 translations for user space. For an ARMv8 platform, this requires four kernel objects:

- Page Global Directory (PGD)
- Page Upper Directory (PUD)
- Page Directory (PD)
- Page Table (PT)

The PUD is capable of mapping a 1GB page, the PD is capable of mapping a 2MB page, and the PT maps a 4KB page [26]. The MMU uses the PGD as the root of the translation. This allows the MMU to translate virtual addresses to physical addresses. However, when stage 1 translations are disabled, the MMU only uses stage 2 translations. ARM implementations vary on how the MMU translates stage 2 translations. On the ZUS+, the MMU begins at Level 1. Therefore, the PGD object is unused, and the PUD object becomes the root of the virtual address space. In order to accommodate both stage 2 translations and the differences in Vspace structure, significant kernel changes we made. The result of the kernel changes was a fully functional sel4test application with seL4 running at EL2 and the beginning of work on the VMM application.

Data61 provides a CAmkES based VMM for ARM applications. DornerWorks originally planned to provide ZUS+ support for this system; however, during the time of this effort, CAmkES did not support ARMv8. Luckily, DornerWorks found a non-CAmkES VMM application that was developed as part of the HACMS project. We first updated the application to the latest seL4 release, and then provided ARMv8 and ZUS+ support for loading Linux images. First, the VMM creates some objects for each new VM, including a new thread in which the VM will run, a Vspace, and a Virtual CPU (VCPU). The VMM then loads the VM image and device tree into the VM's Vspace, before configuring the VM's registers to run at EL1 at the entry point of the VM image. When a thread has a VCPU associated with it, seL4 gives the thread access to specific EL1 registers. seL4 saves the state of these registers to the VCPU object when the VM exits (either to a fault, or to seL4 scheduling a new thread). Right before that same VM runs again, the context gets restored from the VCPU object. Most operating systems expect to start execution in physical memory. The stage 2 mappings make the VM think it has access to the physical addresses compiled into the image. Therefore, the VM can create its own stage 1 mappings and it has no reason to think it is executing under a hypervisor. This allows the VMM to configure and manage multiple VMs.

The two main ways to provide devices to VMs in a virtualized environment are virtualized devices and pass-through devices. To pass-through a device to a VM, the VMM gives it direct access to that device. If that device is using interrupts, then only a single VM can effectively use it with pass-through. Most IO devices use interrupts, so when using pass-through, VMs are limited to the number of actual hardware devices that are available. The other method virtualizes the device by tricking the VM into thinking it has access to hardware resources by using trap-and-emulate methods. For these devices, the VMM does not configure stage 2 mappings, so that when the VM attempts to access that memory, it causes a fault into the VMM. The VMM then determines if the VM is allowed to read or write to that device. If so, the VMM will interface with the actual device on behalf of the VM. Both methods were employed on this project. Device virtualization was used to share a single serial console between multiple VMs, and virtualize the Generic Interrupt Controller (GIC), of which seL4 needs sole access. Device pass-through was used to give a VM direct access to an ethernet device.

Device pass-through works mostly as one would expect; register memory is mapped in to the VM address space and the device IRQs are injected into the VM. However, devices that use direct memory access (DMA) present a problem. As described earlier, VMs only have access to an intermediate physical address so when they provide buffers to a DMA master, the only address they can provide is not actually a physical address. This results in errors or unexpected behavior when a DMA event occurs. The best way to handle this is to use the ARM System Memory Management Unit (SMMU) to intercept these DMA transactions on the AXI bus and translate them to the correct physical address. DornerWorks developed an SMMU driver in the seL4 microkernel itself to manage the SMMU. It uses the same mappings from the MMU along with stream-ids to make sure that access to specific memory ranges are only allowed for specific VMs.

DornerWorks implemented a multiple VM system utilizing the methods described above. This system uses the isolation properties of seL4 to guarantee that VM's cannot interfere with one another and are fully isolated without any shared resources. DornerWorks would like to thank Data61 for providing some work-in-progress code for the VCPU handling as a basis for some of our work.

5.2. *Virtual Channels*

Since DornerWorks implemented a multiple VM system without shared resources, communication between VMs becomes impossible without using physical IO. To provide a means of inter-VM communication, a communication framework, called virtual channels (vchans) was devised. DornerWorks implemented the following pieces, which allow VMs to send messages securely through static, unidirectional channels:

- Linux kernel module
- VMM event handler
- Communication Server

The Linux kernel module is responsible for parsing through the device tree for vchan devices. The device tree entry for vchans contains a region of memory, a label, and a direction (receive or transmit). The kernel module combines the information to make a user accessible device that can either transmit or receive data. When the user writes to a transmit capable device, the message has a checksum calculated before being copied into the memory region and the kernel module performs a hypervisor call, which is handled by the VMM.

The VMM is responsible for two pieces of the vchan system. First, during initialization, the VMM determines how many complete virtual channels exist between separate VMs and creates a communication server that handles events specific to that channel. Second, the VMM needs to handle the Linux kernel module's hypervisor call. The VMM determines which VM made the call on which channel. The VMM then utilizes the seL4 IPC methodology to signal the communication server that an event has occurred on its channel. When initializing the communication server, the VMM shares a single read page, and a read/write page from each VM with the appropriate access rights, such that the communication server cannot write to the read-only page. This ensures a controlled, unidirectional data flow which prevents information leaks.

The final piece of the vchan system is the communication server, which DornerWorks implemented as a new seL4 thread with its own capabilities and address space. The communication server shares a read page with the transmitting VM, such that it can read the message, and a read/write page with the receiving VM. The communication server reads the transmitted message, calculates a simple checksum, and stores the message until the receiving VM asks if there is a message available. When the communication server has stored the message, it responds to the VMM with the length and checksum of the message, which are returned to the VM, which checks the received values against its original calculation.

As part of GVSC requirements, the vchan system needed a controlled data flow between mixed-criticality VMs. The static system configuration ensures that messages cannot pass except between expected channels, but there needed to be a way to ensure that a compromised VM would not leak specific data. As an additional safety feature, DornerWorks implemented a basic filter example in the communication server. If the channel has an associated filter, the communication server will pass the message through the filter before storing it. If the filter passes, the message is stored, otherwise the communication server simply discards the message while returning a valid length and checksum to the VM, such that the VM cannot determine through repeated messages the filters that the VMM has applied.

5.3. *VEA seL4 Results*

DornerWorks was able to leverage seL4 to meet the needs and requirements of GVSC. Both DornerWorks and GVSC identified seL4 as the best solution for VEA because of the formal proof, which would assist in the security certification process. Unfortunately, the proof only applies to a specific hardware configuration. In modifying the kernel, the proof needs to be updated as well. However, the files that DornerWorks modified to support ARMv8 virtualization are not covered by the proof. In order to meet the requirements of the security certification process, the proof needs to be extended to include ARMv8 virtualization on the specified hardware platform. Furthermore, while the kernel code would be formally verified, the VMM functionality that exists in user space would not have any formal backing, but this can be addressed by an appropriate system design.

Another area for process improvement is getting involved with open-source maintainers earlier when developing new features. Data61 and DornerWorks were both working on some ARMv8 virtualization support at the same time. We discussed it early in the project, but could not share much with them during development. This resulted in two different partial implementations that are now being merged by Data61. If developers could work more closely with the maintainers then this could be managed more easily and code would get mainlined more quickly. This would require a discussion early on in the project to figure out exactly what will be open sourced and what can be shared with the code maintainers.

There was quite a lot of work done for this project, both in developing basic functionality for virtualization as well as utilizing it for a prototype system. Since the virtualization support has all been open sourced, the effort needed for future developers to build a virtualized system with seL4 on ARMv8 devices has significantly decreased. This should make it easier for more organizations to integrate seL4 into their solutions and provide more secure systems.

6. CONCLUSION

One of the challenges in utilizing seL4 is the lack of drivers and libraries for user space. The easiest way to overcome this is to run seL4 in VMM mode so the microkernel can execute and manage guest operating systems. Applications can then rely on the drivers, libraries, and stacks that are provided by their preferred OS. This solution can allow seL4 to provide its guaranteed isolation while relying on other sources for user space support. However, applications that have the most rigorous security or safety requirements may not be certifiable on top of an OS such as Linux. The best approach for those applications is to develop them as native seL4 applications, porting or developing the necessary libraries and drivers, and perform at least some degree of partial verification. The amount of verification needed will depend on each project and its requirements. As the seL4 ecosystem continues to improve, it is possible that users will be able to purchase proven libraries and drivers on which to build applications.

One of the biggest challenges that still needs to be solved is updating the proof to support the newest and necessary seL4 features. For example, ARMv8 virtualization is now possible with seL4, but only as an unverified configuration. Theoretically, using virtualization is a great method, but there is not yet a proof or a formal model to back it up. Currently, Data61 is the best organization to update the proof, but they are only a single organization with limited funding and resources. They are also not a U.S. based company, which can make it hard for the DoD to interface with them. To solve this issue, more investment needs to be focused on training people on formal methods so more organizations can provide both new functionality and the proofs to go along with that functionality. The seL4 Center of Excellence (COE) that is being brought up by DARPA and AFRL is one avenue for this, but more resources, such as university scholarships for formal methods, should be investigated to further improve this.

Despite these issues, seL4 still has the most robust claim to security and safety available. Even if specific platforms are only partially verified for a configuration of seL4, it has been shown that it can be fully verified with the right amount of effort. The formal proof of seL4 provides the best assurances that a system can have. The proof means that the security properties of integrity and confidentiality are met, which means that one can actually claim some "shall not" requirements, i.e. things the system is not supposed to do. This is impossible with traditional testing and documentation methods for certification. This fact makes seL4 a great solution for any projects that require rigorous safety or security certifications. Pair that along with the free, open-source nature of the seL4 codebase and one has a great platform enabler for military ground vehicle applications as well as other defense markets.

1. REFERENCES

[1] C. Chaplain, "WEAPON SYSTEMS CYBERSECURITY: DOD Just Beginning to Grapple with Scale of Vulnerabilities," GAO-19-128, Oct 9, 2018. [Online]. Available: https://www.gao.gov/products/GAO-19-128.

[2] N. Studer, R. VanVossen, "Xen and the Art of Certification," *Xen Developer Summit*, 2014. [Online]. Available: https://xenbits.xen.org/people/larsk/XPDS14%20-%20Xen%20and%20the%20Art%20of%20Certification.pdf

[3] M. Larabel, "The Linux Kernel Has Grown By 225k Lines of Code So Far This Year From 3.3k Developers," *Phoronix Media*, Sept 16, 2018. [Online]. Available: https://www.phoronix.com/scan.php?page=news_item&px=Linux-September-2018-Stats

[4] B. Beizer, *Software Testing Techniques*, Van Nostrand Rheinhold, New York, second edition, 1990.

[5] RTCA DO-333, Formal Methods Supplement to DO-178C and DO-278A (December 2011)

[6] G. Klein, M. Norrish, T. Sewell, H. Tuch, S. Winwood, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt and Kolanski, "seL4: formal verification of an OS kernel," in *the ACM SIGOPS 22nd symposium*, Big Sky, Montana, USA, 2009.

[7] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Transactions on Computer Systems*, vol. 32, no. 1, pp. 1-70, 26 02 2014.

[8] "seL4 Docs," seL4 docs. [Online]. Available: https://docs.sel4.systems/FrequentlyAskedQuestions.html#what-does-sel4s-formal-verification-mean. [Accessed: 13-May-2019].

[9] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143-155, 1966.

[10] I. Kuz, G. Klein, C. Lewis and A. Walker, "capDL: A language for describing capability-based systems," in *Proceedings of the first ACM Asia-Pacific workshop*, ACM, 2010, pp. 31-36.

[11] D. Schafer, "seL4 & Agile and Resilient Embedded Systems (ARES)," *seL4 Summit*, Nov 14, 2018. [Online]. Available: https://www.sel4-us.org/summit/presentations/Session_3_Talk_1_Schafer_ARES.pdf

[12] L. Elliott, "U.S. Army Ground Vehicle Applications for the seL4 Microkernel," *seL4 Summit*, Nov 14, 2018. [Online]. Available: https://www.sel4-us.org/summit/presentations/Session_3_Talk_3_Elliott_TARDEC_opsecd.pdf

[13] R. VanVossen, "Practical use of seL4," *seL4 Summit*, Nov 14, 2018. [Online]. Available: https://www.sel4-us.org/summit/presentations/Session_2_Talk_3_VanVossen_Practical_Use_of_seL4.pdf

[14] P. Jungwirth, "Open Source High Assurance System," DoD 2018.3 SBIR Solicitation, Aug 24, 2018. [Online]. Available: https://www.sbir.gov/sbirsearch/detail/1508741

[15] T. Barnett, "High assurance hardware state machine trusted computing base," DoD 2019.2 SBIR Solicitation, May 02, 2019. [Online]. Available: https://www.sbir.gov/sbirsearch/detail/1605861

[16] R. Merrett, "NICTA pushes seL4 microkernel beyond drones," *Techworld Australia*, July 29, 2014. [Online]. Available: https://www.techworld.com.au/article/551020/nicta_pushes_sel4_microkernel_beyond_drones/

[17] R. Pearce, "seL4 kernel to head into space," *Computerworld*, August 09, 2016. [Online]. Available: https://www.computerworld.com.au/article/604761/sel4-kernel-head-into-space/

[18] "iOS Security iOS 12.1," Apple, Nov, 2018. [Online]. Available: https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf

[19] TrustZone, Arm, [Online]. Available: https://developer.arm.com/ip-products/security-ip/trustzone

[20] K. MCLeod, "Usermode OS Components On seL4 With Rump Kernels," p. 80.

[21] A. Warkentin , *UX/RT OS*, *Gitlab*. [Online]. Available: https://gitlab.com/uxrt/uxrt-toplevel/blob/master/architecture_notes. [Accessed: 14-May-2019].

[22] "Genode on seL4 - Building a simple root task from scratch," *Genode*. [Online]. Available: https://genode.org/documentation/articles/sel4_part_1. [Accessed: 14-May-2019].

[23] C. Richardson, *Robigalia*. [online] Robigalia.org. Available at: https://robigalia.org/. [Accessed 14 May 2019].

[24] Free Software Foundation, *GNU General Public License*, 2 ed., 1991.

[25] "Changing Exception levels," Fundamentals of ARMv8-A, Mar 03, 2017. [Online]. Available: https://developer.arm.com/docs/100878/latest/changing-exception-levels

[26] ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. 2017.