IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 23 August 2023, accepted 6 September 2023, date of publication 15 September 2023, date of current version 21 September 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3316031

RESEARCH ARTICLE

Is Formal Verification of seL4 Adequate to Address the Key Security Challenges of Kernel Design?

MINA SOLTANI SIAPOUSH, (Graduate Student Member, IEEE), AND JIM ALVES-FOSS, (Senior Member, IEEE)
Center for Secure and Dependable Systems (CSDS), University of Idaho, Moscow, ID 83844, USA

Corresponding author: Jim Alves-Foss (jimaf@uidaho.edu)

ABSTRACT Formal method tools are used in the initial stages of the software development cycle and have advanced to deal with the design difficulties related to ensuring strong cybersecurity and reliability in high-assurance systems. Operating system kernels are the security keystone of most computer systems. Their continuous advances require formal verification that guarantees the accuracy of their functionalities. As the world's first microkernel to be proven mathematically secure and functionally correct, seL4 has been adopted for use in a range of critical systems, including defense, aerospace, and financial services applications. In spite of the great effort of the seL4 team to present a comprehensive formal verification of the kernels, there are some security aspects of verification that suffer from a limited scope. From an outsider's perspective, this paper aims to evaluate if seL4 formal verification is adequate to address security requirements and surveys the recent state of the art for seL4 microkernels.

INDEX TERMS Microkernel, seL4, formal methods, correctness verification.

I. INTRODUCTION

One of the critical services of a computer system is to allow applications to store and protect sensitive data from any malicious activity. Considering the latest research on sophisticated and targeted attacks worldwide, it is generally accepted that the effectiveness of existing security mechanisms for computer systems needs to be improved. Since there are a lot of examples when the correctness of separate computer components has been successfully established, the formal verification of an entire system has drawn lots of attention. An Operating System (OS) considers the kernel as the crucial component which provides basic services for all other parts of the system. It is the main layer between the OS and underlying hardware devices, the first module of the OS to be loaded, and the last one to be terminated. Formal verification of a kernel is a crucial step toward full system verification.

The associate editor coordinating the review of this manuscript and approving it for publication was Claudio Zunino.

The security of computer systems and protection of critical aspects of the computer's behavior is not new, designers have been addressing how software components access hardware and resources for a long time. Processors were designed to support at least two operating modes: kernel mode and user mode. Kernel mode enables software to have unrestricted access to the system's resources. The OS kernel is loaded into protected memory space and operates in the privileged kernel mode. User mode enables applications to load and execute the program, although not access privileged instructions or resources. The kernel uses the memory space and resources for that application's use and runs the application within that user memory space.

In contrast to conventional large kernels and OSs that execute a wide range of functions, a microkernel's function is to separate the subjects and resources of a system into security policy equivalence classes. In microkernels, the user services and kernel services are implemented in different address spaces so this reduces the size of the microkernel compared to recent large kernels. This paper focuses on a

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

recent and most secure microkernel, seL4 [1], explains the procedure of formal verification used to ensure its security, and investigates its weaknesses and strengths.

A. MOTIVATION

As microkernels have drawn lots of attention in recent years, we start our research by focusing on the recent and most secure microkernel, seL4. The seL4 microkernel has been proven to be mathematically secure and functionally correct, making it a viable option for use in critical systems, including defense, aerospace, and financial services applications. However, the formal verification of seL4 kernels suffers from a limited scope, making it crucial to evaluate its adequacy in addressing security requirements. This paper aims to provide researchers and developers with a comprehensive understanding of the current state of seL4 formal verification and its potential application in high-assurance systems. Our analysis of recent work on seL4 microkernels will help researchers identify gaps in current research, strengths, and limitations of seL4 microkernels, and provide insights into future research directions in a way making them informed decisions about the suitability of seL4 for their specific cybersecurity needs.

The rest of the paper is organized as follows. Section II provides background information on kernels and preliminaries about formal verification. Section III discusses the drawbacks of seL4 formal verification and how recent research addresses them. Section IV evaluates existing work and presents a comparison of research on seL4 formal proof. Finally, Section V concludes the paper and discusses future work.

II. BACKGROUND

This section provides some related background about this research. First, it provides a summary of different types of OS kernels, the seL4 kernel, formal methods, and then discusses the formal verification of microkernels.

A. KERNELS

As mentioned earlier, a kernel is the primary layer that separates the OS and underlying hardware devices, the first module of the OS to be loaded, and the last one to be terminated. Kernels are classified into four groups as described in the following. In this paper, we focus on microkernels and leave others for future work.

1) MONOLITHIC

A monolithic OS has a large kernel and contains many services and components; examples include Multics [2], Unix, Linux, and MS Windows. Monolithic OS kernel These require hardware privilege states for isolation and to control I/O access to peripherals. The kernel provides a series of system calls for processes to ask for services such as inter-process communication (IPC). System calls are literally equivalent to function calls but are relatively slower, due to extra processing required for context switching, a process of saving application state and restoring the kernel state. The main drawbacks of monolithic kernels are complexity and dependencies between system components. The larger size of a kernel makes it hard to maintain. As illustrated in Figure 1 monolithic kernels place OS functionality into a single shared address space executing in kernel mode. As the kernel provides a wide range of services, the kernel code typically consists of a sizable quantity of privileged code, all of which has to be verified.

2) MICROKERNEL

A microkernel is a type of kernel design in which the user services and kernel services are implemented in different address spaces, thus reducing the size of kernel and OS. In microkernels, the foremost services such as memory management, inter-process communication (IPC), and CPU-Scheduling are placed in the kernel, while the remaining services are placed in the user application. This allows users to communicate with those secondary services in the user application space. Section II-B provides a more detailed explanation of microkernels. Some of the better known microkernels are MINIX3, Fiasco [3], QNX [4], Fuchsia [5], Nemesis [6], micro-ITRON [7], and seL4.

3) EXOKERNEL

Exokernels present a different mechanism for component isolation where operations with authority are represented as compiled libraries for each user program. The libraries communicate with hardware by secure bindings executing in a ring which provides an Application Programming Interface (API) for untrusted libraries. A secure binding is a safety feature that separates resource authorization from its real use. Exposing system resources directly to applications and their untrusted libraries is defined at initial use, which eliminates the need for privilege validation during runtime when resources are utilized by applications. Some exokernels include ExoKernel (MIT's research project) [8], Genode [9], Viengoos [10], Nooks [11], and Redox [12].

4) HYBRID

A hybrid kernel is an OS kernel architecture that attempts to combine aspects and benefits of microkernel and monolithic kernel architectures. Some known examples are the Polynomial kernel [13], Radial basis function (RBF) kernel [14], Sigmoid kernel [15], Laplacian kernel [16], ANOVA kernel [17], Triangular kernel [18], and Stable Spline kernel [19].

B. SECURE MICROKERNEL: SEL4

The main issue with microkernels is that system calls are often slow mostly because of IPC [20], and there is a lot of context switching between the User mode and kernel mode to request a service. To address these problems, Liedtke proposed *L4* [21] to show that IPC can be supper-fast, a factor of 10–20 times faster than contemporary microkernels. L4 evolved from an earlier system, L3 which

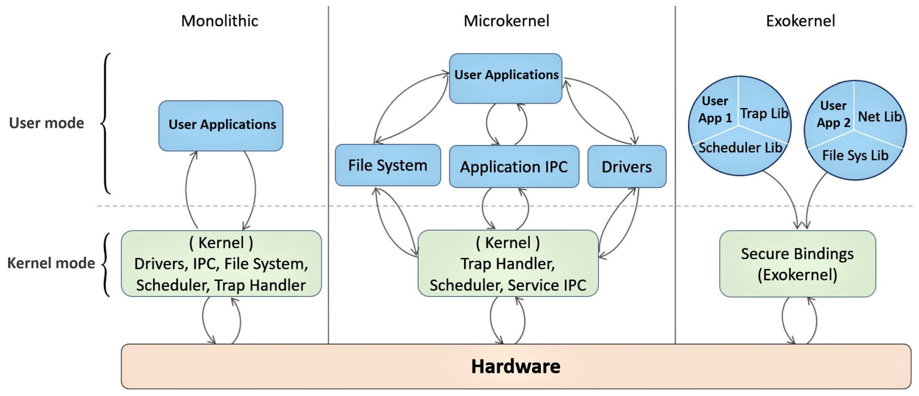


FIGURE 1. Control flow of different types of kernels.

TABLE 1. Summary of key features of recent microkernels.

Microkernel	Supported Platforms	Security Features	Purpose	Verification	License	Size (LOC)
seL4	ARM, x86	Strong	Secure and Real-time	Formally verified	GPLv2 with an exception	50k
QNX	ARM, x86, PowerPC	Moderate	Embedded, Real-time	×	Proprietary	1M
Fiasco	x86, ARM	Good	Experimental	×	Open-source (BSD)	30k
Fuchsia	ARM, x86, RISC-V	Good	General-purpose	Partially verified	Open-source (Apache)	5M
Micro-ITRON	Various	Moderate	Embedded	×	Proprietary	N/A
Nemesis	ARM,x86	Strong	Research, Teaching	×	Open-source (GPL license)	N/A

was the starting point of the journey into microkernels. In this design, a service is only allowed inside the microkernel if moving it outside the kernel–allowing conflicting implementations–prevents the system's required functionality from being executed. Accordingly, the UNSW/NICTA group's research on L4 lead to a model of a third-generation microkernel called *seL4* (Secure L4), which builds on the strengths of the L4 microkernel design, providing an extremely reliable and secure system as the ultimate goal [1].

Verification is what makes seL4 distinct; seL4 has been formally verified (see Section II-C) to provide end-to-end information flow security, meaning that information cannot leak from one component to another component in the system, except through specified channels. This level of assurance is unique among microkernels. With about 50,000 lines of code backed by rigorously checked, machine-generated proofs, seL4 is the most dependable OS kernel [1]. This is why seL4 is designed for use in critical systems that need a high degree of security, such as military and aerospace systems, medical devices, and financial systems. As seL4 keeps developing, its formal proofs must be updated accordingly. The proofs verify a growing list of properties, including functional accuracy, binary accuracy, security properties of integrity, confidentiality, and availability. Some of the verified features include *isolation* to separate components within the system to ensure that one component cannot interfere with or undermine the operation of another component. SeL4 also offers *flexibility*, which makes it a foundation for a variety of systems, ranging from microcontrollers to complex servers.

In terms of performance, seL4 has been designed for efficient operation and low overhead, making it suitable for use in resource-constrained systems. To establish a concept of trust, the seL4 microkernel reduces the amount of code executed at higher privileges by minimizing the thread control block, a data structure that records the information about a thread, including its current state, stack pointer, and register contents. Reducing the TCB size is important for establishing trust because this means that there are fewer opportunities for bugs and vulnerabilities to be exploited by attackers, as there is less code that they can potentially target. Furthermore, seL4 highlights formal and mathematical correctness, ensuring the kernel is *bug-free*. SeL4 also enhances security through fine-grained access control using capabilities, which are tokens that support control over which entity has access to a certain resource in a system. We summarize the key features of microkernels in Table 1, which explicitly illustrate seL4 is outstanding compared to other microkernels in terms of formal verification.

C. FORMAL METHODS IN A NUTSHELL

Formal methods are a set of techniques used to design and verify computer systems that can guarantee the desired properties or requirements. These methods rely on mathematical models and logic to provide a rigorous and systematic approach to designing and verifying software and hardware systems, especially in high-security systems. These systems have strict requirements for security and confidentiality, and any failure in the system can have serious consequences. One of the key concepts in designing high-security systems is the Trusted Computing Base which is the set of functions and components in a system that are critical to maintaining security and confidentiality. It is defined by the United States Department of Defense Orange Book [22] as the part of the computer system that contains all of the elements responsible for supporting the security policy and enforcing the isolation of objects. Hence, by using formal methods, designers can systematically analyze the system to identify potential vulnerabilities.

Formal methods are classified into three approaches i.e., Refinement, Theorem Proving, and Model Checking. Figure 2 shows the classification of these approaches according to their application in modeling and defining the kernels specifications [23].

1) REFINEMENT

Refinement creates complex systems starting from simple ones by adding features incrementally as it can adapt to new and changing requirements. In other words, it is the verifiable conversion of a high-level abstract formal specification into a real low-level executable program. The refinements are complemented with mathematical proofs that validate them. Refinement has been performed for artifacts ranging from modeling and design levels like architectures to implementation and programming levels like source code [24].

2) THEOREM PROVING

Theorem Proving uses mathematical logic to describe the system and the intended features. A formal system, which describes a group of principles and inference guidelines, provides this logic. Discovering a proof for a feature using the system's principles is known as theorem proving. Theorem provers can be categorized into a spectrum ranging from highly automated, general-purpose software interactivity platforms with specialized features. The automated platforms are beneficial as a complete search process and have a good track record of solving different complex issues. The automated formal methods and systematic formal growth of mathematics have benefited more from interactive systems.

3) MODEL CHECKING

Model Checking depends on creating a finite model of a system and checking that the specified feature maintains in that model. Generally, the verification is carried out as a complete search of state space, which is assured to end because the model is finite. Designing data structures and algorithms that enable us to manage search spaces is the primary challenge in model checking. The application of this method began with hardware [25]; currently, it is used to analyze the specifications of software systems. Contrary to theorem proving, model checking is completely automatic and fast, often generating an answer in a matter of minutes. Thus, verifying the accuracy of software designs using model checking is a helpful technique.

However, due to their greater complexity and scale, it is less appropriate for confirming the accuracy of software implementations [26]. For instance, model checking is usually not applied to programs with more than 10,000 lines of code. Model checking can be combined with other verification techniques to handle the complexity of bigger programs. Model checking can be used in combination with other methods to enhance the testing process and guarantee the accuracy of intricate software systems.

D. FORMAL VERIFICATION OF SEL4

The downsizing of microkernels has made them more modular, with various components relying heavily on one another. However, with the use of advanced methods and thoughtful design, it is possible to fully verify an OS microkernel, as stated by the seL4 team. The formal verification of seL4 includes two main refinement phases: a) between abstract and executable specification and b) between executable specs and implementation [28].

1) FIRST REFINEMENT STEP

The first phase of refinement took around 8 person-years and manually generated 117,000 lines of Isabelle/HOL proof script. The abstract specification of the OS is transformed into a more detailed and executable specification. This transformation involved making decisions about the algorithms and data structures that will be used to implement the system. This phase consisted of the conceptual proof and analyzing the design's execution safety and correctness of components. Fundamental prerequisites of the validation are that each operation is precisely described, that accesses to memory are properly classified and that objects that are read from do still valid.

2) SECOND REFINEMENT STEP

During this step, the focus was on developing a systematic approach to generating invariants and the underlying proof obligations. This was done to address the significant amount of proof work required in the first phase, where 80% of the work was spent on presenting invariants of the executable levels. The invariants are crucial for establishing the correlation between the abstract specification and the implementation of the OS. Furthermore, by developing a systematic approach to generating invariants, the proof work required is streamlined and reduces the amount of manual effort involved. The second refinement step is essential to ensure the correctness and safety of the OS's implementation.

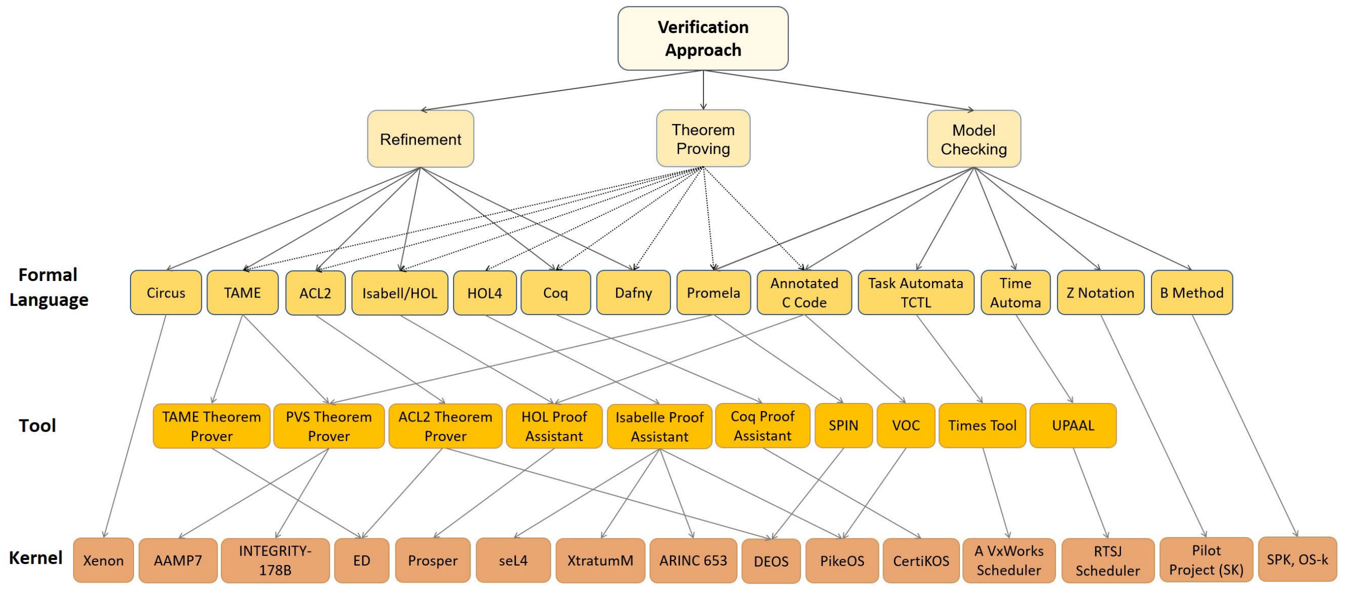


FIGURE 2. Classification of formal methods and tools.

III. WHAT COMPONENTS OF A MICROKERNEL NEED TO BE VERIFIED?

Generally, microkernel verification should include verifying essential components including scheduling, exception management, isolation, access control automation, etc. However, seL4 has some inherent limitations that make it inefficient in full verification. For instance, the initial verification of the seL4 kernel used C and the compiler turned high-level language into machine code. This resulted in the verification results not being entirely valid, as there was a risk of the compiler introducing errors into the code.

To make the seL4 verification process more efficient, some changes need to be made. One approach is to employ a formalized verification process based on machine language, as was done in the team's most recent study, published in 2017 [29]. This approach is more reliable and accurate than the initial kernel verification effort using C, as it eliminates the risk of the compiler introducing errors into the code. Another approach is to use a concurrent program verification approach that is suitable for seL4's coding language. This will enable the verification of highly concurrent components, which may not be possible with the current serial program verification approach.

In the remainder of this section, we discuss recent work that could boost the formal verification of seL4 and related kernels, beyond their machine language-based proofs.

A. LAYERED VERIFICATION

While the seL4 kernel is known for its innovative approach to correctness verification, the process of formally defining and proving the system's properties is slow. The seL4 kernel formalization is composed of over 200,000 lines of Isabelle/HOL code that have undergone a formal definition and verification process and in excess of 8,700 lines of C code. The verification process took 20 person-years of work, highlighting the significant effort and resources required to ensure the system's correctness. Layered verification is a process that separates verification into manageable modules and therefore could reduce the overall proof effort. The major goal of layered verification is to allow simplification of the proof process, especially by abstracting away some lower level details and also allowing for reuse of specifications and proofs of modules outside of a layer that is modified. This section discusses one related project which used layered verification when verifying kernel scheduling.

The idea of separation kernel [30] emanates from the concept of multiple independent levels of security/safety (MILS) [31]. MILS creates several distinct domains with various security degrees on the same hardware. Boettcher et al., used MILS as a two-level mechanism to provide security to system architecture [30]. In terms of policies, analysis of a virtual design is accomplished while inspecting the validated objects and the transmitting channels. One of the key goals of MILS systems is layered verification and allowing a great reduction in proof efforts [32]. Literally, MILS provides implementation and intercommunication channels, to exchange resources through communication channels without threatening the consistency of the system. Thus, it entails security practices that assure full data isolation across the domains and minimize the complexity of validated components.

As a way to provide data isolation features, it is crucial to certify and check how a kernel's schedule section has been implemented. Inappropriate designs of scheduling can negate necessary separation properties and break down the whole system. In this regard, Gao et al. [33] presented a layered verification to verify pointer linked-list through function *'list_add_tail'* and employ a technique to decouple data structure that significantly reduces the complexity of verification. The original code has 320 lines, compared to 32 lines in the restructured old code (including the data structure). They first separate the function's intricate structure into autonomous functions and then write the code in Clight code format by using the Clightgen tool of CompCert [34]. The next step is the creation of the specification of the abstraction/ implementation layer and at last, it verifies the integrity of the specifications of the abstraction/implementation layer and proves the functional correctness of this function. It would be useful to find a way to implement this type of layered verification in seL4.

B. EXCEPTION MANAGEMENT

A vital and time-consuming phase of system development is detecting and managing exceptions that may occur during process execution. Formal modeling and verification of exceptions is a difficult process, and not always well handled in system verification. It is essential to verify exceptions and exception handling to ensure that security vulnerabilities do not exist in this critical part of the system. This section discusses approaches to formalizing and verifying exception handling systems.

When a process is invoked, conditions are checked to validate the invocation context. If a condition is not met, the process is not executed and an exception is signaled to the invoker to run proper handling techniques. As a function module, exception management is generally implemented in assembly language and is in charge of implementing unexpected modifications in the control flow to respond to exceptional events. Unfortunately, to facilitate formal models, the current verification initiatives either neglect to simulate how exceptions are handled or employ methodologies according to the abstraction layer to certify the accuracy of exception management [35]. seL4 implements an event-driven model, where exceptions are delivered to a central handler and processed according to a set of predefined rules. However, it disables nearly all asynchronous interrupts during kernel runtime and implements suspend points using polling. By polling, a program can periodically check for these signals and then take appropriate action based on the results.

There is some research that leverages conceptual specifications to explain how the system behaves against exception management in C [37]. Liu et al. [38] attempted to employ an approach that uses multi-threaded proof to show the correctness of their work. Because there are no formal semantic support for interrupts, their proposed model involves defining the meaning of an interrupt using the same semantics as threads. However, there is some slight distinction between interrupts and threads, so it can be hard to explain the interrupt semantics.

Micro-certified OS kernel (mCertiKOS) [39] is another approach that is designed to be more modular and flexible than seL4, which makes it easier to extend and adapt to different use cases. In mCertiKOS, exceptions and interrupts are managed by a small set of kernel code, which is isolated from the rest of the OS and subject to formal verification. This design helps to ensure that exceptions and interrupts are handled securely and reliably, without compromising the rest of the system. When an exception or interrupt occurs, the relevant handler in the mCertiKOS kernel is executed to handle the event. This may involve altering the state of the system, generating an error message, or taking other actions to respond to the exception. The handling of exceptions and interrupts is designed to help to ensure that the system remains reliable and responsive even in the face of unexpected events.

Ma et al. proposed verification of Real-Time Exception Management SPARCv8 (EMS) [40] which focuses on improving the performance of exception handling in a computer system and verified based on Hoare logic. It is a hybrid mechanism that combines the hardware exception handling approach with software exception handling. The idea is to offload some of the overhead of hardware exception handling to software, while still preserving the benefits of hardware-based exception handling, such as low latency and high reliability. The EMS approach has been shown to provide significant performance improvements compared to traditional hardware exception-handling approaches. seL4 and the EMS approach differ in their goals, with seL4 focusing on security and the EMS approach focusing on performance, but both aim to improve the handling of exceptions in OS.

C. I/O SEPARATION

I/O separation is a design principle used in microkernels, which involves separating Input/Output (I/O) operations from other system functions and moving them into separate, user-space processes or servers. While formal verification of the seL4 microkernel can provide a high level of confidence in its ability to isolate I/O, it is not adequate by itself to ensure that I/O separation is foolproof. If the separation fails, this can lead to security vulnerabilities. This section discusses an approach to formalizing I/O separation, even for commodity devices, and also uses a layered approach similar to that used in Section III-A.

As I/O separation depends on some factors beyond just the code of the microkernel. For example, the hardware architecture and configuration can affect the ability of the system to provide I/O separation. In case the hardware is not properly configured, or if there are bugs or vulnerabilities in the hardware, then this can compromise the ability of the software to provide I/O separation.

SeL4 can maintain static I/O isolation but typical Commodity I/O hardware often cannot distinguish between the isolated program code, and the transfers between I/O. Despite employing outstanding hardware, systems often compromise separation guarantee in favor of improved performance. To address this, Yu et al. [40] developed a formal model for I/O separation, which highlights a separation policy based on the authorization of I/O transfers and independent of hardware. Drivers are separated by I/O kernels in various methods; some do it within the I/O kernels, while others do it within the applications. Device activation is supported by some I/O kernels while static activation only is supported during system startup. Thus, Yu et al.' layered design, which lists important I/O components followed by descriptions of the abstract state, provides a general overview of devices, drivers, and I/O objects, and results in a verified assembly implementation. The components can communicate with one another using a variety of I/O methods. For example, CPUs can read/write memory using Direct Memory Access (DMA), read and write devices using Memory-Mapped I/O (MMIO), and stop the current execution of CPUs via interrupts.

D. AUTOMATING CAPABILITIES

Formal verification of a system validates that the system behaves as specified. However, useful systems are programmable/configurable to allow for deployment for many applications. We can verify that a system supports the configured security policy, but do we know that the configure policy is actually the policy we want? This section discussed automating the configuration of capabilities in seL4 to assist in this process.

After the seL4 system boots up, it hands a set of capabilities to the initial user process, referred to as the root task. Generally, the root task is responsible for starting new processes and assigning capabilities to them. However, manually setting up the capability-based access control managed by the kernel can be time-consuming and prone to mistakes. To address these issues, Kuz and colleagues introduced the Capability Distribution Language (CapDL), a language designed to simplify the process of distributing capabilities [41]. Accordingly, Boyton et al. presented a formal verification for the root task, that enables the starting of a system using a CapDL specification [42] and helps to ease the workload for developers.

One of the visions of seL4 is to use component architectures and also the possibility to verify entire systems. To achieve this, the language called Component Architectures for Microkernel Embedded Systems (CAmkES) [43] was developed by Kuz et al. This architecture description language enables system engineers to write a description of a seL4 system's process, task, and component architecture in a concise form. The CAmkES tool takes this description, generates the necessary RPC stub code for implementing seL4 communication scheme, and produces a CapDL specification for the root task. With the help of CAmkES, components can be set up to communicate according to seL4 patterns, allowing for sending and receiving and calling and replying. It will determine the required number of threads and endpoints and set up the virtual memory for components.

Later, Fernandez et al. extended the capabilities of CAmkES to generate machine-checkable proofs using Isabell/HOL [44]. This marked significant progress in the effort to build a fully verified system. With three elements in place: 1) proven binary correctness of seL4, 2) proven accurate beginning of CapDL, and 3) Machine-verifiable proofs of the RPC stub code for CAmkES, system engineers only need to demonstrate that the "business logic" of their parts is accurate. However, writing CapDL specifications can still be challenging, due to their size and complexity.

E. APPLICATION DEVELOPMENT

Formal verification of a system such as seL4, only verifies the system as shipped. When developers build applications on top of that system, there is a need for some validation that those applications are secure. We know that a developer can build a simulator of an insecure system on top of any secure system. This section surveys approaches to verifying application development, which benefit from using a layered approach knowing that the application is built on top of a secure OS kernel.

Since the efficient functionality of seL4 makes it amenable to high-assurance design, developers may need to create mixed-criticality systems where all of the components are not required to be verified to the highest degree of criticality. Moreover, regarding the system architecture, a firm separation foundation can facilitate modification, without solely involving the recertification of other objects. However, it dictates that they typically only offer lower-level configuration methods, with a focus on memory blocks, functions, etc. Thus, although the use of seL4 is for a full system, the separation assurance of the kernel is not adequate; understanding application logic properly, security policies, and practices for design is significantly challenging.

Several code generation approaches tried to address this issue [45], [46], [47], [48]. As an example, consider the Refinement of AADL Models for Synthesis of Embedded Systems (RAMSES) tool [47]. The refinement phases are directed by developer-defined properties for the desired system, through analyzing the objects and resources that evaluate features versus necessities and platform capabilities. After reaching an adequate model, RAMSES creates a C component infrastructure, that can be placed on Linux, nxtOSEK, and POK. It has been employed in avionics, railway, and robotics domains. In the aeronautics area Cofer et al. [48] have proposed a formally verified implementation of aerial vehicles using multiple tools; for instance, jKind model checker [49], which assures the correctness of components and Isabelle/HOL theorem prover to guarantee that system execution semantics is according to the model. As another example, Trusted Build [47] was proposed in the DARPA

TABLE 2. Comparison of research on different features of seL4 formal verification.

Research	Year	Verified property	Approach	Formal language	Tool	verification effort (LoC)
iDola [38]	2014	Exception management	Refinement	iDola	Tsmart-Edola	~20 k
RPC stub [44]	2015	Automating capabilities	Theorem Proving	Isabelle/HOL	AutoCorres	N/A
mCertiKOS [39]	2016	Exception management	Refinement	Coq	CompCert	3 k
Gao et al. [33]	2021	MILS scheduling	Theorem Proving	Coq	Clightgen	~0.5 k
EMS [35]	2021	Exception management	Theorem Proving	Coq	N/A	15 k
I/O-SM [40]	2021	I/O separation	Refinement	Coq	Dafny	28,518
HAMR [50]	2021	Application development	Refinement	HOL	AADL	40 k

High Assurance Cyber Military Systems (HACMS) Program by Collins Aerospace. This approach created a framework for seL4 from AADL using the CAmkES modeling language and was crucial in outlining the model-based creation for seL4 perspective.

In recent work, Hatcliff et al. [50], introduce HAMR (High-Assurance Model-based Rapid engineering) for embedded cybersystems which is built using AADL. HAMR is a component of the larger set of tools known as BriefCASE, which Collins Aerospace [51] presented. The Model-founded Systems Engineering (MBSE), on which BriefCASE is founded, uses models as the primary means of communication and comprehension between the groups in charge of the system's design. Moreover, Model-driven Development (MDD) models in BriefCASE are counted as the initial approach applied for analysis, testing, verification, and generating source code.

Question: *Is formal verification of seL4 adequate to address security challenges?*

The answer to this question depends on what functionality you need from seL4. If we look at the work discussed in this section, we see that researchers feel there are areas that need improvement. According to the fact that seL4's focus is on providing a strong foundation for building secure systems by low-level security guarantees, a high-level functionality is required to address the features used in this section. For instance, supporting application development in embedded cyber-systems needs libraries, tools, and application programming interfaces for developing applications. As another example, automating capabilities and exception management may require a more advanced process management system that can enforce fine-grained access control policies. In addition, MILS scheduling may require real-time scheduling capabilities that do not exist in seL4. Similarly, I/O separation may require a more sophisticated I/O subsystem that can isolate different components and prevent malicious actors from accessing sensitive information. Table 2 illustrates a detailed overview of verification efforts proposing an improvement of seL4 which is sorted based on the time of release.

IV. FUTURE WORK

Determining which direction will be taken on sel4 formal verification depends on the priorities and goals set by the researchers and developers. However, here are a few possibilities based on current trends in the field:

- *Expanding coverage*. There is ongoing work to formally verify more parts of the seL4 system, including device drivers, libraries, and protocols.
- *Improving scalability*. As seL4 systems grow in scale and complexity, there is a need to improve the scalability of formal verification methods to handle larger and more complex systems.
- *Enhancing security*. Formal verification is essential to ensure the security of seL4. Focusing on verifying security properties, such as confidentiality and integrity, and ensuring the absence of vulnerabilities should be considered.
- *Integrating with other tools*. Integrating seL4 formal verification with other tools, such as automated testing and code generation, to improve the overall development process and increase the efficiency of verification could be another promising work in this field.

Although this paper presents an overview of what properties of enhancement to seL4 and related formal methods, some of the features have not been completely addressed, i.e. automating capabilities.

V. CONCLUSION

The use of formal method tools in the initial stages of the software development cycle has proven crucial in ensuring strong cybersecurity in high-assurance systems. Operating system kernels are a vital component of most computer systems, and their continuous advances require formal verification to guarantee the accuracy of their functionalities. While seL4 has been proven mathematically secure and functionally correct, there are still some security aspects of verification that suffer from a limited scope. This paper evaluates the adequacy of seL4 formal verification in addressing security requirements and surveys recent work that presents an expansion on seL4 formal verification. Based on our research, it is evident that seL4 is well-suited for building secure systems by providing low-level security guarantees. However, high-level functionality such as libraries, tools, and APIs are required to address the features used in this section, such as supporting application development in embedded cyber-systems. Therefore, whether seL4 formal verification is adequate to address security challenges depends on the specific functionality required from seL4. Nonetheless, the continuous advances in microkernel development require a comprehensive approach to formal verification to guarantee the accuracy of the system's functionalities and ensure strong cybersecurity in high-assurance systems.

REFERENCES

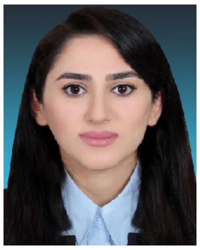
[1] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, ''SeL4: Formal verification of an OS kernel,'' in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ.*, Oct. 2009, pp. 207–220.
[2] M. D. Schroeder, D. D. Clark, and J. H. Saltzer, ''The multics kernel design project,'' *ACM SIGOPS Operating Syst. Rev.*, vol. 11, no. 5, pp. 43–56, Nov. 1977.
[3] M. Hohmuth and H. Härtig, ''Pragmatic nonblocking synchronization for real-time systems,'' in *Proc. USENIX Annu. Tech. Conf., Gen. Track*, 2001, pp. 217–230.
[4] (2022). *QNX*. [Online]. Available: http://www.qnx.com/
[5] (2022). *Fuchsia Overview*. [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/
[6] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, ''The design and implementation of an operating system to support distributed multimedia applications,'' *IEEE J. Sel. Areas Commun.*, vol. 14, no. 7, pp. 1280–1297, 1996.
[7] K. Sakamura, ''ITRON: An overview,'' in *TRON Project 1987 Open-Architecture Computer Systems: Proceedings of the Third TRON Project Symposium*. Tokyo, Japan: Springer, 1987, pp. 29–34, doi: 10.1007/978-4-431-68069-7_3.
[8] C. L. Coffing, ''An x86 protected mode virtual machine monitor for the MIT exokernel,'' Doctoral dissertation, Massachusetts Inst. Technol., Cambridge, MA, USA, 1999.
[9] (2023). *Genode*. [Online]. Available: https://genode.org/
[10] N. H. Walfield, ''Viengoos: A framework for stakeholder-directed resource allocation,'' in *Proc. EuroSys Conf.* Nuremberg, Germany, Apr. 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:55777526
[11] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers, ''Nooks: An architecture for reliable device drivers,'' in *Proc. 10th Workshop ACM SIGOPS Eur. Workshop*, 2002.
[12] (2022). *Redox*. [Online]. Available: https://www.redox-os.org/
[13] J. Fan, N. E. Heckman, and M. P. Wand, ''Local polynomial kernel regression for generalized linear models and quasi-likelihood functions,'' *J. Amer. Stat. Assoc.*, vol. 90, no. 429, pp. 141–150, Mar. 1995.
[14] S. H. Javaran, N. Khaji, and A. Noorzad, ''First kind Bessel function (J-Bessel) as radial basis function for plane dynamic analysis using dual reciprocity boundary element method,'' *Acta Mechanica*, vol. 218, pp. 247–258, May 2011.
[15] H. T. Lin and C. J. Lin, ''A study on sigmoid kernels for SVM and the training of non-PSD kernels by SMO-type methods,'' *Neural Comput.*, vol. 3, p. 16, Mar. 2003.
[16] R. Kondor and H. Pan, ''The multiscale Laplacian graph kernel,'' in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 29, 2016, pp. 1–9.
[17] M. O. Stitson, A. Gammerman, V. Vapnik, V. Vovk, C. Watkins, and J. Weston, ''Support vector regression with ANOVA decomposition kernels,'' in *Advances in Kernel Methods: Support Vector Learning*. Cambridge, MA, USA: MIT Press, 1999, pp. 285–292.
[18] F. Fleuret and H. Sahbi, ''Scale-invariance of support vector machines based on the triangular kernel,'' in *Proc. 3rd Int. Workshop Stat. Comput. Theories Vis.*, 2003, pp. 1–13.
[19] T. Chen, ''Continuous-time DC kernel—A stable generalized first-order spline kernel,'' *IEEE Trans. Autom. Control*, vol. 63, no. 12, pp. 4442–4447, Dec. 2018.
[20] D. Kuznetsov and A. Morrison, ''Privbox: Faster system calls through sandboxed privileged execution,'' in *Proc. USENIX Annu. Tech. Conf.*, 2022, pp. 1–16.
[21] J. Liedtke, ''Improving IPC by kernel design,'' in *Proc. 14th ACM Symp. Operating Syst. Princ.* New York, NY, USA: Association for Computing Machinery, 1993, pp. 175–188.
[22] L. Qiu, Y. Zhang, F. Wang, M. K. Han, and R. Mahajan, ''Trusted computer system evaluation criteria: A general model of wireless interference,'' in *Proc. MobiCom*, Montréal, QC, Canada. New York, NY, USA: Association for Computing Machinery, 2007, pp. 171–182, doi: 10.1145/1287853.1287874.
[23] R. C. Bhushan and D. K. Yadav, ''A survey on formal verification of separation kernels,'' *Recent Adv. Comput. Sci. Commun.*, vol. 15, no. 6, pp. 832–850, Jul. 2022.
[24] A. Cavalcanti, A. Sampaio, and J. Woodcock, *Refinement Techniques in Software Engineering: First Pernambuco Summer School on Software Engineering*, vol. 3167. Recife, Brazil: Springer, 2006.
[25] E. M. Clarke and J. M. Wing, ''Formal methods: State of the art and future directions,'' *ACM Comput. Surveys*, vol. 28, no. 4, pp. 626–643, Dec. 1996.
[26] D. M. G. María and P. Merino, *Model Checking Software*. Berlin, Germany: Springer, 2018.
[27] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, ''Comprehensive formal verification of an OS microkernel,'' *ACM Trans. Comput. Syst.*, vol. 32, no. 1, pp. 1–70, Feb. 2014.
[28] G. Klein, P. Derrin, and K. Elphinstone, ''Experience report: SeL4: Formally verifying a high-performance microkernel,'' in *Proc. 14th ACM SIGPLAN Int. Conf. Funct. Program.*, Aug. 2009.
[29] G. Klein, J. Andronick, G. Keller, D. Matichuk, T. Murray, and L. O'Connor, ''Provably trustworthy systems,'' *Philos. Trans. Royal Soc. A, Math., Phys. Eng. Sci.*, vol. 375, Oct. 2017, Art. no. 20150404.
[30] J. M. Rushby, ''Design and verification of secure systems,'' *ACM SIGOPS Operating Syst. Rev.*, vol. 15, no. 5, pp. 12–21, Dec. 1981.
[31] J. A. Foss, P. W. Oman, C. Taylor, and W. S. Harrison, ''The MILS architecture for high-assurance embedded systems,'' *Int. J. Embedded Syst.*, vol. 2, p. 239, Jan. 2006.
[32] C. Boettcher, R. DeLong, J. Rushby, and W. Sifre, ''The MILS component integration approach to secure information sharing,'' in *Proc. IEEE/AIAA 27th Digit. Avionics Syst. Conf.*, Oct. 2008, pp. 1.C.2-1–1.C.2-14, doi: 10.1109/DASC.2008.4702758.
[33] Y. Gao, X. Yang, W. Guo, and X. Lu, ''Verification of MILS partition scheduling module using layered methods,'' *Int. J. Future Comput. Commun.*, vol. 10, pp. 45–52, Dec. 2021.
[34] R. Krebbers, X. Leroy, and F. Wiedijk, ''Formal C semantics: CompCert and the C standard,'' in *Interactive Theorem Proving*. Berlin, Germany: Springer, 2014.
[35] Z. Ma, L. Qiao, M.-F. Yang, S.-F. Li, and J.-K. Zhang, ''Verification of real time operating system exception management based on SPARCv8,'' *J. Comput. Sci. Technol.*, vol. 36, no. 6, pp. 1367–1387, Dec. 2021.
[36] H. Chen, X. Wu, Z. Shao, J. Lockerman, and R. Gu, ''Toward compositional verification of interruptible OS kernels and device drivers,'' in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2016.
[37] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li, ''A practical verification framework for preemptive OS kernels,'' in *Proc. Int. Conf. Comput. Aided Verification*. Toronto, ON, Canada: Springer, 2016, pp. 59–79.
[38] H. Liu, H. Zhang, Y. Jiang, X. Song, M. Gu, and J. Sun, ''IDola: Bridge modeling to verification and implementation of interrupt-driven systems,'' in *Proc. Theor. Aspects Softw. Eng. Conf.*, Sep. 2014.
[39] (2017). *mCertiKos*. [Online]. Available: https://github.com/zjusbo/mcertikos
[40] M. Yu, V. Gligor, and L. Jia, ''An I/O separation model for formal verification of kernel implementations,'' in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 572–589.
[41] I. Kuz, G. Klein, C. Lewis, and A. Walker, ''CapDL: A language for describing capability-based systems,'' in *Proc. 1st ACM Asia–Pacific Workshop Workshop Syst.*, Aug. 2010, pp. 31–36.
[42] A. Boyton, J. Andronick, C. Bannister, M. Fernandez, X. Gao, D. Greenaway, G. Klein, C. Lewis, and T. Sewell, ''Formally verified system initialisation,'' in *Formal Methods and Software Engineering*. Queenstown, New Zealand: Springer, 2013, pp. 70–85.
[43] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, ''CAmkES: A component model for secure microkernel-based embedded systems,'' *J. Syst. Softw.*, vol. 80, no. 5, pp. 687–699, May 2007.
[44] M. Fernandez, J. Andronick, G. Klein, and I. Kuz, ''Automated verification of RPC stub code,'' in *FM 2015: Formal Methods*. Oslo, Norway: Springer, 2015.
[45] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues, ''Ocarina: An environment for AADL models analysis and automatic code generation for high integrity applications,'' in *Proc. Int. Conf. Reliable Softw. Technol.* Brest, France: Springer, 2009, pp. 237–250.
[46] M. Sudvarg and C. Gill, ''A concurrency framework for priority-aware intercomponent requests in CAmkES on seL4,'' in *Proc. IEEE 28th Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2022, pp. 1–10.
[47] E. Borde, S. Rahmoun, F. Cadoret, L. Pautet, F. Singhoff, and P. Dissaux, ''Architecture models refinement for fine grain timing analysis of embedded systems,'' in *Proc. 25nd IEEE Int. Symp. Rapid Syst. Prototyping*, Oct. 2014, pp. 44–50.

[48] D. Cofer, A. Gacek, J. Backes, M. W. Whalen, L. Pike, A. Foltzer, M. Podhradsky, G. Klein, I. Kuz, J. Andronick, G. Heiser, and D. Stuart, "A formal approach to constructing secure air vehicle software," *Computer*, vol. 51, no. 11, pp. 14–23, Nov. 2018.

[49] A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani, "The JKind model checker," in *Proc. Int. Conf. Comput. Aided Verification*. Oxford, U.K.: Springer, 2018, pp. 20–27.

[50] J. Hatcliff, J. Robby, and T. Carpenter, "HAMR: An AADL multi-platform code generation toolset," in *Proc. Int. Symp. Leveraging Appl. Formal Methods*. Rhodes, Greece: Springer, 2021, pp. 274–295.

[51] D. Cofer, I. Amundson, J. Babar, D. Hardin, K. Slind, P. Alexander, J. Hatcliff, G. Klein, C. Lewis, E. Mercer, and J. Shackleton, "Cyberassured systems engineering at scale," *IEEE Secur. Privacy*, vol. 20, no. 3, pp. 52–64, May 2022.



MINA SOLTANI SIAPOUSH (Graduate Student Member, IEEE) received the M.S. degree in software engineering from Islamic Azad University, Iran, in 2018. She is currently pursuing the Ph.D. degree in computer science with the Center for Secure and Dependable Systems, University of Idaho. Since Fall 2022, she has been a Research Assistant with the Center for Secure and Dependable Systems, University of Idaho. Her research interests include dependable systems, formal verification, software-defined networking (SDN), and big data.

JIM ALVES-FOSS (Senior Member, IEEE) received the B.S. degree in physics and mathematics and computer science and the M.S. and Ph.D. degrees in computer science from the University of California at Davis, Davis, CA, USA, in 1987, 1989, and 1991, respectively.

He has been a Professor with the University of Idaho, since 1991, taking a two-year sabbatical with the University of California at Davis, from 2001 to 2003. Since 1999, he has been the Director of the Center for Secure and Dependable Systems, University of Idaho. He is currently a Distinguished Professor of computer science with the University of Idaho. His research interests include the design and analysis of secure systems, including formal methods, automated vulnerability analysis and repair tools, and secure software development practices.

Dr. Alves-Foss was the Team Captain of one of the seven finalist teams in the DARPA Cyber Grand Challenge, held from 2014 to 2016. His two-person team competed against several larger teams in the development of automated software vulnerability analysis and patching tools.