University of Cincinnati

Date: 3/25/2022

I, Nathan Daughety, hereby submit this original work as part of the requirements for the degree of Doctor of Philosophy in Computer Science & Engineering.

It is entitled:

Design and analysis of a trustworthy, Cross Domain Solution architecture

Student's name: **Nathan Daughety**

This work and its defense approved by:

Committee chair: John Franco, Ph.D.

Committee member: John Emmert, Ph.D.

Committee member: Rashmi Jha, Ph.D.

Committee member: Nan Niu, Ph.D.

Committee member: Marcus Dwan Pendleton, Ph.D.

41809

Design and analysis of a trustworthy, Cross Domain Solution architecture

A dissertation submitted to the

Graduate School

of the University of Cincinnati

in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

in the Department of Electrical Engineering

and Computer Science of the

College of Engineering and Applied Science

by

Nathan Daughety

May 9, 2022

B.S. Software Development, Harding University

May 2017

Committee Chair: J. V. Franco, Ph.D.

ABSTRACT

With the paradigm shift to cloud-based operations, reliable and secure access to and transfer of data between differing security domains has never been more essential. A Cross Domain Solution (CDS) is a guarded interface which serves to execute the secure access and/or transfer of data between isolated and/or differing security domains defined by an administrative security policy. Cross domain security requires trustworthiness at the confluence of the hardware and software components which implement a security policy. Security components must be relied upon to defend against widely encompassing threats – consider insider threats and nation state threat actors which can be both onsite and offsite threat actors – to information assurance. Current implementations of CDS systems use sub-optimal Trusted Computing Bases (TCB) without any formal verification proofs, confirming the gap between blind trust and trustworthiness. Moreover, most CDSs are exclusively operated by Department of Defense agencies and are not readily available to the commercial sectors, nor are they available for independent security verification. Still, more CDSs are only usable in physically isolated environments such as Sensitive Compartmented Information Facilities and are inconsistent with the paradigm shift to cloud environments. Our purpose is to address the question of how trustworthiness can be implemented in a remotely deployable CDS that also supports availability and accessibility to all sectors. In this paper, we present a novel CDS system architecture which is the first to use a formally verified TCB. Additionally, our CDS model is the first of its kind to utilize a computation-isolation approach which allows our CDS to be remotely deployable for use in cloud-based solutions.

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

First, I would like to thank my advisor, Dr. John Franco, for his invaluable advice and feedback during my studies. Additionally, his classes are not only responsible for my deep interest in cyber operations, but the lessons he has taught have proven crucial to my research.

I am beyond grateful to my supervisor, Dr. Marcus Pendleton, for his knowledge and guidance throughout this process. When I called Dr. Pendleton about starting the research for my Ph.D., in the summer of 2020, I had no idea what I was getting into. It has been through his patience and understanding that I have made it this far. He has fought battles for me and given me the chance to succeed.

I want to thank Dr. Marty Emmert, Dr. Nan Niu, and Dr. Rashmi Jha for their valuable feedback and for serving on my dissertation committee. I would also like to thank Dr. Laurent Njilla and Dr. Shouhuai Xu for their help in preparing my research for publication, as well as thank Tyler Reuther for helping to refine the research idea. I thank the Department of Defense (DoD) Cybersecurity Scholarship Program (CySP) which is sponsored by the DoD Chief Information Officer, Management Services and the National Security Agency (NSA) is its Executive Administrator. I am additionally grateful to Deputy Director Robert Kaufman, Captain Alex Resnick and Deputy Director Rebecca Lively for overseeing my internship as a part of the CySP scholarship program and my work. I also would like to thank members of the NSA's National Cross Domain Strategy & Management Office for spirited discussions on this topic.

I am especially thankful to Mrs. Vicki Baker. Without her, I would not have made it to this point. I received a call from her, in 2018, asking me to meet with her in her office at the University of Cincinnati. I walked into her office not knowing if I could even afford to pursue a graduate degree. I walked out with multiple scholarships, a plan for my graduate studies program and the

connections to help me along the way.

Each of the following individuals aided me along this journey by guiding me during my undergraduate studies: I have to give a special thanks to my computer science undergraduate professors, Dr. Scott Ragsdale and Dr. Steve Baber. They have left me with knowledge, but more importantly, with lifetime friendships and a clear vision for the future. They have taught me that there will be no less days and that first-verse adherence yields flawed results. I thank another computer science undergraduate professor, Dr. Frank McCown, whose first words to my freshman software development class reminded me that I was in the right place. I am deeply grateful to Dr. J and Mama B James for their love and guidance. I thank Phil Dixon, not just for the cool handshake greeting and his friendship, but for his continued support. I cannot forget to thank Debra Nesbitt for being my boss, my dorm mom, and my Aunt. Some of the skills I learned from her are reflected in this effort.

The following individuals provided a base of knowledge, creativity, and values which paved the way for this work: I must thank my junior high teacher, Mrs. Rita Rogers. While I use her Algebra lessons on a daily basis, I am especially grateful for her friendship and support. I thank my Art teachers Miss Sylvia and Miss Sandra Young for drawing out my creativity, and providing a classroom which, at times, served as a safe haven. I thank my Coach, Al Riffey, for fighting battles for me, telling me how it is and then making me work for it. Each of these people have helped prepare me for this effort.

I thank my closest and most cherished friends, Tanner McDonald, Harley Halliburton, Jonathan Godsey, and Cade Jenkins for always being at my side.

I am deeply indebted to my parents Mark and Mary Daughety. They have taught me, coached me, raised me, and loved me. They have been by my side in all things, and, more importantly, they have taught me how I can do all things. I am thankful for Brandon, Joanna, and Logan Daughety for their support and examples. I have the best big brother in the world; he has been one of the primary motivators for this effort. I thank my Grampa and Gramma, Larry and Mary Lou Daughety for their love, their example, their inspiration and the home they have shared with me throughout my journey – may it stand forever, with eternal blooms and unclouded skies. I thank my D-Dad and Nana, Dr. Charles and Imogene Aebi for neighboring woods and ancient words. They have been an inspiration on a pilgrimage through barren lands. It has been said that one

must know his reason before pursuing a Ph.D.; my reason, though unconventional, is my D-Dad. Simply that I might follow in his footsteps, D-Dad has been the primary source of inspiration for completing this work. Truly, there will be two empty seats at my defense, one for each of my grandfathers, both representing pillars of guidance.

Most importantly, I thank my wife, Paige Daughety, for her encouragement and love during this process. She has supported me more than I could imagine. She has been everything that I have needed and more – I was led to her for good reason. I thank my daughter, Imogene Daughety, for being a bright spot in every day, lightening stress with every smile and giggle.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Intelligence services use specialized, and often clandestine, sources and methods to gather information about their targets. ... To employ the information in network defense exposes it to leakage or theft, particularly by cyber threat actors. Once such information is compromised, it loses its unique defensive value because actors can modify their plans or adjust their tradecraft. More importantly, compromise of defensive information based on intelligence can also compromise the sources and methods used to get it, eliminating future intelligence.

Therefore, national security communities need mechanisms that will allow effective use of sensitive intelligence information for defense of a broad range of networks, but which can also protect that information from unauthorized access, leakage, and theft. – B. M. Thomas and N. L. Ziring [126]

1.1 Idea Formulation

The initial idea for the project stems from the reading of *Using Classified Intelligence to Defend Unclassified Networks* by B. M. Thomas and N. L. Ziring [126]. The paper lays out some high level approaches to solving the problem of, as the title suggests, how to leverage classified intelligence to defend unclassified networks. Formulated from this paper were the following hypotheses:

(i) The concept of using classified technology to process unclassified data can be extended and applied to a system with capabilities including the transfer of data between two domains of differing trust levels;

(ii) Data confidentiality must be the foremost security requirement for such a system; and

(iii) The system's classified technology and operations must be relied upon to not leak information.

The above hypotheses led to further examination of cross domain solution (CDS) technology with the following questions in mind:

(i) Can a CDS architecture be formed to fulfill such a capability as described in [126], i.e. to defend unclassified networks using classified technology?

(ii) How can a system's reliability be measured, and can a CDS architecture be comprehensively proven to be trustworthy?

(iii) Can a baseline CDS solution be developed for versatility such that it can be applied to a variety of environments and use-cases without extensive modification or costs?

1.2 Overview of the Subject Matter

In the context of secure computing systems, the following problems are often encountered: (i) existing methods for accessing and transferring data between domains are not guaranteed to be trustworthy, (ii) the majority of existing technologies for accessing and transferring data between domains are limited to use and management by the Department of Defense (DoD) and are expensive solutions tailored to a single use-case, and (iii) current solutions are physically isolated, secure appliances with capabilities which are inconsistent with the move to cloud-based solutions. Furthermore, existing solutions are either highly specialized systems which cannot be applied to other use-case environments without incurring unreasonable modification and maintenance costs, or they are ad hoc solutions built upon technologies which lack assured security [13, 26, 47, 57].

In the context of military information systems, a *domain* is an environment which contains a set of computer-based systems, processes, data, controls, and security policies defined by a classification label, which serves in isolation from other systems and can only be accessed using a defined set of rules. Similarly, a *security domain* is a system or set of systems separated from other domains by a boundary defined by an administrative security policy. Oftentimes there is a need for a smaller security domain, called an *enclave*, to reside inside a larger security domain which enforces security

policies for more highly classified data. The objective of the security policy is to uphold trust or classification level, information access and transfer regulations, and data ownership within a domain. Creating a secure connection between security domains necessitates the implementation of multifaceted security policies for information flow management in a CDS. *Cross domain* refers to the access to and/or transport of data across domains of isolated and/or differing classification levels. A *CDS* enforces a security policy on an interface between the discrete security domains. The individual defense technologies used in CDS systems, which implement the security policies, are what allow CDSs to employ layered defense. Note that variations of the terms *high* and *low* are used herein, to describe domains and/or assets of higher and lower security classification or trust levels. Additionally, the terms security domains, classification boundaries and variations of these are used to define a domain in a CDS context.

1.3 Problem Statement

The DoD and the Intelligence Community (IC) manage CDS services, devices, and the standards which CDSs must abide by, almost exclusively, through the National Cross Domain Strategy Management Office (NCDSMO) [48]. Furthermore, [47] details the CDS needs and requirements, originally outlined by the UCDMO, that are exclusively written for DoD agencies. More recently, the National Security Agency (NSA) NCDSMO has documented a strategy for improving CDS technology requirements and security policies called Raise The Bar [99]. While there are a few CDS solutions available outside the DoD/IC such as [40, 50], this community manages the CDS standards and technologies which are leveraged by these systems. This not only creates the problem of general CDS availability outside the DoD, but also means these systems are significantly expensive and must be redesigned for specific environments [48].

Further challenges with the current status quo of CDS designs include reliability and assurance (trustworthiness), remote deployability, and accessibility [48, 82]. [126] also states that current CDS products are available only as secure appliance or "strong box" implementations meaning they reside in a physically isolated environment and are unfit for cloud environments as they are not remotely deployable. As a consequence, existing solutions are either highly specialized systems which cannot be applied to other use-case environments without incurring unreasonable

modification and maintenance costs, or they are ad hoc solutions built upon technologies which lack assured security [13, 26, 47, 57].

These problems can be made manifest by the three CDS architectures of current CDS systems. The first architecture uses *physically isolated domains* (Chapter 2.7.1: Physically Isolated Domains) to maintain one classification level per domain. This separation ensures that an authorized operator must maintain multiple physical infrastructures. One implementation is sometimes referred to as a "swivel-chair setup" because the operator could effectively swivel his or her chair to access each workstation while other implementations use a keyboard, video monitor, and mouse (KVM) switch to access different domains from a single computer [117]. Oftentimes this architecture employs air gaps to transfer data using removable devices. The second architecture uses *partitioned workstations* (Chapter 2.7.2: Partitioned Workstations), which relies on domain virtualization on top of a single host. The host regulates the separation between domains by running a corresponding virtual machine (VM) for each domain or trust level. The third architecture uses *data diodes* (Chapter 2.7.3: Data Diodes), which are analogous to electrical diodes, to restrict the flow of data in one direction (e.g. a domain of low classification level may be permitted to transfer data to a domain of high classification level but not the opposite). These tailored designs may explain why the Committee on National Security Systems (CNSS) called the current CDS architectures *niche CDSs* because they lack accessibility to commercial sectors outside of the military/DoD designations [24, 126]. These architectures are further described in Chapter 2.7: CDS Architectures.

A critical observation regarding security systems, in general, is that many descriptions bundle the well-understood security objectives of confidentiality, integrity, availability, authenticity, and accountability (CIAAA) together. In practice, this not only forces impractical redesign, but enforcing each of these objectives is simply not necessary if the threat model does not require it. Furthermore, each security objective requires the addition of more technologies which may incur risk beyond the defined threat model. The CDS architecture, presented herein, focuses on data confidentiality, i.e. ensuring that data spillage does not occur.

Summarizing the preceding discussion of the problem, the facts presented above pose an elevated risk to data confidentiality, the focus of the vCDS threat model, for multiple reasons. First, a system that has not been mathematically proven trustworthy should not be trusted to securely maintain data. Second, the status quo in both commercial and DoD-specific CDS technology is

inconsistent with the paradigm shift to offsite/cloud computing and does not allow for secure remote deployability as it could expose and compromise the data. Third, *security through obscurity* is exposed as a failed security technique, and the need for independent validation of security properties is revealed.

1.4 Significance of the Problem

CDSs are at the forefront of security research and technology because context-specific data, processes, connections and boundaries need a comprehensive protection plan against a disclosure-focused threat model. Furthermore, the data which are protected by the security policies should be able to be made available at a moment's notice to all parties which require it. Secure threat intelligence sharing (TIS) is a necessity in the age where cloud computing meets cyber warfare [61].

Of further significance is the need for a system which can, with all certainty, be relied upon. It has been established that trusting a system to perform and enforce security as expected without being formally evaluated is commonplace [13, 26, 47, 48, 57, 82]. Without having a CDS that can be relied upon to protect the access and/or transfer of sensitive and/or classified information across remote boundaries during critical operations, the attack could already have materialized by the time the information reached its destination or threat actors could have gained access to and exploited the knowledge of the information.

In general, there is a common need for CDS capabilities in the commercial sector in addition to the military sector. The limited availability of such capabilities to commercial entities exposes not only private information, but even critical infrastructures to tampering [48].

1.4.1 Design Objectives and Requirements

From the presented problem, the following objectives and requirements were constructed:

(i) ensuring data confidentiality, i.e. protection against leaks of classified data/technology, must be the central focus;

(ii) the CDS should be able to be comprehensively and mathematically verified for functional correctness;

(iii) the CDS should be able to be remotely deployable;

(iv) the CDS technology should be available to the commercial sector for use and independent, formal verification; and

(v) the CDS's baseline architecture should provide versatility in the application of the system to different use-cases and environments.

1.5 Research Approach

The processes planned and conducted for realizing the research herein are formalized below:

(i) *Deconstruct current CDS and trusted computing systems to determine how their limitations affect security operations* – explore the significance of formal verification and trustworthy computing systems and the criteria by which these systems are evaluated; the effect of remotely accessible, classified information, on security awareness; and how system availability, outside the DoD, affects critical information which is important to both the DoD and the commercial entities which lack the system capabilities.

(ii) *Develop a concept of the security objectives and requirements necessary to form the basis of a remotely deployable and dynamically adaptable CDS* – develop the minimal necessary security requirements which would be required to develop the CDS, such that it meets the solution objectives of being remotely deployable, widely accessible and comes with functional correctness and object security guarantees.

(iii) *Analyze use-cases and implementations of current systems and envision further use-cases to generate the requirements and the security baseline architecture for the CDS* – examine and deconstruct applications and scenarios in which the system may be used, solidify the security objectives and design potential architectures based on those objectives and adaptability to different applications.

(iv) *Investigate technologies and begin system development* – investigate existing technologies which can be used in the CDS system components and the development process; begin development:

(a) Provide diagrams based on architecture design for each use-case

(b) Consolidate architectures into one adaptable architecture

(c) Select a single use-case to focus on for implementation

(d) Develop goals, test cases and benchmarks for system analysis

(e) Implement the system – build the technology stack and develop the components and processing needed for the selected use-case

(f) Collect metrics – measure the system against the test cases and benchmarks

(g) Refine the system – refine the system to better meet the goals of the test cases and benchmarks and repeat collection and refinement until goals are met

The formalizations above have been further divided and simplified into a research methodology which is shown in Table 1.1, below.

Research Methodology	
1. Review literature and current solutions	Examine and evaluate the state-of-the-art CDS and secure computing systems as detailed in the literature.
2. Examine problems and limitations	Find and deconstruct the limitations and problems based on the literature review in step 1.
3. Hypothesize	Formulate hypotheses based on the potential solution to problems established in step 2.
4. Objectives and Requirements	Formulate solution objectives and requirements necessary to test the hypotheses in step 3.
5. Develop solution	System architecture and implementation must be developed, in addition to specific use-case applications. The architecture must meet the objectives and requirements established in step 4.
6. Evaluate solution	System architecture and implementation must be evaluated and experimented on, in addition to the evaluation of a specific use-case. Furthermore, results should be compared to state-of-the-art solutions and results gathered from step 1.
7. Repeat steps 4-7	Continue as needed for each selected use-case implementation.

Table 1.1: Research Methodology

1.6 Contributions of this Research

The purpose of this work is to systematically examine and correct the challenges of CDS systems, in particular, their lack of: (i) trustworthiness, (ii) remote deployability, and (iii) accessibility,

identified in the references above.

A novel CDS architecture, presented in [33], called *virtual CDS (vCDS)* has been analyzed. The resulting CDS systems overcome the weaknesses of current CDSs with the following contributions:

(i) The architecture allows for public analysis in terms of the trustworthiness of its instantiations – vCDS ensures trustworthiness through execution on top of a formally verified, TCB.

(ii) The resulting vCDS systems are widely accessible, including both DoD/IC and commercial sectors – accessibility to commercial sectors provided by using open-source and commodity software and hardware.

(iii) The resulting vCDS systems allow for secure remote deployability, including remote deployment in cloud environments.

(iv) The vCDS architecture can be leveraged or dynamically adapted to implement critical security mechanisms, such as Intrusion Detection System/Intrusion Prevention System (IDS/IPS) and Firewall technologies, which use tools or Indicators of Compromise (IOC) with high classification levels, such as cryptographic signatures and analytics tools, to defend computer systems with low classification levels. This is achieved by allowing these IPS/IDS of high classification levels to examine traffic in networks of low classification levels, without leaking information about the technologies with high classification levels.

(v) The vCDS architecture is compatible with big data and high performance computing environments where distributed parallel processing, including the access and transfer, of large data sets is commonplace. In the case of a big data platform, vCDS is compatible to platforms like MapReduce in that it moves the computation to the data. In the case of a high performance computing platform, vCDS can move the data to the compute nodes. The versatility of vCDS ensures scalability and low-cost implementations.

(vi) The vCDS architecture can integrate with other security solutions, such as the B2CSM architecture described in [61], which aims at achieving secure intelligence collaboration through data collection, aggregation, analysis, and threat-related information dissemination to critical parties.

(vii) A prototype instantiation of the vCDS architecture was designed and analyzed.

(viii) A stream processor implementation of vCDS has been analyzed and evaluated.

(ix) An auditing tool which seeks to validate the correctness of the implemented security configuration of all vCDS instantiations was developed. This is the first tool to analyze and audit CDS security control implementations. Presented along with this tool are the following contributions:

  (a) The need to verify the implementation of a CDS system security model is addressed.

  (b) This tool is the first and only development of a tool which audits a CDS described via an architecture description language (ADL).

  (c) This work is the first to tailor an ADL for describing a CDS system with the ability to tag components with proper labels which propagate down through the ADL, allowing the algorithm to check the constraints.

  (d) The use of the tailored ADL is extended to include labels and key words which trigger the appropriate protection models and information flow constraints in vCDS to determine whether or not to encrypt the data.

  (e) The tailored ADL is extended to provide system security modeling that is far beyond the status quo.

As evidenced by a thorough search of the relevant literature, vCDS is the first general purpose CDS system, which leverages a TCB that is provably secure and has been comprehensively verified for functional correctness and security guarantees, that can be applied to and deployed in a variety of use-cases and environments.

1.7 Overview of the Dissertation

Chapter 2: Background provides the background of technologies and information which has been deemed essential to understanding the presented research such as the access control and security models, formal verification and functional correctness, TEEs, TCBs, and CDSs. Additionally covered is previous work relating to these specific technologies and information used in this research.

Chapter 3: The vCDS Architecture examines the operational context, such as the system and threat models, the security requirements and objectives, some of the applications of vCDS systems and the architectural design of the vCDS system.

Chapter 4: The vCDS Implementation provides the details about each technology which is useful for realizing each layer of the architecture described in Chapter 3. These specific technologies are those which are used in the prototype system presented herein.

Chapter 5: Auditing a Cross Domain Solution examines the need for CDS auditing and details the implementation of a CDS auditing tool which can be used to audit vCDS instantiations.

Chapter 6: Analysis and Evaluation analyzes the experiments and measurements taken against the presented system as well as how the metrics compare to other systems.

Finally, *Chapter 7: Conclusion* highlights the contributions of this work to the field of cross domain solutions, trusted computing systems and information assurance. This chapter also introduces the future direction of this research.

1.8 Key Findings

This research details the shortcomings associated with the status quo in CDS technology which are systematically examined to determine the underlying causes as well as the effects of these problems. The development and implementation of a vCDS prototype demonstrates the feasibility of applying such a system to a real-time environment which necessitates remote deployability, trustworthy computations and proven data security, and widely available and accessible system components. The functionality of the vCDS prototype displays its adaptability to data security needs and its impact on cross domain capabilities. Furthermore, the system's functional correctness guarantees through formal verification and its provable object security yield system trustworthiness not readily available in CDS systems. Based on the experiments herein, it can be determined that security is not equated with, nor does it necessitate, poor performance. The presented metrics show that the vCDS architecture ensures an implementation which delivers both high security and high performance. Overall, it can be determined that trust is not the same as trustworthiness, that CDS systems can be remotely deployable to offsite computing environments while remaining secure, and that security through obscurity is no excuse for the lack of CDS accessibility to all sectors.

1.9 Publications

This work is partially based on the following previous work:

- [33]: Nathan Daughety et al. "vCDS: A Virtualized Cross Domain Solution Architecture". In: *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*. 2021, pp. 61–68. DOI: 10.1109/MILCOM52596.2021.9652903.

- [35]: Nathan Daughety et al. *Auditing a Software-Defined Cross Domain Solution Architecture*. In Preparation. n.d.

- [34]: Nathan Daughety et al. *Cross Domain Solution System Built on a Formally Verified Security Micro-kernel Running on Processors Enabled with Memory Encryption*. Patent Pending. 2022.

Parts of, and even entire, sections have been adopted and modified from the original publications. Additionally, a few of the diagrams in this work have been re-purposed from the previous work listed above.

Chapter 2

Background

This chapter highlights the objectives and definitions of systems and terms related to the work presented herein and introduces three common CDS categories as well as three common CDS system architectures. Additionally introduced, along with the technological components commonly found in secure computing systems, are the functions and objectives that these components aim to meet, and the distinction between the concepts of trust and trustworthiness as they pertain to secure computing systems. This section will begin with the introduction of several security models, which are categorized as follows: Access Control Models, Domain Security Models, and Data Protection Models.

2.1 Access Control Models

Access control models historically focus on subjects, or the users and processes which access and manipulate the data, and objects, or the data which is being processed. In the context of military systems, each subject is assigned a clearance or privilege and each object is assigned a classification label. In general, access control follows a subject-oriented approach where each file respectively defines a set of access-mode bits, which represent access rights, that allow access to be granted to a subject based on user and group identities or clearance level. Likewise, the subjects and processes of a sufficient clearance level can perform the same operations on an object of a particular classification level.

There are several types of access control models including, but not limited to: mandatory access

control (MAC), discretionary access control (DAC), role and rule based access control (RBAC), and attribute based access control (ABAC). A MAC model is a formal set of rules by which data access is always governed. A DAC security policy provides a user with the authority to extend access to a second user based on his or her own discretion, if and only if that second user is allowed by non-discretionary security to view the information [17]. The RBAC model is a non-discretionary access control model which defines access permissions relative to a subject's position and/or explicit rules defined by the organization. ABAC is a policy based access control model which assigns clearances to subjects and classifications to objects based on attributes of the subject or the resource.

2.1.1 Bell-LaPadula Confidentiality Model

One model that employs both MAC and DAC security policies is the Bell-LaPadula (BLP) model. BLP is a top-down state machine model with the primary purpose of enforcing data confidentiality and controlled access to classified information designed specifically to formalize domain security for the DoD [23]. The BLP model, described formally in [17], enforces the following security properties for access control:

(i) *Simple Security Property*: known as the *no read up* property; a subject at a specific security level cannot read an object at a higher security level

(ii) **-Property (Star Property)*: known as the *no write down* property; a subject at a specific security level cannot write to any object at a lower security level

(iii) *Discretionary Security Property (Strong Star Property)*: an alternative to the *-Property in which all granted access requests must be authorized by a defined protection matrix

(iv) *Tranquility Principle*:

  (a) Strong Tranquility: security levels do not change during normal system operation

  (b) Weak Tranquility: security levels never change in a way that violates the defined security policy

BLP ensures that individuals can only read objects at or below their own security level, individuals can only create objects at or above their own security level if the *-Property is used, and all access

requests must be authorized based on an access control matrix that characterizes the security level and rights of each individual if the Discretionary Security Property is implemented. Furthermore, the weak tranquility principle preserves the principle of least privilege (POLP) assuring that a subject may only access the minimum resources necessary for a particular operation. BLP's primary concern is confidentiality of data [15].

2.1.2 Biba Integrity Model

The MAC model presented by K.J. Biba in [19] focuses on the preservation of data integrity by identifying and enforcing proper data modifications. Listed below are the goals of integrity protection, with respect to the modification of data, detailed in [19]:

(i) *Protect the importance of the data* – prevent object modification by unauthorized subjects

(ii) *Enforce the need-to-modify* – prevent improper object modification by authorized subjects

(iii) *Protection of system services* – prevent the problem of the mutually suspicious subsystem by maintaining internal and external consistency such that no one system service can access the rights of another

The Biba Integrity Model aims to ensure that any higher classified objects remain isolated from lower classified objects and subjects of lower classification. All new objects which are created within a system by a subject are classified at or below the integrity clearance level of the subject which created it. Additionally, subjects can only access objects which are classified at or above their clearance level. The Biba model enforces the following security properties for the preservation of integrity:

(i) *Simple Integrity Property*: known as the *no read down* property; a subject at a specific integrity level cannot read an object at a lower integrity level

(ii) **-Integrity Property (Star Integrity Property)*: known as the *no write up* property; a subject at a specific integrity level cannot write to any object at a higher integrity level

(iii) *Invocation Property*: a subject of lower integrity level cannot request access to or invoke a subject residing at a higher integrity level

The above rules reverse some of the rules defined by BLP, however, because of the lattice structure of Biba, the security policy defined by BLP and the integrity properties of Biba can co-exist and encourage harmonious security and integrity rules.

2.2 Domain Security Models

There are two central domain security models which describe the relationship between security classifications and security domains: *multiple independent levels of security* (MILS) and *multi-level security* (MLS) [24]. Domain security models are architectures which are used to process data that reside at different security domains, that is, incompatible classification levels. The key features in these security architectures are data isolation mechanisms and information flow controls.

2.2.1 Multiple Independent Layers of Security

MILS ensures separation by assigning a single domain to a single security classification level. For example, a high side domain would be segregated from a low side domain using a boundary or separation mechanism. One example of such a mechanism is depicted in 2.15 as an air gap, or physical separation, between domains which can only be bridged by an authorized removable storage device. Another isolation mechanism is a partitioner or separation kernel. The separation kernel enforces the following objectives as defined by [4]:

(i) *Data Separation*: the address spaces of a partition must be independent of other partitions thereby preventing reads and writes of incompatible partitions

(ii) *Information Flow*: if communication channels between differing partitions must exist, then they must be authorized and controlled

(iii) *Sanitization*: all resources must be processed and sanitized before they can be accessed by a process in another partition

(iv) *Damage Limitation*: all breaches or faults must be limited to the partition in which they occur

Figure 2.1 below is a diagram showing domains of different classification levels (i.e high and low) in isolation of one another. No matter which separation mechanism is leveraged by the MILS

architecture, the following properties, otherwise known as the NEAT concept, should be upheld by the security mechanisms in a MILS system [4, 16]:

(i) *Non-bypassable*: the boundary between domains cannot be bypassed and the mechanisms guarding the boundary cannot be avoided;

(ii) *Evaluatable*: mechanisms must be small enough to be formally tested and analyzed for security and functional correctness;

(iii) *Always invoked*: mechanisms are invoked for every action, i.e. all communications and accesses are always individually checked and monitored; and

(iv) *Tamperproof*: no unauthorized modifications of the mechanisms are permitted;

Figure 2.1: Multiple Independent Layers of Security (MILS) Model

2.2.2 Multi-level Security

MLS, on the other hand, governs systems where all data is placed in a single domain, relying on trusted labels and user clearance level for proper security access enforcement [117]. MLS classification levels are not separated into high and low domains, but instead use high and low access control labels as well as security assurance policies to enforce the separation boundary. MLS systems implement the separation and security features through a Reference Validation Mechanism (RVM) such as a reference monitor. The RVM, as is similarly true with MILS security mechanisms, must adhere to NEAT [16]. Additionally, the RVM must:

(i) provide validation of all references made by processes in execution against references authorized for the subject,

(ii) assure that the references are authorized to shared resource objects, and

(iii) assure that the reference access right is correct (i.e read or write, etc.) [10].

Figure 2.2 below shows the subject who may have clearance for individually tagged data which, in this case, is data tagged with either H (for high) or L (for low), or the subject may have clearance for all data which is tagged with either an H or an L. Note that the tags of H and L on the data represent the levels at which the data are classified and the RVM provides the reference checks and separation mechanisms.

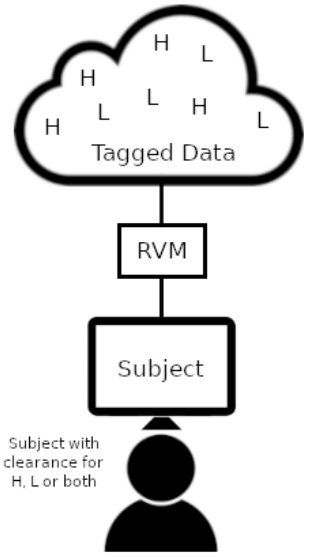

Figure 2.2: Multi-level Security (MLS) Model

2.3 Data Protection Models

Data protection models are security models which define the conditions under which authority may be granted. These types of security models have: (i) "a finite set of access rights", and (ii) "a finite set of rules for modifying the distribution of these access rights" [44]. Introduced in this section are the following models: Take-Grant Model and Capability Model. Note that portions of the following sections, including diagrams, have been adopted from "Auditing a Software-Defined Cross Domain Solution Architecture", in preparation [35].

2.3.1 Take-Grant Model

The classical take and grant scheme (a.k.a. take-grant security model) utilizes a directed graph with rules to express the conditions under which a subject can acquire authority over another object within a system. Each node in the directed graph represents a user or a data record, further referred to as subjects and/or objects depending on their relationship to one another. This system can be formulated as a graph with $n_i \in N$, where n_i is a node representing a particular subject or object, and N is the non-empty set of nodes in a graph, i.e. the set of all subjects and objects in the system. The labelled, directed edges represent one node's possession of authority over another node that is being pointed to by the edge which is formulated as $\alpha \subseteq R$ where R is the non-empty set of all access rights in the system, e.g. $\alpha \subseteq \{r = read, w = write\}$. As defined by [44, 91], the classical take-grant model employs four rules for state transitions which are as follows:

(i) Take: Let n_1, n_2, n_3 be three distinct nodes in the protection graph. Let there be an edge from n_1 to n_2 with a label γ such that $take \in \gamma$, and from n_2 to n_3 labelled $\alpha \subseteq R$. The take rule then defines a new graph by adding an edge from n_1 to n_3 with the label $\beta \subseteq \alpha$. Therefore, n_1 *takes* from n_2 the ability to execute β operations on n_3.

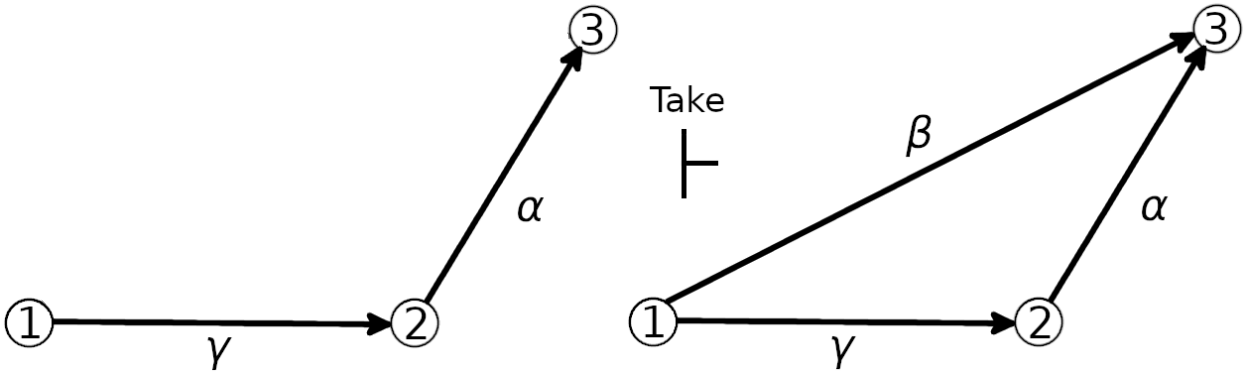

Figure 2.3: Take Rule

(ii) Grant: Let n_1, n_2, n_3 be three distinct nodes in the protection graph. Let there be an edge from n_1 to n_2 with a label γ such that $grant \in \gamma$, and from n_1 to n_3 labelled $\alpha \subseteq R$. The grant rule then defines a new graph by adding an edge from n_2 to n_3 with the label $\beta \subseteq \alpha$. Therefore, n_1 *grants* to n_2 the ability to execute β operations on n_3.

(iii) Create: Let n_1 be a node in the protection graph. The create rule then defines a new graph

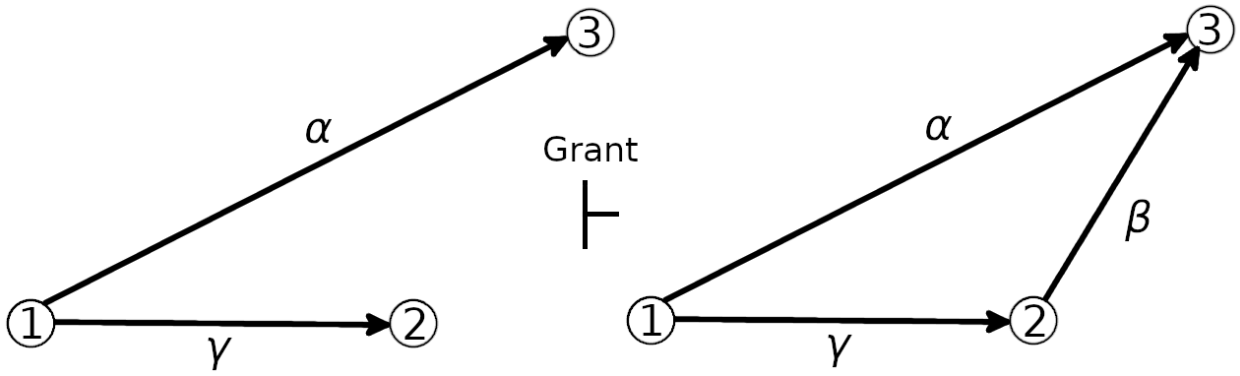

Figure 2.4: Grant Rule

by adding a new object, n_2, and an edge from n_1 to n_2 with a label $\alpha \subseteq R$. Therefore, n_1 *creates* a new object, n_2, which it can execute α operations on.

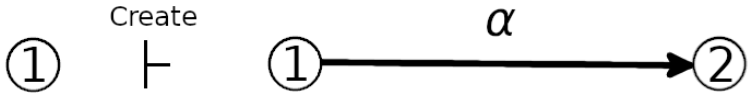

Figure 2.5: Create Rule

(iv) Remove: Let n_1, n_2 be nodes in the protection graph. Let there be an edge from n_1 to n_2 with a label $\alpha \subseteq R$. The remove rule defines a new graph by deleting a subset, β, from α. If $\alpha - \beta = \emptyset$, then the edge is removed. Therefore, n_1 *removes* its ability to execute β operations on n_2.

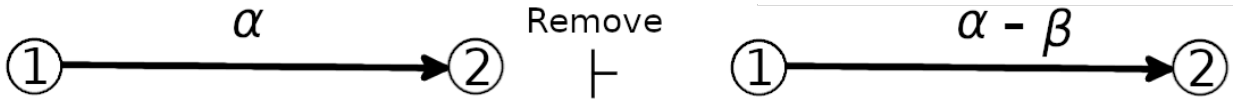

Figure 2.6: Remove Rule

While these rules will vary depending on the safety model in which they are used, the application of these rules determine whether or not rights will or can leak in a particular safety model. Furthermore, [44, 91] have shown that the take-grant model is decidable in linear time and, therefore, object security is decidable in a take and grant based safety model.

2.3.2 Capability Model

The capability protection model supports data security by using access tokens. A capability is an access token, or key, which grants a subject specific authoritative rights to a particular object.

Shown in Fig. 2.7, the capability is implemented as a data structure which encapsulates an object reference and the rights conveyed to that object [87].

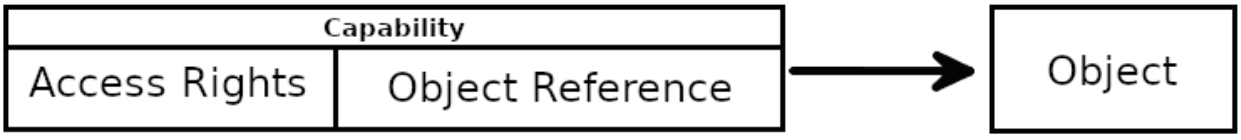

Figure 2.7: A capability is an immutable object reference which encapsulates an object reference and the access rights conveyed to the object

Capabilities operate contrary to forgeable references which identify objects but do not explicitly state which access rights are granted based on the relationship of the privilege of the subject holding the reference to the label of the object. Attempts to access an object through a forgeable reference must be validated against the access control list (ACL) by the operating system. However, these object references are mutable so they can be typecast and modified to permit access to any system object.

A capability, on the other hand, is an immutable reference that enforces the POLP by ensuring that the only way an operation can be performed on a component is by invoking the capability which is pointing to that object; thus restricting the granted rights to the absolute minimum required to perform the operation [65]. "Capabilities are the basis for object protection; a program cannot access an object unless its capability list contains a suitably privileged capability for the object" [87]. In other words, when a process is invoked, it must be handed a capability which defines the object and the operation permitted to take place on the object. This ensures that the system disallows any direct modification of access rights in the capability and that "invoking a capability is the one and only way of performing an operation on a system object" [65].

There are two additional properties of capabilities which are beneficial to the model: access interposition and privilege delegation [65, 79]. The opacity of capabilities provides the interposition of access so that, if a subject is given the reference or capability to an object, it has no method of determining what the object is, restricting the subject only to invocation of methods on the object. The delegation property of capabilities allows one subject, that owns an object, to safely delegate privileges for that object to a second subject by minting an object capability and giving it to the second subject. This allows the second subject to operate on the object without referring to the

delegating subject. The delegating subject can also mint the capability with diminished rights as is the case with most access control models.

2.4 Security Model Safety and Decidability

Note that portions of the following sections have been adopted from "Auditing a Software-Defined Cross Domain Solution Architecture", in preparation [35].

When provided with a system S, an initial graph state, s, and the set of access rules, R, decidability means that an algorithm exists which can determine whether or not S is *safe* with respect to $\alpha \subseteq R$. A safe system is a system in which it is impossible for a node n_i to acquire $\alpha \subseteq R$, which it did not previously possess, in order to reach some new state, s' [81]. In other words, if $\alpha \subseteq R$ cannot be leaked in system S, S is considered safe. Presented in Chapter 2.4.1: Analyzing Security Configurations are the results of [44, 91] who have shown that the take-grant model, and subsequently, the seL4 security model, is decidable in linear time and, therefore, object security is decidable in the take-grant security model.

2.4.1 Analyzing Security Configurations

This section provides a review of the conclusions in the works respectively found in Lipton and Snyder [91] and Elkaduwe et al. [44]. As in the literature, there are no distinctions between subjects and objects, only references to both, synonymously, through the following terms: entities, nodes and vertices. Various states of the system security model itself are referred to as the following: state, graph, system and subsystem. Finally, the following terms to denote the use of an access right are used: authority, rule, label, arc and edge.

Take-Grant Decidability

Presented below are the results of the work in [91] which examines the nonuniform safety problem; Lipton and Snyder present a concrete example of a practical protection system, i.e. the classical take-grant model, and seek to analyze its behavior to determine if a protection violation is possible [91]. It should also be understood that the following descriptions have been aggregated from their original work.

Methodology. Lipton and Snyder begin by presenting two questions, the answer to which should be known by each user of a protection system, represented by $u \in U$, where U is the non-empty set of all system users:

(i) What information, belonging to a user, u, can be accessed by another user, $u' \in U$?

(ii) What information, belonging to u', can be accessed by user, $u \in U$?

The questions are simplified by the following question, given that $\alpha \subseteq R$ where α is a subset of access rights and R is the set of all access rules: *Is it true that a node, p, can be α by a node, q?*

The objective of Lipton and Snider is to show that there are two conditions which are necessary to answer the stated question. Each of the predicates is presented below, with the following definition presented in [91]: Let G be a directed protection graph and $\alpha \subseteq R$, then a node, p, and a node, q, are *connected* if there exists a path between p and q in G, independent of directionality or α label.

Condition 1 *p and q are connected in G [91].*

Condition 2 *There exists a node, x in G and an arc from x to q, with label $\beta \subseteq \alpha$ [91]. In other words, there exists a node x that has access to, i.e. α's, node q.*

These conditions determine the safety of a system, S, with respect to $\alpha \subseteq R$. Therefore, these conditions serve to prove the decidability of a system, i.e. to determine if one node could acquire a particular authority over another node which it did not previously possess. This work is further examined and applied in Chapter 6: Analysis and Evaluation.

2.5 Cross Domain Solutions

As previously stated in Chapter 1: Introduction, a CDS is a system which provides for the access to and/or transfer of data across domains of differing classification or trust levels. One of the mechanisms a CDS enforces between the discrete security domains is a security policy. The security policy is used to enforce the security models described above by upholding trust or classification level, information access and transfer regulations, and data ownership within a domain.

There are three main CDS categories that fit the purpose and needs of different cross domain devices as defined by [37]. These categories are Access Solutions, Transfer Solutions, and Multi-level Solutions. Additionally, each cross domain type has several different implementation architectures. Three of these implementation architectures are focused on and categorized by [117]: Physically Isolated Domains, Partitioned Workstations, and Data Diodes. The following sections detail the use and background of each.

2.6 CDS Categories

The following types of CDSs govern the central focus and use of the system. While a CDS could be set up to allow for both access to and transfer of data, if the system requirements do not call for both capabilities, implementing such a system would likely add more threat risks. It can be noted here that access and transfer solutions typically rely on an MILS domain security architecture whereas a CDS which includes both access and transfer capabilities relies on an MLS domain security architecture.

2.6.1 Access Solutions

The function of an access CDS is to allow the reading and manipulation of data, applications, or platforms in different security domains [37, 100]. For example, an access solution may grant read access in higher domains to data residing at a lower trust level [48]. Access solutions prevent data overlap between security domains by enforcing separation policies between trust boundaries [117]. Essentially, an access solution is much like a "dumb terminal" as it only allows a user to view and/or use the data but does not permit movement or writes to the data. A primitive mandatory access control (MAC) model, such as Bell-LaPadula [17], is used to implement rules such as *no read up* where a user with a low trust level cannot access data with a higher trust level. The purpose of this type of solution is to allow access to information while preventing data spills from high trust to lower trust levels. As previously stated, this solution typically relies on the MILS domain security model.

Depicted below, in Figure 2.8, is one example of an access solution architecture where the high and low domains are separated by a CDS. The subject, or user, is interfacing with Domain High and

wants to access data residing in Domain Low. This form of access CDS is a "browse-down" CDS because it only allows the subject to access data at or below the classification level of the domain to which the subject is connected. In this case, the access CDS does allow for communications. However, the only communications permitted are access requests input by the user to the CDS and the system responses output back to the user. No data is transferred between domains.

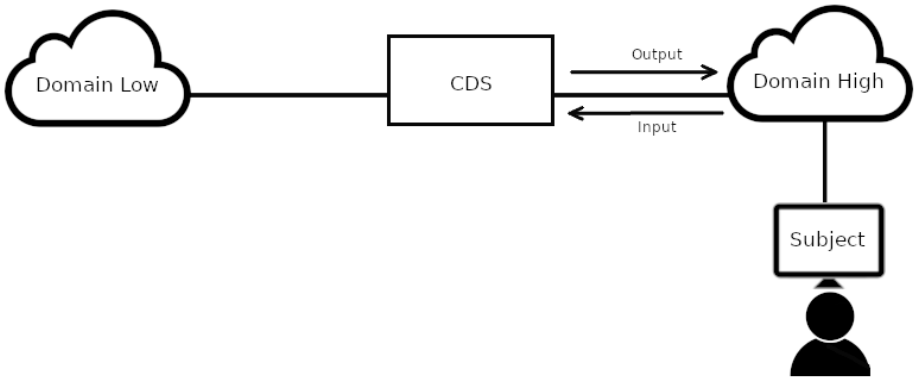

Figure 2.8: Access Solution

While "browse-up" and multi-access solutions exist in theory, these architectures introduce risk and, in a lot of cases, break the defined security models. For example, in a multi-access solution where an authorized subject can access two or more domains at a time, not only do common risks such as the threat of hypervisor and hardware attacks exist, but trust is placed in the hands of the subject not to make any inadvertent or purposeful errors such as inputting or copying information into the wrong domain. The multi-access solution should only be used when necessary and should include each of the above risks as a part of the threat model. In a browse-up solution, the subject would be accessing high classified data from an untrusted, low environment. This introduces unnecessary risks such as a threat actor intercepting high classified data or even sending unauthorized commands to be executed in the high domain and therefore should never be used [24].

2.6.2 Transfer Solutions

The goal of a transfer CDS is to enforce the security policy for the movement of information between different security domains [37, 100]. Transfer devices generally leverage a copy function rather than actually relocating the data to a different domain [48]. In order for a transfer solution to control

connections between domains and determine what function to execute on the information, the data are examined in accordance with security policies defined by the implemented technologies [117]. Security policies regulate the assurance and trust of data before copying it to a higher domain and they restrict higher data from entering into lower security domains. Furthermore, small scale transfer solutions can be implemented as uni-directional using a single data diode to ensure that data only flows in one direction (e.g. low to high; high to low), or bi-directional using a pair of data diodes with data guards to separate data flow control [24]. Data diodes are further discussed in Chapter 2.7.3: Data Diodes and displayed in Figure 2.17. Like access solutions, this solution typically relies on the MILS domain security model.

Figure 2.9 shows a uni-directional "data export" solution where certain data which resides in Domain High is permitted, and optionally filtered, to be transferred to Domain Low. Conversely, Figure 2.10 shows a uni-directional "data import" solution where certain data which resides in Domain Low is permitted to be transferred to, or imported into, Domain High. Finally, a bi-directional transfer solution is depicted in Figure 2.11. The bi-directional solution provides two different avenues of communication: one which carries data away from Domain High and one which carries data to Domain High. By themselves, these individual avenues act as uni-directional data export and/or data import solutions.

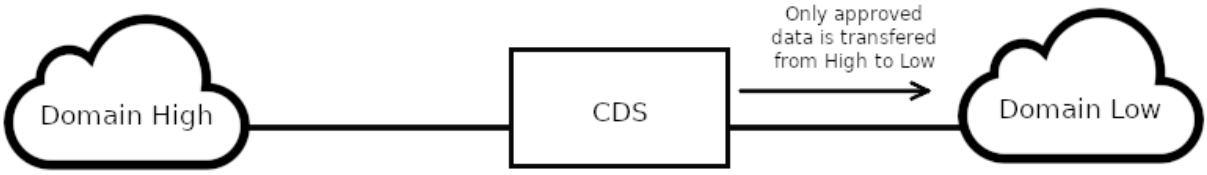

Figure 2.9: Transfer setup from High to Low

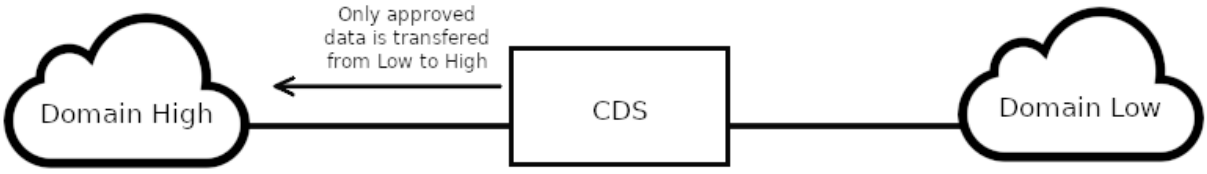

Figure 2.10: Transfer setup from Low to High

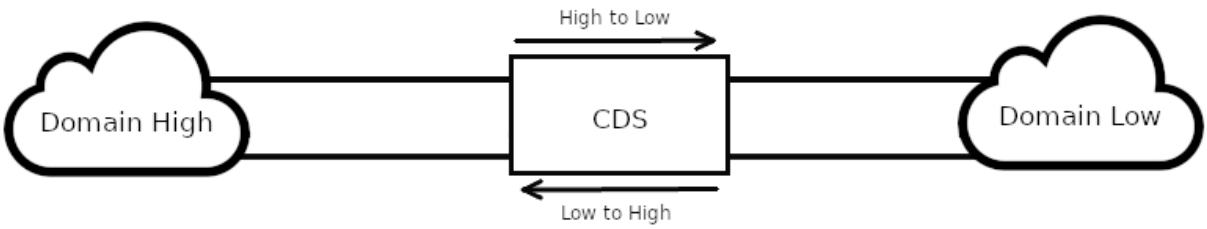

Figure 2.11: Bi-directional Transfer Solution

2.6.3 Multi-level Solutions

Multi-level solutions rely on the MLS domain security model to help provide both access and transfer capabilities through trusted labeling on a single or multi-domain setup. This allows data to be stored at the proper classifications and restricts access to the data based on credentials [100]. Like access solutions, multi-level CDS systems utilize MAC policies, though more evolved, to enforce user authenticity and privilege level. One additional security feature of multi-level solutions is that subjects are not permitted to directly access the security domains allowing subjects of differing clearance levels to work together with different data in the same environment [24]. While multi-level systems improve performance and eliminate the need for several technologies like filters and guards, it is impractical to develop, not only because of implementation difficulty and cost [117], but because of the threat model with a single point of failure and the fact that this type of setup restricts the environments in which it can be used. It is also important to note that based on the threat model and the environment, multi-level CDS architectures have historically lacked in system assurance [41].

Figure 2.12, below, diagrams a multi-level CDS. As shown, Domain High and Domain Low are isolated from one another by the CDS. As discussed in Chapter 2.2.2: Multi-level Security, the RVM resides inside the CDS to provide reference checks and separation mechanisms. Figure 2.13 dissects the internals of the CDS, showing an RVM between the domains, and an RVM between the tagged data and the subject. In this case, the subject can be authorized to access high data, low data, or both high and low data depending on the subject's clearance level. Note that the subject cannot directly access either of the security domains.

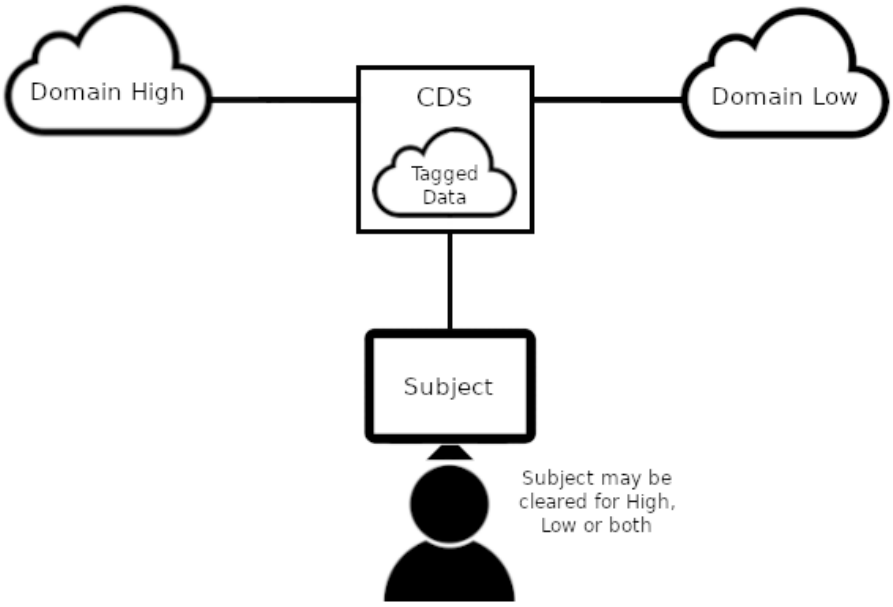

Figure 2.12: Multi-level CDS Architecture

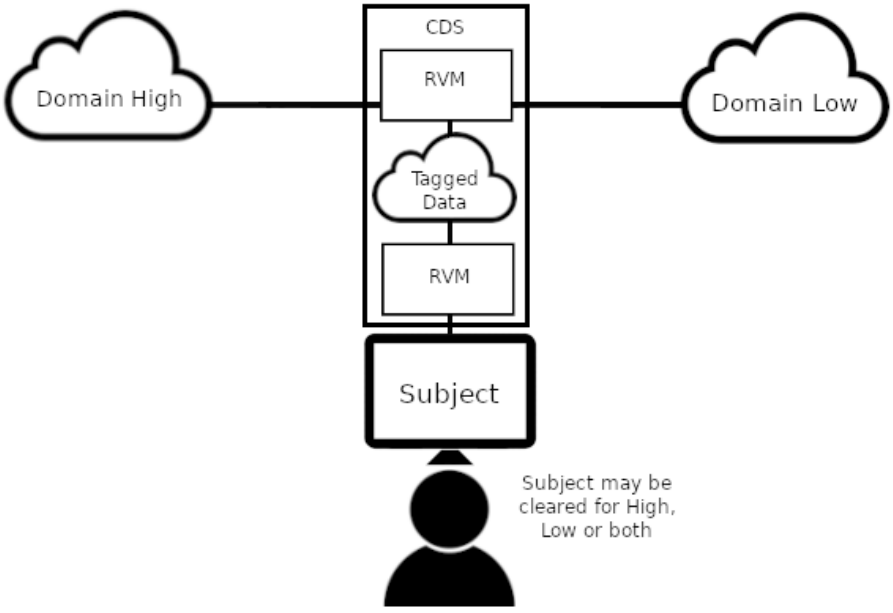

Figure 2.13: Multi-level CDS showing RVM

2.7 CDS Architectures

This section introduces three common CDS architectures which are used in current solutions. Additionally provided for better understanding is a diagram for each. Each of them focuses on data confidentiality protection and are listed as follows: Physically Isolated Domains, Partitioned Workstations, and Data Diodes.

2.7.1 Physically Isolated Domains

The physical isolation approach to domains separates individual security domains to maintain only one classification or trust level per domain. Therefore, in terms of an access CDS, if an authorized operator needs to access two different domains, he or she would need to access two different physical infrastructures. Shown in Figure 2.14 is one implementation, for the segregation of information processing environments in this way, that is sometimes referred to as a "swivel-chair" setup because the operator could effectively swivel his or her chair to access each workstation [117]. Some implementations use a keyboard, video monitor, and mouse (KVM) switch to access different domains from a single terminal. The most common implementation of this architecture as a transfer CDS is called an air gap, depicted in Figure 2.15. Air gapped domains are physically separated domains with the ability to transfer data using only a removable device. While this architecture is the most common, one of the key concerns with it is that it does not allow for remote access or maintenance.

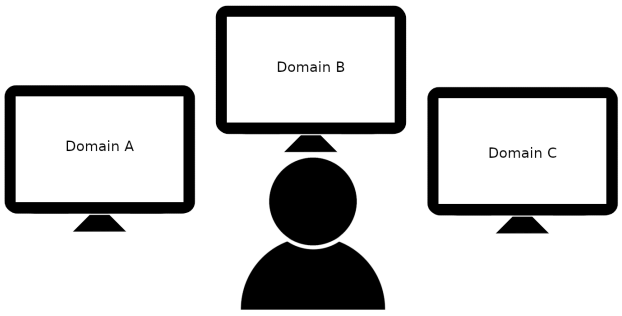

Figure 2.14: Isolated Domains (e.g. "swivel-chair")

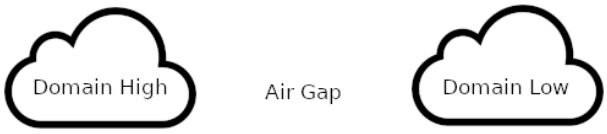

Figure 2.15: Physically isolated domains separated by an Air Gap

2.7.2 Partitioned Workstations

The partitioned workstation architecture relies on domain virtualization on top of a single host. The host regulates the separation between domains by running a corresponding virtual machine (VM) for each domain trust level. Partitioned workstations not only slow down performance due to the need for hardened operating systems, but they are susceptible to common hardware attacks and VM-specific vulnerabilities [117]. The main problem comes down to incomplete assurance promised by state-of-the-art TCBs, not to mention the common neglection to use a TEE to protect data confidentiality.

Figure 2.16 below shows one possible arrangement of the partitioned workstation architecture with a base hardware layer that contains multiple network interface cards (NIC) where each corresponds to a different security domain. Above the hardware layer is the host operating system whose hypervisor runs a virtual operating system for each security domain. A subject with proper clearance and authorization can access each domain from the same workstation.

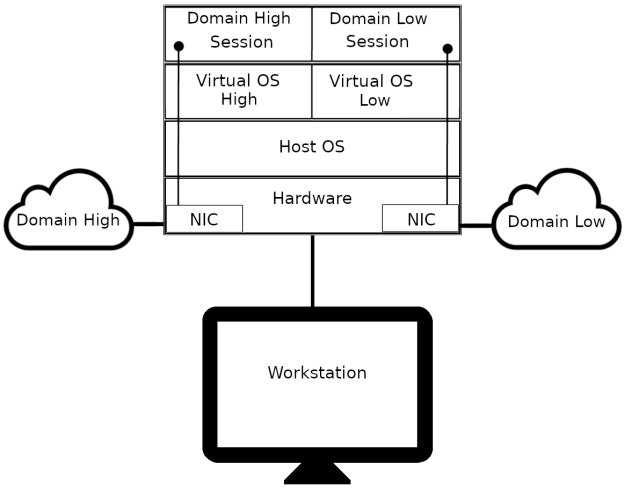

Figure 2.16: Partitioned Workstation

2.7.3 Data Diodes

Shown in Figure 2.17 is the data diode architecture. Data diodes, analogous to electrical diodes, are devices that restrict the flow of data to one direction. In a circuit diagram, the diode symbol contains an arrow which points in the direction of conventional current, i.e. anode to cathode; the arrow contradicts the flow of electrons, i.e. cathode to anode. In this work, depictions of the diode symbol are used, with the arrow pointing in the direction of conventional current, in accordance with the following citations: [38, 49, 52, 71, 77, 90, 106]. An example implementation of a data diode might be in a system where a low side domain is permitted to transfer information to a high side domain but the diode would prevent the high side from leaking data to the low side. One limitation with uni-directional solutions is that it does not implement the TCP/IP stack (and breaks the majority of network protocols) which would be essential to remote deployability [113].

[24, 117] detail bi-directional implementations with a pair of data diodes which allow for the transfer of information in two directions with the addition of content guards that filter data based on a directional security policy. While this solution allows for secure, bi-directional data transfer, there is a key problem. The implementation of content guards in this solution does mitigate the covert passing of malicious data but it does not always prevent data spills [117].

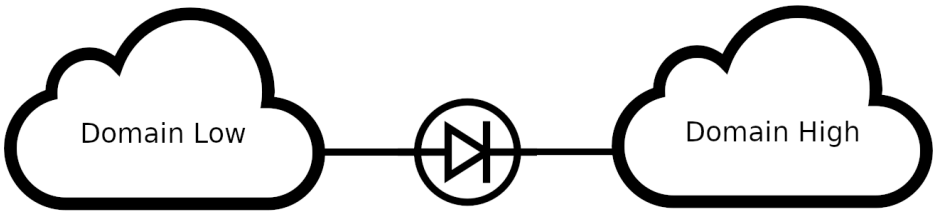

Figure 2.17: Data Diode

2.8 Trusted Execution Environments

A Trusted Execution Environment (TEE) "is a secure, integrity-protected processing environment, consisting of processing, memory, and storage capabilities" [14]. While there is some discrepancy in TEE definitions and implementations outlined in [109], the goal of a TEE is to improve security through runtime-state protection and data restriction to ensure no sensitive data leaves the TEE. This can be achieved through the implementation of a dedicated VM [53] or an environment that

runs alongside but is isolated from the main OS [95], such as a hardware-based enclave supported by security co-processors with the common goal of sensitive data restriction and isolation. A TEE must define mechanisms to "securely attest its trustworthiness", not allowing untrusted code or operations to cause, enable, or prevent any code execution, traps, exceptions, or interruptions [109]. [109] introduces five building blocks of TEEs:

(i) *Secure Boot*: assure that only the correct, unmodified code can be loaded;

(ii) *Secure Scheduling*: assure a balanced and efficient coordination between the TEE and the rest of the system so that tasks running in the TEE do not affect the responsiveness of the main OS;

(iii) *Inter-Environment Communication*: interface for assuring authenticity in the communication between the TEE and other system components;

(iv) *Secure Storage*: data confidentiality and integrity is preserved in storage; and

(v) *Trusted I/O Path*: protect authenticity and confidentiality of the communication between the TEE and peripheral devices.

TEEs are used to protect complex and interconnected systems that require a high level of security with protection against both physical attacks to main memory, and software-based attacks.

There has been an emergence of TEE technology in commodity systems. The most common hardware based TEE systems includes Intel's Software Guard Extensions (SGX), AMD's Secure Encrypted Virtualization/Secure Memory Encryption (SEV/SME), and ARM's TrustZone. While the technology is still maturing, TEE systems have evolved greatly from proprietary solutions to a standards-based approach for use in PC's, mobile devices, and other Internet-connected devices [3]. Furthermore, the push for a viable TEE solution for use in cloud computing environments has begun.

2.9 Trusted Computing Bases

The core element for any security solution is the TCB. Historically, a TCB has referred to several types of computing bases including, but not limited to: a security kernel, a trusted operating

system (TOS), a security filter or individual access control validation mechanism, or an entire trusted computing system [36, 103]. The integral components included in modern TCBs are the *reference monitor* and *security kernel*. The reference monitor serves to provide complete mediation of access, validating access to all objects by authorized subjects. The security kernel provides the lowest level of software to hardware abstraction and employs mechanisms to enforce security at differing trust boundaries.

In order for a TCB solution to be robust, its components must be sound. While the design and construction of TCB implementations have changed some over the years, the objectives and mechanisms necessary to determine a sound TCB, as per [103], have remained:

(i) *Controlled*: the system should provide and properly implement access controls

(ii) *Guarded*: the system should provide self-protecting mechanisms such that no system modification or interference can take place

(iii) *Formally Verifiable*: the system should be designed with the intent to be functional correctness verified using formal and/or semi-formal methods

In other words, the TCB is responsible for isolating security-bound components and upholding the security policy which describes "the conditions under which information and system resources can be made available to the users of the system" [103].

Therefore, a TCB, commonly implemented with a combination of hardware, firmware, and software, is the totality of protection mechanisms responsible for enforcing a security policy [36]. These protection mechanisms ensure that any system components which are not included in the TCB should not need to be trusted for the whole system to remain fully protected [36]. While the TCB definition in [36] remains the standard, emphasis in TCB design, as of late, is on assurance and trustworthiness through formal verification of trusted computing systems [28, 102, 103].

2.10 Trust vs Trustworthy

The DoD Trusted Computer System Evaluation Criteria (TCSEC), first released in 1983, states that the "assurance of correct and complete design and implementation for these systems is gained mostly through testing of the security-relevant portions of the system" where the security-relevant

portion refers to the TCB [36]. Assurance is measured as confidence in the TCB to meet explicitly identified security expectations [31]. The Common Criteria (CC), which replaced the aforementioned evaluation criteria in 2005, introduces the Evaluation Assurance Level (EAL) which goes beyond security testing to introduce formal specifications and formal verification of computing bases [31]. The goal of an EAL measurement is to exercise the distinction between trust and trustworthiness, where a trusted system is a system that is *believed* to be capable of handling a security event and a trustworthy system is *proven* to be capable of handling a security event. In other words, trust can be broken but trustworthiness has been formally proven and cannot be broken. However, as shown in [11], vendors often find loopholes to game the evaluation system, rendering the "illusion of stability" without actually proving trustworthiness. This fact shows the importance of the following system design objectives: the CDS should be able to be comprehensively and mathematically verified for functional correctness and security, and it should be available for independent verification to ensure trustworthiness.

2.11 Formal Verification

Formal verification (FV) is a rigorous process that uses mathematical reasoning to verify that the design's correct behavior is preserved by the implementation [110]. Specifically, functional correctness verification is often conducted by systems of formal, mathematical logic which reason about the correctness of an algorithm or computer program. Hoare logic is one such formal set of logical rules which cover certain "programming language constructs such as loops, exceptions, expressions with side effects, and procedures, together with clear presentations of their soundness and completeness proof" [105]. Hoare logic can be used to show *refinement*. A refinement proof takes the properties of the abstract model and establishes their correspondence with the properties of the refined model representation of the system. In other words, refinement guarantees that any security properties which are proven in the abstract model, using Hoare logic, will also hold in the source code [78, 79].

For the FV process to be conducted, a software specification must be evaluated based on an extensive criteria. For example, when a software product is measured by the CC's EAL, it is measured on a scale from 1 to 7, where 1 means functionally tested without security, levels 2-4

requires structural and methodical design and testing, level 6 requires semi-formal verification, and level 7 claims full, formal verification of functional correctness. Therefore, the higher the EAL, the greater the assurance or confidence in the system. During the evaluation, the software product ideally undergoes exhaustive exploration of all input values and all possible states.

The need for FV is apparent as [1] shows that one third of all software faults take 5000+ execution-years to be revealed. While that means one third of all software bugs are hidden from common execution paths, it also means there are still several vulnerabilities in the system. FV is valuable because a formally verified system is a system that is proven to be correct and without bugs. The key metric in TCB assurance evaluation is the size and complexity of the code base [36]. As the size of the code base grows, added effort, measured in time by [28], grows astronomically. Therefore, FV is not only a rigorous process, but also an expensive one.

2.12 Guards

A guard is a trusted "application which is responsible for analyzing the content of the communication and determining whether this communication is in accordance with the system security policy" and is often implemented in CDS systems [4]. Typically, guard components are implemented as filters which can modify or delete messages, verifiers which can check data integrity, and isolation mechanisms to separate the data. Guards are located at the border of a component's or domain's ingress and egress data channels. Described in the sections below are three such guards: the Content Guard, the Integrity Guard, and the High Assurance Guard (HAG).

Content Guard

The content guard "is a filter that separates the originally transmitted data into legitimate information and illegitimate information", allowing only the legitimate information to pass through [60]. In the context of a CDS, content guards filter out high data, for example, from documents that are transmitted to a low side domain. This filter relies on complex deep inspection and pattern recognition algorithms or manual labelling to remove any data that should not be present based on the domain it resides in. This guard is useful for preventing the covert passing of malicious data, however, due to the complexity of the pattern recognition algorithms and/or human error,

this guard is not as reliable in preventing data spillage [117].

Integrity Guard

Integrity guards, described in [60], are filters designed for storage subsystems where, in the case of an integrity output filter, data are written or transferred out to be later read back in while preserving the integrity of the data. An integrity input filter works in the opposite direction where data are read in and later need to be written back out in an integrity preserving way. In a cross domain pipeline, the integrity preserving mechanism works by first calculating a unique identification tag, much like a hash, just before the data exits the security domain. The unique ID is then checked against the data by the corresponding filter upon entry into a second security domain. If the data returns, it will be re-examined by the output filter to ensure that the data have not been modified. This filter is especially useful for preventing covert passing of malicious data as well as data modification.

High Assurance Guard

The National Institute of Standards and Technology (NIST) defines a HAG as "an enclave boundary protection device that controls access between a local area network that an enterprise system has a requirement to protect, and an external network that is outside the control of the enterprise system, with a high degree of assurance" [83]. In other words, a HAG is a data transfer device that ensures that authorized data, and only authorized data, is transferred from one domain to another [63]. This transfer guard ensures that data, processes, or devices do not violate security policies or interfere with one another. [117] shows that HAG's are efficient at preventing data leakage, broken protocols, hidden or malicious content including steganography and other embedded files, and zero day attacks.

2.13 Protocol Adapters

CDSs leverage environment and use-case appropriate security functionality which often includes mechanisms to protect against protocol-based threats. One such component is called a protocol adapter (PA). PAs are processes that handle CDS communications with external entities through

physical communication interfaces. PAs often enforce security actions such as authentication schemes, packet filtering and/or protocol breaking/termination. Typically, PAs process protocols in the Open Systems Interconnection (OSI) model layers 5-7. Some examples include HyperText Transfer Protocol (HTTP) [70], Advanced Message Queuing Protocol (AMQP) [9], Message Queuing Telemetry Transport (MQTT) [97], Distributed Network Protocol 3 (DNP3) [42], Inter-Control Center Communications Protocol (ICCP) [121], Secure File Transfer Protocol (SFTP) [121], and Transmission Control Protocol (TCP) [121]. Protocol adapters are further discussed in Chapter 3.7: Discussion: Protocol Adapters for their enforcement of data safety.

2.14 Related Work

Researchers have been trying to develop provably secure operating systems (PSOS) using a design methodology that "facilitates the formal statement and proofs of relevant system properties", since the mid-1970's [102]. The concept of a security kernel materialized as a means to derive system security from a formally verified foundation [75]. One such general purpose security kernel [112] was evaluated at the highest level of the NSA's TCSEC – Class A1: Verified Protection. [62] is a system which has been developed on top of the security kernel [112], with the intention of being evaluated at EAL7+. [119] is a trusted operating system which has been evaluated at EAL6. Another notable solution is the XTS-400, an MLS system based on the secure trusted operating program (STOP) operating system which was evaluated at EAL5 [122]. Additional systems geared more for military use include [93, 101, 123, 124].

In an effort to improve performance and make functional correctness more easily verifiable, a family of security microkernels [67], being the critical portions of a kernel, were developed. More recent research has led to the "comprehensive formal verification, including a functional correctness proof of the implementation and a complete proof chain of high-level security properties down to the executable binary" [67, 79]. The only system which has undergone such evaluation and is acclaimed as the fastest performing, most verifiably secure, microkernel in the world is seL4 [65]. This study is very closely related to [65] but is unique because of the utilization of the technology in a novel CDS architecture.

Commercial CDS systems and technology companies, such as the aforementioned [40, 50, 54,

124], along with the data diode solutions [69, 71, 74, 120, 123, 127], whose systems are managed and tested by the DoD/IC, seem to adhere to a current standard of security through obscurity as the specifications and evaluation results are not available for independent verification. It is also important to note that the above respective evaluations, some of which are EAL7+, may not express the system confidence expected because these vendors have historically cheated the system. As a consequence, the specifications and evaluations of such systems may not be sound. This can be justified by the fact that evaluations based on the CC are problematic because:

(i) "usability is ignored";

(ii) the paperwork, not the product, is the test subject; and

(iii) these schemes are known to "squeeze a very volatile and competitive industry into a bureaucratic straightjacket, in order to provide purchasers with the illusion of stability" [11].

From a technical perspective, the presented vCDS is designed to leverage seL4, the previously mentioned state-of-the-art microkernel, which has a number of formally verified properties [65, 67, 79]. Each of these properties, and the proofs of enforcement [32], are available for independent verification.

As evidenced by a thorough search of the relevant literature, vCDS is the first general purpose CDS system to be designed and developed, which leverages a TCB that is provably secure and has been comprehensively verified for functional correctness and security which can be applied to a variety of use-cases and applications. Recall that existing CDS systems are either highly specialized systems which cannot be applied to other environments without incurring unreasonable modification and maintenance costs or they are ad hoc solutions built upon TCBs which lack the assured security of the vCDS system [13, 26, 47, 57].

Chapter 3

The vCDS Architecture

This chapter outlines the vCDS system and threat models along with the security requirements, the architecture and pipeline of vCDS and a security analysis of the architecture. Note that portions of the following chapter, including diagrams marked with "© 2021 IEEE", have been adopted from "vCDS: A Virtualized Cross Domain Solution Architecture", © 2021 IEEE [33].

3.1 System Model

The vCDS system model is based, in part, on the Bell-LaPadula (BLP) model whose primary concern is data confidentiality [15]. BLP ensures that subjects can only read objects at or below their own security level, subjects can only write to objects at or above their own security level, and all access requests must be authorized based on an access control matrix that characterizes the security level and rights of each subject. Furthermore, BLP preserves the *principle of least privilege* assuring that a subject may only access the minimum resources necessary for a particular operation. In fact, the BLP model is a CDS model in that is was designed to confront the "operational needs to move information between networks of different classifications and sensitivity levels" [18]. vCDS faithfully implements this model because it has been rigorously studied to ensure that confidentiality of data is protected, despite compromises to other security objectives such as integrity, availability, authenticity, and/or accountability, and it allows data to flow safely between certain isolated security domains [17, 18].

3.2 Threat Model

The threat model focus is to prevent the compromise of data confidentiality through various information disclosure attacks, specifically focusing on attempts to leak data from the high side to the low side entities. Common vectors by which attackers are able to create data spillage in sensitive computer systems include covert channels, unauthorized access to resources, and disrupting the flow of data by forcing movement against the intent [24]. Furthermore, the threat model includes powerful adversaries who are given insider access and, therefore, are granted privileges that an outsider would not possess [109]. The attack surface includes all communications between the secure enclave and the high side target [126]. For example, powerful attackers who are able to spoof the secure enclave to intercept high-trust communications are a grave threat to data confidentiality. The model also "includes all software attacks and the physical attacks performed on the main memory and its non-volatile memory" [109] and also, physical bus attacks.

Vulnerabilities	Mitigations		
	TEE	TCB	Guard
Side Channels	[7, 58]	[55, 65]	[60]
Disclosure, spillage, manipulation	[5, 76, 89]	[65]	[60]
Logic Errors		[65, 125]	[131]
VM Breakout	[5, 22, 89]	[65]	
Control Hijacking, Injection	[5, 22, 76]	[65]	
Communication/Spoofing	[5, 22]	[65, 125]	

Table 3.1: Threat Model Matrix

Table 3.1 provides insight into the hardware and software vulnerabilities defined in the vCDS threat model as well as references to the various mitigations employed by each technology component (including an optional guard) of vCDS which are further discussed later in Chapter 3.6: Security Analysis.

Table 3.2 provides specific insight into different cache mitigation strategies. Note that Rowhammer-induced memory flips can still halt the machine necessitating a power cycle [58]. Additional common vulnerabilities and exposures (CVE) can be found for the TEE though appropriate mitigations are claimed in [8].

Vulnerabilities	Mitigations
Electromagnetic Attacks (e.g. Rowhammer/RAMBleed)	[7, 58, 85]
Timing Attacks (e.g. Prime and Probe)	[55, 130]
Transient/Speculative Execution (e.g. Spectre, Meltdown)	[8]

Table 3.2: Cache Attack Mitigation

3.3 Security Requirements

There are three essential property measurements expressed in a comprehensive CDS protection plan, as detailed by [111, 126]:

(i) *defensive effectiveness*: timely and accurate prevention and response, system flexibility;

(ii) *confidentiality*: data content protection from unauthorized parties; and

(iii) *operational relevance*: usable and accessible in multiple operational environments.

These three measurements re-enforce the primary security objective to protect confidentiality of data. While this system is equipped to protect and enforce other objectives in the CIAAA, these are orthogonal to the main objective. In order to achieve data confidentiality and enforce the designed protection plan in the presence of the threats mentioned above, the security design needs to ensure the following which are discussed in Chapter 3.6, Security Analysis: Hardware Protections and Memory Encryption, Trustworthy Components, Decidable Object Security and Staticity, Computation Isolation, and Data Flow Restriction.

3.4 Architectural Design

The following subsections detail two of the basic architectural designs which vCDS can provide. It can be noted that, given the versatility of the vCDS pipeline, both of these designs have optional components and have dynamic arrangements for utilization depending on the use-case to which

these architectures are applied. The two architectures are the General Purpose Architecture and the Co-processor Architecture.

3.4.1 General Purpose Architecture

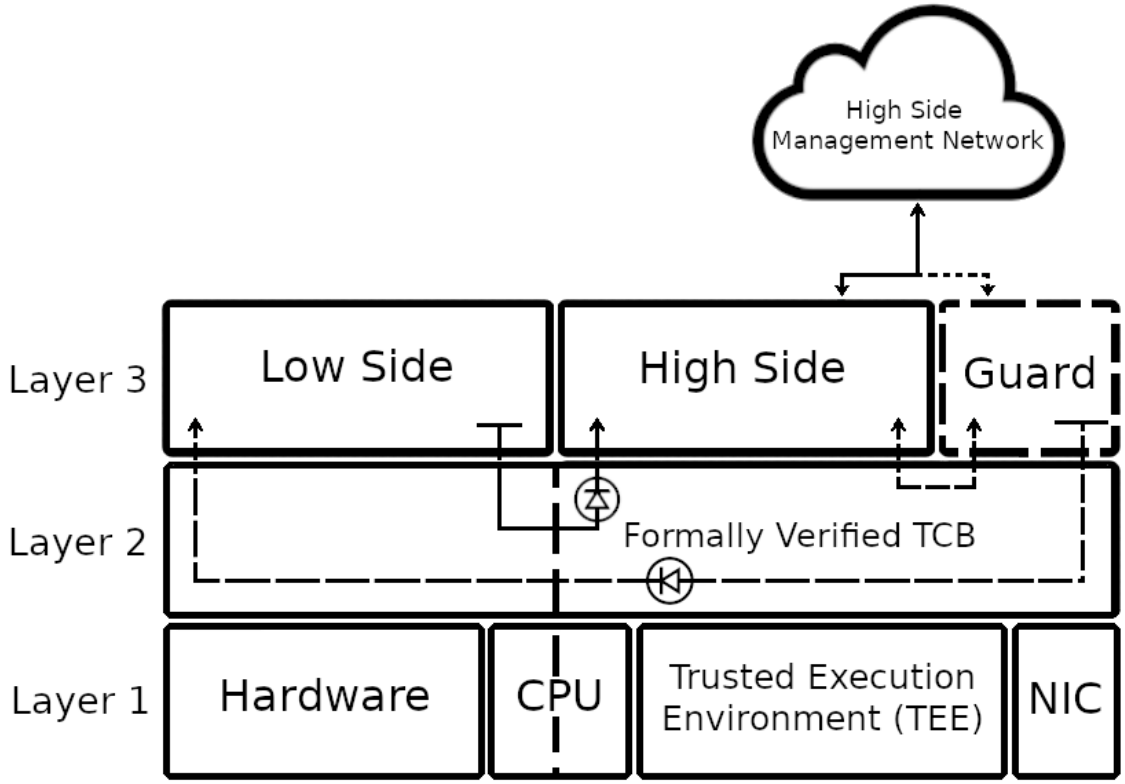

Figure 3.1: General purpose vCDS architecture © 2021 IEEE

In order to achieve the security objectives mentioned above, the following CDS security architecture is presented, and depicted in Figure 3.1, with three layers which include the (i) Hardware, (ii) Computing Base, and (iii) Software Components.

Overview

Within the functional block diagram there is a dotted line, which is referred to as the delineator, that separates all Low Side operations from all High Side operations. The Guard component is optional and therefore is enclosed in a dotted block. Additionally, there is an optional High Side Management Network (C2) for protected communications with the High Side components. Elsewhere in the diagram, all solid, directed lines refer to required communication channels, where

as all dotted, directed lines refer to optional communication channels.

Hardware

Found in Layer 1, the hardware-based TEE capability corresponds to all High Side (Layer 3) data and computations while the Low Side (Layer 3) leverages the basic hardware capabilities. The basic hardware, shown to the left side of the delineator in Layer 1, contains very few added security components as it is the base hardware of the low security boundary. The TEE, to the right of the delineator, is specifc to all High Side operations. Within the hardware level there is also a central processing unit (CPU) and a NIC. The CPU manages all processing of data for the components residing in Layers 2 and 3, using strict isolation mechanisms. The NIC functions only with the High Side and Guard on Layer 3.

Computing Base

In Layer 2, on top of the TEE, is a Formally Verified TCB, which serves as the fundamental component in this system and allows vCDS to be easily adapted to different implementation requirements and CDS architectures. The TCB ensures integrity and confidentiality through a trustworthy code base and access controls providing isolation and *staticity* which is further discussed in Chapter 3.6: Security Analysis.

Software Components

In Layer 3, there are two separate processes running on top of the Formally Verified TCB: one that represents the High Side and one that represents the Low Side. The Low Side component manages the low classified data while the High Side manages the higher classified data.

The High Side leverages only one device and driver in order to communicate with a C2: a NIC. The High Side tunneling strategy allows the isolated high enclave to communicate with components of the same classification which, in this case, is an optional Guard and the C2. Functions of the latter are relative to the system, for example, the C2 in a CDS with the primary purpose of analyzing and filtering network traffic would need to regularly push signature updates and blocking actions to sensitive intelligence sensors and traffic analyzers as well as receive alerts should a malicious packet be discovered. In a distributed computing CDS system, the C2 would regularly push code to each

computing node which would then run an operation defined by the code and send the results back to the C2.

Before data passes from high to low, there is an optional Guard that functions as a filter to ensure that no high sensitive data are passed to low. This Guard can have several additional functions like sending alerts back to the High Side, but in any case, it has bi-directional communication with the High Side as it resides at the same classification level. This is an important distinction from the rest of the data flow model because the direction of data flow is restricted with a *data diode* [1] as depicted with a diode symbol in Figure 3.1. The data diode is further discussed in Chapter 3.6: Security Analysis.

Information Flow

The flow of information depends heavily on the implemented use-case, but, in general, the flow within vCDS begins once the data are within the Low Side and moves as follows:

(i) The data will flow through the Formally Verified TCB via a data diode before reaching any High Side components.

(ii) Once within the High Side processing environment, the data may flow out through an egrees route, which passes through the TCB, to the Guard or in through an ingress route, also passing through the TCB, to the High Side from the Guard. If required by the use-case, the bi-directional channel between the Guard and the High Side can be separated into dual routes which combine to effectively meet the requirements for a bi-directional channel but with more strictly enforced access controls. Recall that all communications with the Guard are optional as the Guard itself is optional depending on the use case.

(iii) The data will then optionally flow through the Formally Verified TCB via a data diode, from one of the High Side components, and back to the Low Side.

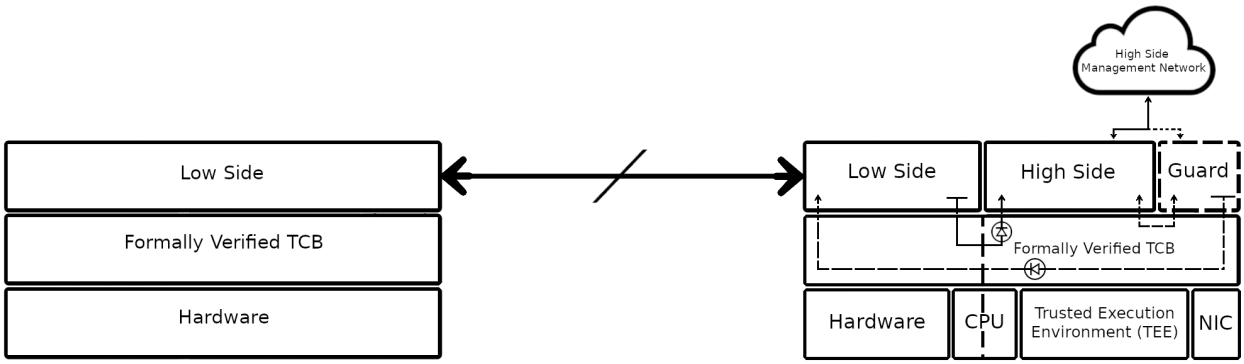

Figure 3.2: Co-processor vCDS architecture

3.4.2 Co-processor Architecture

Figure 3.2 is a schematic which illustrates the possibility of further isolation of low components. This architecture leverages three central, high-level components: Untrusted Host Processor, Protected Co-Processor, and the Bi-directional Bus.

Untrusted Host Processor

On the left portion of the diagram is the host processor. The host processor runs a strictly low security domain on top of the Formally Verified TCB. Additionally, the host employs basic hardware components and may or may not employ additional security mechanisms or controls. This domain may or may not have a NIC for communication with other low security environments, but in any case, this host is the isolated entry point for this CDS architecture.

Protected Co-Processor

The right portion of the diagram depicts the protected co-processor. The co-processor is an implementation of the generic CDS, previously detailed in Figure 3.1. Other than the communication channel with the untrusted host processor, the architecture of the co-processor is the same as the General Purpose Architecture of vCDS.

[1]Recall from Chapter 2.7.3: Data Diodes that in a circuit diagram, the diode symbol contains an arrow which points in the direction of conventional current, i.e. anode to cathode; the arrow contradicts the flow of electrons, i.e. cathode to anode. In this work, depictions of the diode symbol are used, with the arrow pointing in the direction of conventional current, in accordance with the following citations: [38, 49, 52, 71, 77, 90, 106].

Bi-directional Bus

Between the host and co-processor, the architecture leverages a physical, bi-directional bus for use as a communication channel through which all data that is to be moved, either from the host to the co-processor or in reverse, must pass. There are no added security controls for the bus itself; the bus acts as an additional isolation mechanism between low operations and high operations.

Information Flow

The data first enters the untrusted host processor. This could be by way of a network connection passing through a NIC, a removable physical device, creation of data directly within the domain, or some other method. If the data must undergo or be processed by the high operations, then they will pass through the bi-directional bus into the Low Side of the co-processor. Within the co-processor, information flows identically to that of vCDS's General Purpose Architecture with the exception of the additional bi-directional communication channel. Once the data are back in the Low Side of the co-processor, they may travel back through the bus and into the host processor.

3.5 Applications

The vCDS system model and architectures abstract a family of systems which can support a range of applications such as those listed below. Note that portions of the following sections, including diagrams, have been adopted from "Cross Domain Solution System Built on a Formally Verified Security Microkernel Running on Processors Enabled with Memory Encryption", a patent in preparation [34].

3.5.1 Stream Processor

The research discussed in [126] aims to address the problem of using classified data and technology for a security task on an unclassified host without revealing any information about the classified resources to the host. The vCDS system is adaptable to implement an Intrusion Detection System/Intrusion Prevention System (IDS/IPS) and/or firewall technologies which use highly classified tools, or Indicators of Compromise (IOC), such as cryptographic signatures and analytics tools, to defend a host that is either unclassified or resides at a low security boundary. For example,

one implementation is a network sensor (IPS/IDS) that is composed of classified technology but is required to examine traffic on an unclassified system. The sensor employs IDS/IPS technology which must not leak any information, including covert channel information such as time or storage, about itself while operating in a low environment.

Figure 3.3 below peels back the curtain and reveals the inner mechanisms governing vCDS when implemented as a stream processor. The system is comprised of three layers, the hardware layer, the TCB layer, and the domain layer. Each layer is split into two sides as represented by the dotted vertical line in the figure. The left half represents Low Side, unencrypted processing; the right half represents encrypted High Side processing. Residing in the hardware layer is a NIC for untrusted, Low Side communications, and a NIC for protected, High Side communications with a High Side management network. Additionally, there is a CPU for all processing which, again, uses encryption when processing High Side data, and a TEE for use by the High Side. The next layer in the diagram is the Formally Verified TCB through which all communication channels pass.

In the domain layer, inside the Low Side, is a Packet Bridge which could receive and process all communications received by the NIC. Once the data passes through the Packet Bridge, they are tagged by a cryptographic Integrity Tag mechanism, such as a hash, to be later verified for data integrity. Once the data have been tagged, it will pass through a data diode to the high domains, shaded with diagonal lines, ensuring that the data only travels in one direction. The High Side will then process the data through and IDS/IPS and/or Firewall. This is where the goal of processing unclassified data with classified technology is realized. From here, there is a bi-directional path which provides the data with an ingress and egress route between the High Side domain and an optional, High Side, filter which is running as a native TCB process. If the data travels to the TCB Native Process, the data tag is verified to ensure that no data was changed as they were being processed on the High Side. Further checks in the native process can take place inside a Disposition mechanism. If any of the security checks fail, a notification will be sent from the TCB Native Process to the High Side and the data are prevented from leaving the High Side domain. If the checks pass, the data will exit through a data diode and return back to the Low Side. Note that the data, depending on what is required by the High Side, may or may not leave the High Side. Additionally, the data may never enter the TCB Native Process.

Overall, the flow of the data, depicted in Figure 3.3, are as follows: the data are received via the

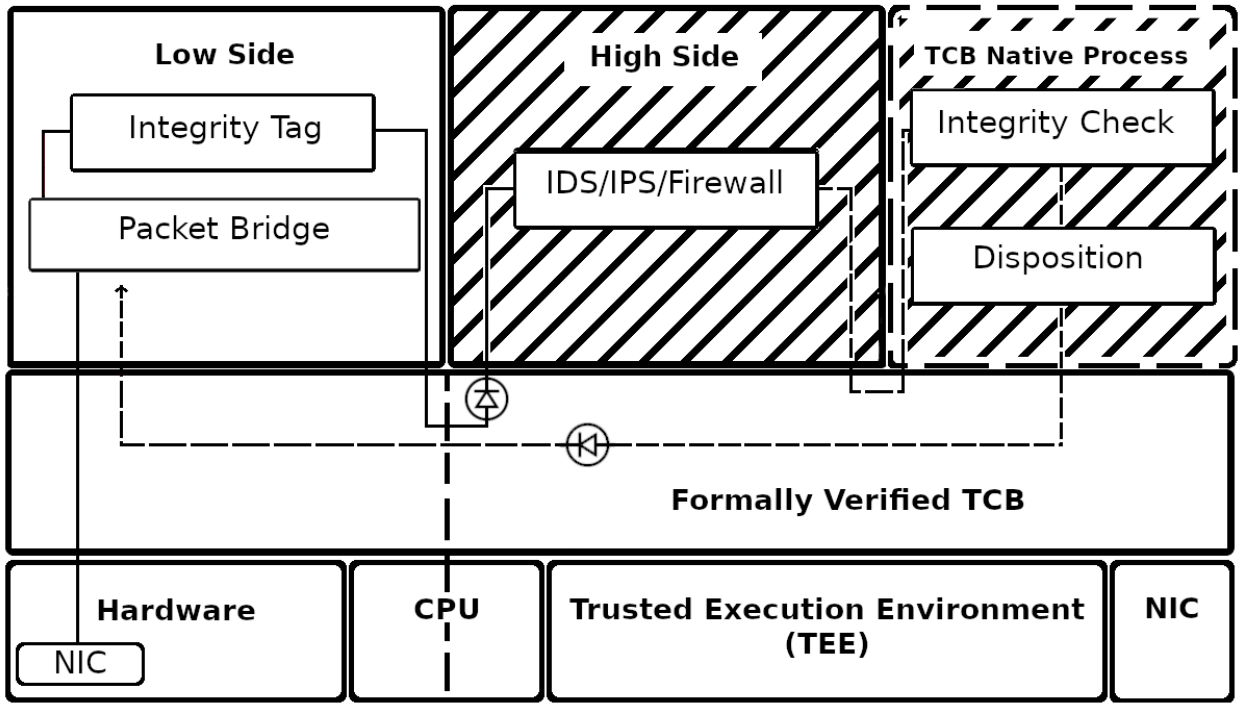

Figure 3.3: Stream Processor Architecture

Low Side NIC and processed by the packet bridge. The data are then identified with an Integrity Tag. From the Low Side, the data moves in one egress direction, controlled by a data diode, to the High Side domain where they pass through an IDS/IPS/Firewall. The next step depends on the system architecture based on the system implementation of a TCB Native Process (guard) or absence thereof. If there isn't a guard, the data will simply transfer from the High Side back to the Low Side through the data diode. However, if the guard exists as it does in the functional block diagram, the data will flow into the guard and will be examined by an Integrity Check. Upon a successful integrity audit, the data are passed out through the Disposition portion of the guard and back to the Low Side, via the data diode.

3.5.2 Data Sharing

Secure information sharing must be realized between compliant parties in systems such as Fusion Centers, Real Time Tactical Information Centers, and Real Time Crime Centers where threat intelligence sharing (TIS) is required. One application of vCDS is in a blockchain-based, TIS,

environment for robust and automated cyber security management (CSM). [25] state that, because validation of state transitions during consensus is necessary, all data concerning the blockchain are public and "existing smart contract systems thus lack confidentiality or privacy: they cannot safely store or compute on sensitive data". This problem is one which vCDS aims to correct. Furthermore, vCDS can be adapted to a system architecture based upon the B2CSM architecture described in [61] with the goal of achieving secure intelligence collaboration through data collection, aggregation, analysis, and threat-related information dissemination to critical parties.

Figure 3.4 below shows one potential vCDS architecture which seeks to correct the problem of confidentiality concerning smart contracts while achieving the goal of TIS. Once again, the architecture is comprised of three layers where each layer is split into two sides, high and low, as represented by the dotted vertical line in the figure. The first layer represents the hardware, with full encryption capabilities for the high side. The second layer in the diagram is the Formally Verified TCB through which all communication channels pass. The top layer shows the publicly available Blockchain placed on the Low Side while the Smart Contract resides within the High Side. Both high domains are shaded with diagonal lines. In this way, similarly to Ekiden [25], vCDS architecture separates *computation* from *consensus*. The Filter Guard in this figure is, once again optional. The Filter Guard, if used, could be purposed to ensure that no metadata from the Smart Contract is leaked back to the Blockchain.

Much like the Stream Processor, the data flow is as follows: the public intelligence data which resides on the Blockchain in the Low Side are directed to the High side via the one-way data diode link. The High Side performs the Smart Contract computations and then, if the guard is implemented, sends the data and results through the Filter Guard to filter out any metadata and results before sending the intelligence through a data diode to the Low Side. Once again, all transfer channels pass through the Formally Verified TCB and are therefore protected.

3.5.3 Big Data/High Performance Computing

Today's information processing workload on cloud/big data infrastructure makes apparent the need for scalable, remotely deployable, and high performance CDS systems. The current funnelling approach to data manipulation across classification boundaries in Big Data/Cloud environments degrades system performance to the point of ineffectiveness. These platforms require too much data

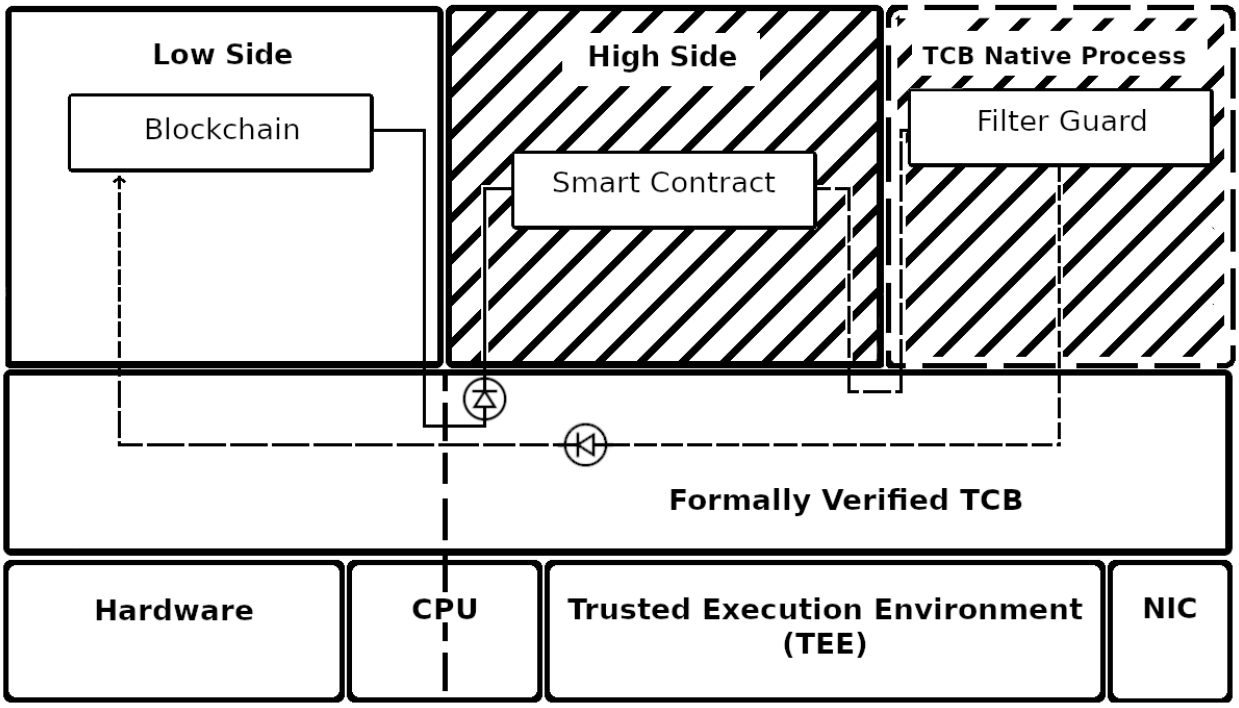

Figure 3.4: Data Sharing Architecture

to move efficiently as current approaches do which is inconsistent with MapReduce – the distributed computing paradigm for big data where the computation is moved to the data nodes as opposed to the data being moved to the computation [88]. One note about current platforms is that operational ineffectiveness oftentimes forces users to violate the security policy and access regulations in order to carry out an operation [82]. More concerning is that permanent solutions to this problem do not currently exist and the DoD is merely implementing several short term technologies which, more than likely, introduce more vulnerabilities into the system [82]. Furthermore, CDS appliances that are able to handle the immense amount of data are highly specialized, individually developed solutions: presenting scalability and cost of maintenance as major limitations [47].

Our CDS system is adaptable to both the big data platform (BDP) and high performance computing (HPC) environments where distributed parallel processing, including the access and transfer, of large data sets is commonplace. In the case of a big data platform, vCDS is consistent with MapReduce in that it moves the computation to the data. Relating to the tool presented herein, the computation which is moved resides in the enclave. In the case of a high performance

computing platform, vCDS can be adapted to move the data to the compute nodes. vCDS is a versatile solution that provides the needed scalability and low cost implementations in a field ruled only by highly specialized systems [47, 82]. The implementation of the vCDS architecture can achieve high performance/BDP goals as a general purpose, security baseline solution with very little cost, and is adaptable to the particular use case.

Illustrated in Figure 3.5 is a functional block diagram for the BDP/HPC application. Beginning again at the hardware layer, there are components used to process both the Low Side and the High Side functionality. On the Low Side, there exists the basic hardware components which allow for processing of untrusted, low data and operations. On the High Side hardware layer, there is a hardware TEE which enables secure memory computations for the high security components. Above the hardware is the TCB layer. The Formally Verified TCB provides the assurance of functional correctness and security, along with the uni-directional channels which pass data from one security domain to another. The channels are enforced with a data diode to ensure the uni-directional flow of the data from the Low Side and into the High Side boundary.

Additionally, within the High Side boundary, there is an optional, bi-directional, data flow channel from the High Side to the high TCB Native Process. From the native process, there is another optional uni-directional data flow channel, enforced by a data diode, which serves to transfer data from one of the High Side components and back to the Low Side. At the top level of Figure 3.5, there are the untrusted and trusted domains represented respectively by the Low side and the High Side, which includes the TCB Native Process. Depending on the implementation, the Low Side can implement a Hadoop file system (HDFS) or a Lustre file system which, much like the common file system, house the data. Related, the High Side domain could employ the Hadoop process, MapReduce, or an MPI Node. The TCB Native Process, once again, is an optional guard that contains an obfuscation function which, for example, could poly-instantiate the data, adding noise to the channel.

Overall, the flow of the data, is as follows: the data are first accessed via the Hadoop or Lustre file system and move in one direction, controlled by a data diode, to the High Side domain where they pass through to MapReduce or the MPI Node. The next step depends on the system architecture based on the system implementation of a guard or absence thereof. If there isn't a guard, the data will simply transfer from the High Side, back to the Low Side through the data

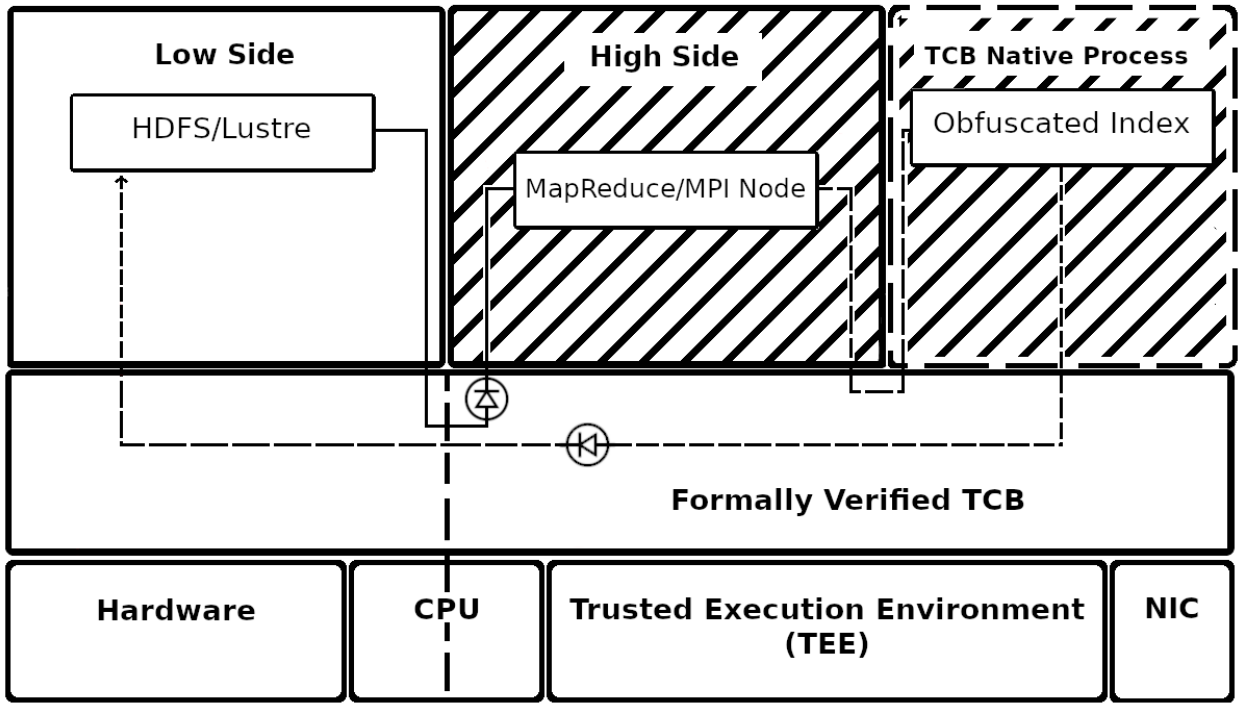

Figure 3.5: Big Data Architecture

diode. However, if the guard exists as it does in the functional block diagram, the data will flow into the guard, and be obfuscated by a poly-instantiation function. The data will then travel back to the Low Side via the data diode.

3.6 Security Analysis

The methods by which the security architecture achieves the security objectives and requirements mentioned above are analyzed below.

3.6.1 Hardware Protections and Memory Encryption

Relative to the High Side, the TEE is used to mitigate attacks from more privileged software and physical attacks with full, transparent memory encryption as well as protection of memory at rest, memory in transit, and memory in use which helps mitigate the vCDS Threat Model (Chapter 3.2). Additionally, a padding mechanism, similar to the one described in [68], is added to increase the execution time of the data processing and authentication mechanisms to mitigate data leakage

through timing analysis. Note that [55] presents a method to provably eliminate timing channels and cross-domain temporal interference in a formally verified microkernel, discussed in Chapter 7.2: Discussion, though it has not yet been integrated as of this work. Furthermore, [130] shows that timing channels can be successfully eliminated in a RISC-V processor.

It is important to note here that without the TEE, vCDS provides complete formal verification but lacks the security measures required for a secure, remotely deployable system. With the TEE, the system achieves the objective of secure remote deployability, however, at the time of this writing, no effort has been made to formally verify the enhancement of this solution. The formal proofs for the enhancement are intended for future development.

3.6.2 Trustworthy Components

The formally verified code base assures that no software vulnerabilities exist in its operation and that the system is proven trustworthy. This proof is one key element which differentiates current trusted CDS systems from the presented trustworthy CDS system: the vCDS system architecture is able to rely on the functional correctness provided by the formal verification which ensures that the system is trustworthy to adhere to defensive effectiveness and ensure data confidentiality. Furthermore, vCDS components are open for independent verification [32] which further validates the confidence in its correctness.

3.6.3 Decidable Object Security and Staticity

A capability-based access control model governs all kernel services so that any applications wanting to perform an operation must invoke a capability that has sufficient access rights for the service making object security decidable [44, 91, 125]. There is no implicit memory allocation within the kernel, only explicit request via capability invocation [44]. Furthermore, all hardware resource partitioning is governed by capability distribution, that is, authority distribution. A system resource manager enforces a resource management policy such that it is guaranteed that no entity can obtain any capability to another entity (subject or object) without the preexistence, i.e. pre-allocation, of the particular capability [44]. This *authority confinement* enforces an upper bound on authority changes [115]. The component architecture model combines with the capability model to enforce the *staticity property* [33] – a property which ensures that configurations occur before compile time

so that all channels, i.e. interfaces and connections, and privileges are pre-allocated and that no channels or added privileges can exist outside of what is predefined. Therefore, the threat vectors involving attacks stemming from dynamic creation of channels and reconfiguration of privileges are mitigated. The access control model also allows the system to grant specific communication capabilities which enable authorized and controlled communication between components, thus enabling, with a high degree of assurance, component isolation because only operations that are explicitly authorized by capability possession are permitted [125]. Component isolation and communication authenticity further mitigate threat vectors outlined in the vCDS Threat Model (Chapter 3.2) by ensuring that no user or process can access resources without authorization.

3.6.4 Computation Isolation

In addition to kernel and kernel-enforced component isolation, there exists component and computation isolation within the High and Low Side domains. On the Low Side, the necessary drivers and data management services and computation are isolated from the High Side. In contrast, the High Side hides all sensitive intelligence used to analyze the low data from the Low Side.

3.6.5 Data Flow Restriction

The data diode ensures that data can only travel from low to high and never back to low, thus mitigating the confidentiality threat model [98]. If the data must travel from high to low through a corresponding data diode, the data will first pass through the guard which ensures that no sensitive data are passed to the low, again mitigating threat vectors in the vCDS Threat Model (Chapter 3.2). The stream processor application leverages the integrity filter as described in Chapter 2.12: Integrity Guard. The integrity filter creates and checks data identifiers to make certain that data from the high does not leak, accidentally or purposefully, to the Low Side.

For whichever use case vCDS is implemented, secure communication between components of the same classification level helps to mitigate the threat vectors in the vCDS threat model, while providing a secure path of remote deployability.

3.7 Discussion: Protocol Adapters

Although confidentiality is the focus of this model, PAs, like filters and guards, can be added to the architecture to further enforce data safety. Data safety involves protection from data attacks, data hiding, and data disclosure [30]. The addition of such mechanisms would require expansion of the vCDS baseline threat model. In the vCDS architecture, a new component would be instantiated as a particular PA, like those mentioned in Chapter 2.13: Protocol Adapters. In the cases when multiple PAs are required, an individual component would be added and instantiated for each one.

Figure 3.6 below presents an example of what the vCDS architecture may look like when leveraging consumer and producer PAs. For example, a PA consumer may be located between the Low Side and the High Side while a PA producer would be located between the Guard and Low Side return connection. The flow of data would then be as follows: the data would proceed from the Low Side to the consumer which would terminate the protocol by striping only the essential information, for example the payload; then the data would be processed by the High Side and Guard before being reconstructed by the producer and transferred back to the Low Side. Other configurations of the architecture may place the PAs of varying objectives at other points depending on the use-case and requirements.

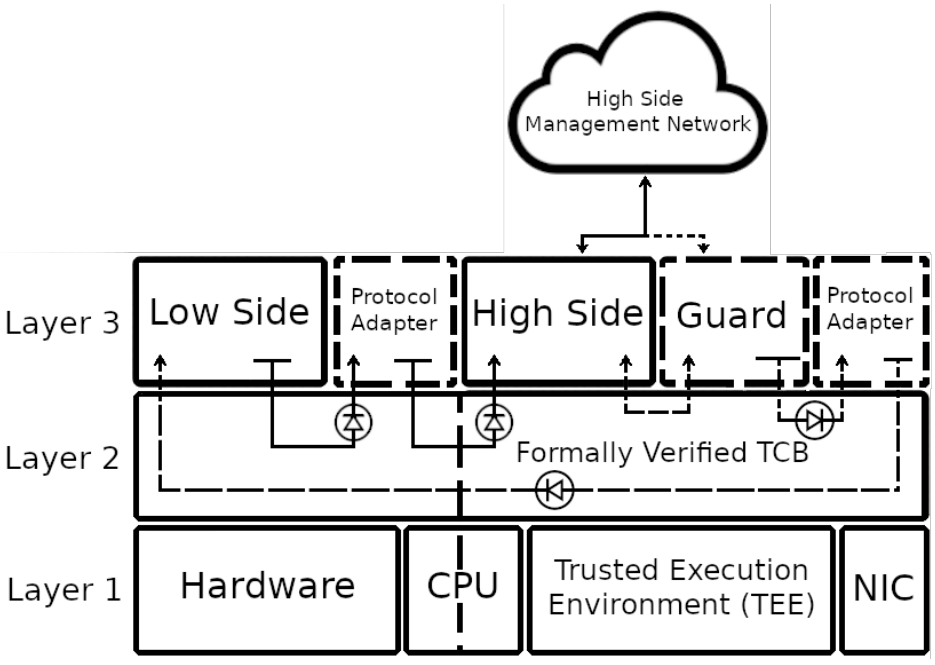

Figure 3.6: vCDS Protocol Adapter Integration

Chapter 4

The vCDS Implementation

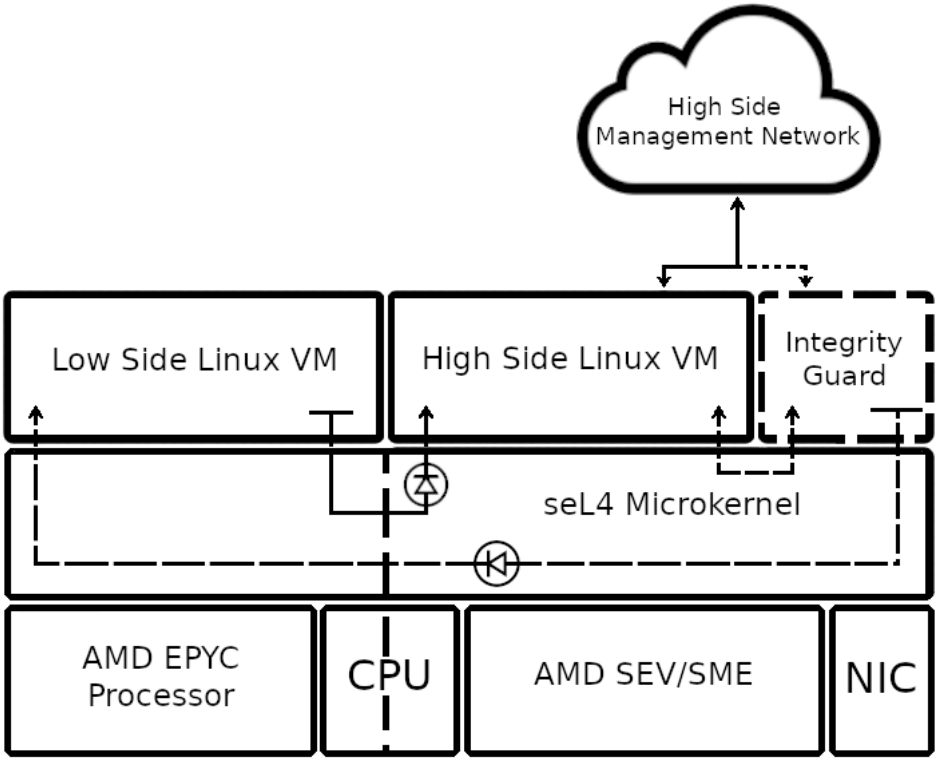

Figure 4.1: vCDS Stream Processor Instantiation © 2021 IEEE

Note that portions of the following chapter, including diagrams marked with "© 2021 IEEE", have been adopted from "vCDS: A Virtualized Cross Domain Solution Architecture", © 2021 IEEE [33].

The following chapter details the resilient, cross-layer implementation of vCDS and each system

component, shown in Figure 4.1, from the selection of hardware TEE to the use and development of software applications used by the high and low components. The implementation described herein, as tailored to the stream processor application which can be referred to as a network sensor (IPS/IDS), was elected for the purposes of better understanding the architecture. The sensor application is further discussed and evaluated in Chapter 6: Analysis and Evaluation. To further emphasize the mobility and flexibility of vCDS, a prototype of the solution is implemented, both on QEMU, an open source machine emulator and virtualizer [108], and directly on machine hardware to emphasize the use of a hardware-based TEE.

To understand the following implementations, it is useful to visualize the vCDS project directory tree, including the key source files which help to make up the individual components of the stream processor. Note that only the smallest necessary portion of the files are listed in Figure 4.2.

4.1 For Realizing Layer 1 in the Architecture: AMD EPYC with SEV/SME

As mentioned above, experiments were run with the stream processor on QEMU, in addition to those hardware platforms with the AMD Zen 3 and Intel Nehalem microarchitectures [108]. One of these microarchitectures, AMD Zen 3, is implemented on AMD's EPYC processor. In order to instantiate the hardware layer of vCDS, the Dell PowerEdge R6525 rack server with AMD EPYC processor was selected [72].

4.1.1 Implementation

All Low Side computations require the basic hardware capabilities – no concern is taken to incorporate security mechanisms on the Low Side in this use-case, as the Low Side represents an untrusted domain. The AMD EPYC processor provides the basic hardware mechanisms and capabilities necessary for implementation of each Low Side component and to carry out all Low Side computations. On the other hand, the High Side does require security and data protection mechanisms for storing and processing classified data. The AMD EPYC processor presents the Secure Encrypted Virtualization (SEV) and the Secure Memory Encryption (SME) technologies which help administer essential protections – for *data-at-rest*, *data-in-transit*, and *data-in-use* – against

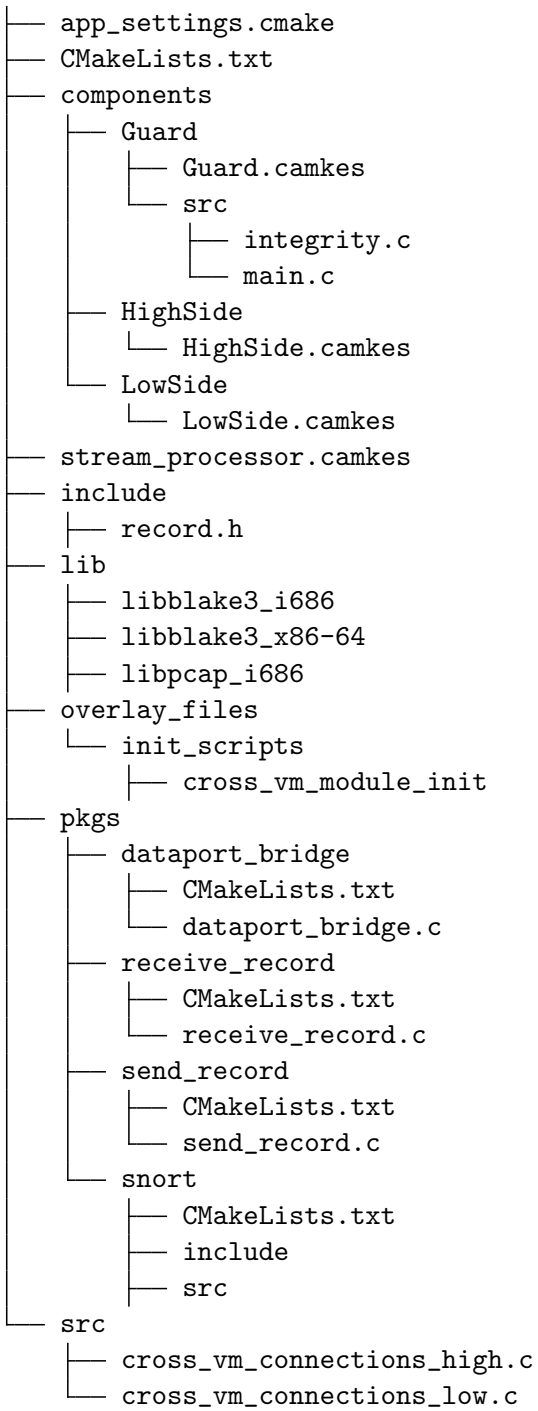

Figure 4.2: Stream Processor Project Tree

the threat vectors outlined in Chapter 3.2: Threat Model. With the provision of these technologies, the processor provides full memory encryption to meet the security requirements, outlined in Chapter 3.3: Security Requirements, for use as a TEE to secure the High Side components at the hardware level.

Secure Memory Encryption

The SME mechanism is a scalable architectural capability for main memory encryption. Two hardware components are leveraged for the main memory encryption: the AMD Secure Processor and the Advanced Encryption Standard (AES), 128-bit, hardware encryption engine [5]. Encryption is performed by on-die memory controllers which each include an AES-128 engine to encrypt the data when written to main memory and, when provided with the key, decrypt the data when they are read [76]. The AES-128 engine, when the memory encryption bit is enabled by the operating system or the hypervisor in the page table entry, automatically executes the cryptographic operations. It is worth noting that the cryptographic operations do incur additional latency for the main memory access, but has no significant impact as recorded by [76]. Additionally, because SME is only utilized for High Side operations, it is expected that performance is less impacted.

The Secure Processor is a 32-bit microcontroller that functions as a dedicated security processor in order to provide secure, NIST SP 800-90 compliant, key generation and key management. The AES encryption key is randomly generated and managed upon each system reset and is not accessible to, nor does it require access to, any software running on the CPU [76]. SME also provides a transparent SME mode (TSME) which allows even legacy operating systems and hypervisors to be protected by full memory encryption without modifying the software.

Secure Encrypted Virtualization

The SEV mechanism integrates the SME capabilities with AMD-V virtualization architecture in order to provide cryptographic isolation to VMs. This security model contrasts the traditional ring-based security, where subjects have full access to objects at or below their own classification level, by isolating each level so one does not have access to the resources of another [76]. Therefore, a hypervisor cannot access the resources of a VM guest even though it typically resides at a higher classification or privilege level, thereby mitigating threat vectors described in Chapter 3.2: Threat

Model.

Once SEV is enabled, the hardware will tag all data with its address space identifier (ASID) to mark to which VM the data belongs. The tag protects data within the system on chip (SOC); when the data are outside the SOC, that is leaving or entering, the AES-128 engine encrypts/decrypts and protects the data [76]. Furthermore, each VM and the hypervisor have an ASID and an encryption key such that any data associated with a VM tag and encryption key is restricted to that VM.

An additional benefit of SEV is Encrypted State (SEV-ES). SEV-ES ensures the encryption of all CPU register contents whenever the VM stops running. This is especially useful for preventing any leakage of data from the registers to the hypervisor. SEV-ES can also detect any malicious attempts to modify register contents as the VM is encrypted with the guest encryption key such that integrity is protected [6].

One especially advantageous outcome of using SEV in the vCDS system is that its security isolation is applicable in cloud environments. Specifically, SEV ensures that data-in-use is cryptographically protected and isolated from the hosting software and other VMs and processes [76]. As is true with SME, SEV does not require any software modifications to provide cryptographic isolation.

4.1.2 Security Analysis

Hardware accelerated memory encryption ensures that all data are cryptographically protected from physical attacks, in addition to attacks on main memory and data-in-use. VM and hypervisor isolation by SEV ensures that if an attacker has access to the hypervisor, host, or has control over one VM, the attacker will not be able to read the memory of any other VM as the memory will be encrypted. The vCDS system takes advantage of the per-VM encryption keys to ensure that High Side data confidentiality is maintained in the hardware. Additionally, when the vCDS use-case calls for a high side remote management network (C2) or a cloud-based environment, the isolation of the VM from the host and the hypervisor supports the goal of remote deployability [76].

4.2 For Realizing Layer 2 in the Architecture: seL4

The Secure Embedded L4 (seL4) operating system microkernel and hypervisor has been chosen and implemented as the TCB, in order to leverage the trustworthiness provided through its formal verification and security guarantees. This section provides the background and technical details relating to the seL4 microkernel. It is also important to distinguish between the previous use of the acronym TCB and how it is used in Chapter 4.2, For Realizing Layer 2 in the Architecture: seL4 . Previously, and in all other sections, TCB refers to the Trusted Computing Base, however, in this section only, TCB will refer to the thread control block.

4.2.1 seL4 Technical Overview

The seL4 microkernel isolates execution contexts, i.e. *threads*, through a capability based security model, and supports message passing services for communication between theses contexts. The kernel provides objects for thread control block (TCB), address spaces, message passing, device primitives and capability spaces, which can be used to interact with other objects within the system. Recall that a capability is the encapsulation of a reference to a kernel object and the rights associated with that object. Once the kernel boots, all physical memory resources are allocated to the root thread as a data structure, *BootInfo* that contains interrupt request (IRQ), memory, I/O port information as well as capability references to *untyped* kernel objects [46, 114]. Untyped kernel objects reside in untyped memory which correspond to "a block of contiguous physical memory with a specific size" [114]. These objects leverage a Boolean device to indicate kernel-writable memory. In the case of device memory, which is "not backed by RAM but some other device", the kernel is unable to write to it. It can however be retyped as a frame object: "physical memory frames, which can be mapped into virtual memory" [114]. It is useful to note that, while at the hardware level, threads sharing the same virtual memory space can traditionally be referred to as siblings, the concept of a process does not exist in seL4; instead, the kernel uses threads [114]. Therefore, seL4 differs from most contemporary kernels in that one address space does not necessarily correspond to one process [114]. seL4 manages its threads by keeping track of schedulable, executable resources called TCB objects, shown in Figure 4.3. Unlike contemporary kernels, TCB objects must be manually *retyped* in seL4.

Another difference between contemporary kernels and seL4 can be realized when an object is retyped and handed to the thread as a capability reference to a specific kernel object. The *type* of the new object will determine what invocations can be made on that object. seL4 leverages multiple, directed graphs, or trees, of capabilities called *CSpaces*, also shown in Figure 4.3. Each CSpace is made up of one or more *CNode* objects, each CNode block contains an array of *CSlots*, and each CSlot corresponds to a capability. Note that because a CNode is an object, one CNode may contain a capability reference to another CNode. In effect, a CSpace is the full range of capabilities accessible to an individual thread or a shared execution context [114]. Therefore, when a new kernel object is minted by retyping an untyped kernel object, the capability reference is set within the thread's CSpace. Additionally, a thread that contains a capability to a chunk of untyped memory, can retype it as virtual memory. Similarly to CSpaces, seL4 provides each thread with a tree like object for managing virtual memory called a *VSpace*. For further isolation, a thread can run inside the virtual address space referenced by the capability. While each TCB object contains capabilities which define a thread's CSpace and VSpace, multiple threads can share portions of, or entire, CSpaces and VSpaces [84].

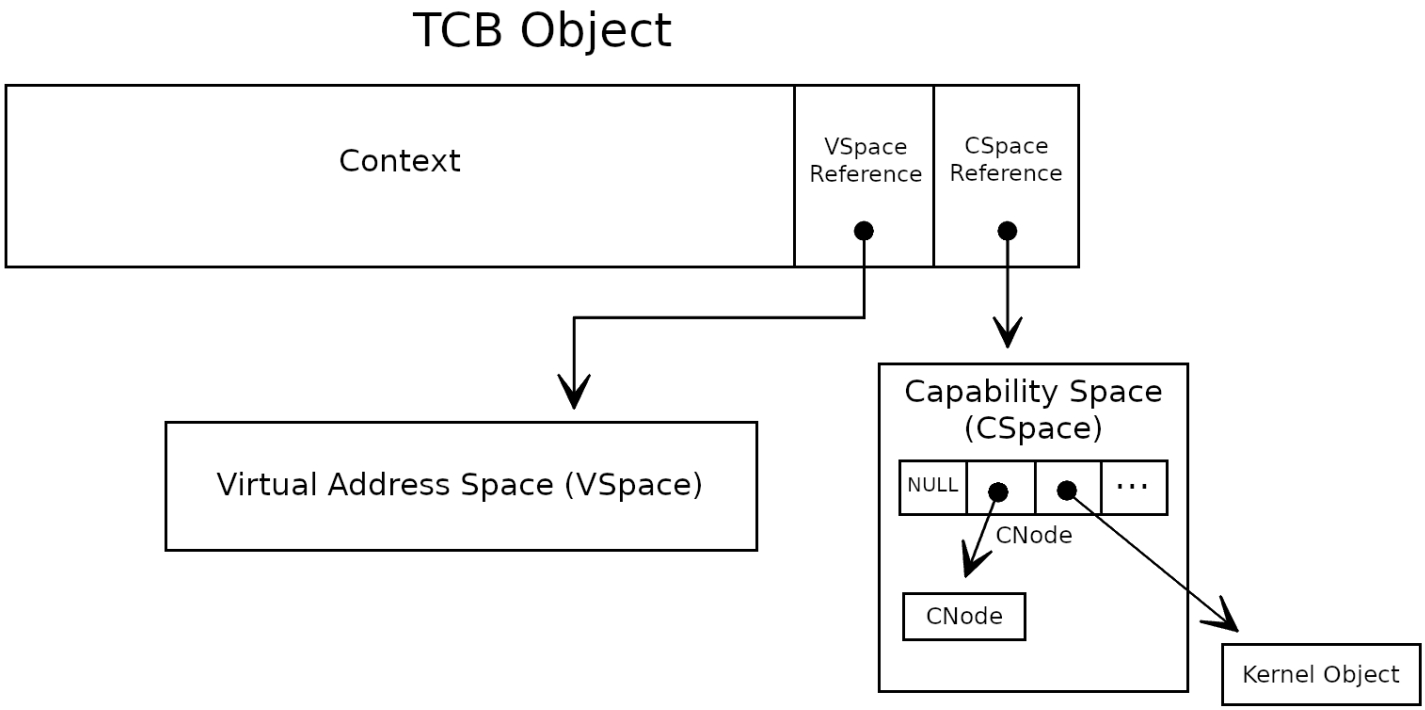

Figure 4.3: Thread Control Block

4.2.2 Implementation

seL4 achieves its security properties and meets its security requirements through several mechanisms and evaluations: Capability-based Security Access Control Model, Communication Mechanisms, seL4 Data Protection Model, and seL4 Formal Verification.

Capability-based Security Access Control Model

The capability-based security access control model components used in seL4 are *capabilities*. Capabilities are "access tokens which support very fine-grained control over which entity can access a particular resource in a system" [65]. A capability, which is an immutable object reference, or pointer, enforces the PLOP by ensuring that the only way an operation can be performed on a component is by invoking the capability which is pointing to that object, thus restricting the granted rights to the absolute minimum required to perform the operation. Recall that BLP preserves the POLP in a similar fashion. The capability, sometimes referred to as a fat pointer, encapsulates the kernel object reference as well as the rights conveyed to the kernel object. The object itself could be a TCB, for example.

The seL4 whitepaper [65] makes the distinction between ACLs, generally regarded as OS capabilities, and seL4 object capabilities. As previously discussed in Chapter 2: Background, general access control takes on a subject-oriented approach where each file respectively defines a set of access-mode bits which allow access to a subject based on user and group identities. All subjects or threads of a particular user or group can perform the same operations on an object. seL4 capabilities differ by providing object-oriented access control where a particular thread must have been given a capability for a particular operation on an object. In other words, when a thread is invoked, it must be handed a capability which defines the object and the operation permitted to take place on the object. The opacity of capabilities provides the interposition of access so that, if a subject is given the capability reference to an object, it has no method of determining what the object is, restricting the subject only to invocation of methods on the object. Furthermore, one subject that owns an object can safely delegate privileges for that object to a second subject by minting an object capability and giving it to the second subject. This allows the second subject to operate on the object without referring to the delegating subject. The delegating subject can also

mint the capability with diminished rights. Once again, seL4 capabilities ensure the POLP and maintain one avenue or channel for each operation on an object.

Communication Mechanisms

Endpoints are communication ports which facilitate the transfer of small amounts of data and capabilities between threads and address spaces, through the kernel. Notifications are a signaling mechanism which logically represent an array of binary semaphores. As shown in Figure 4.4, capabilities to Endpoint and Notification objects have a unique member which categorizes them as either a *badged capability* or an *unbadged capability*. Badged capabilities are capabilities with a

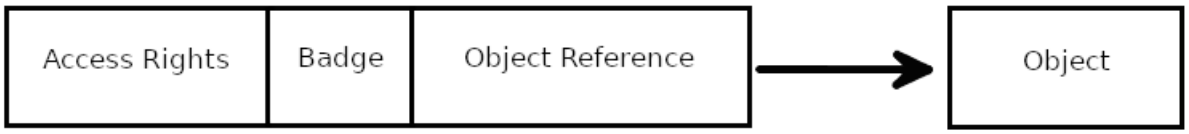

Figure 4.4: Endpoint capabilities have a badge field which is used to identify the parties involved in a communication

defined data word, i.e. a badge, which specifies information related to the access of the capability, such as the delineation of those parties involved in the communication, and whether the the capability is Notification or Endpoint specific [84, 125]. Unbadged capabilities are Endpoint capabilities with a zero badge. The use of these badges will be examined in the following discussion. seL4 provides the use of three different communication mechanisms, each with specific and independent use-cases: seL4 IPC, Notifications, and Shared Memory.

seL4 IPC. seL4 IPC is no longer the common "inter-process communication", as defined by the message passing primitives of typical operating systems; instead, it is the seL4 mechanism used to synchronously transfer capabilities and small amounts of data between threads [114]. In order for a thread to send a message, it must invoke an Endpoint capability that is in its CSpace. IPC message passing is facilitated by Endpoint capability references to kernel objects, that is, Endpoint objects are invoked to send and received IPC messages, depending on the respective capability a thread possesses. Recall that a badge in a badged capability can mark the capability as an Endpoint capability. Badges may also be used such that the sender and receiver can identify one another. These types of capabilities are especially useful in the case of the seL4 IPC rendezvous model, with

synchronous Endpoint objects that utilize blocking threads, where multiple parties are expected to participated in communications.

Each Endpoint object contains a queue of threads which are waiting to send or receive messages. Each thread within an Endpoint has an IPC buffer, consisting of a bounded area of message registers that house the capabilities and data of the IPC message. The IPC buffer in the sending thread will be copied, by the kernel, to the receiving thread's IPC buffer [114]. The sending thread will block until the message is consumed by the receiving thread. IPC closely resembles a remote procedure call in that it is a mechanism "used for implementing cross-domain function calls" and is not to be used as a mechanism for shipping data or synchronizing activities [64]. Furthermore, IPC should not be used for sending large amounts of data or one-way notifications. Therefore, the only reason seL4 IPC should be employed is for a thread to invoke functions within a different address space of another thread.

Notifications. seL4 Notifications are *asynchronous* signals which can be sent from one thread to another. In order for a thread to send a notification, it must invoke a capability to a Notification object that is in its CSpace. Notification capabilities, like Endpoint capabilities, can be badged such that the receiver can identify the sender. Each Notification object is formed by a data word, which acts as an array of binary semaphores, and a queue of TCBs waiting for notifications. Each Notification object has three states: (i) Waiting: TCBs are queued on this Notification and are waiting to be signaled; (ii) Active: TCBs have signaled data on this Notification; and (iii) Idle: No TCBs are queued and no TCBs have signaled this object since it was last set to idle [114]. When a thread signals a Notification object, the following event will occur, depending on the state of the object: (i) Waiting: the TCB at the head of the queue is woken and the capability badge is sent to it, however, if the queue should be empty, the notification object is converted to idle; (ii) Active: the badge of the capability used to signal the notification object is bitwise ORed with the notification object's data word; and (iii) Idle: the notification object's data word is set to the badge of the capability used to send the signal, and the object is transitioned to active [114]. When a thread waits on a notification object, the following event will occur, depending on the state of the object: (i) Waiting: the TCB is pushed to the queue; (ii) Active: the TCB will receive the notification object's data word which is then reset to 0 and the object is transitioned to idle; and (iii) Idle: the TCB is pushed to the queue, and the object is converted to waiting [114]. Notifications are

primarily used for interrupt handling and to synchronize access to shared memory [114].

Shared Memory. seL4 shared memory mappings are offered in the form of buffers which can be shared between threads. Shared memory is specifically useful for shipping bulk data between threads. seL4 IPC is distinguished from contemporary IPC through shared memory by the amount of data to be shared.

seL4 Data Protection Model

Note that portions of this section have been adopted from "Auditing a Software-Defined Cross Domain Solution Architecture", in preparation [35].

The protection model employed in seL4 is based on the take-grant model described in Chapter 2: Background and is shown by [44, 91] to be decidable. It is useful to note here that, when the take-grant model was proven decidable by [91], the *take* operation, described in Chapter 2.3.1: Take-Grant Model, was included in the proof of the theorem. The take operation, in the case of seL4, is a dangerous operation. Given n_1, n_2, n_3, from Chapter 2.3.1: Take-Grant Model, the take rule permits the node n_1 to take the authority, α, from n_2, to operate on n_3. If this operation were permitted, n_1 may acquire authority to operate on n_3 without explicitly being granted that authority by n_2. This would break the security proofs of seL4. Therefore, the take operation is omitted and only the grant operation can be used to propagate authority. The new theorem that applies to the seL4 implementation of the security model is proven in [44]. A security configuration analysis tool is one contribution to mitigate this problem and is introduced in Chapter 5: Auditing a Cross Domain Solution, along with further examinations of modifications made to the take-grant model. The purpose of the tool, presented in [35], is to verify the correctness of the security configuration of a vCDS instantiation.

seL4 Formal Verification

"Complete formal verification is the only known way to guarantee that a system is free of programming errors" [78, 79]. The seL4 microkernel has undergone comprehensive, formal, machine-checked verification, such that there exists a mathematical proof which ensures the following:

(i) seL4's implementation is consistent with its specification;

(ii) the implementation is free from programmer-induced defects;

(iii) the specification satisfies desirable high level properties such as:

  (a) termination: the kernel will never crash, and

  (b) execution safety: the kernel will never perform an unsafe operation,

  that carry through to the code and binary levels; and

(iv) seL4's access control and data protection models are formally proven to provide security guarantees [56, 78, 79].

In order to achieve this level of functional correctness verification, the well-known Isabelle/HOL interactive formal proof assistant was used to generate an executable specification, in order to show *refinement* using Hoare logic, as previously described in Chapter 2.11: Formal Verification [104, 105]. Further proof is provided to show that the executable specification of seL4 correctly implements the abstract model, in addition to proofs of security enforcement, which were first carried out over seL4's abstract specification and then to it's implementation using refinement proofs which embody functional correctness. Therefore, the seL4 microkernel is the first comprehensively verified, general-purpose OS kernel which provides complete proof of the high-level security and safety requirements, i.e. integrity, confidentiality and availability, as well as full, functional correctness down to the executable machine code [79].

4.2.3 Security Analysis

seL4 is the only existing capability-based microkernel system with a proof of functional correctness which "guarantees that every behavior of the kernel is predicted by its formal abstract specification" [65]. Furthermore, the functional correctness property which has been proven for seL4 is "much stronger and more precise than what automated techniques like model checking, static analysis or kernel implementations in type-safe languages can achieve" because the kernel is analyzed to ensure safe execution, and also to produce "a full specification and proof for the kernel's precise behavior" [78]. The proofs also show that the kernel is free of, and therefore cannot be affected by, deadlocks or livelocks, buffer overflows or memory leaks, arithmetic overflows or exceptions, use of

uninitialized variables or null pointer dereferences [46]. In other words, the seL4 proofs guarantee that there can be no undefined behavior and no bugs will be realized in the system.

The microkernel is also provably secure: "seL4 comes with further proofs of security enforcement" [79], and employs the Take-Grant protection model, described in Chapter 4.2.2: seL4 Data Protection Model, such that, "in a correctly configured seL4 -based system, the kernel guarantees the classical security properties of confidentiality, integrity and availability" [65]. One of the stated contributions of this research was to provide an audit tool which verifies the correctness of the security configuration. This tool is detailed in Chapter 6: Analysis and Evaluation.

seL4 further ensures safety of time-critical systems and is the world's fastest performing microkernel while providing fine-grained access control [20, 65]. Therefore, the seL4 microkernel garners the highest performance in its class [67]. Additionally, a key design goal implemented in seL4 is to provide "strong isolation between mutually distrusting components", making it ideal for vCDS instantiations [80]. The seL4 isolation mechanisms are supported when used as a hypervisor, allowing the secure execution of multiple systems of differing trust levels on top of seL4. The system also renders statically defined components, processes and channels which are immutable after compile time; this is another property which makes this microkernel and hypervisor ideal for vCDS.

4.3 For Abstracting Layer 2 and Linking to Layer 3 Components: CAmkES

In order to abstract away the low-level seL4 components, the component architecture for microkernel-based embedded systems (CAmkES) framework has been chosen. This component framework allows the development and manipulation of a CDS instantiation on top of the static architecture of seL4. CAmkES abstracts over low-level kernel mechanisms, providing communication primitives and support for decomposing a system into functional units [80].

4.3.1 Implementation

The CAmkES architecture description language (ADL) describes three high-level system abstractions and their interactions which combine to form a composition: Components, Interfaces, and Connectors [114]. Components are objects encapsulated by the microkernel, interfaces define com-

ponent invocation, and connectors are one-to-one links between the interfaces. CAmkES leverages a compiler to translate the ADL into the capability distribution language (capDL). capDL describes the kernel objects that are needed by an seL4 application, in addition to the distribution of the capability references to those objects [114]. In other words, the capDL defines a system's entities (subjects and objects) and access rights. seL4 enforces the specified access rights and only those rights, so that whichever state the system is in, it is guaranteed to behave as described by the ADL.

4.3.2 Components

A CAmkES component is a type of functional entity [114]. The functional entity itself is the set of all programs, code, and data which represent an instance of the component. For the purposes of understanding, both components and instances of a component will be referred to as components herein. Relating to the stream processor, the High Side and Low Side domains, as well as the Guard are the CAmkES components.

4.3.3 Interfaces

An interface is an interaction point of a component, that is, it defines a point to which another point is linked for the purposes of communication. CAmkES provides the use of three interfaces: (i) procedures, (ii) events, and (iii) ports. A procedure is an interface over which function calls can be made; each procedure has methods which can be independently invoked. An event interface is an asynchronous signal interface of a component and a port is an interface type that represents shared memory semantics [114]. In the case of the stream processor, only events and ports are implemented. However, all three of these interfaces correspond to a connector, the use of which will be examined in Chapter 4.3.4: Connectors.

4.3.4 Connectors

Connectors define the interfaces used to connect one component to another. A connection is an instance of a connector, that allows two CAmkES components to communicate with each other [114]. seL4 system calls must be invoked in order to send data through a connection; the system calls are hidden by the CAmkES abstraction layer. CAmkES provides three connector types which each correspond to an interface: (i) Shared Data, (ii) Notifications, and (iii) Remote Procedure

Calls (RPC). Additionally, there is a special use-case for CAmkES connectors which is covered in Chapter 4.3.4: Cross VM Connectors.

Shared Data

CAmkES components use Dataports as shared data connectors between port interfaces used by components and guest processes, to enable one component to pass large amounts of data to another component. The interfaces are made available to CAmkES components as shared memory regions at runtime; seL4 has a corresponding `seL4SharedData` connector. Dataports are the typed shared memory mappings which are leveraged in order to pass data across the high and low security domain boundaries.

An important property of these shared memory mappings is that CAmkES provides access controls such that components may have restricted access to them. In order to implement a data diode explicitly and enforce the domain boundaries, the access rights granted to each component for use of their Dataports must be configured in the ADL. This configuration allows the passing of data structures through the protected seL4 microkernel via a unidirectional interface without the possibility of leaking information through the component or microkernel layers. One thing to note is that CAmkES Dataport permissions map directly to the seL4 page mapping flags. In the case of a "write-only" flag, however, there, generally, is no write-only equivalent in the paging models of most hardware platforms. This is not a bug and, in the data diode implementation, does not convey any extra abilities to the sender (Appendix D: Inquiry to seL4 Developers).

Dataports are initialized between a CAmkES component and a VM guest when a processes makes the `ioctl` system call, a call which connects the Dataport to a shared memory region. When the system call is made on the file that is associated with the Dataport, specifying a page-aligned size for the shared memory region, a kernel module within the guest VM will allocate the requested sized, page-aligned buffer. The module will then notify the hypervisor, causing the hypervisor to modify the VM guest's address space in order to create the shared memory region to connect to the CAmkES component. A process within the VM is then able to map the shared memory into their own address space.

Notifications

CAmkES components use event interfaces which, as previously stated, are asynchronous signal interfaces of a component; seL4 has a corresponding `seL4Notification` connector. In CAmkES there are two types of event connectors: emits and consumes. In this paragraph, these connectors are referred to as events in the sense that their instantiations process an action, or event, which is to take place. Emits events allow a process to emit a signal or notification to a CAmkES component. This is made possible through an `emits_event` kernel module inside the guest VM which makes a hypercall to the seL4 hypervisor, or VMM, to trigger the event and resume the VM. The emitter, or sending thread, is non-blocked and can send data at any time and at any rate. Consumes events are the receivers of emits events in that they allow a process in the VM to wait for an event sent by a CAmkES component. In this case, the VMM will receive an emits event that is destined for another guest VM and place the event identifier in shared memory between itself and the `consumes_event` kernel module. Then the VMM will interrupt the guest VM that is running the consumes kernel module; the module is registered to handle the interrupt from the VMM and it reads the event identifier from shared memory. The consumer, i.e. the receiving thread, can either be blocked or non-blocked so that it could simply just wait or it could handle other events while waiting for the asynchronous signal from the emitter.

Remote Procedure Calls

RPC calls are synchronous procedure calls that allow a component to provide functionality to another component; seL4 has a corresponding `seL4RPCCall` connector. In seL4 the RPC CAmkES connector abstracts the invocation of a function that is provided by another component as a regular function call, executed by the client component [65]. The client component can simply call the procedure as if it were one of its members. Though this particular type of connector is not used in the stream processor implementation (although it certainly could be used), this connector is especially useful in the Big Data/High Performance Computing application as it allows the movement of the computations to the data when the use-case requires it.

Cross VM Connectors

Cross VM connections are all events and memory regions that are shared between Linux VM processes and the CAmkES components. The cross VM connection kernel module is an invaluable part of vCDS functionality for virtualized use-case applications like the stream processor (Chapter 3.5.1: Stream Processor). Cross VM connector mechanisms are provided, both by seL4 and CAmkES, in order to create a communication channel between processes in a guest VM and CAmkES components. The channel formed by these mechanisms is what allows the secure transfer capability to be executed.

The `connection` module provides the functionality for parsing cross VM connections which are advertised as peripheral component interconnect (PCI) devices by employing userspace input/output (UIO) devices to allow processes in Linux to `mmap` the PCI base address registers (BAR). Some of the inner workings of the module are highlighted in Listing 4.1, however, the cross VM connection module is publicly available [114], and is therefore omitted from Appendix A: vCDS Source Code. In line 1, BAR0, the event BAR, is set up by getting the bus start address for the PCI device region and mapping it into the UIO device physical memory address. The event BAR is used specifically to handle the previously discussed emits and consumes events through the Dataports. Line 3 sets the byte length of the PCI region of `addr` from line 1 and line 4 sets the internal address to the offset virtual address of the BAR's physical address so that the device can be accessed from within the module. Lines 6 through 14 repeat the above UIO device initialization for the remaining BAR mappings which correspond to the Dataports.

```
1  uio->mem[0].addr = pci_resource_start(dev, 0);
2  ...
3  uio->mem[0].size = pci_resource_len(dev, 0);
4  uio->mem[0].internal_addr = pci_ioremap_bar(dev, 0);
5  ...
6  for (i = 1; i < MAX_UIO_MAPS; i++)
7  {
8      uio->mem[i].addr = pci_resource_start(dev, i);
9      ...
10     uio->mem[i].internal_addr = ioremap_cache(pci_resource_start(dev, i),
11                                               pci_resource_len(dev, i));
```

```
12      ...
13      uio->mem[i].size = pci_resource_len(dev, i);
14      ...
15  }
```

Listing 4.1: Connection Module

4.3.5 Visualizing the CAmkES Model

A CAmkES visualizer, VisualCAmkES, was developed to aid in modelling CAmkES components and connections [114]. Modifications were made to the visualizer in order to generate Figures 4.5 and 4.6, and even then, the visualization does not accurately capture the directional data flow properties through dataports, nor does it capture the use of Cross VM Connectors. However, the depictions below are useful to model the components and general connections which have been implemented.

Consider the LowSide, HighSide, and Guard components of Figure 4.5. The box symbols represent Shared Data Dataports which connect via the leftmost and rightmost lines to each respective component. The triangle symbols represent emits and consumes Notifications which actually show directionality, i.e. the triangle points in the direction of the consumer. Figure 4.6 further provides the description of the connections, showing the number and type of each. These two figures provide an overview of the CAmkES components and their respective connections leveraged in vCDS.

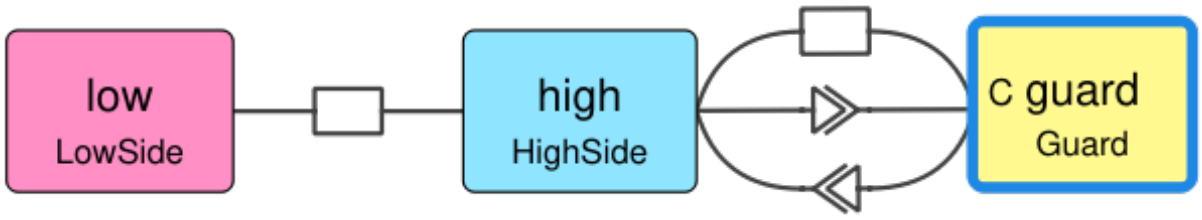

Figure 4.5: VisualCAmkES Components and Connections

4.3.6 Security Analysis

CAmkES grants all vCDS systems the assurance that the Components, Interfaces, and Connectors which have been specified in the ADL provide an accurate representation of all possible interactions

```
low contains:  lowToHighConn
        Number of connections is: 1
        dataport: 1
        provides: 0
        consumes: 0
        uses: 0
        emits: 0
high contains:  readyConn doneConn lowToHighConn highToGuardBridgeConn
        Number of connections is: 4
        dataport: 2
        provides: 0
        consumes: 1
        uses: 0
        emits: 1
guard contains:  readyConn doneConn highToGuardBridgeConn
        Number of connections is: 3
        dataport: 1
        provides: 0
        consumes: 1
        uses: 0
        emits: 1
```

Figure 4.6: VisualCAmkES Component Description

and that any interaction beyond what is specified will not materialize [65]. The implemented Dataport configuration is an explicit data diode that allows the passing of data structures through the protected seL4 kernel via a unidirectional interface without the possibility of leaking information through the component or kernel layers. Additionally, the analysis tool developed for this research, further described in Chapter 6: Analysis and Evaluation, takes the capDL as input and seeks to verify the security configuration of the system based on the system description in the capDL.

4.4 For Realizing Layer 3 in the Architecture: Linux and seL4 Native Process

To represent the High and Low Side domains, VMs are utilized to run a custom-built Linux kernel due to practicality of implementation (Appendix A: Custom Linux Kernel Config File). The Low Side handles incoming traffic and initiates transport of the appropriate data to the High Side for processing. Communication from the low component to the high component, through the aforementioned Dataports, is achieved through cross VM connectors. The cross VM connectors allow for the configuration of each process, within the guest VMs, to communicate with the CAmkES components. The High Side can then access and process the data by opening and reading from the Dataports. Communication from high to low within the network sensor (IPS/IDS) application is a

separate channel protected by a guard. The guard or filter which has been developed is an optional component which was specifically selected for use in the stream processor application.

4.4.1 Implementation

In this subsection are the details of the implementation of each mechanism and an explanation of the unique and essential aspects of the stream processor code. It is now useful to discuss the access right granted to each component and differentiate these rights from the access rights granted to each Dataport connection between those components.

As per BLP, described in Chapter 3.1: System Model, the "no read up" and "no write down" properties are enforced with the following access rights shown in table 4.1. Note that the High Side refers to the domain as the Guard is a party to high computations. Therefore, the High Side and the Guard components have the same access rights to the data which is being processed.

Component	**Access Rights**
Low Side	write-only
High Side	read-only
Guard	read-only

Table 4.1: Component Access Rights

The Low Side to High Side connection is equivalent to the access rights granted to the Low Side. Additionally, the connection between the High Side and/or the Guard back to the Low Side is equivalent to access rights granted to the Guard and High Side, shown in Table 4.1. The difference, shown in Table 4.2, is recognized in the High Side to Guard and the Guard to the High Side connections. The rights granted to these connections differ because the High Side is permitted to send messages to the Guard and, likewise, the Guard is permitted to communicate back to the High Side. This is possible without compromising data confidentiality because the Guard and the High Side reside at the same classification or trust level. Further discussion on the components' connections and how explicit rights enforce the data diode will reference Table 4.2.

Dataport Connection	**Access Rights**
Low Side to High Side	write-only
Guard to High Side	read/write
Guard and/or High Side to Low Side	read-only

Table 4.2: Shared Memory Access Rights

Domains

As stated above, each domain has been implemented as a VM running Linux. In this case, the Linux kernel `version 5.4.139` was chosen. This version was selected in order to make the stream processor compatible with multiple hardware platforms for testing which include, but are not limited to, those platforms discussed in Chapter 4.1: For Realizing Layer 1 in the Architecture: AMD EPYC with SEV/SME. The configuration file for the custom build is located in Appendix A: Custom Linux Kernel Config File.

Listing 4.2, below, shows the kernel image being located, decompressed, and added to the VM file server. The `AddToFileServer` receives two required arguments, the name which the kernel image will be referred to within the file server – `"bzImage"` – and the location of the image within the file system – the value of `decompressed_kernel`. Additional dependent arguments can be passed through the optional parameter, `DEPENDS`. Lines 2 and 3 show the High Side and Low Side root file systems being located while lines 15 and 16 show the file systems being added to the respective archive ("copy in and copy out" – CPIO) after having been packed with all necessary files and modules. Note that the `GetDefaultLinuxRootfsFile` function retrieves the custom-built default Linux `rootfs.cpio` as the default root file system for both the High Side and Low Side.

```
1  GetDefaultLinuxKernelFile(kernel_file)
2  GetDefaultLinuxRootfsFile(high_rootfs_file)
3  GetDefaultLinuxRootfsFile(low_rootfs_file)
4
5  DecompressLinuxKernel(extract_linux_kernel
6      decompressed_kernel
7      ${kernel_file}
8  )
9  AddToFileServer("bzImage"
10     ${decompressed_kernel}
11     DEPENDS extract_linux_kernel
12 )
13 ...
14 # Add high and low rootfs images to file server
15 AddToFileServer("low_rootfs.cpio" ${low_rootfs_file})
```

```
16 AddToFileServer("high_rootfs.cpio" ${high_rootfs_file})
```

Listing 4.2: Linux Kernel and File System Build Setup

The code in Listing 4.3, from *stream_processor.camkes* which is the main CAmkES project file, shows the definitions of the high and low VM components. On line 1, `VM_COMPOSITION_DEF` is a C preprocessor macro which defines the hardware multiplexing components and configurations: PCI bus configuration, a real time clock (RTC), a serial server, and a time server. The main concern of this macro is to define and configure any components that all VMs and/or hypervisors need. `VM_PER_VM_COMP_DEF` is a macro which has been modified to take two arguments: the domain name and a VM number. This macro defines each VM component and establishes all necessary connections to the VM components: connections between the VM and the file server; connections between the serial server and each VM component, as well as the serial input emulated by seL4; connections between the VM and the RTC, as well as the time server; and connections between the VM and the PCI configuration space. The PCI devices themselves are defined by each respective VM component as discussed below.

```
1 VM_COMPOSITION_DEF()
2
3         VM_PER_VM_COMP_DEF(LowSide, 0)  // component LowSide vm0
4         VM_PER_VM_COMP_DEF(HighSide, 1) // component HighSide vm1
```

Listing 4.3: Domain Component Definition

In order to better understand the following instantiated high and low domains, it is useful to understand the `Record` data structure which is defined in *record.h*:

```
1 typedef struct {
2   int isValid;   // Holds value of pass/fail
3   int isDone;    // Notification from guard to High
4   Tag tag;       // Blake3 checksum
5   Packet packet; // Packet data
6 } Record;
```

Listing 4.4: Record Struct

The purpose of the `Record` struct is to hold the data and information about the data so they can easily be accessed and processed by any authorized components which require it. The `Record` data

structure has four members which are described as follows: (i) `isValid`: Boolean value representing the outcome of an integrity check – 1 for successful, 0 for failed; (ii) `isDone`: Boolean value which represents the processing status of the guard – 1 for finished, 0 for in progress; (iii) `tag`: structure which holds a 32-byte char pointer that references a checksum; and (iv) `packet`: structure which contains the data as well as some metadata about the data.

Low Side Domain.

The Low Side domain implementation is presented first because of the sequential data flow through the CAmkES components, shown in Figure 4.1. Additionally, the Low Side implementation is not nearly as complex as the High Side implementation and is therefore useful in laying the foundation of vCDS domains. The Low Side domain implementation is organized and presented in the following way: (i) CAmkES Low Side component definition, (ii) CAmKES Low Side component configuration, (iii) Low Side connections, and (iv) Low Side root file system build and setup.

(i) CAmkES Low Side component definition

Listing 4.5 shows the `DeclareCAmkESVM` function being called from *CMakeLists.txt* in the project's root directory to declare the LowSide component which was defined in the *stream_processor.camkes* file, shown in Listing 4.3. The function also provides the support to pass in an additional source file as well as an include directory containing the necessary header files, such as the aforementioned *record.h*.

```
1 DeclareCAmkESVM(LowSide
2     EXTRA_SOURCES ${connectionSourceLow}
3     INCLUDES include
4 )
```

Listing 4.5: Low Side VM Declaration

(ii) CAmKES Low Side component configuration

The Low Side CAmkES component attributes for the base VM definition, such as the RTC, interrupt handlers for events, and the file server attributes, are set in the `VM_INIT_DEF` function, shown in Listing 4.6. Line 5 defines a shared memory Dataport which is of the previously defined `Record` data type. The `record` Dataport is used to communicate with, that is, to send data to, the High Side.

```
component LowSide {
    VM_INIT_DEF()

    include "record.h";
    dataport Record record;
}
```

Listing 4.6: Low Side Component Definition

Listing 4.7 below, from Appendix A.3: LowSide.camkes, shows the Low Side assembly, i.e. the complete description of the full component, being configured. `VM_PER_VM_CONFIG_DEF` initializes VM memory configuration attributes common to all VM guest instances. Lines 4 and 5 give the Low Side access to blocks of untyped memory, 8 megabyte and 4 megabyte respectively, so that the Low Side can retype them as its virtual memory. Lines 6 and 7 specify the VM guest's heap size allotment and the RAM size respectively. The lines following specify the command line console, kernel image file, root file system archive, and the I/O space at line 12. Finally, on line 13, the `record` Dataport is configured with a size allotment of 4096 bytes.

```
configuration {
    VM_PER_VM_CONFIG_DEF(0)

    vm0.simple_untyped23_pool = 21; // 2^23 (8 MB)
    vm0.simple_untyped22_pool = 1;  // 2^22 (4 MB)
    vm0.heap_size = 0x2000000;
    vm0.guest_ram_mb = 128;
    vm0.kernel_cmdline = LOW_SIDE_CMDLINE;
    vm0.kernel_image = "bzImage";
    vm0.kernel_relocs = "bzImage";
    vm0.initrd_image = "low_rootfs.cpio";
    vm0.iospace_domain = 0x0f;
    vm0.record_size = 4096;
}
```

Listing 4.7: Low Side Component Configuration

It is especially useful to the network sensor application to provide network access to the

untrusted guest VM so that it can receive untrusted communications from other untrusted sources. An Ethernet device is configured by giving the VM guest passthrough access to the Ethernet controller. Line 1 of Listing 4.8 sets the I/O space and lines 3 through 5 set the I/O ports. The starting and ending regions were retrieved from the native Linux instance by using the `lspci` command. Lines 7 through 19 set the PCI Ethernet controller device information such as the bus, device, and function numbers, as well as the additional memory regions. Finally, the IRQ information is set in lines 21 through 23.

```
1  vm0_config.pci_devices_iospace = 1;
2
3  vm0_config.ioports = [
4      {"start":0x3000, "end":0x3100, "pci_device":00, "name":"Ethernet5"},
5  ];
6
7  vm0_config.pci_devices = [
8      {
9          "name":"Ethernet5",
10       "bus":08,
11       "dev":00,
12       "fun":1,
13       "irq":"Ethernet5",
14       "memory":[
15           {"paddr":0xb4304000, "size":0x1000, "page_bits":64},
16         {"paddr":0xb4300000, "size":0x4000, "page_bits":64},
17       ],
18   },
19 ];
20
21 vm0_config.irqs = [
22     {"name":"Ethernet5", "source":0x12, "level_trig":1, "active_low":1, "dest"
      :10},
23 ];
```

Listing 4.8: Passthrough Ethernet Configuration

(iii) Low Side connections

Listing 4.9 below shows the instantiation of a connector by forming a connection from the Low Side VM `record` Dataport interface to the High Side VM `record` Dataport interface. Specifically, `seL4SharedDataWithCaps` is a connector used to form a connection between Dataport interfaces where both the "to" and the "from" sides have access to capabilities to the frames which back the Dataport. This connector is required in order to connect processes in a VM guest to other CAmkES components which is done at runtime when the hypervisor inserts new memory mappings into the guest VM's address space in order to establish shared memory. On line 6 the `record` Dataport's access rights are explicitly set. The enforcement of BLP access control for this implementation means that the Low Side has write only privileges so that "no read up" is enforced and that High Side data confidentiality is maintained. This implements the "from" side of the data diode such that the Low Side can only transfer data to the High Side and, again from the Low Side's perspective, data cannot be read back.

```
1 connection seL4SharedDataWithCaps lowToHighConn(
2     from vm0.record,
3     to vm1.record
4 );
5 ...
6 vm0.record_access = "W";  // write only
```

Listing 4.9: Low Side connections

(iv) Low Side root file system build and setup

The Low Side leverages a few external packages and additional files to complete processing operations. For the purpose of the stream processor, two are discussed: (a) the *send_record* project, and (b) the Cross VM Connection Kernel Module.

(a) *send_record* project

send_record is an external project which provides the Low Side VM with the functionality to send or transfer data along the CAmkES shared memory Dataport interface with the High Side. Recall that this is the first step along the data flow path, depicted in Figure 4.1. The following paragraph refers to Listing 4.10 below.

The `ExternalProject_Add` function (lines 1 through 7) declares the `send_record-app` ap-

plication and creates a custom target to build from the provided `SOURCE_DIR` on line 2. `AddExternalProjFilesToOverlay` is a helper function responsible for packing the application files generated by the external project into the overlay target, in this case `overlay_low`. Essentially, this function creates a target for the generated binary in the external project and adds it to the root file system. The function passes four required arguments and, in this case, one additional optional argument: (1) the external project target – `send_record-app`, (2) the external project's install directory – `${CMAKE_CURRENT_BINARY_DIR}/send_record-app`, (3) the overlay target to which to add the file – `overlay_low`, (4) the location of install within the root file system – `"usr/sbin"`, and (5) the file to add from the external project – `send_record`.

```
1  ExternalProject_Add(send_record-app
2      SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/pkgs/send_record
3      BINARY_DIR ${CMAKE_CURRENT_BINARY_DIR}/send_record-app
4      INSTALL_COMMAND ""
5      BUILD_ALWAYS ON
6      EXCLUDE_FROM_ALL
7  )
8
9  AddExternalProjFilesToOverlay(send_record-app
10     ${CMAKE_CURRENT_BINARY_DIR}/send_record-app
11     overlay_low
12     "usr/bin"
13     FILES send_record
14 )
```

Listing 4.10: Declare Send Record External Project

The `send_record-app` functionality is granted by one key system call, `mmap`, shown in Listing 4.11. The `mmap` function is a system call used to create a new mapping in the virtual memory address space of the current running process. The Low Side calls the `mmap` function on the file associated with the shared memory Dataport, captured by the `file_descriptor` parameter. This provides the mapping of the memory, which resides in the Dataport, to its own address space so that it can be processed. Note that within the call to `mmap`, the respective enforced software-level file permissions are explicitly equivalent

to those set for the Dataport configurations in the Root Project stream_processor.camkes file from Appendix A: vCDS Source Code. If the memory mapping fails, no data will be transferred to the High Side, upholding the system model, Chapter 3.1: System Model.

```
Record *dataport = (Record*)mmap(NULL, length, prot, flags,
    file_descriptor, 1 * getpagesize());
```

Listing 4.11: Send Record External Project

(b) Cross VM Connection Kernel Module

Listing 4.12 leverages `DefineLinuxModule` to define a new Linux module, `connection`, which is then added to the Low Side overlay, `overlay_low`, by `AddFileToOverlayDir`. In both functions, `connection-module` is used to define the location to the compiled Linux kernel module and `connection-target` is used to define the target name for the kernel module. The first argument in `DefineLinuxModule` is the location of the connection module project directory, and the final argument provides the Linux kernel source directory. The first argument of `AddFileToOverlayDir`, `"connection.ko"`, defines the name of the installed module and the third argument, `"lib/modules/5.4.139/kernel/drivers/vmm"`, defines the location within the overlay where the module will be placed.

```
DefineLinuxModule(
    ${CAMKES_VM_LINUX_DIR}/camkes-linux-artifacts/camkes-linux-modules/
    camkes-connector-modules/connection
    connection-module
    connection-target
    KERNEL_DIR ${CMAKE_CURRENT_SOURCE_DIR}/5.4.139/linux-5.4.139/
)
...
AddFileToOverlayDir("connection.ko"
    ${connection-module}
    "lib/modules/5.4.139/kernel/drivers/vmm"
    overlay_low
    DEPENDS connection-target
)
```

Listing 4.12: Adding the Cross-VM kernel module to the Low Side overlay target

Recall from the declaration of the Low Side CAmkES VM, in Listing 4.5, that `connectionSourceLow` was added as an additional source file to be compiled. This source file *cross_vm_connections_low.c* explicitly defines the connection handles and initializes them. Listing 4.13 below shows the Low Side `connections` which correspond to the Low Side Dataport, `record`. The data structure below is a struct of connections, which, in this case, only contains one. The connection itself is also a data structure which receives a handle to the VM as well as an additional function and an additional data structure which will be examined when the High Side connections are introduced.

```
static struct camkes_crossvm_connection connections[] = {
   {&record_handle, NULL, {.id = -1, .reg_callback = NULL}},
};
```

Listing 4.13: Low Side Connection Source

Listing 4.14 shows `AddFileToOverlayDir` adding the cross VM module initialization shell script as `"S90cross_vm_module_init"` to the `"etc/init.d"` directory within the Low Side overlay. Listing 4.15 shows the use of `insmod` to insert the module into the Linux kernel which will run at boot because it was placed in `"etc/init.d"`.

```
AddFileToOverlayDir("S90cross_vm_module_init"
    ${CMAKE_CURRENT_SOURCE_DIR}/overlay_files/init_scripts/
    cross_vm_module_init
    "etc/init.d"
    overlay_high
)
```

Listing 4.14: Add script to insert the Cross-VM module into the High Side Linux kernel

```
insmod /lib/modules/5.4.139/kernel/drivers/vmm/connection.ko
```

Listing 4.15: Insert Module Script

Once all packages and files intended for the Low Side have been packed into `overlay_low`, the overlay is then packed into the `rootfs.cpio` archive. Listing 4.16 shows the function call to `AddOverlayDirToRootfs` which is a helper function responsible for installing the defined target overlay, `overlay_low`, onto the stated rootfs image, `low_rootfs_file`. Recall that both the high

rootfs and the low rootfs files are built on top of the same default file system: the custom-built default Linux rootfs file. The next argument, `"buildroot"` specifies the distribution of the rootfs image. `rootfs_install` specifies how the image will be installed which, in this case, is as the Low Side guest VM file system that will be used when the Low Side VM boots. The final arguments specify the output location and the target of the overlayed rootfs image.

```
AddOverlayDirToRootfs(
    overlay_low
    ${low_rootfs_file}
    "buildroot"
    "rootfs_install"
    low_rootfs_file
    low_rootfs_target
)
```

Listing 4.16: Packing the High Side Overlay

High Side Domain. As was done with the Low Side domain, the High Side domain implementation is organized and presented in the following way: (i) CAmkES High Side component definition, (ii) CAmKES High Side component configuration, (iii) High Side connections, and (iv) High Side root file system build and setup.

(i) CAmkES High Side component definition

Listing 4.17 shows the `DeclareCAmkESVM` function being called, this time, to declare the `HighSide` component. Once again, the helper provides the support to pass in an additional source file and an include directory containing the necessary header files.

```
DeclareCAmkESVM(HighSide
    EXTRA_SOURCES ${connectionSourceHigh}
    INCLUDES include
)
```

Listing 4.17: High Side VM Declaration

(ii) CAmKES High Side component configuration

Like with the Low Side, the High Side component attributes for the base VM definition – the RTC, interrupt handlers for events, and the file server attributes – are set by the `VM_INIT_DEF` function in Listing 4.18. The emits and consumes event connections are defined in lines 4 and 5. Shown from these definitions, the High Side emits a `Ready` signal to another component, which is the Guard, and consumes a `Done` signal, also from the Guard. Lines 8 and 9 define shared memory Dataports which will store the previously described `Record` data structure. The `record` Dataport is used to receive data from the Low Side, and the `record_bridge` is used to communicate bidirectionally with the Guard.

```
1  component HighSide {
2      VM_INIT_DEF()
3
4      consumes Done done;
5      emits Ready ready;
6
7      include "record.h";
8      dataport Record record;
9      dataport Record record_bridge;
10 }
```

Listing 4.18: High Side Component Definition

Listing 4.19, from Appendix A.4: HighSide.camkes, shows the High Side assembly being configured.

`VM_PER_VM_CONFIG_DEF` sets several important memory attributes such as the file server shared memory size and the global endpoint address used for notifications which are especially important to the High Side. Lines 4 and 5 give the High Side access to 8 megabyte and 4 megabyte blocks of untyped memory, respectively, so that the High Side can retype them as its virtual memory. Lines 6 and 7 specify the VM guest's heap size allotment and the RAM size respectively. The lines following specify the command line console, kernel image file, root file system archive, and the I/O space. Beginning at line 14, the `record` and `record_bridge` Dataports are configured with differing identification numbers – an important distinction when concerned with their respective access rights – but with the same size allotments as they will be handling the same amounts of data.

```
configuration {
    VM_PER_VM_CONFIG_DEF(1)

    vm1.simple_untyped23_pool = 21; // 2^23 (8MB)
    vm1.simple_untyped22_pool = 1;  // 2^22 (4MB)
    vm1.heap_size = 0x2000000;
    vm1.guest_ram_mb = 128;
    vm1.kernel_cmdline = HIGH_SIDE_CMDLINE;
    vm1.kernel_image = "bzImage";
    vm1.kernel_relocs = "bzImage";
    vm1.initrd_image = "high_rootfs.cpio";
    vm1.iospace_domain = 0x10;

    vm1.record_id = 1;
    vm1.record_size = 4096;
    vm1.record_bridge_id = 2;
    vm1.record_bridge_size = 4096;
}
```

Listing 4.19: High Side Component Configuration

(iii) High Side connections

The listing below, Listing 4.20, defines the event connections, `readyConn` and `doneConn`, as well as the Dataport connections, `lowToHighConn` and `highToGuardBridgeConn`. Lines 1 through 4 establish the connection from the High Side to the Guard – an emits event on the High Side, `vm1`, and a consumes event on the Guard, `guard`. The `seL4Notification` is an asynchronous event connector such that a consumer will wait for an asynchronous signal from the emitter. The emitter is non-blocked and can send data at any time. The consumer may be either blocked or non-blocked. In the case that the consumer executes faster than the emitter, this connector has the advantage of allowing the consumer to execute additional jobs while waiting for the signal from the emitter. Likewise, lines 5 through 8 establish the connection from the Guard to the High Side – an emits event on the Guard and a consumes event on the High Side.

Line 10 instantiates a connector by forming a connection between the High Side VM and the Low Side VM Dataports. Specifically, `seL4SharedDataWithCaps` is a connector used to form

a connection between Dataport interfaces where the "to" side has access to capabilities to the frames backing the Dataport. Once again, this is a requirement for Dataport connections between VMs in order to connect processes in a VM guest to other CAmkES components. This same connector is used to form a connection between the High Side `record_bridge` Dataport and the Guard `record` Dataport. Note that, in the connection formed from the High Side to the Guard in line 14, the High Side uses a different Dataport than that which it uses to retrieve data from the Low Side. Once again, this is because the High Side and the Guard reside at the same classification level. Also, notice that no connections are made where the Low Side, `vm0`, is on the "to" side of the connection. This is because this particular stream processor implementation does not require a connection back to the Low Side; however, in a use-case that would require it, it can easily be added by using the `seL4SharedDataWithCaps` connector.

Finally, in lines 19 and 20, each Dataport's access rights are explicitly set. The enforcement of BLP access control for this implementation means that the High Side has read only privileges, i.e. "no write down", for data that comes from the Low Side. This implements the "to" side of the data diode in that the High Side can only receive data from the Low Side and is not permitted to transfer data back to the Low Side via this interface. The High Side has both read and write privileges to and from the Guard as they reside at the same classification level; for this interface, there is no data diode. Note that if the High Side were given the ability to transfer the data back to the Low Side in this use-case, it would have to form a new connection with a new Dataport which would require new access controls in order to enforce the data diode.

```
1 connection seL4Notification readyConn(
2     from vm1.ready,
3     to guard.ready
4 );
5 connection seL4Notification doneConn(
6     from guard.done,
7     to vm1.done
8 );
9 ...
```

```
connection seL4SharedDataWithCaps lowToHighConn(
    from vm0.record,
    to vm1.record
);
connection seL4SharedDataWithCaps highToGuardBridgeConn(
    from vm1.record_bridge,
    to guard.record
);
...
vm1.record_access = "R";          // read only
vm1.record_bridge_access = "RW"; // read/write
```

Listing 4.20: High Side connections

(iv) High Side root file system build and setup

Additionally, the High Side leverages several external packages and additional files to complete processing operations. For the purpose of the stream processor, three are discussed: (a) *receive_record* project, (b) Snort project, and (c) Cross VM Connection Kernel Module.

(a) *receive_record* project

The first package is an external project called `receive_record-app` which provides the High Side VM with the functionality to retrieve data from the CAmkES shared memory Dataport and optionally, if specified as it is in the stream processor application, provide data access to the Guard component, described in Chapter 4.4.1: seL4 Native Process: the Guard. The following paragraphs closely parallel the previous examination of the Low Side file system build.

In Listing 4.21, the `ExternalProject_Add` function declares `receive_record-app` and creates a custom target to build. Note the `CMAKE_ARGS` where the C compiler for compiling the project is specified; the variable `CMAKE_C_COMPILER` is set in the external project's *CMakeLists.txt* file. This is because several of the external projects which are added to the stream processor had to be built for different architectures by using a cross compilation toolchain. This challenge is further discussed in Chapter 4.7: Implementation Challenges.

`AddExternalProjFilesToOverlay` is a helper function responsible for packing the application files generated by the external project into the overlay target, in this case `overlay_high`. Once again, this function creates a target for the generated binary in the external project and adds it to the root file system. The function passes four required arguments and, like with the `send_record-app`, one additional argument: (1) the external project target – `receive_record-app`, (2) the external project's install directory – `${CMAKE_CURRENT_BINARY_DIR}/receive_record-app`, (3) the overlay target to which to add the file – `overlay_high`, (4) the location of install within the root file system – `"usr/sbin"`, and (5) the file to add from the external project – `receive_record`.

```
ExternalProject_Add(receive_record-app
    SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/pkgs/receive_record
    BINARY_DIR ${CMAKE_CURRENT_BINARY_DIR}/receive_record-app
    INSTALL_COMMAND ""
    BUILD_ALWAYS ON
    EXCLUDE_FROM_ALL
    CMAKE_ARGS -DCMAKE_C_COMPILER=\${CMAKE_C_COMPILER}
)

AddExternalProjFilesToOverlay(receive_record-app
    ${CMAKE_CURRENT_BINARY_DIR}/receive_record-app
    overlay_high
    "usr/sbin"
    FILES receive_record
)
```

Listing 4.21: Declare Receive Record External Project

The `receive_record-app` functionality is granted by one central function call, shown in Listing 4.22. The `mmap` system call is used the same way here, as with the Low Side, to create a new mapping in the virtual memory address space of the current running process. In this case, the High Side calls the `mmap` function on the file associated with the Dataport, captured by the `record` parameter, in order to map the memory which resides in the Dataport to its own address space so that it can be processed. Likewise shown, the High Side maps the Dataport bridge file, captured by the `record_bridge` parameter,

to its address space where it can then write data to be accessed and processed by the Guard.

```
Record *record_dataport = (Record*)mmap(NULL, size, record_prot, flags,
    record, 1 * getpagesize());
...
Record *record_bridge_dataport = (Record*)mmap(NULL, size, bridge_prot,
    flags, record_bridge, 1 * getpagesize());
```

Listing 4.22: Receive Record External Project

Note that, in the case of each call to `mmap`, the respective enforced software-level file permissions are explicitly equivalent to those set for the Dataport configurations in the Root Project stream_processor.camkes file in Appendix A.2. If the memory mapping fails, no data are received by any of the High Side components and therefore all high data are protected. If the mapping fails on the write to the `record_bridge` Dataport, then the Guard will not be permitted by the High Side domain to read or process any of the data. In the case where the High Side expects to send the data to the Guard for processing before being sent back to the Low Side, the High Side will stop processing the data immediately so that no data are sent back to the Low Side, ensuring confidentiality.

(b) Snort project

Another package included on the High Side is the IDS/IPS, Snort. This package is used to examine and process the data as they pass through the High Side. Snort is especially useful when the Ethernet passthrough device is leveraged on the Low Side to receive network packets. The packets received on the Low Side can be transferred to the High Side where they will be scanned by Snort to determine subsequent handling of the data. Listing 4.23, similar to what was done with *receive_record*, shows the `ExternalProject_Add` function which creates a custom target to build the application. The `AddExternalProjFilesToOverlay` function, as stated previously, accepts four required arguments and one additional argument in order to add the files generated from the external project to the High Side overlay.

```
ExternalProject_Add(snort-app
    SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/pkgs/snort
```

```
    BINARY_DIR ${CMAKE_CURRENT_BINARY_DIR}/snort-app
    INSTALL_COMMAND ""
    BUILD_ALWAYS ON
    EXCLUDE_FROM_ALL
)

AddExternalProjFilesToOverlay(snort-app
     ${CMAKE_CURRENT_BINARY_DIR}/snort-app
     overlay_high
     "usr/sbin"
     FILES snort
)
```

Listing 4.23: Declare Snort Package

(c) Cross VM Connection Kernel Module

Like with the Low Side, a new Linux `connection` module is defined using `DefineLinuxModule`. Listing 4.24 below shows how the module is added to the High Side overlay, `overlay_high`, by calling `AddFileToOverlayDir`. Once again, `connection-module` refers to the location of the compiled Linux kernel module and `connection-target` is used to define the target name for the kernel module. `"connection.ko"` defines the name of the installed module, and `"lib/modules/5.4.139/kernel/drivers/vmm"`, defines the location within `overlay_high` where the module will be placed.

```
AddFileToOverlayDir("connection.ko"
    ${connection-module}
    "lib/modules/5.4.139/kernel/drivers/vmm"
    overlay_high
    DEPENDS connection-target
)
```

Listing 4.24: Adding the Cross-VM kernel module to the High Side overlay target

Recall, once again, from the declaration of the High Side CAmkES VM that the file represented by the `connectionSourceHigh` variable was added as an additional source file to be compiled. This source file, *cross_vm_connections_high.c*, explicitly defines the connection handles and initializes them. Listed below are the High Side connections which

correspond to the High Side Dataports. The data structure in Listing 4.25 is a `struct` of connections. Each connection itself is also a struct which takes a handle to the VM, an emit function, and a consumes event struct as arguments. As previously discussed, the only emits/consumes events in the stream processor application are sent between the High Side and the Guard, thus they only pass through the `record_bridge` Dataport. In the case of the bridge Dataport, there is a `ready_emit` function – CAmkES prefixes the name of the interface instance, "ready", to the function being called across the interface, `_emit()`, so that the application can transparently interact with the event – and a consumes event struct with a positive identifier which has no use for a callback for this implementation. In contrast, the `record` Dataport, which servers as the connection between the High Side and the Low Side, does not have an emits function and has a negative consumes event identifier which means it is not used.

```
static struct camkes_crossvm_connection connections[] = {
   { &record_handle, NULL, {.id = -1, .reg_callback = NULL }},
   { &record_bridge_handle, ready_emit, {.id = 1, .reg_callback = NULL}},
};
```

Listing 4.25: Connection Source

Listing 4.26 shows `AddFileToOverlayDir` adding the cross VM module initialization shell script as `"S90cross_vm_module_init"` to the `"etc/init.d"` directory, this time, within the High Side overlay. As with the Low Side, `insmod` is used in a shell script to insert the module into the Linux kernel, placed in `"etc/init.d"`, which will run at boot.

```
AddFileToOverlayDir("S90cross_vm_module_init"
    ${CMAKE_CURRENT_SOURCE_DIR}/overlay_files/init_scripts/
    cross_vm_module_init
    "etc/init.d"
    overlay_high
)
```

Listing 4.26: Add script to insert the Cross-VM module into the High Side Linux kernel

Once all packages and files intended for the High Side have been packed into the overlay, the overlay is then packed into the `rootfs.cpio` archive. Listing 4.27 shows the same function

call to `AddOverlayDirToRootfs` as was used for the Low Side, which installs the target overlay, `overlay_high`, onto the rootfs image, `high_rootfs_file`. The final arguments closely mirror those which were passed for the Low Side: `"buildroot"` specifies the distribution of the rootfs image, `rootfs_install` specifies that the image will be installed as the guest VM file system it will use when it boots, and the final arguments specify the output location and the target of the overlayed rootfs image respectively.

```
1    AddOverlayDirToRootfs(
2        overlay_high
3        ${high_rootfs_file}
4        "buildroot"
5        "rootfs_install"
6        high_rootfs_file
7        high_rootfs_target
8    )
9
```

Listing 4.27: Packing the High Side Overlay

seL4 Native Process: the Guard.

The seL4 native process is really a thread; recall from Chapter 4.2.1: seL4 Technical Overview that seL4 does not really have a concept of processes. The thread is the execution context that is responsible for running a CAmkES component which, in this case, is the Guard. Defining the Guard component through CAmkES is very straight forward as seen in Listing 4.28.

```
1 component Guard guard;
```

Listing 4.28: Guard Definition

Unlike the VM components, the Guard is very simple but powerful. The listing below shows the component configuration which specifies the Guard with the `control` keyword. Control makes the Guard an active component by providing it with a `run` method (see Listing 4.32). Similarly to the High Side, the Guard also consumes an event which, this time, is a `Ready` event and emits a `Done` event. Finally, on line 8, the Guard is given access to a shared memory Dataport which is a `Record` data structure type.

```
component Guard {
    control;

    consumes Ready ready;
    emits Done done;

    include "record.h";
    dataport Record record;
}
```

Listing 4.29: Component Configuration

The `record` Dataport, shown in Listing 4.30, is configured as follows: the Dataport is of size 4096 bytes and has read and write access to the shared memory. Recall that there is a bidirectional connection between the Guard and the High Side so no data diode is enforced. This particular implementation of the stream processor use-case does not call for a connection from the Guard back to the Low Side. However, if the use-case required an additional connection from the Guard to the Low Side, the new connection would be formed with a new Dataport which would require explicit access controls to form the data diode.

```
guard.record_size = 4096;
...
guard.record_access = "RW";
```

Listing 4.30: Dataport Configuration

Now the packages and dependencies which are needed to build and run the Guard will be described. Listing 4.31 uses the `DeclareCAmkESComponent` helper function at line 12, similarly to `DeclareCAmkESVM` which was used for the domain components, declares the Guard component with its source and header files. As seen from lines 2 through 10, and lines 13 through 22, there are several internal and external files which must be added to achieve the desired functionality. Specifically, there three internal source files which are essential in achieving this functionality: *main.c*, *integrity.c*, and *disposition.c*.

```
file(GLOB LIB_DIR lib/libblake3_x86-64)
set(DEPS
    ${LIB_DIR}/src/blake3.c
```

```
    ${LIB_DIR}/src/blake3_dispatch.c
    ${LIB_DIR}/src/blake3_portable.c
    ${LIB_DIR}/src/blake3_avx2_x86-64_unix.S
    ${LIB_DIR}/src/blake3_avx512_x86-64_unix.S
    ${LIB_DIR}/src/blake3_sse2_x86-64_unix.S
    ${LIB_DIR}/src/blake3_sse41_x86-64_unix.S
)

DeclareCAmkESComponent(Guard
    SOURCES
    ${DEPS}
    ${CMAKE_CURRENT_SOURCE_DIR}/components/Guard/src/main.c
    ${CMAKE_CURRENT_SOURCE_DIR}/components/Guard/src/integrity.c
    ${CMAKE_CURRENT_SOURCE_DIR}/components/Guard/src/disposition.c
    INCLUDES
    include
    ${LIB_DIR}/include
    /usr/lib/gcc/x86_64-linux-gnu/${GCC_VERSION}/include
)
```

Listing 4.31: Setting the Guard Dependencies

The Guard's *main.c* is shown in Listing 4.32. As previously stated, the `control` keyword makes the Guard an active component and provides it with a control thread that executes a `run` function one time during the boot of the system. The function itself is quite simple: in order to keep the thread running, all operations are placed inside an infinite loop. Line 5 of the code calls a function which waits and polls for an emits event from the High Side. If an event is received from the High Side, it means that the High Side has written data to shared memory for the Guard to begin operations. The next function calls are for the integrity and disposition guard operations which will be discussed later. Line 10 sets a value to be emitted back, when `done_emit` is called, to the High Side to notify that the Guard has completed.

```
int run(void)
{
   while (1)
   {
```

```
5       ready_wait();
6
7       checkIntegrity();
8       runDisposition();
9
10      record->isDone = 1;
11      done_emit();
12   }
13   return 0;
14}
```

Listing 4.32: Guard Run Method

As described in Chapter 3: The vCDS Architecture, the integrity checker leverages the Blake3 hashing algorithm. Shown in Listing 4.33, the integrity portion of the Guard leverages two central functions: `calculateCheckSum` and `checkIntegrity`. As seen above `checkIntegrity` is called from the initial control thread's `run` function. This function first calls `calculateCheckSum` to initialize the Blake3 hasher in lines 3 and 4, then load the hasher in line 6 with the data value to be hashed. The checksum is then calculated in line 7 and placed in the `checksum` byte array.

Finally, in line 15, the `checkIntegrity` function enlists the help of another function, `tagMatches`, which compares the tag that was originally calculated on the Low Side with the checksum that was just produced. If the hashes match, the `isValid` flag is set to 1 – `true` – otherwise, the value is set to 0 – `false`. This value will be read later by the High Side to determine if the data are permitted to pass back to the Low Side. The assignment to `isValid` shows the reason why `seL4GlobalAsynch` was used for the event connection from the Guard to the High Side. Here, a value is set within the Dataport to be passed back to the High Side for processing and verification.

```
1 void calculateCheckSum(CDS_BYTE *checksum)
2 {
3    blake3_hasher hasher;
4    blake3_hasher_init(&hasher);
5
6    blake3_hasher_update(&hasher, record->packet.value, record->packet.size);
7    blake3_hasher_finalize(&hasher, checksum, BLAKE3_OUT_LEN);
8 }
```

```
9
10 void checkIntegrity()
11 {
12    CDS_BYTE checksum[BLAKE3_OUT_LEN];
13    calculateCheckSum(checksum);
14
15    if (tagMatches(checksum))
16      record->isValid = 1;
17    else
18      record->isValid = 0;
19 }
```

Listing 4.33: Integrity Guard

For the purposes of experimentation and analysis, the disposition portion of the Guard has been disabled because the Ethernet passthrough was not required for the use-case environment. However, in an environment where the passthrough is enabled, the disposition will check portions of the data against a white list or black list. In one implementation that leverages this functionality, the source and/or destination internet protocol (IP) addresses were extracted from each packet and checked against a list of black-listed IP addresses and a list of white-listed IP addresses. If the disposition guard finds something of importance, that is, there is a match from one of the lists, the guard will send a notification back to the High Side which will determine the appropriate course of action necessary to resolve the alert.

4.4.2 Security Analysis

One of the processes applied to the data on the High Side is an IPS/firewall, which, in this case, is Snort [118]. The filter adheres to the primary objective of integrity by implementing both an integrity guard and a firewall which can be referred to as the disposition guard. The integrity guard ensures that, upon crossing a trust boundary, the data have not been modified. This particular stream processor implementation leverages the speed and security of the Blake3 cryptographic hashing algorithm in order to check the integrity of the data and verify that no High Side data are disclosed to the Low Side. The disposition guard, depending on the design, filters packets in or out based on a list of source IP addresses, ports, and other properties to further mitigate threats.

It is noteworthy to reveal that the Guard, though built upon seL4 is not formally verified, as Blake3 is also not formally verified. This highlights the need for formal verification of functional correctness and memory safety. One of the steps that has been taken to mitigate the risks associated with the lack of formal verification is to practice n-version programming (NVP) [2, 59, 131]. Practically, multiple different versions of the guard implementations have been added to the Low and High sides which must agree in order for acceptance to take place. To further address memory safety, an additional implementation of Blake3 in Rust, a memory-safe systems programming language, was leveraged.

Furthermore, it is useful to note that the Guard is implemented towards compliance with one of the NCDSMO's Raise-The-Bar concepts which is similar to the NEAT concept, described in Chapter 2.2.1: Multiple Independent Layers of Security, called RAIN [51].

1. *Redundant* – security-relevant mechanisms must be executed multiple times.
2. *Always Invoked* – security-relevant mechanisms are always executed.
3. *Independent Implementations* – security-relevant mechanisms have multiple implementations.
4. *Non-Bypassable* – security-relevant mechanisms cannot be bypassed.

The difference between the described stream processor Guard instantiation and a RAIN compliant filter is the lack of redundancy in the Guard. Redundancy (R) [94] has since been implemented on subsequent versions of the stream processor implementation. The use of NVP ensures that there are independent implementations (I) [59] of the Guard and the formally verified properties of the TCB ensure that the Guard is always invoked (A) and non-bypassable (N) [21, 32, 98].

4.5 Data Diode Solution

In addition to the vCDS implementation with virtual domains, a bare-bones data diode using the vCDS architecture was implemented. The reason for this implementation was to measure the performance of a vCDS instantiation without the use of virtualized domains, in order to better compare vCDS to real-world solutions. The results of this evaluation can be seen in Chapter 6: Analysis and Evaluation.

The main difference between the implementation of the aforementioned stream processor and the bare-bones data diode is that both the High Side and the Low Side are seL4 native processes. In the *CMakeLists.txt* project file of the stream processor, the `DeclareCAmkESVM` function is used to declare the CAmkES VM. In the data diode instantiation, the High and Low Sides are declared using `DeclareCAmkESComponent` as follows:

```
DeclareCAmkESComponent(Low SOURCES low.c)

DeclareCAmkESComponent(High SOURCES high.c)
```

Listing 4.34: Declare Diode Components

Similarly, the High and Low components are defined in Listing 4.35 (note that no Guard was implemented). The method chosen for the transfer of data was also the same: CAmkES Dataports. Not shown is the implementation of a `TimeServer` component for benchmarking the solution. The `TimeServer` component definition and implementation is given in the seL4 documentation [114].

```

component Low {
  ...
  dataport Buf buffer;
}

component High {
  ...
  dataport Buf buffer;
}
```

Listing 4.35: Diode Component Definitions

The Dataports were configured similarly to the stream processor and are shown in Listing 4.36, with the remainder of the assembly. Though the differences are minimal, note the use of `seL4SharedData` instead of `seL4SharedDataWithCaps` as with the cross VM Dataports.

```
assembly {
    composition {
          component Low low;
          component High high;
```

```
5
6           connection seL4SharedData lowToHighConn(
7               from low.buffer,
8               to high.buffer
9           );
10      }
11
12      configuration {
13          ...
14          low.buffer_access = "W";
15          high.buffer_access = "R";
16      }
17  }
```

Listing 4.36: Diode Dataports Assembly

4.6 vCDS Build Process

The vCDS build process has been fully automated, not just for the stream processor application, but for any instantiation of vCDS provided the proper component modifications have taken place as required by the use-case. It is noteworthy that the custom built Linux kernel base requires modifications relative to the domain which it will represent. In the case of the stream processor, the addition of these modifications are fully automated and conducted during the build process. For a different use-case, like the Data Diode Solution above, the process is similar once the modifications have taken place. Note that the build process has been fully automated for the Data Diode Solution as well.

4.7 Implementation Challenges

The implementation of vCDS came with many challenges including, but not limited to the following:

(i) the libraries which were essential to the Low Side, such as `libblake3` and `libpcap`, had to be compiled for both i686 and x86_64 architectures so that they could be used in the VM and with the seL4 native process;

(ii) full automation of the build process took a significant amount of time do to the scale of the system and component implementations with multiple architectures;

(iii) several versions of the custom Linux kernel and cross-compile toolchains had to be built and on hand for switching between machine architectures which included the servers and different development machines;

(iv) the benchmark values from the initial experiments were calculated on four layers of virtualization and did not garner competitive results;

(v) VisualCAmkES (see Chapter 4.3.5: Visualizing the CAmkES Model) did not work out of the box and had to be modified extensively to work properly;

(vi) the `time` C library could not be used to establish the benchmarks for the data diode solution as the functions were not supported for the native seL4 processes. This was worked around by the development of a native seL4 process which acts as a time server.

Chapter 5

Auditing a Cross Domain Solution

As was discussed in Chapter 6.1: seL4 Proofs, seL4 has been formally verified for the object security guarantees of confidentiality, integrity, and availability [44]. However, building a system which leverages a formally verified TCB does not necessarily mean that the system is secure out of the box. In fact, in order for a system to be proven trustworthy, that system's security configuration must be verified to ensure that no access rights can be propagated beyond what is specified in the system specification. The security analysis of vCDS is discussed below as well as an evaluation using the presented auditing tool. Note that the following sections, including diagrams, have been adopted from "Auditing a Software-Defined Cross Domain Solution Architecture", in preparation [35].

5.1 Introduction

In the context of security controls, a *leak* refers to the addition or acquisition of authority over an object by a subject that did not previously have that authority. A system must have proper security configurations in order to prevent unintended results such as privilege leaks. One critical goal of any system's security model should be its trustworthy implementation. The status quo in CDS implementations rely on trust rather than trustworthiness. Trust in a system is the firm belief that the system will perform as expected. Trustworthiness in a system is the proven property of that system to perform as expected. One system which seeks to provide a trustworthy implementation for cross domain capabilities is vCDS [33].

vCDS is the first and, at the time of this writing, the only CDS built upon a formally verified trusted computing base (TCB). The TCB has been formally verified for functional correctness and security guarantees of confidentiality, integrity, and availability (CIA) [44, 65]. In one instantiation of vCDS's trustworthy architecture, the seL4 microkernel and hypervisor was leveraged. seL4 serves as the lower level component of a system which can be abstracted by CAmkES, the component architecture for microkernel-based embedded systems [114]. An important part of vCDS is the architecture description language (ADL) provided by CAmkES. The ADL describes the components, interfaces, connectors, and privileges which make up a system. Given these combined architectural components, vCDS is a complete, formally verified CDS and comes with the security guarantees of CIA which can be verified, through the ADL, to ensure that the system does not violate any CDS constraints [44].

5.2 Problem Statement

Building a system which leverages a formally verified TCB does not necessarily mean that the system is secure out of the box; the security guarantees provided by the TCB only hold if the system's security controls have been configured correctly. The serious nature of security misconfiguration is underscored by Open Web Application Security Project's (OWASP) rating it as number five in the top ten most critical security concerns in applications in 2021 [107].

In order for a vCDS instantiation to be proven trustworthy, the system must be audited to verify that no CDS information flow properties are violated. In other words, the ADL specification of the vCDS system security properties must be verified against the specification for the vCDS implementation. The problem of description verification often stems from poorly modeled system security properties. Another issue that arises is that current CDS solution components are either not described in an ADL, or component descriptions are not properly expressed in such a way as to differentiate between the levels of protection necessitated by the sensitivity of the components.

5.3 Contributions

In the following sections,

(i) the need to verify the implementation of a CDS system security model is addressed.

(ii) A tool is presented which, as evidenced by a thorough search of the relevant literature, is the first to analyze and audit CDS security control implementations. The implementation of the algorithm parses the system description and verifies that no CDS constraints are violated such that there are no operations which will lead to a leak of rights or data. Furthermore, as again evidenced by a thorough search of the relevant literature,

(iii) this is the first and only development of a tool which audits a CDS described via an ADL, and the first to

(iv) tailor an ADL for describing a CDS system with the ability to tag components with proper labels which propagate down through the ADL, allowing the algorithm to check the constraints.

(v) The use of the ADL is extended to include labels and key words which trigger the appropriate protection models and information flow constraints in vCDS to determine whether or not to encrypt the data. This capability is not only important for auditing a CDS system and allowing for proper labelling but

(vi) is extended to provide system security modeling that is far beyond the status quo.

5.4 Overview of the vCDS Security Infrastructure

Presented in Chapter 3: The vCDS Architecture was a layered vCDS architecture which includes the optional hardware protections, the software computing base, and the components [33]. The primary focus of this section is to present the computing base with some carryover into the domain components through a discussion on the security infrastructure of vCDS.

Figure 5.1 omits the hardware and TEE layers and instead focuses on the TCB, the component architecture abstraction layer, and the component layer which depicts three of the components employed in the stream processor, described in [33]: (i) Low Side, (ii) High Side, and (iii) Guard. The Low Side corresponds to a lower classification or trust level and functions to retrieve the data and, if the use-case requires it, calculate an integrity tag before writing the data to the High Side.

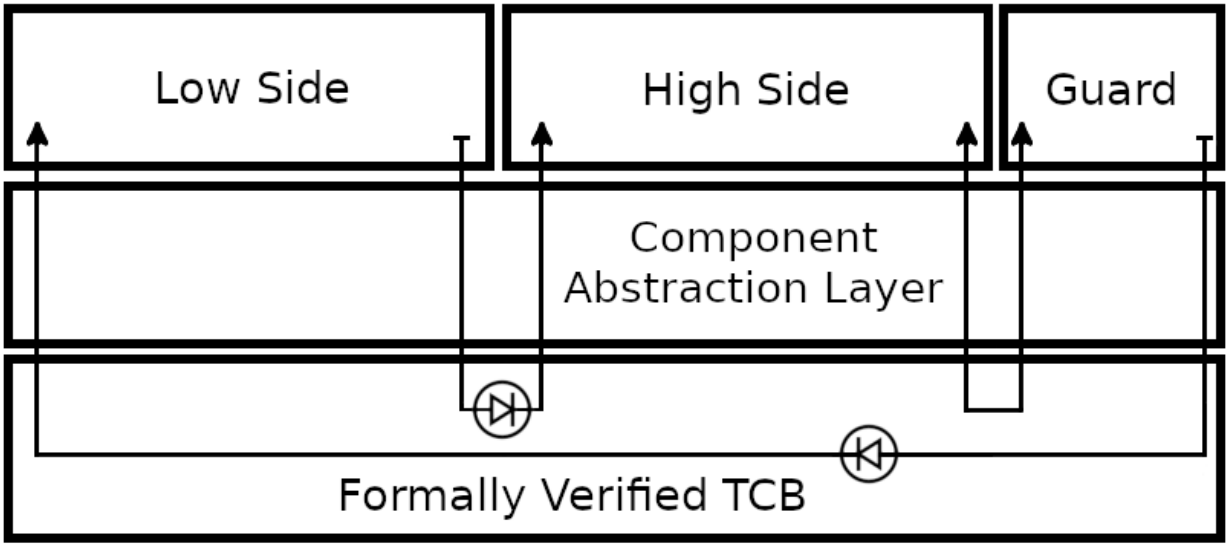

Figure 5.1: vCDS stream processor architecture

The High Side corresponds to a higher classification or trust level and will read the data from the Low Side and execute any predefined, trusted operations on the data. The High Side may then write the data to the Guard which, depending on the use-case, may function to check the integrity of the data, ensuring that no modifications to the data have occurred and that no operational data have been added to the stream. If the data are cleared, the Guard may pass the data back to the Low Side component. If the data have been modified, the Guard will notify the High Side and determine the next course of action. Guard operations are typically automatic and are implemented relative to the use-case and environment [60].

5.5 vCDS Security Model

The vCDS TCB leverages the capability model described in Chapter 2.3.2: Capability Model. At kernel boot time, all physical memory resources, such as *untyped* kernel objects, are allocated [46]. Contiguous memory blocks hold untyped memory objects, some of which are writable by the kernel [114]. Each untyped object can be retyped as a capability reference to a specific kernel object. Invocations can then be made on the object, depending on the type of the object and the access rights encapsulated in the capabililty.

The leveraged TCB provides security enforcement proofs as it employs a protection model

inspired by the classical take-grant model such that when properly configured, the system "guarantees the classical security properties of confidentiality, integrity and availability" [65, 79]. The employed security protection model is essentially the take-grant protection model with the following modifications as detailed in [44]:

(i) *Create*: Creating a new object occurs by retyping an untyped object that was created in memory at boot time. The create rule only applies if an object has an outgoing edge representing the create authority.

(ii) *Remove*: Capabilities are immutable so the remove operation will remove an object's entire edge as opposed to a portion of the object's authority. Therefore, in order to take away a portion of authority, the TCB's security model will remove the more privileged edge and then create a less privileged edge.

(iii) *Revoke*: The revoke rule is added in the security protection model and is a combination of removal operations. Revoke allows the kernel to remove a set of capabilities from an object.

(iv) *Take*: All authority propagations are grant operations and therefore, the TCB does not employ the take rule. In this particular case, the take operation is a dangerous operation. Given n_1, n_2, n_3, from Chapter 2.3.1: Take-Grant Model, the take rule permits the node n_1 to take the authority, α, from n_2, to operate on n_3. If this operation were permitted, n_1 may acquire authority to operate on n_3 without explicitly being granted that authority by n_2. This would break the security proofs of the TCB. Therefore, the take operation is omitted and only the grant operation can be used to propagate authority.

5.6 Access Control Model Decidability

Elkaduwe et al. show the seL4 access control model, described in Chapter 5.5: vCDS Security Model, to be decidable through proofs presented in [44]. Furthermore, the access control model has been formalized and the security analysis of the TCB has been machine-checked. Presented below are the theorems and necessary lemmas which lend themselves to reasoning about the decidability of the model. It should also be understood that the following descriptions have been aggregated from the original work in [44].

5.6.1 Methodology

The goal of the proofs presented by Elkaduwe et al. [44] is to formalize the seL4 access control model and show that the security model prevents *authority leaks*. For example, in a state, s, any ruling that node, n_x, has over another node, n_i, cannot be exposed in such a way that n_x would pass any authority to n_i in s or any future state, s'. In terms of the capability model, preventing an authority leak means that a subsystem cannot give any capability references to physical memory or communication channels to any other subsystem [44].

The definition of a *sane* state as it pertains to the introduction of new vertices, per Elkaduwe et al.: a sane state is a formulation of kernel objects, i.e. a graph with vertices, in which the following three properties hold:

(i) The new vertex being examined must exist in the graph,

(ii) there exist no dangling capability references and

(iii) no newly created vertices overlap with any existing object's memory region [44].

Reviewed below are the theorems presented in the paper which seek to answer the following question: *Is it possible to prevent some node from leaking a capability to some other node in any future state of the graph?*

Theorem 1 ([44]) *In any sane state, if two existing entities are not connected, they will never be able to leak authority to each other.*

Theorem 1 is described as the contrapositive of the following lemma, presented in the paper: in any sane state s, if one existing node, n_x, can spill a subset of the possible access rights, $\alpha \subseteq R$, to any other existing node, n_i, in some future state, s', then the nodes n_x and n_i must be connected in the current state. Therefore, Theorem 1 states that, in a sane state, two entities which are not connected will never be able to leak authority to one another [44]. This theorem does prove the "standard take-grant non-leakage property for authority distribution in seL4", however, Theorem 1 alone does not completely satisfy the question of whether or not spillage can occur. The theorem does not account for entities that do not exist in the current state being analyzed, but may, however, be created in some future state.

Theorem 2 (Isolation of authority [44]) *Given a sane state s, a non-empty subsystem n_s in s, and a capability c with a target identity n in s, if the authority of the subsystem does not exceed c in s, then it will not exceed c in any future state of the system.*

To correct the limitations of Theorem 1, Theorem 2 is introduced. Additionally referred to as an *Isolation Theorem*, it proves the non-leakage property when *authority confinement* can be used to implement *isolated subsystems* which can create additional entities in future states. Authority confinement is a term presented in the paper to describe

(i) the "strong isolation guarantees between components" such that any unexpected behavior by a particular subsystem is restricted to that subsystem and

(ii) the "isolation of authority" such that any particular authority cannot be increased in a subsystem [44].

An isolated subsystem is a graph of connected vertices with arcs, i.e. access control labels, such that any particular vertex in one subsystem, may not acquire a capability with a particular authority over a vertex belonging to another subsystem, without the prior existence of that authority in that other subsystem. In other words, arcs within a subsystem's graph cannot be exfiltrated to another subsystem and those arcs within another subsystem cannot infiltrate the graph. Additionally, Elkaduwe et al. show that an already existing authority cannot be increased in the subsystem. More formally, Theorem 2 states that, given a non-empty subsystem, i.e. a subsystem where objects and authorities exist, in a sane state, s, and a capability, c, with some authority over a node, n, in the subsystem, if the authority of s does not exceed c, then the authority will not exceed c in any future state of the system, s' [44]. Therefore, it can be concluded that Theorem 2 proves the entirety of the non-leakage property such that "subsystems can neither exceed their authority over physical memory nor their authority over communication channels to other subsystems" [44].

5.7 CAmkES Security Enforcement

CAmkES abstracts low-level kernel mechanisms. The CAmkES ADL describes the components, interfaces, and connectors which make up a system. Components refer to the data, code, and programs encapsulated by the microkernel, which, in the case of vCDS, represent the trust domains;

interfaces define component invocation; and connectors are one-to-one links between the interfaces. The CAmkES compiler translates the ADL into the capability distribution language (capDL): "a low-level specification of the system's initial configuration of kernel objects and capabilities" [80].

CAmkES provides multiple types of connectors, however, vCDS primarily utilizes Dataports which represent shared memory regions. These port interfaces provide an avenue for one component to pass bulk data to another component, including across the boundaries between domains of differing trust levels [64]. One advantage of using CAmkES is that it allows for the implementation of explicit access controls on each of these Dataports. These access controls form a portion of the inputs which are processed by the audit tool, presented in Chapter 5.9: Security Control Audit Tool.

A further advantage of utilizing this method is the enforcement of a data diode. A data diode is a link through which data may only flow in one explicit direction. Data diode enforcement ensures that the priority of preventing data leakage is upheld and no leaks can occur over the link in the opposing direction.

5.8 vCDS Stream Processor Audit

Now the impact of the audit tool, described in Chapter 5.9: Security Control Audit Tool, is demonstrated by leveraging it to analyze and verify the security configuration of the vCDS stream processor application. This analysis requires the security configuration of the application as well as the system description. Recall that the vCDS application implements four components for the stream processor use-case: (i) Low Side, (ii) High Side, (iii) Guard, and (iv) High Side Management Network.

Listing 5.1 shows the definition of these components using the keywords `LowSide` and `HighSide`. These are an example of the keywords which have been tailored to trigger the appropriate protection models and information flow constraints in vCDS. For example, the `LowSide` keyword specifies a component which resides in a low classification space whereas the `HighSide` keyword specifies a component residing in a high classification space. Listing 5.1 also shows the pseudocode for a connection between components. Figure 5.2 depicts the connectivity as well as the security configuration of the vCDS system. Specifically, the direction of the arrow represents a write operation.

The component pointed to, i.e. on the receiving end of the arrow, is permitted to perform a read operation.

```
component LowSide lowDomain;
component HighSide highDomain;
component HighSide guard;
component HighSide managementNetwork;

/* Define Connections Below
  connection <connector-type> conn(
      from component,
      to component
  );
*/
```

Listing 5.1: vCDS CAmkES Component Definitions

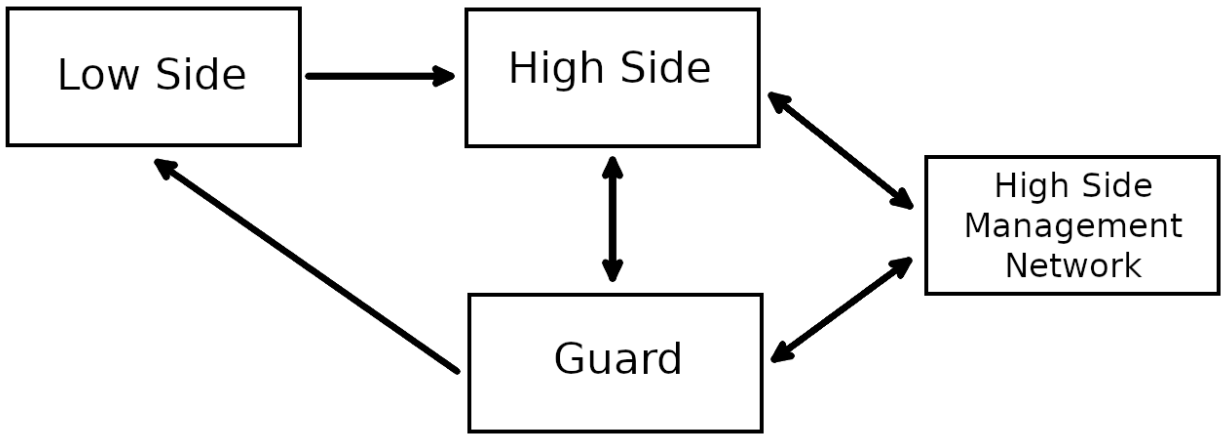

Figure 5.2: vCDS stream processor security configuration

Recall that the central focus of vCDS is data confidentiality. Therefore, the Low Side must not be permitted to read content residing at a higher label. The Low Side may only write to the High Side. The High Side must not be permitted to write to a lower label. Therefore, the High Side may only read from the Low Side. The link from the High Side to the Guard, and from the High Side to the High Side Management Network allow for both read and write privileges, bidirectionally. This is possible because all three components have the same label. Finally, the Guard may be permitted to write to the Low Side. This is made possible through the service which the Guard provides:

ensuring that no data are leaked from any of the three high-classified components back to the Low Side. The Low Side may read from the Guard only after an integrity check has been successful and permits the operation. The Low Side may not read from any other high component under any circumstances.

The audit tool verifies that the above description of the security configuration, visualized in Figure 5.2, is accurate to the goal of data confidentiality. The results of the audit, excluding the High Side Management Network rights, are shown in Figure 5.1. Furthermore, the results are sound and no additional channels or rights can be acquired due to the authority confinement provided by the formally verified TCB [33].

From Component	Access Rights	To Component
Low Side	Write	High Side
Low Side	Controlled Read	Guard
High Side	Read	Low Side
High Side	Read/Write	Guard
Guard	Read/Write	High Side
Guard	Controlled Write	Low Side

Table 5.1: vCDS Security Configuration Audit Results

5.9 Security Control Audit Tool

This section provides an introduction to the methodology and implementation of an analysis tool which seeks to verify the correctness of the security configuration of vCDS [33].

5.9.1 Application of the Isolation Theorem to capDL

As described in Chapter 5.7: CAmkES Security Enforcement, capDL defines a system's configuration of kernel objects and capabilities, which notably includes access rights. The Isolation Theorem, in [44], shows that authority, which currently exists within a system, can never increase. This theorem can be applied to subsystems described within capDL to determine if authority leak could occur. Therefore, the Isolation Theorem serves as a function through which to input a particular subsystem, from capDL. The output of the function would be the resulting authority that could be propagated to a subsystem through the use of the access rights given to each subject within the system.

5.9.2 Implementation

The implementation of the audit tool can be broken down into five sections: (i) capDL parser, (ii) graph constructor, (iii) connection generator, (iv) verifier, and (v) visualizer. The auditor pipeline is depicted in Figure 5.3 and is made up of two phases: (i) Collection and (ii) Audit.

The `main` entry point begins by retrieving the capDL, which was constructed by the vCDS build, and then sending it as input to the Collection phase. The Collection phase combines a set of functions which serve to calculate both the intended solution as defined by the specification in the capDL as well as all possible propagations from each of the connections. The Audit phase follows the Collection phase where the intended configuration solution is compared against all possible connections to determine if any additional connection or access right propagations can occur. If the vCDS security configuration passes the audit, the output is a visualization of the system components and their respective connections, otherwise, the output serves as an alert which highlights the security control implementation error.

capDL Parser

The parser module contains the `parseCapDL` function which is called to read the vCDS specification from capDL and filters out unnecessary information. It then retrieves all information which is pertinent to the algorithm such as the components, the connectors, the connector types, and the access rights for each of the connectors. The output of the parser is the system assembly which is the description of each component, their connectors and corresponding access controls.

Graph Constructor

The graph constructor module takes the system assembly which was the output from the parser as input. It matches the connector of one component to the connector of the component to which the former connects. This is done for each of the connectors belonging to a single component, until each component has been collected. The constructor then calls the `constructGraph` function to create a graph of the system based on the provided assembly. More specifically, the generated system graph is a directed graph which models the system controls for leakage prevention from higher to lower trust levels. The directed graph consists of:

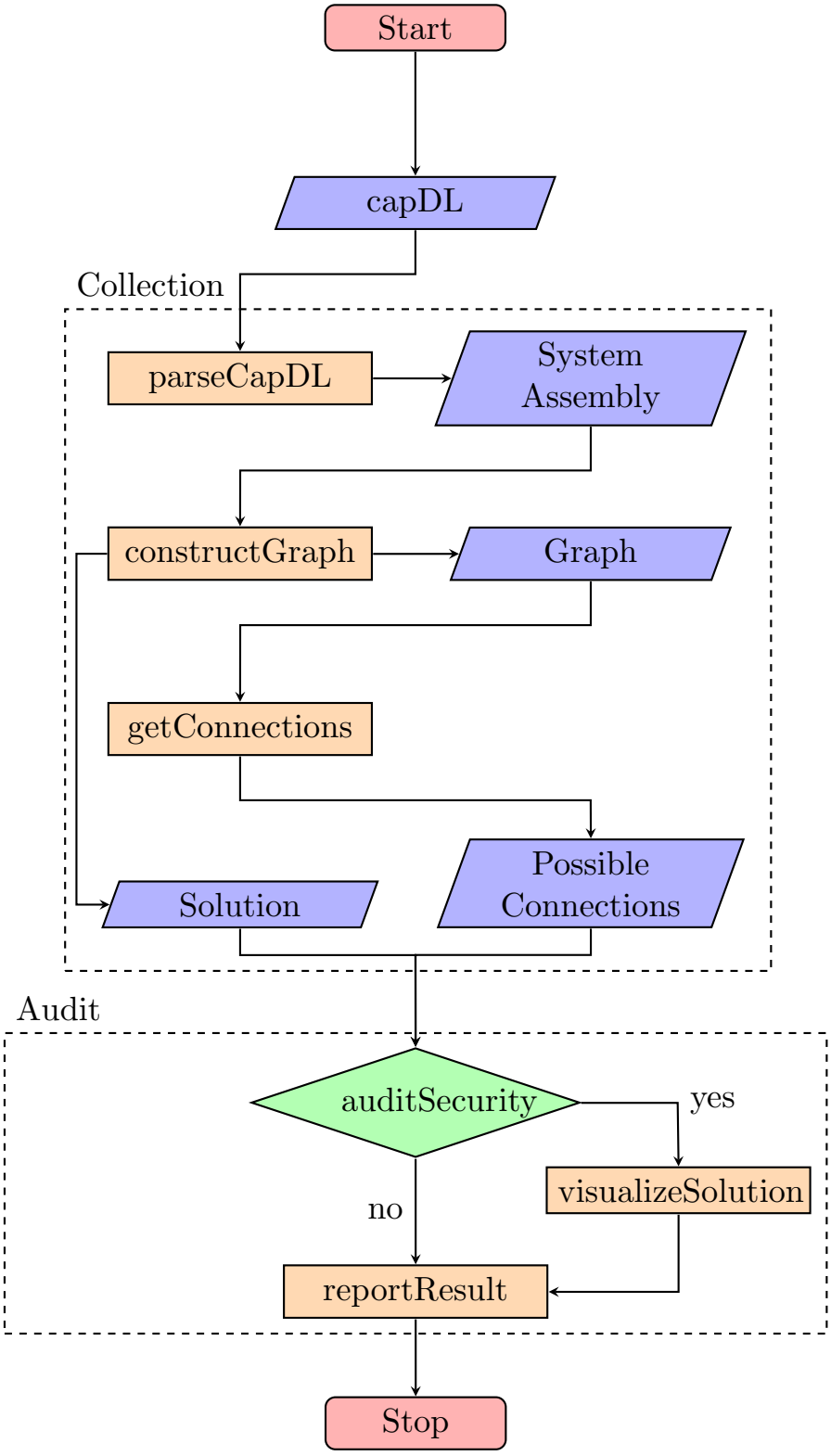

Figure 5.3: Audit Tool Pipeline

(i) a set of nodes, which correspond to the system components parsed from capDL;

(ii) a set of arcs, which represent the connections from one component to another; and

(iii) a set of labels along the arcs, where a label corresponds to the rights conveyed to the *from* component through the connection on the *to* component side.

Connection Generator

The connection generator is the core of this algorithm and is implemented such that the Isolation Theorem, given by [44], is deemed true. Additionally, for evaluation purposes (see Chapter 5.10: Analysis), implementations of the conditions given by [91] were added to emphasize the danger of the *take* rule, described in Chapter 5.5: vCDS Security Model, and to show the correctness of the verification mechanism. Recall that [91] presents the take rule which permits a particular node, n_1, to take some specified authority from another node, n_2 in order to perform an operation on a node, n_3, which n_1 did not previously have the authority to perform. This would allow an authority propagation and would break the security proofs of seL4. The use of both algorithms are further examined in Chapter 5.10: Analysis. The correct implementation of the tool relies on the Isolation Theorem, proven formally in [44].

This module calculates every possible connection and label which can be propagated to any node in the graph, i.e. every possible access right propagation that can occur for all possible connections between components. The algorithm implemented by the connection generator is given by Algorithm 1.

Additionally, Algorithm 2 is developed to enforce the Isolation Theorem from [44]. Recall that the Isolation Theorem ensures that a subject's authority over an object or communication channel in a current state cannot be exceeded in any future state, i.e. authority leakage is prevented.

Verifier

The verifier module implementation is trivial with respect to the operations on its inputs. The module takes the output of possible connections from the connection generator and the output from the Graph Constructor as input. The verifier then audits the graph constructed from the capDL against the graph of all possible connections and authorities provided by the connection

Algorithm 1: connectionGenerator

Input: digraph $G = (V, E)$, where V is the set of vertices and E is the set of edges; λ_G labeling function of E

Output: array M of dimension $|V|\,x\,|V|$

```
1 foreach p ∈ V do
2     foreach q ∈ V do
3         if p ≠ q and (p,q) in E then
4             α = λ_G( (p,q) )
              /* Isolation Theorem holds, i.e.  p can α q                    */
5             if hasAuthority(p, q, α) then
6                 M[p,q] = α
7             else
8                 M[p,q] = NIL
9 return M
```

Algorithm 2: hasAuthority

Input: p, q, α

Output: Boolean

```
1 if ∃ the authority α, from p to q then
2     return True
3 return False
```

generator. If the two graphs are congruent, i.e. each contain the same number and type of nodes and the same privileges, then authority leaks are nonexistent in the system described by the capDL. The results of the verification step are reported.

Visualizer

The visualizer module serves to provide a visual representation of the security model of the system. Essentially, the visualizer builds and outputs a diagram of the subsystems within the directed graph – a diagram of the components, their respective connections and authorities, much like what is presented in Figure 5.2, is generated.

5.10 Analysis

The analysis of this tool begins with the analysis of the take-grant security model. Recall that in [91], Lipton and Snyder show the security model to be decidable in linear time, $O(n)$. Subsequently,

recall that the seL4 security model, which is based on the take-grant security model, is presented in [44]. Based on the theorems and lemmas explicitly stated, which lead to the proof of the Isolation Theorem, Elkaduwe et al. formally prove that this theorem ensures that no authority propagations may occur in an seL4-based system. Moreover, Elkaduwe et al. show that the seL4 security model, i.e. object security, is decidable in linear time, $O(n)$. This audit tool is based on the theorems and lemmas proven in [44, 91] and can therefore be *trusted* to be correct. Further examination to prove this tool *trustworthy* is discussed in Chapter 5.10.4: Future Uses and Modifications.

Figure 5.2 shows the worst case execution time (WCET) for each component of the tool. The overall execution time of the tool is $O(n^2)$. This execution time is realized due to the $O(n^2)$ from the capDL parser, the graph constructor, the connection generator, and the verifier.

Component	**WCET**
capDL Parser	$O(n^2)$
Graph Constructor	$O(n^2)$
Connection Generator	$O(n^2)$
Verifier	$O(n^2)$
Visualizer	$O(n)$

Table 5.2: WCET of Audit Tool Components

5.10.1 Versatility

One of the further nice properties of this tool is its versatility to audit systems other than vCDS [33]. With little to no modifications, this tool functions with any system which leverages the CAmkES framework on the seL4 microkernel. Therefore, it can be used to analyze and audit the security configuration of vCDS systems and any other systems which utilize CAmkES on seL4.

5.10.2 Theorem Examination

When examining condition 2, expressed by Lipton and Snyder in [91], in the context of a vCDS audit, the take operation permits rule propagations which would otherwise be prohibited. For example, when auditing the specification of the stream processor application, some additional rules, relative to Figure 5.1, are permitted, as shown in Figure 5.3. Upon first review, these results reflect a significant error in the implementation of the security controls because the Low Side can read and write to both the Guard and the High Side. However, when further examined, one might

From Component	Access Rights	To Component
Guard	Read	Low Side
Low Side	Write	Guard
Low Side	Read/Write	High Side

Table 5.3: vCDS Audit Results Permitting Take Rule

conclude that the data diode link between the Low Side and the High Side has failed altogether. One can be certain, however, that this is not the case because the data diode comes with formal proofs of correctness and the take rule, as previously discussed, is not implemented in seL4 for the reasons above.

5.10.3 Limitations

The limitations of this tool are reflections of the limitations of the capDL which is generated by the CAmkES compiler from the ADL. Specifically, certain endpoint connectors do not translate to the capDL. This, however, may not provide the functionality that is desired by other CAmkES/seL4 applications. A second limitation is that, while the Isolation Theorem has been proven trustworthy, this software is not formally verified; this is further addressed in Chapter 5.10.4: Future Uses and Modifications.

5.10.4 Future Uses and Modifications

First and foremost, for this audit tool to be trustworthy, it must be comprehensively verified for functional correctness with respect to its specification. Proving this software correct is the next step in providing the assurance that all security configurations in vCDS and like systems are correct. Secondly, as new products evolve from vCDS, the building blocks of seL4 and CAmkES , and any CDS described via an ADL, it will be useful to analyze and verify the respective configurations with this tool to ensure proper security enforcement and improve the future development of formally verified security systems.

5.11 Final Audit Tool Analysis

In the previous sections, the need for verifying the implementation of a CDS has been addressed. The contributions of Elkaduwe et al. to the problem of decidable object security and how their

conclusions provide an important function in this work have been reviewed. Additionally, an algorithm has been presented as well as an implementation in the form of a security audit tool which can be leveraged to analyze and audit the security configurations of the above listed systems. Once again, this is, as evidenced by a thorough search of the relevant literature, the only algorithm which seeks to verify the correctness of a CDS. This tool is also the first tool which audits a CDS described by an ADL. Additionally presented is an ADL which has been tailored for describing a CDS system with the ability to tag components in such a way as to check the system constraints and trigger the appropriate protection models and information flow constraints in vCDS to determine whether or not to encrypt the data. Finally, this tool has been extended to generate a system security model to improve the status quo in system security modelling. The hope is that this work inspires the further development of provably secure and trustworthy security computing systems with verified security controls.

Chapter 6

Analysis and Evaluation

This chapter begins with an analysis of the proof assumptions of seL4 as they pertain to vCDS and ends with the performances of different vCDS instantiations are evaluated.

6.1 seL4 Proofs

The seL4 proofs show the formally verified properties of the microkernel. Examined in this section are the properties which are enforced by the proofs, the implications of these properties, and the assumptions which must be made to complete the formal verification process, as described in [32].

6.1.1 Enforced Properties

Discussed below are the properties enforced by the proofs relating to the comprehensive formal verification of seL4. The proofs cover the following properties: (i) functional correctness, (ii) confidentiality, (iii) integrity, and (iv) availability. More specifically, the proofs, as described in [79], include the following:

(i) *Functional correctness [78]*: Recall that a *refinement* proof takes the properties of the abstract model and establishes their correspondence with the properties of the refined model representation of the system. Therefore, refinement ensures that any security properties which are proven in the abstract model will also hold in the source code [78, 79]. This particular proof is a proof of functional correctness between the kernel's C code and its specification. However,

when combined with the following proofs, these proofs "constitute a functional correctness property in the strongest sense" [79].

(ii) *Functional correctness for a "high-performance, hand-optimised inter-process communication (IPC) fastpath" [79]*: Recall that the IPC fastpath is a "hand-tuned implementation of the IPC paths through the seL4 kernel" which are used to pass messages between components. This proof shows that when fastpath is enabled, the kernel implementation of the abstract specification is correct.

(iii) *Correct access-control enforcement* [115]: Specifically, this proof ensures correct enforcement of integrity and authority confinement. As stated in the paper, "integrity provides an upper bound on write operations ... authority confinement provides an upper bound on how authority may change" [115].

(iv) *Information-flow noninterference [98]*: This proof shows that seL4 can be configured as a separation kernel to provide strong isolation through "isolated partitions with controlled communications channels" [98]. Furthermore, seL4 is proven to enforce the properties of confidentiality and integrity on storage and communication channels.

(v) *User-level system initialization* [21]: This proof connects to the access-control model of the two security proofs above. Generally applied to all capability-based systems, an initializer receives as input a formal description of the desired initialization state. The initializer then creates the components, communication channels and access rights based on the description. This proof ensures that the state, output from the initialzer, is the correct, desired state.

(vi) *Binary functional correctness* [116]: Binary correctness is shown through the "refinement between the semantics of the kernel binary after compilation/linking and the C source code semantics used in the functional correctness proof" [79]. In other words, this is a refinement theorem between the source code and the binary. This proof shows that the binary is correct and that the compiler and linker do not need to be proven trustworthy (or even trusted), thus eliminating the necessity for a comprehensively formally verified toolchain.

(vii) *Static analysis of the seL4 binary to provide a sound worst-case execution time (WCET) profile of all system calls [20]*: Although this is not a proof, but rather an analysis, this

analysis makes seL4 an ideal choice for safety-critical and time-critical systems as it shows that seL4 is the world's fastest performing microkernel.

6.1.2 Implications

As stated above, the proofs directly prove the properties of (i) functional correctness, (ii) confidentiality, (iii) integrity, and (iv) availability. As per [32], these proofs carry with them implications of "the absence of whole classes of common programming errors". A subset of these errors, listed in [32], are as follows:

(i) *Buffer overflows*: Buffer overflows, sometimes referred to as overruns, occur when the amount of data exceeds a memory buffer's storage capacity. This can be used to crash the software or inject malicious code to change the execution path of the program, leading to data leakage. These proofs show that buffer overflows are unsuccessful on seL4.

(ii) *Pointer errors*: There are many common pointer errors such as uninitialized pointers, dangling pointers, null pointer dereferences and more. Uninitialized pointers occur when attempting to reference a pointer that does not yet point to a valid address. Dangling pointers occur when attempting to reference a pointer whose memory has already been freed. A null pointer dereference occurs when a pointer, which contains the value of `NULL`, is manipulated as if it pointed to a valid memory address. Additionally, it is possible to assign a pointer to the wrong datatype such as passing a `char*` to a function which expects a `double*`, for example. These problems are commonly exposed in the C programming language and often lead to system crashes, undefined behavior, lost data and/or memory leaks. These pointer related errors cannot occur in seL4.

(iii) *Memory leaks*: Memory leaks commonly occur when memory, which is no longer needed, is not released or freed. The opposite is true as well – memory can be freed while it remains in use. These proofs show that memory leaks cannot occur in seL4.

(iv) *Arithmetic overflows and exceptions*: Given that machines have a finite amount of memory storage, there are many problems which can occur if the use of numbers greater than what can be represented by 32 or 64 bits (or more depending on the architecture) is attempted.

Additionally, if an attempt to execute an undefined operation occurred, such as dividing some number by zero, an exception would be thrown. These exceptions would be cause for a system crash. Arithmetic overflows and exceptions cannot occur in seL4.

(v) *Undefined behaviour*: Undefined behavior occurs due to the lack of restrictions on the behavior of a system. Many of the previous examples of errors are examples of undefined behavior. The proofs show system restrictions such that no undefined behavior can occur in seL4.

6.1.3 Assumptions

Each of the proofs and implications of the proofs listed above are founded on a few basic assumptions. Essentially, seL4 comes with three assumptions as stated in [65]:

(i) the hardware behaves as expected;

(ii) the specification is correct; and

(iii) the theorem prover is correct.

The hardware assumptions include faults which may remain in certain low-level kernel components such as the TLB, cache handling, handwritten assembly, and the boot code [32]. If the hardware contains bugs, then the verification of seL4 has no benefit. With regard to a correct specification, "there is always a gap between the world of mathematics and the physical world ... the advantage of formal reasoning is that you know exactly what this gap is" [65]. In order to mitigate this, properties about the specification, such as security, have been proven through "comprehensive formal verification, including a functional correctness proof of the implementation and a complete proof chain of high-level security properties down to the executable binary" [67, 79]. The assumption that the theorem prover must be correct is the least concerning because the core of the theorem prover (Isabelle/HOL) used to prove seL4 has "checked many proofs small and large from a wide field of formal reasoning, so the chance of it containing a correctness-critical bug is extremely small" [65]. In addition to assuming the theorem prover is correct, one must assume that the axioms of high-order logic (HOL) are also correct. If the axioms are logically inconsistent, there is a much more significant problem with mathematics. Each of these assumptions, while limitations of the system, are not limitations of the formal verification process and can be eliminated [32].

6.2 vCDS Performance Throughput

This section serves to demonstrate the performance of vCDS through carefully selected benchmarks which measure the throughput transfer of data from the Low Side domain to the High Side domain. Additionally, results below show four stages of experiments.

6.2.1 Overhead Formula

The overhead is calculated for some of the experiments used to calculate throughput, given the throughput measurements to compare the performance results of vCDS to the Native Linux Process. The following formula is used to calculate overhead where the throughput performance of vCDS is P_{vCDS} and that of the Native Linux Process is, likewise, P_{Linux}:

$$\left| \frac{P_{Linux} - P_{vCDS}}{P_{Linux}} \right| \cdot 100$$

6.2.2 CPU Information

All throughput measurements were run on an Intel® Xeon® Processor E5-2690V4 with 112 GB of DDR4 memory (76.8 GB/s bandwidth) and a 35 MB cache. Further CPU specifications may be found in [29].

6.2.3 Results

Shown below is the resulting throughput of vCDS, and the system overhead compared to a native Linux process used to transport data through shared memory. Additionally, the vCDS performance is compared to that of the state-of-the-art in commercial hardware data diodes.

vCDS vs Native Linux Process

The first result, shown in Table 6.1, is in the context of a vCDS solution with three layers of virtualization: VM development machine, QEMU virtualizer, and seL4 VMM. The table displays the vCDS pipeline throughput, that is, the number of bytes which can be processed in one second of operation, compared to the throughput of a native Linux process which uses shared memory as a transfer channel. For comparison purposes, the overhead of the development system is shown.

Throughput (Bytes/Second)		
vCDS	Native Linux Process	vCDS Overhead
33,193,800	447,585,750	92.6%

Table 6.1: vCDS vs Native Linux Process Throughput (3 Layers of Virtualization)

Table 6.2 depicts the vCDS pipeline throughput in the context of a vCDS solution running directly on hardware with one layer of virtualization: seL4 VMM. There are two different throughput measurements, a partial transfer where the data are available to the second domain without any processing taking place, and a full transfer where the second domain does process the data. Specifically, the data are processed by Snort on the High Side as well as the filter running on the Guard. Additionally for comparison, the vCDS overhead is calculated.

Throughput (Bytes/Second)			
Pipeline Transfer	vCDS	Native Linux Process	vCDS Overhead
Partial	326,705,400	403,745,175	19.1%
Full	150,696,450	214,413,750	29.7%

Table 6.2: vCDS vs Native Linux Process Throughput (1 Layer of Virtualization)

vCDS Pipeline vs Commercial Hardware Data Diodes

Table 6.3 below shows, once again, the partial pipeline throughput of vCDS. However, the vCDS performance throughput is measured against the hardware data diode products with the highest claimed throughput specifications provided by the following commercial data diode vendors: OWL Cyber Defense, Belden, BAE Systems (evaluated to EAL7+), Vado Security, Fox-It (EAL7+), Arbit Cyber Defence Systems (EAL5+) and Waterfall Security Solutions [40, 69, 71, 74, 120, 123, 127]. OPSWAT [71] provides the specifications for several of the solutions listed below in the table. Additional citations are also provided for specific solutions. As the results state, [71, 74, 123] claim the highest throughput performance at 10 Gbps (1,250,000,000 Bps) for enterprise solutions while the majority of general solutions boast 1 Gbps (125,000,000 Bps). vCDS results in a throughput transfer of 2.6+ Gbps (326,250,000+ Bps). It can be noted that most commercial solutions provided by each of the listed vendors generally only provide one solution with high throughput. This is because the high throughput solutions are geared toward enterprise environments where security

is often compromised as shown in Chapter 2: Background. Many of these vendors carry security-focused, tactical solutions which range in performance from 5 Mbps (625,000 Bps) to 100 Mbps (12,500,000 Bps) [71]. These security focused solutions often implement additional filtering and sanitization techniques which may decrease the throughput. While the full transfer throughput of vCDS yields superior performance compared to the throughput of tactical solutions, it is a challenge to compare performance because the filtering mechanisms used within other solutions are unknown and therefore could significantly decrease throughput. Overall, vCDS yields a throughput that is 26 times greater than the security-focused solutions and 2.6 times greater than the throughput of the median solutions.

Throughput (Bytes/Second)			
Tactical Solutions: [69, 71]	General Solutions: [39, 71, 120, 127]	vCDS	Enterprise Solutions: [71, 74, 123]
12,500,000	125,000,000	326,705,400	1,250,000,000

Table 6.3: vCDS Pipeline vs Commercial Data Diode Throughput

vCDS Data Diode Only Solution

The throughput of the bare-bones, software data diode vCDS solution is presented in Table 6.4 below. Note that Infodas [73] claims to have the fastest software data diode in the world with a transfer rate up to 9.1 Gb/s. The provided CPU specifications of the Infodas solution are as follows: Intel® Core™ i7-5700EQ Quad Core CPU, 32GB DDR3L memory (25.6 GB/s bandwidth), and a 6MB cache. More specifications can be found in [73]. The software based data diode solution has a transfer rate of up to 12.2 Gb/s on the hardware detailed in Chapter 6.2.2: CPU Information. Though inconclusive, it is possible that the Infodas solution may outperform the vCDS data diode if tested on the same hardware. For this to happen, the source code from the Infodas solution would be required. Additionally, it is notable, from Table 6.5, that taking the vCDS software data diode throughput compared to the native Linux process shared memory throughput where read and write only access was specified yields an overhead of -241.2%.

Throughput (Bytes/Second)			
Tactical Solutions: [69, 71]	General Solutions: [39, 71, 120, 127]	Enterprise Solutions: [71, 74, 123]	vCDS Data Diode
12,500,000	125,000,000	1,250,000,000	1,527,280,640

Table 6.4: vCDS Data Diode vs Commercial Data Diode Throughput

Throughput (Bytes/Second)		
vCDS	Native Linux Process	vCDS Overhead
1,527,280,640	447,585,750	-241.2%

Table 6.5: vCDS Data Diode vs Native Linux Process Throughput (0 Layers of Virtualization)

Chapter 7

Conclusion

7.1 Contributions

This work has systematically examined and presented a solution which aims to correct the challenges of CDS systems, in particular, lack of: (i) trustworthiness, (ii) remote deployablility, and (iii) accessibility. A novel CDS architecture, presented in [33], called *virtual CDS (vCDS)* has been analyzed. The resulting vCDS systems have been shown to overcome the weaknesses of current CDSs with the following contributions:

(i) The architecture allows for public analysis in terms of the trustworthiness of its instantiations – vCDS ensures trustworthiness through execution on top of a formally verified, TCB.

(ii) The resulting vCDS systems are widely accessible, including both DoD/IC and commercial sectors – accessibility to commercial sectors provided by using commodity software and hardware.

(iii) The resulting vCDS systems allow for secure remote deployability, including remote deployment in cloud environments.

(iv) The vCDS architecture can be leveraged or adapted to implement critical security mechanisms, such as Intrusion Detection System/Intrusion Prevention System (IDS/IPS) and Firewall technologies, which use tools or Indicators of Compromise (IOC) with high classification levels, such as cryptographic signatures and analytics tools, to defend computer systems with

low classification levels. This is achieved by allowing these IPS/IDS of high classification levels to vet traffic in networks of low classification levels, without leaking information about the technologies with high classification levels.

(v) vCDS can leverage additional, third-party, filtering and diode mechanisms to include both hardware and software based mechanism.

(vi) The vCDS architecture is compatible with big data and high performance computing environments where distributed parallel processing, including the access and transfer, of large data sets is commonplace. In the case of a big data platform, vCDS is compatible to platforms like MapReduce in that it moves the computation to the data. In the case of a high performance computing platform, vCDS can move the data to the compute nodes. The versatility of vCDS provides the scalability and low-cost implementations.

(vii) The vCDS architecture can integrate with other security solutions, such as the B2CSM architecture described in [61], which aims at achieving secure intelligence collaboration through data collection, aggregation, analysis, and threat-related information dissemination to critical parties.

(viii) A prototype stream processor instantiation of the vCDS architecture was designed and analyzed.

(ix) An auditing tool which seeks to validate the correctness of the implemented security configuration of all vCDS instantiations was developed. This is the first tool to analyze and audit CDS security control implementations. Presented along with this tool were the following contributions:

  (a) The need to verify the implementation of a CDS system security model is addressed.

  (b) This tool is the first and only development of a tool which audits a CDS described via an architecture description language (ADL).

  (c) This work is the first to tailor an ADL for describing a CDS system with the ability to tag components with proper labels which propagate down through the ADL, allowing the algorithm to check the constraints.

(d) The use of the tailored ADL is extended to include labels and key words which trigger the appropriate protection models and information flow constraints in vCDS to determine whether or not to encrypt the data.

(e) The tailored ADL is extended to provide system security modeling that is far beyond the status quo.

As evidenced by a thorough search of the relevant literature, vCDS is the first general purpose CDS system, which leverages a TCB that is provably secure and has been comprehensively verified for functional correctness and security guarantees, that can be applied to and deployed in a variety of use-cases and environments.

7.2 Discussion

This section provides the details of the limitations of this work and the mitigations which have been put in place to combat these limitations. Additionally discussed are the future intended efforts which will seek to improve upon this work. Finally, presented below are some recommendations for future work.

7.2.1 Limitations

The discussion below examines each of the limitations and potential problems with the current design and implementation with hopes to garner interest and inspire further research of this technology. These limitations of vCDS are inherited from its building blocks.

(i) While seL4 proofs show functional correctness, the proofs, at present, do not capture time properties. This presents an issue when examining the potential of an attacker opening a covert timing channel to expose data or information about the data which could lead to spillage.

(ii) CAmkES, on seL4, is guaranteed to provide proofs of security enforcement [79] when properly configured. However, unless properly validated, there is no way to know that the security enforcement is correct.

(iii) The use of AMD SEV/SME exposes the vCDS system to some potential threats that are not considered in the threat model [7, 43, 89, 96].

7.2.2 Mitigations

The following mitigations to the above limitations are incorporated and/or proposed and listed numerically to correspond with the limitations above:

(i) vCDS utilizes a padding mechanism in order to pad the execution time to an upper bound as described in [68]. The proof for this in seL4 is in the works [20, 68, 92] though it comes with several difficulties [27, 55]. [66] explains that two important cache practices are necessary for timing channel protection: spatial partitioning and cache flushing. By themselves, spatial partitioning is limited by the fact that the core caches (L1, TLB, branch predictor, pre-fetchers) are virtually indexed or not controlled by the OS so they cannot be spatially partitioned, and, while cache flushing covers the problems with spatial partitioning, it is useless for concurrent access. Therefore, both are necessary to prove timing protection. Additionally, [130] shows that timing channels can be successfully eliminated in a RISC-V processor. The hope is to incorporate this in the future (Chapter 7.2.4: Future Work).

(ii) This limitation is mitigated with the development of the auditing tool, described in Chapter 5: Auditing a Cross Domain Solution,

(iii) Finally, one mitigation could be the use of RISC-V Keystone enclave to instantiate the TEE. This is also further discussed in Chapter 7.2.4: Future Work.

7.2.3 NCDSMO: Concerns of vCDS

Discussed here are some concerns that the NCDSMO has expressed about vCDS going forward. Most of these concerns closely relate to CDS filtering mechanisms and the Raise The Bar (RTB) strategy [99]. Additionally presented are the ways in which vCDS is capable of meeting, or has met, these concerns.

Concerns

(i) Real-world protocol adapters and filters are written with enormous amounts of code and cannot, realistically, be proven correct. The attacker would then need only to defeat the protocols and filters to fail the system open, while the kernel itself would have no knowledge of the failure.

(ii) Filters must be able to enforce data safety, i.e. safety from data attacks, data hiding, and data disclosure [30]. In other words, the data must be semantically checked for irregularities such as malicious code or hidden data as well as sensitive data that could be leaked. The NCDSMO would like to see vCDS implement additional filters to perform such safety checks.

(iii) Filters must be compliant with the RAIN concept [51].

Rebuttal

(i) Without any channels from the Guard to the Low Side, the advantage of using the formally verified (for security, flow restriction, channel restriction, and functional correctness – see Chapter 3.6: Security Analysis) TCB is that it does not have to rely on any filtering mechanisms to ensure that data is not leaked from high to low. No additional channels can be created, the data cannot flow directly from high to low, and any implemented filters will always be invoked and cannot be bypassed. Knowing these facts and that any filters implemented on top of vCDS are susceptible to their own vulnerabilities, if there is a channel from the Guard to the Low Side, then data confidentiality could potentially be compromised. However, if the integrity guard is formally verified (see Chapter 7.2.4: Guard Formal Verification), then it can be assured that data confidentiality will not be compromised from high to low which is the primary concern of the vCDS architecture (Chapter 3.2: Threat Model).

(ii) In the case of the stream processor implementation, when the High Side moves data to the Low Side, data disclosure is prevented (Chapter 4.4.2: Security Analysis). Note that the described integrity guard is one implementation and recall from Chapter 3.6: Security Analysis that the movement of data from high to low classification is optional. In the case where data moves from low to high via the data diode (Chapter 3.6.5: Data Flow Restriction), data leakage

and disclosure are prevented. However, the architecture security model does not directly address the other data safety concerns: safety from data attacks and data hiding. The reason for this is two fold: (i) vCDS was designed as a bare-bones, baseline solution with a threat model (Chapter 3.2: Threat Model) focused primarily on ensuring data confidentiality from high to low; and (ii) the vCDS architecture assumes that data safety filtering mechanisms will be implemented and added according to the use-case requirements. Recall that any existing hardware or software based filters may be added to vCDS, simply by declaring a new component and associated connectors as shown in Chapter 4: The vCDS Implementation. As for the instantiation of vCDS for additional use-cases and corresponding filters, this is discussed in Chapter 7.2.4: Towards Extension to Operational Applications.

(iii) Discussed in Chapter 4.4.2: Security Analysis, the filter leveraged in the described stream processor implementation is nearly compliant with RAIN. Additional implementations are, and future implementations will be, compliant with the RAIN concept.

Supporting Evidence that vCDS Corrects Problems Now

With the concerns of the NCDSMO, came evidence that vCDS can be impactful now.

First, it can be confirmed that security through obscurity is a technique that is relied upon for CDS and other security technologies. The advantage of vCDS is that it does not need to rely on obscurity because, to a large extent, it is provably correct and provably protected. To the extent that vCDS is not provably correct or protected, the vulnerabilities are well understood. Formal verification of those portions is intended for future work (Chapter 7.2.4: Future Work).

Second, and as a compliment to the first point, the versatility of vCDS should not be underestimated but rather used as an advantage. The vCDS architecture was designed such that, because its vulnerabilities are well understood, any necessary mechanisms can be quickly added to an instantiation. One of the issues with status quo CDS systems (Chapter 1.3: Problem Statement) is that they are highly specialized systems where different protocol adapters and filters are required for each use-case such that CDSs cannot be applied to other use-case environments without extensive modification [13, 47]. vCDS is different as the architecture serves as a "build your own CDS" architecture upon which many use-cases may be built.

Third, current solutions, while "independently verified", are verified in closed labs operating with biases in favor of the product. Furthermore, those labs are governed by, and therefore produce results only as good as, the requirements [11]. The TCB upon which vCDS is built is completely open to anyone for independent verification meaning that it does not just meet lab requirements, it meets mathematical and logical requirements [32].

7.2.4 Future Work

In addition to addressing the limitations mentioned above, there are several research directions for future studies. A subset of these studies and their importance are discussed below. Note that portions of the following sections have been adopted from "vCDS: A Virtualized Cross Domain Solution Architecture", © 2021 IEEE [33].

RISC-V

In order to further enhance the security of vCDS systems while reducing their cost and addressing scalability limitations associated with vendor hardware, it is interesting to instantiate the abstract TEE used in the vCDS architecture via RISC-V based Keystone Enclave. This is possible because Keystone [86] claims to solve many of the limitations surrounding AMD's SEV/SME and Intel's SGX schemes, and it is available for independent, hardware formal verification [33]. Furthermore, timing channels can be eliminated [130].

Guard Formal Verification

Recall from Chapter 4.4.2: Security Analysis that the Guard is not formally verified. The Guard code base is quite small so formal verification is a necessary future goal. This would ensure that the integrity check could not be compromised and that data leakage prevention is assured.

Another possibility is to adopt the use of a formally verified implementation of a cryptographic algorithm [12, 128] to replace the use of Blake3. This would require the verification of a second implementation of the same algorithm in order to fulfill the need for independent implementations [51, 59]. Additionally, any effect on vCDS performance is unknown at this time.

Towards Extension to Operational Applications

Third, it is useful to address additional environment/use-case, security, and scalability needs of current systems. The intention is to apply the vCDS architecture to such use-cases as were described in Chapter 3.5: Applications. The instantiation of the vCDS architecture for additional use-cases or the modification of one of the described use-cases would likely involve implementing or leveraging additional filtering mechanisms. This will be a significant addition in directing the use of the vCDS system in different operational environments.

Load Balancing Strategy

Fourth, an aim is to evaluate the implementation of different load balancing strategies concerning the movement of data between domains. Currently being explored are the following three options as they related to CAmkES Dataports:

(i) Multi-Dataport, single buffer: multiple threads are leveraged with a single Dataport per thread and each Dataport passes a single buffer

(ii) Single Dataport, multi-buffer: a single Dataport is designed to contain multiple buffers where each buffer corresponds to a single thread

(iii) Multi-Dataport, multi-buffer: this strategy combines the above strategies using multiple Dataports and buffers, separating each into groups by buffer content attributes such as origin, priority and size

In addition to these strategies, there is an aim to adopt a circular buffer strategy to provide a more dynamic functionality, implemented in either the buffer or Dataport levels. The hope is that the results of each evaluation would reveal the most advantageous ways to improve performance relative to the use-case that is implemented.

Remote Boot of High Domain

Fifth, as a strategy to prevent persistence of data after processing has concluded within the High Side enclave, there is an aim to implement a remote management capability of the high domain

from the C2 (high side management network). The management capability will include the lifecycle from remote creation of the data to remote shutdown of the high domain.

Collaborative Efforts

Finally, interest in a CDS built upon a formally verified TCB was explicitly expressed at the seL4 Summit 2020 by the U.S. Army [45]. Numerous commercial companies such as the Cohere Technology Group and L3Harris Technologies have expressed interest in collaborations and potentially using a future product based on vCDS. Additionally, the NSA has reached out with inquiries about vCDS and about potential future collaborations. In the future, the hope is to pursue collaboration with the U.S. Army, the NSA, DARPA, and other entities in the commercial sector.

7.3 Recommendations

This work may serve as an important step in the right direction as it:

(i) yields a solution which is closer to, and, in certain use-case environments, serves as, a comprehensively formally verified CDS system;

(ii) improves upon the status quo in CDS security;

(iii) makes secure CDS technology more accessible to all sectors;

(iv) promotes remote deployability; and

(v) promotes system versatility such that it can be applied to many different use-cases and environments.

With the impacts listed above in mind, the following recommendations for future improvements of the vCDS system are presented:

(i) formally verify the integrity guard implementation;

(ii) instantiate the TEE using the RISC-V Keystone enclave in addition to the timing protections presented in [68];

(iii) formally verify the coupling of a RISK-V based TEE with the TCB of vCDS;

(iv) the auditing tool should be formally verified in order to ensure a trustworthy implementation of vCDS security configurations; and

(v) a system should be developed for quick evaluation of load balancing strategies for each use-case implementation of vCDS.

Additionally presented are the following recommendations for any future use and development of CDS systems:

(i) CDS systems must be able to be relied upon, i.e. security must be architected into both hardware and software, with all components, including filters, being formally verified for security and functional correctness, in order to build secure, trustworthy CDS systems;

(ii) future development of CDS systems should utilize the architecture of vCDS, not only for its formally verified properties, security, and versatility, but also because vCDS implementations align with the DoD's best interests (see Appendix C: Towards Trustworthy Cross-Domain Technologies;

(iii) the practice of verify once, reuse many should be adopted in order to mitigate both cost and time to market;

(iv) describing CDS systems via an ADL is essential for the verification of CDS security configurations and should therefore be a requirement; and

(v) making use of the vCDS auditing tool further improves the time to market for a CDS which fully aligns with the best interests of the DoD.

For further recommendations, see Appendix C: Towards Trustworthy Cross-Domain Technologies.

7.4 Closing Remarks

In this work, there has been a review and examination of the status quo in state-of-the-art CDS technology. Additionally the problems, limitations and concerns of current CDS systems have been discussed. To combat these problems, limitations and concerns, the *Virtualized Cross Domain*

Solution (vCDS) system, originally presented in [33], has been examined. vCDS is the first CDS system to be constructed on top of a comprehensively, formally verified trusted computing base (TCB). The main objectives of vCDS: (i) trustworthiness, (ii) multi-sector availability, (iii) remote deployability and (iv) versatility, have been re-enforced through an in-depth analysis of the implementation of the stream processor application. As previously discussed, this work "provides a new and much improved security baseline tailored to (i) defensive effectiveness, (ii) data confidentiality and (iii) operational relevance" [33].

Furthermore, a CDS auditing tool was developed which, as evidenced by a thorough search of the relevant literature, is the first to:

(i) analyze and audit CDS security control implementations such that no CDS constraints are violated and no operations will lead to a leak of rights or data;

(ii) audit a CDS described via an ADL;

(iii) tailor an ADL for describing a CDS system with the ability to tag components with proper labels which propagate down through the ADL, allowing the algorithm to check the constraints;

(iv) include labels and key words in the ADL which trigger the appropriate protection models and information flow constraints in vCDS to determine whether or not to encrypt the data; and

(v) provide system security modeling that is far beyond the status quo.

Finally, the work presented herein does not conform to status-quo of security through obscurity and ad hoc solutions. The security and limitations of vCDS are well understood and open for independent evaluation. The hope is that this work inspires and influences further improvements in CDS systems with the common goals of (i) trustworthiness, (ii) multi-sector availability, (iii) remote deployability and (iv) versatility.

Bibliography

[1] Edward N. Adams. "Optimizing Preventive Service of Software Products". In: *IBM J. Res. Dev.* 28.1 (Jan. 1984), pp. 2–14. ISSN: 0018-8646. DOI: 10.1147/rd.281.0002. URL: https://doi.org/10.1147/rd.281.0002.

[2] Mohammad Alam. *Analysis of Different Software Fault Tolerance Techniques*. May 2009.

[3] Secure Technology Alliance. "Trusted Execution and Environment (TEE) and 101: A Primer". In: *A Secure Technology Alliance Mobile Council White Paper*. Secure Technology Alliance. 191 Clarksville Road Princeton Junction, NJ 08550: Secure Technology Alliance, June 2018.

[4] Jim Alves-Foss et al. "The MILS architecture for high-assurance embedded systems". In: *International Journal of Embedded Systems* 2 (2006), pp. 239–247.

[5] AMD. *Enhance your Cloud Security with AMD EPYC™ Hardware Memory Encryption*. Tech. rep. Oct. 2018.

[6] AMD. *Extending secure encrypted virtualization with Sev-ES*. 2018. URL: https://events19.linuxfoundation.org/wp-content/uploads/2017/12/Extending-Secure-Encrypted-Virtualization-with-SEV-ES-Thomas-Lendacky-AMD.pdf.

[7] AMD. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. Tech. rep. Jan. 2020.

[8] AMD. *Software Techniques for Managing Speculation on AMD Processors*. Sept. 2020. URL: https://developer.amd.com/wp-content/resources/56305_3.01.pdf.

[9] *AMQP protocol adapter*. Aug. 2021. URL: https://owlcyberdefense.com/resource/amqp-protocol-adapter/.

[10] J.P. Anderson. *Computer security technology planning study. Volumes I and II*. Tech. rep. ESD-TR-73-51. The Mitre Corporation, Air Force Electronic Systems Division, Hanscom AFB, Badford, MA, Oct. 1972.

[11] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. 2nd ed. Wiley Publishing, 2008. ISBN: 9780470068526.

[12] Andrew W. Appel. "Verification of a Cryptographic Primitive: SHA-256". In: *ACM Trans. Program. Lang. Syst.* 37.2 (Apr. 2015). ISSN: 0164-0925. DOI: 10.1145/2701415. URL: https://doi.org/10.1145/2701415.

[13] United States Government US Army. *Joint Publication JP 3-12 Cyberspace Operations June 2018*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2018. ISBN: 1723569224. DOI: 10.5555/3285221.

[14] N. Asokan et al. "Mobile Trusted Computing". In: *Proceedings of the IEEE* 102.8 (Aug. 2014), pp. 1189–1206. DOI: 10.1109/jproc.2014.2332007.

[15] Ryan Ausanka-Crues. "Methods for Access Control: Advances and Limitations". In: Claremont, CA, 2001.

[16] Bill Beckwith et al. "High Assurance Safe and Secure Distributed Systems and Information Sharing". In: *Infotech@Aerospace*. 2005. DOI: 10.2514/6.2005-6930. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2005-6930. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2005-6930.

[17] D. E. Bell and L. J. Lapadula. *Secure Computer System: Unified Exposition and Multics Interpretation*. Tech. rep. National Technical Information Service (NTIS) U. S. Department of Commerce, Mar. 1976.

[18] D.E. Bell. "Looking back at the Bell-La Padula model". In: *21st Annual Computer Security Applications Conference (ACSAC'05)*. 2005, 15 pp.–351. DOI: 10.1109/CSAC.2005.37.

[19] Ken Biba. *Integrity Considerations for Secure Computer Systems*. Apr. 1977.

[20] B. Blackham et al. "Timing Analysis of a Protected Operating System Kernel". In: *IEEE 32nd Real-Time Systems Symposium*. 2011, pp. 339–348. DOI: 10.1109/RTSS.2011.38.

[21] Andrew Boyton et al. “Formally Verified System Initialisation”. In: *Formal Methods and Software Engineering*. Ed. by Lindsay Groves and Jing Sun. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 70–85. ISBN: 978-3-642-41202-8.

[22] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. “Insecure Until Proven Updated: Analyzing AMD SEV’s Remote Attestation”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 1087–1099. ISBN: 9781450367479. DOI: 10.1145/3319535.3354216. URL: https://doi.org/10.1145/3319535.3354216.

[23] Laurentiu Burdusel. “A Secure Communication System for classified documents over public network”. In: IEEE, 2010, pp. 485–488.

[24] Australian Cyber Security Center. “Fundamentals of Cross Domain Solutions”. English. In: *MENA Report* (June 2020). Name - Cybersecurity and Infrastructure Security Agency; Copyright - © 2019 Al Bawaba (Albawaba.com) Provided by SyndiGate Media Inc. (Syndigate.info); Last updated - 2019-12-06; SubjectsTermNotLitGenreText - United States–US. URL: https://search-proquest-com.proxy.libraries.uc.edu/docview/2322199850?accountid=2909.

[25] Raymond Cheng et al. “Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution”. In: *CoRR* abs/1804.05141 (2018). arXiv: 1804.05141. URL: http://arxiv.org/abs/1804.05141.

[26] CloudShield. “CloudShield CS-4000: Trusted Network Security Platform (TNSP)”. In: (2018).

[27] David Cock et al. “The Last Mile: An Empirical Study of Timing Channels on SeL4”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 570–581. ISBN: 9781450329576. DOI: 10.1145/2660267.2660294. URL: https://doi.org/10.1145/2660267.2660294.

[28] Ed Colbert and Barry Boehm. “Cost Estimation for Secure Software and Systems”. In: *ISPA / SCEA 2008 Joint International Conference*. Center for Systems and Software Engineering,

University of Southern California. 941 W. 37th Pl., Sal 328, Los Angeles, CA 90089-0781, Jan. 2006.

[29] Intel Corporation. *Intel® Xeon® processor E5-2690 V4 (35m cache, 2.60 GHz) - product specifications*. Feb. 2022. URL: https://www.intel.com/content/www/us/en/products/sku/91770/intel-xeon-processor-e52690-v4-35m-cache-2-60-ghz/specifications.html.

[30] Roger L Costello. *XML Risks and Mitigations*. 2013. URL: https://www.mitre.org/sites/default/files/pdf/13_2445.pdf.

[31] Common Criteria. "Common Criteria for Information Technology Security Evaluation". In: *Common Criteria*. July 2009.

[32] CSIRO's Data61. *The Proof*. 2020. URL: https://www.sel4.systems/Info/FAQ/proof.pml.

[33] Nathan Daughety et al. "vCDS: A Virtualized Cross Domain Solution Architecture". In: *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*. 2021, pp. 61–68. DOI: 10.1109/MILCOM52596.2021.9652903.

[34] Nathan Daughety et al. *Cross Domain Solution System Built on a Formally Verified Security Micro-kernel Running on Processors Enabled with Memory Encryption*. Patent Pending. 2022.

[35] Nathan Daughety et al. *Auditing a Software-Defined Cross Domain Solution Architecture*. In Preparation. n.d.

[36] Department of Defense. "Department of Defense Trusted Computer System Evaluation Criteria". In: DoD 5200.28-STD. Dec. 1985.

[37] Department of Defense. *Department of Defense Instruction: Cross Domain Security Policy*. Aug. 2017.

[38] OWL Cyber Defense. *What's the Difference Between Firewalls and Data Diodes?* May 2019. URL: https://owlcyberdefense.com/wp-content/uploads/2019/05/19-OWL-DataDiodes-Firewalls.pdf.

[39] OWL Cyber Defense. *OPDS-1000 Perimeter Defense Solution*. Mar. 2021. URL: https://owlcyberdefense.com/product/opds-1000/.

[40] OWL Cyber Defense. *OWL Cyber Defense Cross Domain Solutions*. https://owlcyberdefense.com/. 2021.

[41] Rance J. DeLong and LynuxWorks. *MLS with MILS*. 2006.

[42] *DNP3 protocol adapter*. 2021. URL: https://owlcyberdefense.com/products/data-diode-products/software-modules/dnp3/.

[43] Zhao-Hui Du et al. *Secure Encrypted Virtualization is Unsecure*. 2017. DOI: 10.48550/ARXIV.1712.05090. URL: https://arxiv.org/abs/1712.05090.

[44] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. "Verified Protection Model of the seL4 Microkernel". In: University of New South Wales Sydney, Australia, 2008.

[45] Leonard Elliot. *Army CCDC-GVSC Vision and Progress*. seL4® Center of Excellence 2020 Summit, Jan. 2021. URL: https://www.youtube.com/watch?v=EqnKeRcDQu0.

[46] Nicholas Evancich. *seL4 Overview and Tutorial*. 2020. URL: http://secdev.ieee.org/wp-content/uploads/2020/11/t1-03-evancich.pdf.

[47] B. Farroha, M. Whitfield, and D. Farroha. "Enabling Net-Centricity through Cross Domain Information Sharing". In: *IEEE SysCon 2009 —3rd Annual IEEE International Systems Conference, 2009 Vancouver, Canada, March 23–26, 2009*. Vancouver, Canada: IEEE, Mar. 2009.

[48] Bassam S. Farroha Deborah L. Farroha et al. "Challenges and Alternatives in Building a Secure Information Sharing Environment through a Community-Driven Cross Domain Infrastructure". In: IEEE, 2009.

[49] Fibersystem. *Data diodes for isolated and classified networks*. 2021. URL: https://www.fibersystem.com/data-diodes/.

[50] Forcepoint. *Forcepoint Cross Domain Solutions*. https://www.forcepoint.com/solutions/need/cross-domain. 2021.

[51] Forcepoint. *Forcepoint Meets Current National Cross Domain Startegy Managment Office Raise-The-Bar Guidelines for Cross Domain*. Oct. 2021. URL: https://www.forcepoint.com/sites/default/files/resources/files/solution_brief_cross_domain_raise_the_bar_en.pdf.

[52] Miguel Borges de Freitas et al. "SDN-Enabled Virtual Data Diode". In: *Computer Security*. Ed. by Sokratis K. Katsikas et al. Cham: Springer International Publishing, 2019, pp. 102–118. ISBN: 978-3-030-12786-2.

[53] Tal Garfinkel et al. "Terra: A Virtual Machine-Based Platform for Trusted Computing". In: Bolton Landing, New York, USA: ACM, Oct. 2003.

[54] Garrison. *Transformational Cross-Domain Technology*. 2021. URL: https://www.garrison.com/en/cross-domain.

[55] Qian Ge et al. *Time Protection: The Missing OS Abstraction*. 2019.

[56] Phillip Gibbons. *seL4: Formal Verification of an OS Kernel*. 2021. URL: https://www.cs.cmu.edu/~15712/lectures/17-seL4.pdf.

[57] Looking Glass. *LookingGlass IRD-100 Data Sheet: Stealth Threat Response at the Network Edge*. Feb. 2019.

[58] Daniel Gruss et al. "Another Flip in the Wall of Rowhammer Defenses". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 245–261. DOI: 10.1109/SP.2018.00031.

[59] Robert S. Hanmer and Lucent Lane. *N-Version Programming*. 2009.

[60] Michael Hanspach and Jorg Keller. "In Guards We Trust: Security and Privacy in Operating Systems Revisited". In: *2013 International Conference on Social Computing*. IEEE, Sept. 2013. DOI: 10.1109/socialcom.2013.87.

[61] Songlin He et al. *Blockchain-Based Automated Cyber Security Management*. 2020.

[62] Mark R. Heckman, Roger R. Schell, and Edwards E. Reed. "A High-assurance, Virtual Guard Architecture". In: *IEEE* (2012).

[63] Mark R. Heckman, Roger R. Schell, and Edwards E. Reed. "Towards Formal Evaluation of a High-Assurance Guard". In: *6th Layered Assurance Workshop*. Ed. by Christoph Schuba. Orlando, Forida, USA, Dec. 2012, pp. 25–32.

[64] Gernot Heiser. *How to (and how not to) use seL4 IPC*. https://microkerneldude.wordpress.com/2019/03/07/how-to-and-how-not-to-use-sel4-ipc/. Mar. 2019.

[65] Gernot Heiser. "The seL4 Microkernel – An Introduction". In: *The seL4 Foundation*. Vol. 1.2. LF Projects, LLC. June 2020.

[66] Gernot Heiser. *The seL4 Report: State of the Union*. YouTube, Nov. 2020. URL: https://www.youtube.com/watch?v=_2KgrFm2Fz4&t=1961s.

[67] Gernot Heiser and Kevin Elphinstone. "L4 Microkernels: The Lessons from 20 Years of Research and Deployment". In: *ACM Trans. Comput. Syst.* 34.1 (Apr. 2016). ISSN: 0734-2071. DOI: 10.1145/2893177. URL: https://doi-org.proxy.libraries.uc.edu/10.1145/2893177.

[68] Gernot Heiser, Gerwin Klein, and Toby Murray. "Can We Prove Time Protection?" In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '19. Bertinoro, Italy: Association for Computing Machinery, 2019, pp. 23–29. ISBN: 9781450367271. DOI: 10.1145/3317550.3321431. URL: https://doi.org/10.1145/3317550.3321431.

[69] Hirschmann. *Rail Data Diode*. Oct. 2021. URL: https://catalog.belden.com/techdata/EN/19DINRailAdapter_techdata.pdf.

[70] *HTTP protocol adapter*. 2021. URL: https://owlcyberdefense.com/products/data-diode-products/software-modules/http/.

[71] OPSWAT INC. *Data Diode Guide*. 2021. URL: https://www.opswat.com/products/data-diode.

[72] Dell Inc. *PowerEdge R65625*. 2021. URL: https://i.dell.com/sites/csdocuments/Product_Docs/en/poweredge-r6525-spec-sheet.pdf.

[73] infodas. *SDoT Diode*. Mar. 2021. URL: https://www.infodas.de/en/products/sdot_cross_domain_solutions/data_diode/.

[74] FOX IT. *Fox Data Diode*. 2021. URL: https://www.fox-it.com/nl-en/fox-crypto/fox-datadiode/.

[75] T. Jaeger. *Operating System Security*. Synthesis Lectures on Information Security, Privacy, and Trust. Morgan and Claypool, 2008, p. 218. DOI: 10.2200/S00126ED1V01Y200808SPT001.

[76] David Kaplan, Jeremy Powell, and Tom Woller. "AMD Memory Encryption". In: *AMD Developer Central, Advanced Micro Devices, Inc.* (Apr. 2016). URL: ttps://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[77] Mehmet Kara. "A Model for Secure Data Sharing Between Different Security Level Networks". In: *Computer Engineering and Information Technology* 6:1 (Jan. 2017). DOI: 10.4172/2324-9307.1000163.

[78] Gerwin Klein et al. "seL4: formal verification of an OS kernel". In: *SOSP '09*. 2009. URL: https://www.sigops.org/s/conferences/sosp/2009/papers/klein-sosp09.pdf.

[79] Gerwin Klein et al. "Comprehensive Formal Verification of an OS Microkernel". In: *ACM Transactions on Computer Systems (TOCS)* 32 (Feb. 2014). DOI: 10.1145/2560537.

[80] Gerwin Klein et al. "Formally Verified Software in the Real World". In: *Commun. ACM* 61.10 (Sept. 2018), pp. 68–77. ISSN: 0001-0782. DOI: 10.1145/3230627. URL: https://doi.org/10.1145/3230627.

[81] Manuel Koch, Luigi Mancini, and Francesco Parisi Presicce. "Decidability of Safety in Graph-Based Models for Access Control". In: Oct. 2002, pp. 229–243. ISBN: 978-3-540-44345-2. DOI: 10.1007/3-540-45853-0_14.

[82] Bernard F. Koelsch and Army War College Carlisle Barracks PA. *Solving the Cross-Domain Conundrum*. English. 2013.

[83] D. Richard Kuhn et al. "Introduction to Public Key Technology and the Federal PKI Infrastructure". In: *SP 800-32*. National Institue of Standards and Technology. Feb. 2001.

[84] Ihor Kuz et al. "capDL: A Language for Describing Capability-Based Systems". In: June 2010, pp. 31–36. DOI: 10.1145/1851276.1851284. URL: http://conferences.sigcomm.org/sigcomm/2010/papers/apsys/p31.pdf.

[85] Andrew Kwong et al. "RAMBleed: Reading Bits in Memory Without Accessing Them". In: *41st IEEE Symposium on Security and Privacy (S&P)*. 2020.

[86] Dayeol Lee et al. "Keystone: an open framework for architecting trusted execution environments". In: Apr. 2020, pp. 1–16. DOI: 10.1145/3342195.3387532.

[87] Henry M. Levy. *Capability-Based Computer Systems*. USA: Butterworth-Heinemann, 1984. ISBN: 0932376223.

[88] Feng Li et al. "Distributed Data Management Using MapReduce". In: *ACM Comput. Surv.* 46.3 (Jan. 2014). ISSN: 0360-0300. DOI: 10.1145/2503009. URL: https://doi-org.proxy.libraries.uc.edu/10.1145/2503009.

[89] Mengyuan Li et al. "Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1257–1272. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan.

[90] Bryan Lim. *Cybersecurity Young Writers Series Article 2: Data DIODES EXPLAINED*. Ed. by NyanEditor Tun Zaw. May 2021. URL: https://athenadynamics.com/demystifying-data-diodes-data-diodes-explained/.

[91] R. J. Lipton and L. Snyder. "A Linear Time Algorithm for Deciding Subject Security". In: *J. ACM* 24.3 (July 1977), pp. 455–464. ISSN: 0004-5411. DOI: 10.1145/322017.322025. URL: https://doi.org/10.1145/322017.322025.

[92] Anna Lyons et al. "Scheduling-context capabilities: a principled, light-weight operating-system mechanism for managing time". In: Apr. 2018, pp. 1–16. DOI: 10.1145/3190508.3190539.

[93] SIGNAL Magazine. *Enlighten IT is Providing Cyber Situational Awareness for Big Data Platforms*. urlhttps://www.afcea.org/content/enlighten-it-providing-cyber-situational-awareness-big-data-platforms. Apr. 2021.

[94] Andrea Mattavelli. "Understanding the Redundancy of Software Systems". In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ICSE Companion 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 698–701. ISBN: 9781450327688. DOI: 10.1145/2591062.2591077. URL: https://doi.org/10.1145/2591062.2591077.

[95] Brian McGillion et al. "Open-TEE – An Open Virtual Trusted Execution Environment". In: *2015 IEEE Trustcom/BigDataSE/ISPA*. IEEE, Aug. 2015. DOI: 10.1109/trustcom.2015.400.

[96] Phillip Mestas. *Securing AMD SEV*. 2020.

[97] *Mqtt Protocol Adapter*. Sept. 2021. URL: https://owlcyberdefense.com/resource/mqtt-protocol-adapter/.

[98] Toby Murray et al. "seL4: From General Purpose to a Proof of Information Flow Enforcement". In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 415–429. DOI: 10.1109/SP.2013.35.

[99] N.S.A. *Raise The Bar*. URL: https://www.nsa.gov/Cybersecurity/Partnership/National-Cross-Domain-Strategy-Management-Office/.

[100] Committee on National Security Systems (CNSS). *Committee on National Security Systems(CNSS) Glossary*. Apr. 2015.

[101] Naval-technology.com. *US Navy contracts Lockheed for Radiant Mercury Intelligence Sharing System*. Feb. 2015. URL: https://www.naval-technology.com/uncategorised/newsus-navy-contracts-lockheed-for-radiant-mercury-intelligence-sharing-system-4521866/.

[102] P. Neumann et al. *A Provably Secure Operating System*. 1975.

[103] G. H. Nibaldi. "Specification of a Trusted Computing Base (TCB)". In: *The Mitre Corporation*. AF19628-80-C-00011. The Mitre Corporation. Bedford, Massachusetts, Nov. 1979.

[104] Leonor Prensa Nieto. "The Rely-Guarantee Method in Isabelle/HOL". In: *Programming Languages and Systems*. Ed. by Pierpaolo Degano. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 348–362. ISBN: 978-3-540-36575-4.

[105] T. Nipkow. *Hoare Logics in Isabelle/HOL*. 2002. URL: https://www21.in.tum.de/~nipkow/pubs/MOD2001.pdf.

[106] Hamed Okhravi and F.T. Sheldon. "Data diodes in support of trustworthy cyber infrastructure". In: (Jan. 2010), p. 23. DOI: 10.1145/1852666.1852692.

[107] *OWASP Top 10*. 2021. URL: https://owasp.org/Top10/.

[108] QEMU. *QEMU: the FAST! processor emulator*. https://www.qemu.org/. 2021.

[109] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. "Trusted Execution Environment: What It is, and What It is Not". In: *2015 IEEE Trustcom/BigDataSE/ISPA*. IEEE, Aug. 2015. DOI: 10.1109/trustcom.2015.357.

[110] Alok Sanghavi. "What is formal verification?" In: *EE Times-Asia* (May 2010).

[111] K. Scarfone and P. Mell. "Guide to Intrusion and Detection and Prevention Systems and (IDPS)". In: Gaithersburg, Maryland: National Institute of Standards and Technology, NIST, 2009.

[112] R. Schell, T. Tao, and M. Heckman. *Designing the GEMSOS security kernel for security and performance*. 1985.

[113] Austin Scott and Richard Carbone. "Tactical Data Diodes in Industrial Automation and Control Systems". In: *SANS Institute Information Security Reading Room*. SANS Institute, 2020.

[114] *seL4 Docs*. 2020. URL: https://docs.sel4.systems/projects/camkes/manual.html.

[115] Thomas Sewell et al. "seL4 Enforces Integrity". In: *Interactive Theorem Proving*. Ed. by Marko van Eekelen et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 325–340. ISBN: 978-3-642-22863-6.

[116] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. "Translation Validation for a Verified OS Kernel". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 471–482. ISBN: 9781450320146. DOI: 10.1145/2491956.2462183. URL: https://doi.org/10.1145/2491956.2462183.

[117] Scott Smith. "Shedding Light on Cross Domain Solutions". In: *SANS Institute Information Security Reading Room* (Nov. 2015).

[118] Snort. *Snort: Open Source Intrustion Prevention System*. https://www.snort.org/. 2021.

[119] Green Hills Software. *Safety-Critical Products: INTEGRITY-178B Real-Time Operating System*. 2005.

[120] Waterfall Security Solutions. *WF-500 DIN RAIL*. July 2021. URL: https://waterfall-security.com/din-rail/.

[121] *SSH File Transfer Protocol (SFTP)*. Mar. 2020. URL: https://owlcyberdefense.com/wp-content/uploads/2020/03/20-OWL-D080-V1-SFTP.pdf.

[122] BAE Systems. *XTS-400 UK EAL5 security target - XTS-400 version 6.4(UKE)*. Nov. 2007. URL: https://www.commoncriteriaportal.org/files/epfiles/xts-400.pdf.

[123] BAE Systems. *Data diode solution*. Apr. 2020. URL: https://www.baesystems.com/en/product/data-diode-solution.

[124] BAE Systems. *XTS® Guard 7*. Apr. 2020. URL: https://www.baesystems.com/en/product/xts--guard-7.

[125] Data61 Trustworthy Systems Team. "seL4 Reference Manual". In: Version 12.1.0. General Dynamics C4 Systems. June 2021. URL: https://sel4.systems/Info/Docs/seL4-manual-latest.pdf.

[126] B. M. Thomas and N. L. Ziring. "Using Classified Intelligence to Defend Unclassified Networks". In: *2015 48th Hawaii International Conference on System Sciences*. Jan. 2014, pp. 2298–2307. DOI: 10.1109/HICSS.2015.275.

[127] Vado. *Vado Unidirectional gateway*. 2021. URL: https://www.vadosecurity.com/.

[128] Dexi Wang et al. "Verification of Implementations of Cryptographic Hash Functions". In: *IEEE Access* 5 (2017), pp. 7816–7825. DOI: 10.1109/ACCESS.2017.2697918.

[129] Shaun Waterman. "Assured Cross-Domain Access Through Hardware-Based Security: Sponsored Content". In: *SIGNAL* (Aug. 2021). URL: https://www.afcea.org/content/assured-cross-domain-access-through-hardware-based-security-sponsored-content.

[130] Nils Wistoff et al. "Prevention of Microarchitectural Covert Channels on an Open-Source 64-bit RISC-V Core". In: *ArXiv* abs/2005.02193 (May 2020).

[131] R H Wyman and G L Johnson. *Defense against common mode failures in protection system design*. Aug. 1997. URL: https://www.osti.gov/biblio/358846.

Appendix A

Appendix A: vCDS Source Code

A.1 Root Project CMakeLists.txt

```
#*********************************************************
#* Author: Nathan Daughety
#* File: CMakeLists.txt
#* Description: root CMake input file for building
#*     the stream processor application
#*********************************************************/

cmake_minimum_required(VERSION 3.8.2)
project(flea_qemu)

# Edit as needed
set(GCC_VERSION 9) # 9 or 7.5.0

#---------
# Includes
# --------
# CAmkES VM
find_package(camkes-vm REQUIRED)
include(${CAMKES_VM_HELPERS_PATH})

# CAmkES VM Linux
find_package(camkes-vm-linux REQUIRED)
```

```
include(${CAMKES_VM_LINUX_HELPERS_PATH})

# Linux Module
include(${CAMKES_VM_LINUX_MODULE_HELPERS_PATH})

# External
include(ExternalProject)

# Cross VM connection source
file(GLOB connectionSourceLow src/cross_vm_connections_low.c)
file(GLOB connectionSourceHigh src/cross_vm_connections_high.c)



# ---------------------------------
# Linux Kernel and File System Setup
# ---------------------------------
GetDefaultLinuxKernelFile(kernel_file)

GetDefaultLinuxRootfsFile(high_rootfs_file)
GetDefaultLinuxRootfsFile(low_rootfs_file)

DecompressLinuxKernel(extract_linux_kernel
    decompressed_kernel
    ${kernel_file}
)

AddToFileServer("bzImage"
    ${decompressed_kernel}
    DEPENDS extract_linux_kernel
)



# -------------------
# Kernel Module Setup
```

```
# --------------------
# Compile CrossVM Dataport Connection Module
DefineLinuxModule(
     ${CAMKES_VM_LINUX_DIR}/camkes-linux-artifacts/camkes-linux-modules/camkes-
    connector-modules/connection
     connection-module
     connection-target
          KERNEL_DIR ${CMAKE_CURRENT_SOURCE_DIR}/5.4.139/linux-5.4.139/
)



# -------------
# LowSide Setup
# -------------
DeclareCAmkESVM(LowSide
    EXTRA_SOURCES ${connectionSourceLow}
    INCLUDES include
)

# Add dataport write binary
ExternalProject_Add(dataport_low-app
    SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/pkgs/dataport
    BINARY_DIR ${CMAKE_CURRENT_BINARY_DIR}/dataport_low-app
    INSTALL_COMMAND ""
    BUILD_ALWAYS ON
    EXCLUDE_FROM_ALL
    CMAKE_ARGS -DCMAKE_C_COMPILER=${CMAKE_C_COMPILER}
)

AddExternalProjFilesToOverlay(dataport_low-app
    ${CMAKE_CURRENT_BINARY_DIR}/dataport_low-app
    overlay_low
    "usr/sbin"
    FILES dataport_write dataport_read
)
```

```
94
95
96  ExternalProject_Add(send_record-app
97      SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/pkgs/send_record
98      BINARY_DIR ${CMAKE_CURRENT_BINARY_DIR}/send_record-app
99      INSTALL_COMMAND ""
100     BUILD_ALWAYS ON
101     EXCLUDE_FROM_ALL
102 )
103
104 AddExternalProjFilesToOverlay(send_record-app
105     ${CMAKE_CURRENT_BINARY_DIR}/send_record-app
106     overlay_low
107     "usr/bin"
108     FILES send_record
109 )
110
111 AddFileToOverlayDir("connection.ko"
112     ${connection-module}
113     #    "lib/modules/4.8.16/kernel/drivers/vmm"
114     "lib/modules/5.4.139/kernel/drivers/vmm"
115     overlay_low
116     DEPENDS
117     connection-target
118 )
119
120 AddFileToOverlayDir("S90cross_vm_module_init"
121     ${CMAKE_CURRENT_SOURCE_DIR}/overlay_files/init_scripts/cross_vm_module_init
122     "etc/init.d"
123     overlay_low
124 )
125
126 #[[
127 AddFileToOverlayDir("pipeline_check.pcap"
128     ${CMAKE_CURRENT_SOURCE_DIR}/overlay_files/benchmark/pipeline_check.pcap
129     "/"
```

```
    overlay_low
)
]]

# Pack the overlay into the rootfs archive
AddOverlayDirToRootfs(
    overlay_low
    ${low_rootfs_file}
    "buildroot"
    "rootfs_install"
    low_rootfs_file
    low_rootfs_target
)



# --------------
# HighSide Setup
# --------------
DeclareCAmkESVM(HighSide
    EXTRA_SOURCES ${connectionSourceHigh}
    INCLUDES include
)

# Dataport Read/Write Binary
ExternalProject_Add(dataport_high-app
    SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/pkgs/dataport
    BINARY_DIR ${CMAKE_CURRENT_BINARY_DIR}/dataport_high-app
    INSTALL_COMMAND ""
    BUILD_ALWAYS ON
    EXCLUDE_FROM_ALL
    CMAKE_ARGS -DCMAKE_C_COMPILER=${CMAKE_C_COMPILER}
)

AddExternalProjFilesToOverlay(dataport_high-app
     ${CMAKE_CURRENT_BINARY_DIR}/dataport_high-app
```

```
     overlay_high
     "usr/sbin"
     FILES dataport_read dataport_write
)

# Receive Record Binary
ExternalProject_Add(receive_record-app
    SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/pkgs/receive_record
    BINARY_DIR ${CMAKE_CURRENT_BINARY_DIR}/receive_record-app
    INSTALL_COMMAND ""
    BUILD_ALWAYS ON
    EXCLUDE_FROM_ALL
    CMAKE_ARGS -DCMAKE_C_COMPILER=${CMAKE_C_COMPILER}
)

AddExternalProjFilesToOverlay(receive_record-app
    ${CMAKE_CURRENT_BINARY_DIR}/receive_record-app
    overlay_high
    "usr/sbin"
    FILES receive_record
)

# Snort Binary
ExternalProject_Add(snort-app
    SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/pkgs/snort
    BINARY_DIR ${CMAKE_CURRENT_BINARY_DIR}/snort-app
    INSTALL_COMMAND ""
    BUILD_ALWAYS ON
    EXCLUDE_FROM_ALL
)

AddExternalProjFilesToOverlay(snort-app
     ${CMAKE_CURRENT_BINARY_DIR}/snort-app
     overlay_high
     "usr/sbin"
     FILES snort
```

```
)

AddFileToOverlayDir("connection.ko"
    ${connection-module}
    #    "lib/modules/4.8.16/kernel/drivers/vmm"
    "lib/modules/5.4.139/kernel/drivers/vmm"
    overlay_high
    DEPENDS connection-target
)

AddFileToOverlayDir("S90cross_vm_module_init"
    ${CMAKE_CURRENT_SOURCE_DIR}/overlay_files/init_scripts/cross_vm_module_init
    "etc/init.d"
    overlay_high
)


# Pack the overlay into the rootfs archive
AddOverlayDirToRootfs(
    overlay_high
    ${high_rootfs_file}
    "buildroot"
    "rootfs_install"
    high_rootfs_file
    high_rootfs_target
)



# -----------
# Guard Setup
# -----------
# Set files for libblake3
file(GLOB LIB_DIR lib/libblake3_x86-64)
set(DEPS
    ${LIB_DIR}/src/blake3.c
```

```
    ${LIB_DIR}/src/blake3_dispatch.c
    ${LIB_DIR}/src/blake3_portable.c
    ${LIB_DIR}/src/blake3_avx2_x86-64_unix.S
    ${LIB_DIR}/src/blake3_avx512_x86-64_unix.S
    ${LIB_DIR}/src/blake3_sse2_x86-64_unix.S
    ${LIB_DIR}/src/blake3_sse41_x86-64_unix.S
)

DeclareCAmkESComponent(Guard
    SOURCES
    ${DEPS}
    ${CMAKE_CURRENT_SOURCE_DIR}/components/Guard/src/main.c
    ${CMAKE_CURRENT_SOURCE_DIR}/components/Guard/src/integrity.c
    ${CMAKE_CURRENT_SOURCE_DIR}/components/Guard/src/disposition.c
    INCLUDES
    include
    ${LIB_DIR}/include
    /usr/lib/gcc/x86_64-linux-gnu/${GCC_VERSION}/include
)



# ---------------
# Temporary Files
# ---------------
AddFileToOverlayDir(
    "S90crossvm_test"
    ${CMAKE_CURRENT_SOURCE_DIR}/overlay_files/init_scripts/cross_vm_test_low
    "etc/init.d"
    overlay_low
)

AddFileToOverlayDir(
    "S90crossvm_test"
    ${CMAKE_CURRENT_SOURCE_DIR}/overlay_files/init_scripts/cross_vm_test_high
    "etc/init.d"
```

```
    overlay_high
)



# --------
# Finalize
# --------
# Add high and low rootfs images to file server
AddToFileServer("low_rootfs.cpio" ${low_rootfs_file})
AddToFileServer("high_rootfs.cpio" ${high_rootfs_file})

# Initialise CAmkES Root Server
DeclareCAmkESVMRootServer(flea_qemu.camkes)
```

Listing A.1: CMakeLists.txt File

A.2 Root Project stream_processor.camkes

```
/*********************************************************
 * Author: Nathan Daughety
 * File: stream_processor.camkes
 * Description: Define and configure components and
 *     connections
 *********************************************************/

#ifdef HAVE_AUTOCONF
#include <autoconf.h>
#endif

#include <configurations/vm.h>

import <VM/vm.camkes>;
import "components/LowSide/LowSide.camkes";
import "components/HighSide/HighSide.camkes";
import "components/Guard/Guard.camkes";

```

```
19
20
21 assembly {
22     composition {
23         VM_COMPOSITION_DEF()
24
25         VM_PER_VM_COMP_DEF(LowSide, 0)  // component LowSide vm0
26         VM_PER_VM_COMP_DEF(HighSide, 1) // component HighSide vm1
27
28         component Guard guard;
29
30         connection seL4Notification readyConn(
31           from vm1.ready,
32             to guard.ready
33         );
34         connection seL4GlobalAsynch doneConn(
35           from guard.done,
36             to vm1.done
37       );
38
39        /* seL4SharedDataWithCaps -- connects dataport interfaces where the
40         * "to" side has access to capabilities to the frames backing the
41         * dataport -- required for cross VM dataports in order to connect
42         * processes in a VM guest to other CAmkES components
43         */
44         connection seL4SharedDataWithCaps lowToHighConn(
45           from vm0.record,
46             to vm1.record
47         );
48
49         connection seL4SharedDataWithCaps highToGuardBridgeConn(
50           from vm1.record_bridge,
51           to guard.record
52       );
53     }
54
```

```
    configuration {
        VM_CONFIGURATION_DEF() // general vm configuration

      guard.record_size = 4096;

       /* Our data diode (one way communication) is defined by
        * access rights to shared memory (dataport)
      +----------------+--------------+
      |   Component    |   Access     |
      |                |   Rights     |
      +----------------+--------------+
      |    LowSide     | write-only   |
      +----------------+--------------+
      |    HighSide    | read-only    |
      +----------------+--------------+
      |     Guard      | read-only    |
      +----------------+--------------+

        Note: the bridged dataport accessed only by the Guard
        and the HighSide are read/write because they are
        at the same classification level and read/write
        is needed to create the bridge
        */
        vm0.record_access = "W";         // write only
        vm1.record_access = "R";         // read only
        vm1.record_bridge_access = "RW"; // read/write
        guard.record_access      = "RW"; // read/write
    }
}
```

Listing A.2: Stream Processor CAmkES File

A.3 LowSide.camkes

```
/*****************************************************
 * Author: Nathan Daughety
 * File: HighSide.camkes
```

```
 * Description: Define LowSide component and attributes
 *     with the guest vm configurations
 *********************************************************/

import <VM/vm.camkes>;
#include <configurations/vm.h>

#define LOW_SIDE_CMDLINE "earlyprintk=ttyS0,115200 console=ttyS0,115200 \
      i8042.nokbd=y i8042.nomux=y i8042.noaux=y io_delay=udelay \
      noisapnp pci=nomsi debug root=/dev/mem"


component LowSide {
   /* Define LowSide VM with attributes
    * specified in "../../../../vm/components/VM/configurations/vm.h"
    */
    VM_INIT_DEF()

   /* Shared memory used to write data packets
    * which will be read by high side
    */
    include "record.h";
    dataport Record record;
}


assembly {
  composition {}
  configuration {
     VM_PER_VM_CONFIG_DEF(0)

     vm0.simple_untyped23_pool = 21; // 2^23 (8 MB)
     vm0.simple_untyped22_pool = 1;  // 2^22 (4 MB)
     vm0.heap_size = 0x2000000;
     vm0.guest_ram_mb = 128;
     vm0.kernel_cmdline = LOW_SIDE_CMDLINE;
```

```
    vm0.kernel_image = "bzImage";
    vm0.kernel_relocs = "bzImage";
    vm0.initrd_image = "low_rootfs.cpio";
    vm0.iospace_domain = 0x0f;

    // Dataport ID and size
    vm0.record_size = 4096;
  }
}
```

Listing A.3: Low Side CAmkES File

A.4 HighSide.camkes

```
/*********************************************************
 * Author: Nathan Daughety
 * File: HighSide.camkes
 * Description: Define HighSide component and attributes
 *    with the guest vm configurations
 *********************************************************/

import <VM/vm.camkes>;
#include <configurations/vm.h>


#define HIGH_SIDE_CMDLINE "earlyprintk=ttyS0,115200 console=ttyS0,115200 \
      i8042.nokbd=y i8042.nomux=y i8042.noaux=y io_delay=udelay noisapnp \
      pci=nomsi debug root=/dev/mem"


component HighSide {
   /* Define HighSide VM with attributes
    * specified in "../../../../vm/components/VM/configurations/vm.h"
    */
    VM_INIT_DEF()

   /* HighSide emits ready to Guard to begin
```

```
    * init or filtering.  HighSide consumes
    * messages from Guard notifying task done
    */
    consumes Done done;
    emits Ready ready;

   /* Shared memory used to read data packets
    * written from the low side
    */
    include "record.h";
    dataport Record record;
    dataport Record record_bridge;
}


assembly {
  composition {}
  configuration {
     VM_PER_VM_CONFIG_DEF(1)

     vm1.simple_untyped23_pool = 21; // 2^23 (8MB)
     vm1.simple_untyped22_pool = 1;  // 2^22 (4MB)
     vm1.heap_size = 0x2000000;
     vm1.guest_ram_mb = 128;
     vm1.kernel_cmdline = HIGH_SIDE_CMDLINE;
     vm1.kernel_image = "bzImage";
     vm1.kernel_relocs = "bzImage";
     vm1.initrd_image = "high_rootfs.cpio";
     vm1.iospace_domain = 0x10;

     // Dataport ID and size
     vm1.record_id = 1;
     vm1.record_size = 4096;
     vm1.record_bridge_id = 2;
     vm1.record_bridge_size = 4096;
  }
```

```
}
```

Listing A.4: High Side CAmkES File

A.5 Guard.camkes

```
/*********************************************************
 * Author: Nathan Daughety
 * File: Guard.camkes
 * Description: Define Guard component to be a filter
 *    between HighSide and LowSide
 *********************************************************/

component Guard {
  /* The control keyword makes this an
   * active component with a run method
   */
   control;

  /* The Guard consumes a ready event from the
   * HighSide and checks tag. Once finished,
   * a Done message is emitted to the HighSide.
   */
   consumes Ready ready;
   emits Done done;

  /* Shared memory used to read data packets
   * written from the low side
   */
   include "record.h";
   dataport Record record;
}
```

Listing A.5: Guard CAmkES File

A.6 Custom Linux Kernel Config File

```
#
# Automatically generated file; DO NOT EDIT.
# Linux/x86 5.4.139 Kernel Configuration
#

#
# Compiler: gcc (Ubuntu 10.3.0-1ubuntu1) 10.3.0
##
CONFIG_CC_IS_GCC=y
CONFIG_GCC_VERSION=100300
CONFIG_CLANG_VERSION=0
CONFIG_CC_CAN_LINK=y
CONFIG_CC_HAS_ASM_GOTO=y
CONFIG_CC_HAS_ASM_INLINE=y
CONFIG_IRQ_WORK=y
CONFIG_BUILDTIME_EXTABLE_SORT=y
CONFIG_THREAD_INFO_IN_TASK=y

#
# General setup
#
CONFIG_BROKEN_ON_SMP=y
CONFIG_INIT_ENV_ARG_LIMIT=32
CONFIG_LOCALVERSION=""
CONFIG_BUILD_SALT=""
CONFIG_HAVE_KERNEL_GZIP=y
CONFIG_HAVE_KERNEL_BZIP2=y
CONFIG_HAVE_KERNEL_LZMA=y
CONFIG_HAVE_KERNEL_XZ=y
CONFIG_HAVE_KERNEL_LZO=y
CONFIG_HAVE_KERNEL_LZ4=y
CONFIG_KERNEL_GZIP=y
CONFIG_DEFAULT_HOSTNAME="(none)"
CONFIG_SWAP=y
CONFIG_SYSVIPC=y
```

```
CONFIG_SYSVIPC_SYSCTL=y
CONFIG_POSIX_MQUEUE=y
CONFIG_POSIX_MQUEUE_SYSCTL=y
CONFIG_USELIB=y
CONFIG_HAVE_ARCH_AUDITSYSCALL=y

#
# IRQ subsystem
#
CONFIG_GENERIC_IRQ_PROBE=y
CONFIG_GENERIC_IRQ_SHOW=y
CONFIG_IRQ_DOMAIN=y
CONFIG_GENERIC_IRQ_RESERVATION_MODE=y
CONFIG_IRQ_FORCED_THREADING=y
CONFIG_SPARSE_IRQ=y
# end of IRQ subsystem

CONFIG_CLOCKSOURCE_WATCHDOG=y
CONFIG_ARCH_CLOCKSOURCE_DATA=y
CONFIG_ARCH_CLOCKSOURCE_INIT=y
CONFIG_CLOCKSOURCE_VALIDATE_LAST_CYCLE=y
CONFIG_GENERIC_TIME_VSYSCALL=y
CONFIG_GENERIC_CLOCKEVENTS=y
CONFIG_GENERIC_CLOCKEVENTS_MIN_ADJUST=y
CONFIG_GENERIC_CMOS_UPDATE=y

#
# Timers subsystem
#
CONFIG_TICK_ONESHOT=y
CONFIG_NO_HZ_COMMON=y
CONFIG_NO_HZ_IDLE=y
CONFIG_HIGH_RES_TIMERS=y
# end of Timers subsystem

CONFIG_PREEMPT_VOLUNTARY=y
```

```
72
73 #
74 # CPU/Task time and stats accounting
75 #
76 CONFIG_TICK_CPU_ACCOUNTING=y
77 CONFIG_TASKSTATS=y
78 CONFIG_TASK_DELAY_ACCT=y
79 CONFIG_TASK_XACCT=y
80 CONFIG_TASK_IO_ACCOUNTING=y
81 # end of CPU/Task time and stats accounting
82
83 #
84 # RCU Subsystem
85 #
86 CONFIG_TINY_RCU=y
87 CONFIG_SRCU=y
88 CONFIG_TINY_SRCU=y
89 # end of RCU Subsystem
90
91 CONFIG_IKCONFIG=m
92 CONFIG_IKCONFIG_PROC=y
93 CONFIG_LOG_BUF_SHIFT=18
94 CONFIG_PRINTK_SAFE_LOG_BUF_SHIFT=13
95 CONFIG_HAVE_UNSTABLE_SCHED_CLOCK=y
96
97 #
98 # Scheduler features
99 #
100 # end of Scheduler features
101
102 CONFIG_ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH=y
103 CONFIG_CGROUPS=y
104 CONFIG_CGROUP_SCHED=y
105 CONFIG_FAIR_GROUP_SCHED=y
106 CONFIG_CGROUP_FREEZER=y
107 CONFIG_CGROUP_CPUACCT=y
```

```
CONFIG_NAMESPACES=y
CONFIG_UTS_NS=y
CONFIG_IPC_NS=y
CONFIG_PID_NS=y
CONFIG_NET_NS=y
CONFIG_SCHED_AUTOGROUP=y
CONFIG_RELAY=y
CONFIG_BLK_DEV_INITRD=y
CONFIG_INITRAMFS_SOURCE=""
CONFIG_RD_GZIP=y
CONFIG_RD_BZIP2=y
CONFIG_RD_LZMA=y
CONFIG_RD_XZ=y
CONFIG_RD_LZO=y
CONFIG_RD_LZ4=y
CONFIG_CC_OPTIMIZE_FOR_SIZE=y
CONFIG_SYSCTL=y
CONFIG_HAVE_UID16=y
CONFIG_SYSCTL_EXCEPTION_TRACE=y
CONFIG_HAVE_PCSPKR_PLATFORM=y
CONFIG_BPF=y
CONFIG_UID16=y
CONFIG_MULTIUSER=y
CONFIG_SGETMASK_SYSCALL=y
CONFIG_SYSFS_SYSCALL=y
CONFIG_FHANDLE=y
CONFIG_POSIX_TIMERS=y
CONFIG_PRINTK=y
CONFIG_PRINTK_NMI=y
CONFIG_BUG=y
CONFIG_ELF_CORE=y
CONFIG_PCSPKR_PLATFORM=y
CONFIG_BASE_FULL=y
CONFIG_FUTEX=y
CONFIG_FUTEX_PI=y
CONFIG_EPOLL=y
```

```
CONFIG_SIGNALFD=y
CONFIG_TIMERFD=y
CONFIG_EVENTFD=y
CONFIG_SHMEM=y
CONFIG_AIO=y
CONFIG_IO_URING=y
CONFIG_ADVISE_SYSCALLS=y
CONFIG_MEMBARRIER=y
CONFIG_KALLSYMS=y
CONFIG_KALLSYMS_BASE_RELATIVE=y
CONFIG_ARCH_HAS_MEMBARRIER_SYNC_CORE=y
CONFIG_RSEQ=y
CONFIG_HAVE_PERF_EVENTS=y

#
# Kernel Performance Events And Counters
#
CONFIG_PERF_EVENTS=y
# end of Kernel Performance Events And Counters

CONFIG_VM_EVENT_COUNTERS=y
CONFIG_SLUB_DEBUG=y
CONFIG_SLUB=y
CONFIG_SLAB_MERGE_DEFAULT=y
CONFIG_PROFILING=y
# end of General setup

CONFIG_X86_32=y
CONFIG_X86=y
CONFIG_INSTRUCTION_DECODER=y
CONFIG_OUTPUT_FORMAT="elf32-i386"
CONFIG_ARCH_DEFCONFIG="arch/x86/configs/i386_defconfig"
CONFIG_LOCKDEP_SUPPORT=y
CONFIG_STACKTRACE_SUPPORT=y
CONFIG_MMU=y
CONFIG_ARCH_MMAP_RND_BITS_MIN=8
```

```
CONFIG_ARCH_MMAP_RND_BITS_MAX=16
CONFIG_ARCH_MMAP_RND_COMPAT_BITS_MIN=8
CONFIG_ARCH_MMAP_RND_COMPAT_BITS_MAX=16
CONFIG_GENERIC_ISA_DMA=y
CONFIG_GENERIC_BUG=y
CONFIG_ARCH_MAY_HAVE_PC_FDC=y
CONFIG_GENERIC_CALIBRATE_DELAY=y
CONFIG_ARCH_HAS_CPU_RELAX=y
CONFIG_ARCH_HAS_CACHE_LINE_SIZE=y
CONFIG_ARCH_HAS_FILTER_PGPROT=y
CONFIG_HAVE_SETUP_PER_CPU_AREA=y
CONFIG_NEED_PER_CPU_EMBED_FIRST_CHUNK=y
CONFIG_NEED_PER_CPU_PAGE_FIRST_CHUNK=y
CONFIG_ARCH_HIBERNATION_POSSIBLE=y
CONFIG_ARCH_SUSPEND_POSSIBLE=y
CONFIG_ARCH_WANT_GENERAL_HUGETLB=y
CONFIG_ARCH_SUPPORTS_DEBUG_PAGEALLOC=y
CONFIG_ARCH_SUPPORTS_UPROBES=y
CONFIG_FIX_EARLYCON_MEM=y
CONFIG_PGTABLE_LEVELS=2
CONFIG_CC_HAS_SANE_STACKPROTECTOR=y

#
# Processor type and features
#
CONFIG_ZONE_DMA=y
CONFIG_SMP=n
CONFIG_X86_32_SMP=n
CONFIG_X86_FEATURE_NAMES=y
CONFIG_RETPOLINE=y
CONFIG_X86_EXTENDED_PLATFORM=y
CONFIG_IOSF_MBI=y
CONFIG_SCHED_OMIT_FRAME_POINTER=y
CONFIG_M686=y
CONFIG_X86_GENERIC=y
CONFIG_X86_INTERNODE_CACHE_SHIFT=6
```

```
CONFIG_X86_L1_CACHE_SHIFT=6
CONFIG_X86_INTEL_USERCOPY=y
CONFIG_X86_USE_PPRO_CHECKSUM=y
CONFIG_X86_TSC=y
CONFIG_X86_CMPXCHG64=y
CONFIG_X86_CMOV=y
CONFIG_X86_MINIMUM_CPU_FAMILY=6
CONFIG_X86_DEBUGCTLMSR=y
CONFIG_CPU_SUP_INTEL=y
CONFIG_CPU_SUP_AMD=y
CONFIG_CPU_SUP_HYGON=y
CONFIG_CPU_SUP_CENTAUR=y
CONFIG_CPU_SUP_TRANSMETA_32=y
CONFIG_CPU_SUP_ZHAOXIN=y
CONFIG_DMI=y
CONFIG_NR_CPUS_RANGE_BEGIN=1
CONFIG_NR_CPUS_RANGE_END=1
CONFIG_NR_CPUS_DEFAULT=1
CONFIG_NR_CPUS=1
CONFIG_X86_UP_APIC=n

#
# Performance monitoring
#
CONFIG_PERF_EVENTS_INTEL_UNCORE=y
CONFIG_PERF_EVENTS_INTEL_RAPL=y
CONFIG_PERF_EVENTS_INTEL_CSTATE=y
# end of Performance monitoring

CONFIG_X86_16BIT=y
CONFIG_X86_ESPFIX32=y
CONFIG_X86_REBOOTFIXUPS=y
CONFIG_MICROCODE=y
CONFIG_MICROCODE_INTEL=y
CONFIG_MICROCODE_OLD_INTERFACE=y
CONFIG_X86_MSR=y
```

```
CONFIG_X86_CPUID=y
CONFIG_NOHIGHMEM=y
CONFIG_PAGE_OFFSET=0xC0000000
CONFIG_ARCH_FLATMEM_ENABLE=y
CONFIG_ARCH_SPARSEMEM_ENABLE=y
CONFIG_ARCH_SELECT_MEMORY_MODEL=y
CONFIG_ILLEGAL_POINTER_VALUE=0
CONFIG_X86_CHECK_BIOS_CORRUPTION=y
CONFIG_X86_BOOTPARAM_MEMORY_CORRUPTION_CHECK=y
CONFIG_X86_RESERVE_LOW=64
CONFIG_MTRR=y
CONFIG_X86_PAT=y
CONFIG_ARCH_USES_PG_UNCACHED=y
CONFIG_ARCH_RANDOM=y
CONFIG_X86_SMAP=y
CONFIG_X86_INTEL_UMIP=y
CONFIG_X86_INTEL_TSX_MODE_OFF=y
CONFIG_SECCOMP=y
CONFIG_HZ_100=y
CONFIG_HZ=100
CONFIG_SCHED_HRTICK=y
CONFIG_KEXEC=y
CONFIG_PHYSICAL_START=0x1000000
CONFIG_RELOCATABLE=y
CONFIG_X86_NEED_RELOCS=y
CONFIG_PHYSICAL_ALIGN=0x200000
CONFIG_MODIFY_LDT_SYSCALL=y
# end of Processor type and features

#
# Power management and ACPI options
#
CONFIG_SUSPEND=n
CONFIG_HIBERNATION=n
CONFIG_PM=n
CONFIG_ARCH_SUPPORTS_ACPI=y
```

```
CONFIG_ACPI=n

#
# CPU Frequency scaling
#
CONFIG_CPU_FREQ=y
CONFIG_CPU_FREQ_GOV_ATTR_SET=y
CONFIG_CPU_FREQ_GOV_COMMON=y
CONFIG_CPU_FREQ_DEFAULT_GOV_USERSPACE=y
CONFIG_CPU_FREQ_GOV_PERFORMANCE=y
CONFIG_CPU_FREQ_GOV_USERSPACE=y
CONFIG_CPU_FREQ_GOV_ONDEMAND=y

#
# CPU Idle
#
CONFIG_CPU_IDLE=y
CONFIG_CPU_IDLE_GOV_LADDER=y
CONFIG_CPU_IDLE_GOV_MENU=y
# end of CPU Idle

CONFIG_INTEL_IDLE=y
# end of Power management and ACPI options

#
# Bus options (PCI etc.)
#
CONFIG_PCI_GOANY=y
CONFIG_PCI_BIOS=y
CONFIG_PCI_DIRECT=y
CONFIG_ISA_DMA_API=y
CONFIG_AMD_NB=y
# end of Bus options (PCI etc.)

#
# Binary Emulations
```

```
#
CONFIG_COMPAT_32=y
# end of Binary Emulations

CONFIG_HAVE_ATOMIC_IOMAP=y

#
# Firmware Drivers
#
CONFIG_FIRMWARE_MEMMAP=y
CONFIG_DMIID=y
CONFIG_DMI_SCAN_MACHINE_NON_EFI_FALLBACK=y
# end of Firmware Drivers

CONFIG_HAVE_KVM=y
CONFIG_VIRTUALIZATION=y

#
# General architecture-dependent options
#
CONFIG_CRASH_CORE=y
CONFIG_KEXEC_CORE=y
CONFIG_OPROFILE=m
CONFIG_OPROFILE_EVENT_MULTIPLEX=y
CONFIG_HAVE_OPROFILE=y
CONFIG_OPROFILE_NMI_TIMER=y
CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS=y
CONFIG_ARCH_USE_BUILTIN_BSWAP=y
CONFIG_HAVE_IOREMAP_PROT=y
CONFIG_HAVE_KPROBES=y
CONFIG_HAVE_KRETPROBES=y
CONFIG_HAVE_OPTPROBES=y
CONFIG_HAVE_KPROBES_ON_FTRACE=y
CONFIG_HAVE_FUNCTION_ERROR_INJECTION=y
CONFIG_HAVE_NMI=y
CONFIG_HAVE_ARCH_TRACEHOOK=y
```

```
CONFIG_HAVE_DMA_CONTIGUOUS=y
CONFIG_GENERIC_SMP_IDLE_THREAD=y
CONFIG_ARCH_HAS_FORTIFY_SOURCE=y
CONFIG_ARCH_HAS_SET_MEMORY=y
CONFIG_ARCH_HAS_SET_DIRECT_MAP=y
CONFIG_HAVE_ARCH_THREAD_STRUCT_WHITELIST=y
CONFIG_ARCH_WANTS_DYNAMIC_TASK_STRUCT=y
CONFIG_ARCH_32BIT_OFF_T=y
CONFIG_HAVE_ASM_MODVERSIONS=y
CONFIG_HAVE_REGS_AND_STACK_ACCESS_API=y
CONFIG_HAVE_RSEQ=y
CONFIG_HAVE_FUNCTION_ARG_ACCESS_API=y
CONFIG_HAVE_CLK=y
CONFIG_HAVE_HW_BREAKPOINT=y
CONFIG_HAVE_MIXED_BREAKPOINTS_REGS=y
CONFIG_HAVE_USER_RETURN_NOTIFIER=y
CONFIG_HAVE_PERF_EVENTS_NMI=y
CONFIG_HAVE_HARDLOCKUP_DETECTOR_PERF=y
CONFIG_HAVE_PERF_REGS=y
CONFIG_HAVE_PERF_USER_STACK_DUMP=y
CONFIG_HAVE_ARCH_JUMP_LABEL=y
CONFIG_HAVE_ARCH_JUMP_LABEL_RELATIVE=y
CONFIG_ARCH_HAVE_NMI_SAFE_CMPXCHG=y
CONFIG_HAVE_ALIGNED_STRUCT_PAGE=y
CONFIG_HAVE_CMPXCHG_LOCAL=y
CONFIG_HAVE_CMPXCHG_DOUBLE=y
CONFIG_ARCH_WANT_IPC_PARSE_VERSION=y
CONFIG_HAVE_ARCH_SECCOMP_FILTER=y
CONFIG_SECCOMP_FILTER=y
CONFIG_HAVE_ARCH_STACKLEAK=y
CONFIG_HAVE_STACKPROTECTOR=y
CONFIG_CC_HAS_STACKPROTECTOR_NONE=y
CONFIG_STACKPROTECTOR=y
CONFIG_STACKPROTECTOR_STRONG=y
CONFIG_HAVE_ARCH_WITHIN_STACK_FRAMES=y
CONFIG_HAVE_IRQ_TIME_ACCOUNTING=y
```

```
CONFIG_HAVE_MOVE_PMD=y
CONFIG_HAVE_ARCH_TRANSPARENT_HUGEPAGE=y
CONFIG_ARCH_WANT_HUGE_PMD_SHARE=y
CONFIG_HAVE_MOD_ARCH_SPECIFIC=y
CONFIG_MODULES_USE_ELF_REL=y
CONFIG_ARCH_HAS_ELF_RANDOMIZE=y
CONFIG_HAVE_ARCH_MMAP_RND_BITS=y
CONFIG_HAVE_EXIT_THREAD=y
CONFIG_ARCH_MMAP_RND_BITS=8
CONFIG_HAVE_COPY_THREAD_TLS=y
CONFIG_CLONE_BACKWARDS=y
CONFIG_OLD_SIGSUSPEND3=y
CONFIG_OLD_SIGACTION=y
CONFIG_64BIT_TIME=y
CONFIG_COMPAT_32BIT_TIME=y
CONFIG_ARCH_HAS_STRICT_KERNEL_RWX=y
CONFIG_STRICT_KERNEL_RWX=y
CONFIG_ARCH_HAS_STRICT_MODULE_RWX=y
CONFIG_STRICT_MODULE_RWX=y
CONFIG_ARCH_HAS_REFCOUNT=y
CONFIG_HAVE_ARCH_PREL32_RELOCATIONS=y
CONFIG_ARCH_HAS_MEM_ENCRYPT=y

#
# GCOV-based kernel profiling
#
CONFIG_ARCH_HAS_GCOV_PROFILE_ALL=y
# end of GCOV-based kernel profiling

CONFIG_PLUGIN_HOSTCC=""
CONFIG_HAVE_GCC_PLUGINS=y
# end of General architecture-dependent options

CONFIG_RT_MUTEXES=y
CONFIG_BASE_SMALL=0
CONFIG_MODULES=y
```

```
CONFIG_MODULE_UNLOAD=y
CONFIG_MODULE_FORCE_UNLOAD=y
CONFIG_MODULES_TREE_LOOKUP=y
CONFIG_BLOCK=y
CONFIG_BLK_SCSI_REQUEST=y
CONFIG_BLK_DEV_BSG=y
CONFIG_BLK_DEBUG_FS=y

#
# Partition Types
#
CONFIG_PARTITION_ADVANCED=y
CONFIG_OSF_PARTITION=y
CONFIG_AMIGA_PARTITION=y
CONFIG_MAC_PARTITION=y
CONFIG_MSDOS_PARTITION=y
CONFIG_BSD_DISKLABEL=y
CONFIG_MINIX_SUBPARTITION=y
CONFIG_SOLARIS_X86_PARTITION=y
CONFIG_UNIXWARE_DISKLABEL=y
CONFIG_SGI_PARTITION=y
CONFIG_SUN_PARTITION=y
CONFIG_KARMA_PARTITION=y
CONFIG_EFI_PARTITION=y
# end of Partition Types

CONFIG_BLK_MQ_PCI=y
CONFIG_BLK_MQ_VIRTIO=y
CONFIG_BLK_MQ_RDMA=y

#
# IO Schedulers
#
CONFIG_MQ_IOSCHED_DEADLINE=y
CONFIG_MQ_IOSCHED_KYBER=y
# end of IO Schedulers
```

```
468
469 CONFIG_INLINE_SPIN_UNLOCK_IRQ=y
470 CONFIG_INLINE_READ_UNLOCK=y
471 CONFIG_INLINE_READ_UNLOCK_IRQ=y
472 CONFIG_INLINE_WRITE_UNLOCK=y
473 CONFIG_INLINE_WRITE_UNLOCK_IRQ=y
474 CONFIG_ARCH_SUPPORTS_ATOMIC_RMW=y
475 CONFIG_ARCH_USE_QUEUED_SPINLOCKS=y
476 CONFIG_ARCH_USE_QUEUED_RWLOCKS=y
477 CONFIG_ARCH_HAS_SYNC_CORE_BEFORE_USERMODE=y
478 CONFIG_FREEZER=y
479
480 #
481 # Executable file formats
482 #
483 CONFIG_BINFMT_ELF=y
484 CONFIG_ELFCORE=y
485 CONFIG_CORE_DUMP_DEFAULT_ELF_HEADERS=y
486 CONFIG_BINFMT_SCRIPT=y
487 CONFIG_BINFMT_MISC=y
488 CONFIG_COREDUMP=y
489 # end of Executable file formats
490
491 #
492 # Memory Management options
493 #
494 CONFIG_SELECT_MEMORY_MODEL=y
495 CONFIG_FLATMEM_MANUAL=y
496 CONFIG_FLATMEM=y
497 CONFIG_FLAT_NODE_MEM_MAP=y
498 CONFIG_SPARSEMEM_STATIC=y
499 CONFIG_HAVE_MEMBLOCK_NODE_MAP=y
500 CONFIG_HAVE_FAST_GUP=y
501 CONFIG_SPLIT_PTLOCK_CPUS=4
502 CONFIG_COMPACTION=y
503 CONFIG_MIGRATION=y
```

```
CONFIG_BOUNCE=y
CONFIG_VIRT_TO_BUS=y
CONFIG_MMU_NOTIFIER=y
CONFIG_DEFAULT_MMAP_MIN_ADDR=4096
CONFIG_NEED_PER_CPU_KM=y
CONFIG_GENERIC_EARLY_IOREMAP=y
CONFIG_ARCH_HAS_PTE_SPECIAL=y
# end of Memory Management options

CONFIG_NET=y
CONFIG_NET_INGRESS=y
CONFIG_SKB_EXTENSIONS=y

#
# Networking options
#
CONFIG_PACKET=y
CONFIG_UNIX=y
CONFIG_UNIX_SCM=y
CONFIG_XFRM=y
CONFIG_XFRM_ALGO=y
CONFIG_XFRM_USER=y
CONFIG_INET=y
CONFIG_IP_MULTICAST=y
CONFIG_IP_ADVANCED_ROUTER=y
CONFIG_IP_MULTIPLE_TABLES=y
CONFIG_IP_ROUTE_MULTIPATH=y
CONFIG_IP_ROUTE_VERBOSE=y
CONFIG_IP_PNP=y
CONFIG_IP_PNP_DHCP=y
CONFIG_IP_PNP_BOOTP=y
CONFIG_IP_PNP_RARP=y
CONFIG_NET_IP_TUNNEL=y
CONFIG_IP_MROUTE_COMMON=y
CONFIG_IP_MROUTE=y
CONFIG_IP_PIMSM_V1=y
```

```
CONFIG_IP_PIMSM_V2=y
CONFIG_SYN_COOKIES=y
CONFIG_INET_TUNNEL=y
CONFIG_TCP_CONG_ADVANCED=y
CONFIG_TCP_CONG_CUBIC=y
CONFIG_DEFAULT_CUBIC=y
CONFIG_DEFAULT_TCP_CONG="cubic"
CONFIG_TCP_MD5SIG=y
CONFIG_IPV6=y
CONFIG_INET6_AH=y
CONFIG_INET6_ESP=y
CONFIG_IPV6_SIT=y
CONFIG_IPV6_NDISC_NODETYPE=y
CONFIG_NETLABEL=y
CONFIG_NETWORK_SECMARK=y
CONFIG_NET_PTP_CLASSIFY=y
CONFIG_NETFILTER=y

#
# Core Netfilter Configuration
#
CONFIG_NETFILTER_INGRESS=y
CONFIG_NETFILTER_NETLINK=y
CONFIG_NETFILTER_NETLINK_LOG=y
CONFIG_NF_CONNTRACK=y
CONFIG_NF_LOG_COMMON=m
CONFIG_NF_CONNTRACK_SECMARK=y
CONFIG_NF_CONNTRACK_PROCFS=y
CONFIG_NF_CONNTRACK_FTP=y
CONFIG_NF_CONNTRACK_IRC=y
CONFIG_NF_CONNTRACK_SIP=y
CONFIG_NF_CT_NETLINK=y
CONFIG_NF_NAT=m
CONFIG_NF_NAT_FTP=m
CONFIG_NF_NAT_IRC=m
CONFIG_NF_NAT_SIP=m
```

```
CONFIG_NF_NAT_MASQUERADE=y
CONFIG_NETFILTER_XTABLES=y

#
# Xtables combined modules
#
CONFIG_NETFILTER_XT_MARK=m

#
# Xtables targets
#
CONFIG_NETFILTER_XT_TARGET_CONNSECMARK=y
CONFIG_NETFILTER_XT_TARGET_LOG=m
CONFIG_NETFILTER_XT_NAT=m
CONFIG_NETFILTER_XT_TARGET_NFLOG=y
CONFIG_NETFILTER_XT_TARGET_MASQUERADE=m
CONFIG_NETFILTER_XT_TARGET_SECMARK=y
CONFIG_NETFILTER_XT_TARGET_TCPMSS=y

#
# Xtables matches
#
CONFIG_NETFILTER_XT_MATCH_ADDRTYPE=m
CONFIG_NETFILTER_XT_MATCH_CONNTRACK=y
CONFIG_NETFILTER_XT_MATCH_POLICY=y
CONFIG_NETFILTER_XT_MATCH_STATE=y
# end of Core Netfilter Configuration


#
# IP: Netfilter Configuration
#
CONFIG_NF_DEFRAG_IPV4=y
CONFIG_NF_LOG_ARP=m
CONFIG_NF_LOG_IPV4=m
CONFIG_NF_REJECT_IPV4=y
```

```
CONFIG_IP_NF_IPTABLES=y
CONFIG_IP_NF_FILTER=y
CONFIG_IP_NF_TARGET_REJECT=y
CONFIG_IP_NF_NAT=m
CONFIG_IP_NF_TARGET_MASQUERADE=m
CONFIG_IP_NF_MANGLE=y
# end of IP: Netfilter Configuration

#
# IPv6: Netfilter Configuration
#
CONFIG_NF_REJECT_IPV6=y
CONFIG_NF_LOG_IPV6=m
CONFIG_IP6_NF_IPTABLES=y
CONFIG_IP6_NF_MATCH_IPV6HEADER=y
CONFIG_IP6_NF_FILTER=y
CONFIG_IP6_NF_TARGET_REJECT=y
CONFIG_IP6_NF_MANGLE=y
# end of IPv6: Netfilter Configuration

CONFIG_NF_DEFRAG_IPV6=y
CONFIG_HAVE_NET_DSA=y
CONFIG_NET_SCHED=y

#
# Queueing/Scheduling
#

#
# Classification
#
CONFIG_NET_CLS=y
CONFIG_NET_EMATCH=y
CONFIG_NET_EMATCH_STACK=32
CONFIG_NET_CLS_ACT=y
CONFIG_NET_SCH_FIFO=y
```

```
CONFIG_DNS_RESOLVER=y
CONFIG_NET_RX_BUSY_POLL=y
CONFIG_BQL=y
# end of Networking options

CONFIG_CAN=m
CONFIG_CAN_RAW=m
CONFIG_CAN_BCM=m
CONFIG_CAN_GW=m

#
# CAN Device Drivers
#
CONFIG_CAN_DEV=m
CONFIG_CAN_CALC_BITTIMING=y
CONFIG_CAN_SJA1000=m
CONFIG_CAN_SJA1000_ISA=m

CONFIG_CAN_DEBUG_DEVICES=y
# end of CAN Device Drivers

CONFIG_FIB_RULES=y
CONFIG_DST_CACHE=y
CONFIG_GRO_CELLS=y
CONFIG_FAILOVER=y
CONFIG_HAVE_EBPF_JIT=y

#
# Device Drivers
#
CONFIG_HAVE_EISA=y
CONFIG_HAVE_PCI=y
CONFIG_PCI=y
CONFIG_PCI_DOMAINS=y
CONFIG_PCIEPORTBUS=y
CONFIG_PCIEASPM=y
```

```
CONFIG_PCIEASPM_DEFAULT=y
CONFIG_PCI_MSI=n
CONFIG_PCI_QUIRKS=y
CONFIG_PCI_LOCKLESS_CONFIG=y
CONFIG_PCI_LABEL=y

#
# Generic Driver Options
#
CONFIG_UEVENT_HELPER=y
CONFIG_UEVENT_HELPER_PATH="/sbin/hotplug"
CONFIG_DEVTMPFS=y
CONFIG_DEVTMPFS_MOUNT=y

#
# Firmware loader
#
CONFIG_FW_LOADER=y
CONFIG_EXTRA_FIRMWARE=""
# end of Firmware loader

CONFIG_ALLOW_DEV_COREDUMP=y
CONFIG_DEBUG_DEVRES=y
CONFIG_GENERIC_CPU_AUTOPROBE=y
CONFIG_GENERIC_CPU_VULNERABILITIES=y
CONFIG_DMA_SHARED_BUFFER=y
# end of Generic Driver Options

CONFIG_CONNECTOR=y
CONFIG_PROC_EVENTS=y
CONFIG_ARCH_MIGHT_HAVE_PC_PARPORT=y
CONFIG_BLK_DEV=y
CONFIG_CDROM=y
CONFIG_BLK_DEV_LOOP=y
CONFIG_BLK_DEV_LOOP_MIN_COUNT=8

```

```
CONFIG_HAVE_IDE=y

#
# SCSI device support
#
CONFIG_SCSI_MOD=y
CONFIG_SCSI=y
CONFIG_SCSI_DMA=y
CONFIG_SCSI_PROC_FS=y

#
# SCSI support type (disk, tape, CD-ROM)
#
CONFIG_BLK_DEV_SD=y
CONFIG_BLK_DEV_SR=y
CONFIG_CHR_DEV_SG=y
CONFIG_SCSI_CONSTANTS=y

#
# SCSI Transports
#
CONFIG_SCSI_SPI_ATTRS=y
# end of SCSI Transports

# end of SCSI device support

CONFIG_ATA=y
CONFIG_ATA_VERBOSE_ERROR=y
CONFIG_SATA_PMP=y

#
# Controllers with non-SFF native interface
#
CONFIG_SATA_AHCI=y
CONFIG_SATA_MOBILE_LPM_POLICY=0
CONFIG_ATA_SFF=y
```

```
#
# SFF controllers with custom DMA interface
#
CONFIG_ATA_BMDMA=y

#
# SATA SFF controllers with BMDMA
#
CONFIG_ATA_PIIX=y

#
# PIO-only SFF controllers
#

#
# Generic fallback / legacy drivers
#
CONFIG_ATA_GENERIC=y

CONFIG_NETDEVICES=y
CONFIG_MII=y
CONFIG_NET_CORE=y
CONFIG_NETCONSOLE=y
CONFIG_NETPOLL=y
CONFIG_NET_POLL_CONTROLLER=y
CONFIG_VIRTIO_NET=y

#
# CAIF transport drivers
#

CONFIG_ETHERNET=y
CONFIG_NET_VENDOR_AGERE=y
CONFIG_NET_VENDOR_ALACRITECH=y
CONFIG_NET_VENDOR_AMAZON=y
```

```
CONFIG_NET_VENDOR_AMD=y
CONFIG_PCNET32=y
CONFIG_NET_VENDOR_AQUANTIA=y
CONFIG_NET_VENDOR_CADENCE=y
CONFIG_NET_VENDOR_CAVIUM=y
CONFIG_NET_VENDOR_CORTINA=y
CONFIG_NET_VENDOR_EZCHIP=y
CONFIG_NET_VENDOR_GOOGLE=y
CONFIG_NET_VENDOR_HUAWEI=y
CONFIG_NET_VENDOR_I825XX=y
CONFIG_NET_VENDOR_INTEL=y
CONFIG_E100=y
CONFIG_E1000=y
CONFIG_E1000E=y
CONFIG_E1000E_HWTS=y
CONFIG_IGB=y
CONFIG_IGB_HWMON=y
CONFIG_NET_VENDOR_MICROCHIP=y
CONFIG_NET_VENDOR_MICROSEMI=y
CONFIG_NET_VENDOR_NETERION=y
CONFIG_NET_VENDOR_NETRONOME=y
CONFIG_NET_VENDOR_NI=y
CONFIG_NET_VENDOR_PACKET_ENGINES=y
CONFIG_NET_VENDOR_PENSANDO=y
CONFIG_NET_VENDOR_QUALCOMM=y
CONFIG_NET_VENDOR_RENESAS=y
CONFIG_NET_VENDOR_ROCKER=y
CONFIG_NET_VENDOR_SOLARFLARE=y
CONFIG_NET_VENDOR_SOCIONEXT=y
CONFIG_NET_VENDOR_SYNOPSYS=y
CONFIG_NET_VENDOR_XILINX=y
CONFIG_USB_NET_DRIVERS=y

#
# Enable WiMAX (Networking options) to see the WiMAX drivers
#
```

```
CONFIG_NET_FAILOVER=y

#
# Input device support
#
CONFIG_INPUT=y
CONFIG_INPUT_LEDS=y
CONFIG_INPUT_FF_MEMLESS=y

#
# Userland interfaces
#
CONFIG_INPUT_EVDEV=y

#
# Hardware I/O ports
#
CONFIG_ARCH_MIGHT_HAVE_PC_SERIO=y
# end of Hardware I/O ports
# end of Input device support

#
# Character devices
#
CONFIG_TTY=y
CONFIG_VT=y
CONFIG_CONSOLE_TRANSLATIONS=y
CONFIG_VT_CONSOLE=y
CONFIG_HW_CONSOLE=y
CONFIG_VT_HW_CONSOLE_BINDING=y
CONFIG_UNIX98_PTYS=y
CONFIG_LDISC_AUTOLOAD=y
CONFIG_DEVMEM=y
CONFIG_DEVKMEM=y

#
```

```
# Serial drivers
#
CONFIG_SERIAL_EARLYCON=y
CONFIG_SERIAL_8250=y
CONFIG_SERIAL_8250_DEPRECATED_OPTIONS=y
CONFIG_SERIAL_8250_CONSOLE=y
CONFIG_SERIAL_8250_DMA=y
CONFIG_SERIAL_8250_PCI=y
CONFIG_SERIAL_8250_EXAR=y
CONFIG_SERIAL_8250_NR_UARTS=32
CONFIG_SERIAL_8250_RUNTIME_UARTS=4
CONFIG_SERIAL_8250_EXTENDED=y
CONFIG_SERIAL_8250_SHARE_IRQ=y
CONFIG_SERIAL_8250_DWLIB=y
CONFIG_SERIAL_8250_LPSS=y
CONFIG_SERIAL_8250_MID=y

#
# Non-8250 serial port support
#
CONFIG_SERIAL_CORE=y
CONFIG_SERIAL_CORE_CONSOLE=y
# end of Serial drivers

CONFIG_SERIAL_MCTRL_GPIO=y
CONFIG_HW_RANDOM=y
CONFIG_HW_RANDOM_INTEL=y
CONFIG_NVRAM=y
CONFIG_RAW_DRIVER=y
CONFIG_MAX_RAW_DEVS=256
CONFIG_DEVPORT=y
# end of Character devices

#
# I2C support
#
```

```
CONFIG_I2C=y
CONFIG_I2C_BOARDINFO=y
CONFIG_I2C_COMPAT=y
CONFIG_I2C_CHARDEV=y
CONFIG_I2C_MUX=y

CONFIG_I2C_HELPER_AUTO=y
CONFIG_I2C_SMBUS=y
CONFIG_I2C_ALGOBIT=y

#
# PC SMBus host controller drivers
#
CONFIG_I2C_I801=y
# end of I2C support

CONFIG_PPS=y

#
# PTP clock support
#
CONFIG_PTP_1588_CLOCK=y
# end of PTP clock support

CONFIG_GPIOLIB=y
CONFIG_GPIOLIB_FASTPATH_LIMIT=512
CONFIG_GPIO_SYSFS=y

#
# Memory mapped GPIO drivers
#
CONFIG_GPIO_ICH=y
# end of Memory mapped GPIO drivers

CONFIG_POWER_SUPPLY=y
CONFIG_POWER_SUPPLY_HWMON=y
```

```
CONFIG_HWMON=y

#
# Native drivers
#
CONFIG_SENSORS_MAX6650=m
CONFIG_THERMAL=y
CONFIG_THERMAL_EMERGENCY_POWEROFF_DELAY_MS=0
CONFIG_THERMAL_HWMON=y
CONFIG_THERMAL_WRITABLE_TRIPS=y
CONFIG_THERMAL_DEFAULT_GOV_STEP_WISE=y
CONFIG_THERMAL_GOV_FAIR_SHARE=y
CONFIG_THERMAL_GOV_STEP_WISE=y
CONFIG_THERMAL_GOV_USER_SPACE=y

CONFIG_WATCHDOG=y
CONFIG_WATCHDOG_HANDLE_BOOT_ENABLED=y
CONFIG_WATCHDOG_OPEN_TIMEOUT=0

#
# USB-based Watchdog Cards
#
CONFIG_SSB_POSSIBLE=y
CONFIG_BCMA_POSSIBLE=y

#
# Multifunction device drivers
#
CONFIG_MFD_CORE=y
CONFIG_LPC_ICH=y
# end of Multifunction device drivers


#
# Graphics support
#
```

```
CONFIG_INTEL_GTT=y
CONFIG_VGA_ARB=y
CONFIG_VGA_ARB_MAX_GPUS=16
CONFIG_DRM=y
CONFIG_DRM_MIPI_DSI=y
CONFIG_DRM_KMS_HELPER=y
CONFIG_DRM_KMS_FB_HELPER=y
CONFIG_DRM_FBDEV_EMULATION=y
CONFIG_DRM_FBDEV_OVERALLOC=100

CONFIG_DRM_I915=y
CONFIG_DRM_I915_FORCE_PROBE=""
CONFIG_DRM_I915_CAPTURE_ERROR=y
CONFIG_DRM_I915_COMPRESS_ERROR=y
CONFIG_DRM_I915_USERPTR=y
CONFIG_DRM_I915_USERFAULT_AUTOSUSPEND=250
CONFIG_DRM_I915_SPIN_REQUEST=5
CONFIG_DRM_PANEL=y

CONFIG_DRM_BRIDGE=y
CONFIG_DRM_PANEL_BRIDGE=y

CONFIG_DRM_PANEL_ORIENTATION_QUIRKS=y

#
# Frame buffer Devices
#
CONFIG_FB_CMDLINE=y
CONFIG_FB_NOTIFY=y
CONFIG_FB=y
CONFIG_FB_CFB_FILLRECT=y
CONFIG_FB_CFB_COPYAREA=y
CONFIG_FB_CFB_IMAGEBLIT=y
CONFIG_FB_SYS_FILLRECT=y
CONFIG_FB_SYS_COPYAREA=y
CONFIG_FB_SYS_IMAGEBLIT=y
```

```
CONFIG_FB_SYS_FOPS=y
CONFIG_FB_DEFERRED_IO=y
CONFIG_FB_MODE_HELPERS=y
CONFIG_FB_TILEBLITTING=y

CONFIG_HDMI=y

#
# Console display driver support
#
CONFIG_VGA_CONSOLE=y
CONFIG_DUMMY_CONSOLE=y
CONFIG_DUMMY_CONSOLE_COLUMNS=80
CONFIG_DUMMY_CONSOLE_ROWS=25
CONFIG_FRAMEBUFFER_CONSOLE=y
CONFIG_FRAMEBUFFER_CONSOLE_DETECT_PRIMARY=y
# end of Console display driver support

# end of Graphics support


#
# HID support
#
CONFIG_HID=y
CONFIG_HIDRAW=y
CONFIG_HID_GENERIC=y

#
# Special HID drivers
#
CONFIG_HID_A4TECH=y
CONFIG_HID_APPLE=y
CONFIG_HID_BELKIN=y
CONFIG_HID_CHERRY=y
CONFIG_HID_CHICONY=y
```

```
CONFIG_HID_CYPRESS=y
CONFIG_HID_EZKEY=y
CONFIG_HID_GYRATION=y
CONFIG_HID_ITE=y
CONFIG_HID_KENSINGTON=y
CONFIG_HID_LOGITECH=y
CONFIG_LOGITECH_FF=y
CONFIG_LOGIWHEELS_FF=y
CONFIG_HID_REDRAGON=y
CONFIG_HID_MICROSOFT=y
CONFIG_HID_MONTEREY=y
CONFIG_HID_NTRIG=y
CONFIG_HID_PANTHERLORD=y
CONFIG_PANTHERLORD_FF=y
CONFIG_HID_PETALYNX=y
CONFIG_HID_SAMSUNG=y
CONFIG_HID_SONY=y
CONFIG_HID_SUNPLUS=y
CONFIG_HID_TOPSEED=y
# end of Special HID drivers

#
# USB HID support
#
CONFIG_USB_HID=y
CONFIG_HID_PID=y
CONFIG_USB_HIDDEV=y
# end of USB HID support
# end of HID support

CONFIG_USB_OHCI_LITTLE_ENDIAN=y
CONFIG_USB_SUPPORT=y
CONFIG_USB_COMMON=y
CONFIG_USB_ARCH_HAS_HCD=y
CONFIG_USB=y
CONFIG_USB_PCI=y
```

```
CONFIG_USB_ANNOUNCE_NEW_DEVICES=y

#
# Miscellaneous USB options
#
CONFIG_USB_DEFAULT_PERSIST=y
CONFIG_USB_AUTOSUSPEND_DELAY=2
CONFIG_USB_MON=y

#
# USB Host Controller Drivers
#
CONFIG_USB_EHCI_HCD=y
CONFIG_USB_EHCI_PCI=y
CONFIG_USB_OHCI_HCD=y
CONFIG_USB_OHCI_HCD_PCI=y
CONFIG_USB_UHCI_HCD=y

#
# USB Device Class drivers
#
CONFIG_USB_PRINTER=y

#
# also be needed; see USB_STORAGE Help for more info
#
CONFIG_USB_STORAGE=y

CONFIG_NEW_LEDS=y
CONFIG_LEDS_CLASS=y

#
# LED Triggers
#
CONFIG_LEDS_TRIGGERS=y

```

```
#
# iptables trigger is under Netfilter config (LED target)
#
CONFIG_INFINIBAND=y
CONFIG_INFINIBAND_ADDR_TRANS=y
CONFIG_INFINIBAND_VIRT_DMA=y
CONFIG_EDAC_ATOMIC_SCRUB=y
CONFIG_EDAC_SUPPORT=y
CONFIG_RTC_LIB=y
CONFIG_RTC_MC146818_LIB=y
CONFIG_RTC_CLASS=y
CONFIG_RTC_SYSTOHC=y
CONFIG_RTC_SYSTOHC_DEVICE="rtc0"
CONFIG_RTC_NVMEM=y

#
# RTC interfaces
#
CONFIG_RTC_INTF_SYSFS=y
CONFIG_RTC_INTF_PROC=y
CONFIG_RTC_INTF_DEV=y

#
# I2C RTC drivers
#

#
# SPI RTC drivers
#
CONFIG_RTC_I2C_AND_SPI=y

#
# Platform RTC drivers
#
CONFIG_RTC_DRV_CMOS=y

```

```
#
# HID Sensor RTC drivers
#
CONFIG_DMADEVICES=y

#
# DMA Devices
#
CONFIG_DMA_ENGINE=y
CONFIG_DMA_VIRTUAL_CHANNELS=y
CONFIG_DW_DMAC_CORE=y
CONFIG_HSU_DMA=y

#
# DMA Clients
#

#
# DMABUF options
#
CONFIG_SYNC_FILE=y
# end of DMABUF options

CONFIG_UIO=y
CONFIG_UIO_PDRV_GENIRQ=y
CONFIG_UIO_DMEM_GENIRQ=y
CONFIG_UIO_PCI_GENERIC=y
CONFIG_VIRT_DRIVERS=y
CONFIG_VIRTIO=y
CONFIG_VIRTIO_MENU=y
CONFIG_VIRTIO_PCI=y
CONFIG_VIRTIO_PCI_LEGACY=y

CONFIG_PMC_ATOM=y
CONFIG_CLKDEV_LOOKUP=y
CONFIG_HAVE_CLK_PREPARE=y
```

```
CONFIG_COMMON_CLK=y


#
# Clock Source drivers
#
CONFIG_CLKSRC_I8253=y
CONFIG_CLKEVT_I8253=y
CONFIG_I8253_LOCK=y
CONFIG_CLKBLD_I8253=y
# end of Clock Source drivers
# end of SOC (System On Chip) specific Drivers

CONFIG_VME_BUS=y

#
# VME Bridge Drivers
#
CONFIG_VME_TSI148=m

#
# VME Board Drivers
#

#
# VME Device Drivers
#

CONFIG_NVMEM=y
CONFIG_NVMEM_SYSFS=y
# end of Device Drivers

#
# File systems
#
CONFIG_DCACHE_WORD_ACCESS=y
```

```
CONFIG_FS_IOMAP=y
CONFIG_EXT4_FS=y
CONFIG_EXT4_USE_FOR_EXT2=y
CONFIG_EXT4_FS_POSIX_ACL=y
CONFIG_EXT4_FS_SECURITY=y
CONFIG_JBD2=y
CONFIG_FS_MBCACHE=y
CONFIG_FS_POSIX_ACL=y
CONFIG_EXPORTFS=y
CONFIG_FILE_LOCKING=y
CONFIG_MANDATORY_FILE_LOCKING=y
CONFIG_FSNOTIFY=y
CONFIG_DNOTIFY=y
CONFIG_INOTIFY_USER=y
CONFIG_QUOTA=y
CONFIG_QUOTA_NETLINK_INTERFACE=y
CONFIG_QUOTA_TREE=y
CONFIG_QFMT_V2=y
CONFIG_QUOTACTL=y
CONFIG_AUTOFS4_FS=y
CONFIG_AUTOFS_FS=y
CONFIG_OVERLAY_FS=y
CONFIG_OVERLAY_FS_REDIRECT_ALWAYS_FOLLOW=y

#
# CD-ROM/DVD Filesystems
#
CONFIG_ISO9660_FS=y
CONFIG_JOLIET=y
CONFIG_ZISOFS=y
# end of CD-ROM/DVD Filesystems

#
# DOS/FAT/NT Filesystems
#
CONFIG_FAT_FS=y
```

```
CONFIG_MSDOS_FS=y
CONFIG_VFAT_FS=y
CONFIG_FAT_DEFAULT_CODEPAGE=437
CONFIG_FAT_DEFAULT_IOCHARSET="iso8859-1"
# end of DOS/FAT/NT Filesystems

#
# Pseudo filesystems
#
CONFIG_PROC_FS=y
CONFIG_PROC_KCORE=y
CONFIG_PROC_SYSCTL=y
CONFIG_PROC_PAGE_MONITOR=y
CONFIG_PROC_PID_ARCH_STATUS=y
CONFIG_KERNFS=y
CONFIG_SYSFS=y
CONFIG_TMPFS=y
CONFIG_TMPFS_POSIX_ACL=y
CONFIG_TMPFS_XATTR=y
CONFIG_HUGETLBFS=y
CONFIG_HUGETLB_PAGE=y
CONFIG_MEMFD_CREATE=y
# end of Pseudo filesystems

CONFIG_MISC_FILESYSTEMS=y
CONFIG_SQUASHFS=y
CONFIG_SQUASHFS_FILE_CACHE=y
CONFIG_SQUASHFS_DECOMP_SINGLE=y
CONFIG_SQUASHFS_XATTR=y
CONFIG_SQUASHFS_ZLIB=y
CONFIG_SQUASHFS_LZ4=y
CONFIG_SQUASHFS_XZ=y
CONFIG_SQUASHFS_4K_DEVBLK_SIZE=y
CONFIG_SQUASHFS_EMBEDDED=y
CONFIG_SQUASHFS_FRAGMENT_CACHE_SIZE=3
CONFIG_NETWORK_FILESYSTEMS=y
```

```
CONFIG_NFS_FS=y
CONFIG_NFS_V2=y
CONFIG_NFS_V3=y
CONFIG_NFS_V3_ACL=y
CONFIG_NFS_V4=y
CONFIG_ROOT_NFS=y
CONFIG_NFS_USE_KERNEL_DNS=y
CONFIG_GRACE_PERIOD=y
CONFIG_LOCKD=y
CONFIG_LOCKD_V4=y
CONFIG_NFS_ACL_SUPPORT=y
CONFIG_NFS_COMMON=y
CONFIG_SUNRPC=y
CONFIG_SUNRPC_GSS=y
CONFIG_SUNRPC_XPRT_RDMA=y
CONFIG_NLS=y
CONFIG_NLS_DEFAULT="utf8"
CONFIG_NLS_CODEPAGE_437=y
CONFIG_NLS_ASCII=y
CONFIG_NLS_ISO8859_1=y
CONFIG_NLS_UTF8=y
# end of File systems

#
# Security options
#
CONFIG_KEYS=y
CONFIG_SECURITY=y
CONFIG_SECURITY_NETWORK=y
CONFIG_HAVE_HARDENED_USERCOPY_ALLOCATOR=y
CONFIG_INTEGRITY=y
CONFIG_DEFAULT_SECURITY_DAC=y
CONFIG_LSM="lockdown,yama,loadpin,safesetid,integrity"

#
# Memory initialization
```

```
#
CONFIG_INIT_STACK_NONE=y
# end of Memory initialization
# end of Kernel hardening options
# end of Security options

CONFIG_CRYPTO=y

#
# Crypto core or helper
#
CONFIG_CRYPTO_ALGAPI=y
CONFIG_CRYPTO_ALGAPI2=y
CONFIG_CRYPTO_AEAD=y
CONFIG_CRYPTO_AEAD2=y
CONFIG_CRYPTO_BLKCIPHER=y
CONFIG_CRYPTO_BLKCIPHER2=y
CONFIG_CRYPTO_HASH=y
CONFIG_CRYPTO_HASH2=y
CONFIG_CRYPTO_RNG=y
CONFIG_CRYPTO_RNG2=y
CONFIG_CRYPTO_RNG_DEFAULT=y
CONFIG_CRYPTO_AKCIPHER2=y
CONFIG_CRYPTO_KPP2=y
CONFIG_CRYPTO_ACOMP2=y
CONFIG_CRYPTO_MANAGER=y
CONFIG_CRYPTO_MANAGER2=y
CONFIG_CRYPTO_MANAGER_DISABLE_TESTS=y
CONFIG_CRYPTO_NULL=y
CONFIG_CRYPTO_NULL2=y
CONFIG_CRYPTO_AUTHENC=y
CONFIG_CRYPTO_ENGINE=m

#
# Public-key cryptography
#
```

```
1368
1369 #
1370 # Authenticated Encryption with Associated Data
1371 #
1372 CONFIG_CRYPTO_CCM=y
1373 CONFIG_CRYPTO_SEQIV=y
1374 CONFIG_CRYPTO_ECHAINIV=y
1375
1376 #
1377 # Block modes
1378 #
1379 CONFIG_CRYPTO_CBC=y
1380 CONFIG_CRYPTO_CTR=y
1381
1382 #
1383 # Hash modes
1384 #
1385 CONFIG_CRYPTO_HMAC=y
1386
1387 #
1388 # Digest
1389 #
1390 CONFIG_CRYPTO_CRC32C=y
1391 CONFIG_CRYPTO_MD5=y
1392 CONFIG_CRYPTO_SHA1=y
1393 CONFIG_CRYPTO_LIB_SHA256=y
1394 CONFIG_CRYPTO_SHA256=y
1395
1396 #
1397 # Ciphers
1398 #
1399 CONFIG_CRYPTO_LIB_AES=y
1400 CONFIG_CRYPTO_AES=y
1401 CONFIG_CRYPTO_LIB_ARC4=y
1402 CONFIG_CRYPTO_ARC4=y
1403 CONFIG_CRYPTO_LIB_DES=y
```

```
CONFIG_CRYPTO_DES=y

#
# Compression
#

#
# Random Number Generation
#
CONFIG_CRYPTO_DRBG_MENU=y
CONFIG_CRYPTO_DRBG_HMAC=y
CONFIG_CRYPTO_DRBG=y
CONFIG_CRYPTO_JITTERENTROPY=y
CONFIG_CRYPTO_HW=y
CONFIG_CRYPTO_DEV_VIRTIO=m

#
# Library routines
#
CONFIG_BITREVERSE=y
CONFIG_GENERIC_STRNCPY_FROM_USER=y
CONFIG_GENERIC_STRNLEN_USER=y
CONFIG_GENERIC_NET_UTILS=y
CONFIG_GENERIC_FIND_FIRST_BIT=y
CONFIG_RATIONAL=y
CONFIG_GENERIC_PCI_IOMAP=y
CONFIG_GENERIC_IOMAP=y
CONFIG_ARCH_HAS_FAST_MULTIPLIER=y
CONFIG_CRC_CCITT=y
CONFIG_CRC16=y
CONFIG_CRC32=y
CONFIG_CRC32_SLICEBY8=y
CONFIG_ZLIB_INFLATE=y
CONFIG_ZLIB_DEFLATE=y
CONFIG_LZO_DECOMPRESS=y
CONFIG_LZ4_DECOMPRESS=y
```

```
CONFIG_XZ_DEC=y
CONFIG_XZ_DEC_X86=y
CONFIG_XZ_DEC_POWERPC=y
CONFIG_XZ_DEC_IA64=y
CONFIG_XZ_DEC_ARM=y
CONFIG_XZ_DEC_ARMTHUMB=y
CONFIG_XZ_DEC_SPARC=y
CONFIG_XZ_DEC_BCJ=y
CONFIG_DECOMPRESS_GZIP=y
CONFIG_DECOMPRESS_BZIP2=y
CONFIG_DECOMPRESS_LZMA=y
CONFIG_DECOMPRESS_XZ=y
CONFIG_DECOMPRESS_LZO=y
CONFIG_DECOMPRESS_LZ4=y
CONFIG_GENERIC_ALLOCATOR=y
CONFIG_INTERVAL_TREE=y
CONFIG_ASSOCIATIVE_ARRAY=y
CONFIG_HAS_IOMEM=y
CONFIG_HAS_IOPORT_MAP=y
CONFIG_HAS_DMA=y
CONFIG_NEED_SG_DMA_LENGTH=y
CONFIG_SGL_ALLOC=y
CONFIG_CHECK_SIGNATURE=y
CONFIG_DQL=y
CONFIG_GLOB=y
CONFIG_NLATTR=y
CONFIG_IRQ_POLL=y
CONFIG_DIMLIB=y
CONFIG_OID_REGISTRY=y
CONFIG_HAVE_GENERIC_VDSO=y
CONFIG_GENERIC_GETTIMEOFDAY=y
CONFIG_GENERIC_VDSO_32=y
CONFIG_FONT_SUPPORT=y
CONFIG_FONT_8x8=y
CONFIG_FONT_8x16=y
CONFIG_SG_POOL=y
```

```
CONFIG_ARCH_STACKWALK=y
CONFIG_SBITMAP=y
# end of Library routines

#
# printk and dmesg options
#
CONFIG_PRINTK_TIME=y
CONFIG_CONSOLE_LOGLEVEL_DEFAULT=7
CONFIG_CONSOLE_LOGLEVEL_QUIET=4
CONFIG_MESSAGE_LOGLEVEL_DEFAULT=4
# end of printk and dmesg options

#
# Compile-time checks and compiler options
#
CONFIG_ENABLE_MUST_CHECK=y
CONFIG_FRAME_WARN=2048
CONFIG_DEBUG_FS=y
CONFIG_OPTIMIZE_INLINING=y
CONFIG_SECTION_MISMATCH_WARN_ONLY=y
CONFIG_FRAME_POINTER=y
# end of Compile-time checks and compiler options

CONFIG_MAGIC_SYSRQ=y
CONFIG_MAGIC_SYSRQ_DEFAULT_ENABLE=0x1
CONFIG_MAGIC_SYSRQ_SERIAL=y
CONFIG_DEBUG_KERNEL=y
CONFIG_DEBUG_MISC=y

#
# Memory Debugging
#
CONFIG_HAVE_DEBUG_KMEMLEAK=y
CONFIG_DEBUG_STACK_USAGE=y
CONFIG_ARCH_HAS_DEBUG_VIRTUAL=y
```

```
CONFIG_DEBUG_MEMORY_INIT=y
CONFIG_HAVE_DEBUG_STACKOVERFLOW=y
CONFIG_DEBUG_STACKOVERFLOW=y
CONFIG_CC_HAS_KASAN_GENERIC=y
CONFIG_KASAN_STACK=1
# end of Memory Debugging

CONFIG_CC_HAS_SANCOV_TRACE_PC=y

CONFIG_PANIC_ON_OOPS_VALUE=0
CONFIG_PANIC_TIMEOUT=0
CONFIG_SCHED_INFO=y
CONFIG_SCHEDSTATS=y

#
# Lock Debugging (spinlocks, mutexes, etc...)
#
CONFIG_LOCK_DEBUGGING_SUPPORT=y
# end of Lock Debugging (spinlocks, mutexes, etc...)

CONFIG_STACKTRACE=y
CONFIG_DEBUG_BUGVERBOSE=y

CONFIG_USER_STACKTRACE_SUPPORT=y
CONFIG_HAVE_FUNCTION_TRACER=y
CONFIG_HAVE_FUNCTION_GRAPH_TRACER=y
CONFIG_HAVE_DYNAMIC_FTRACE=y
CONFIG_HAVE_DYNAMIC_FTRACE_WITH_REGS=y
CONFIG_HAVE_FTRACE_MCOUNT_RECORD=y
CONFIG_HAVE_SYSCALL_TRACEPOINTS=y
CONFIG_HAVE_C_RECORDMCOUNT=y
CONFIG_TRACE_CLOCK=y
CONFIG_RING_BUFFER=y
CONFIG_RING_BUFFER_ALLOW_SWAP=y
CONFIG_TRACING_SUPPORT=y
CONFIG_PROVIDE_OHCI1394_DMA_INIT=y
```

```
CONFIG_RUNTIME_TESTING_MENU=y
CONFIG_HAVE_ARCH_KGDB=y
CONFIG_ARCH_HAS_UBSAN_SANITIZE_ALL=y
CONFIG_UBSAN_ALIGNMENT=y
CONFIG_ARCH_HAS_DEVMEM_IS_ALLOWED=y
CONFIG_TRACE_IRQFLAGS_SUPPORT=y
CONFIG_EARLY_PRINTK_USB=y
CONFIG_X86_VERBOSE_BOOTUP=y
CONFIG_EARLY_PRINTK=y
CONFIG_EARLY_PRINTK_DBGP=y
CONFIG_DOUBLEFAULT=y
CONFIG_HAVE_MMIOTRACE_SUPPORT=y
CONFIG_IO_DELAY_0X80=y
CONFIG_DEBUG_BOOT_PARAMS=y
CONFIG_X86_DEBUG_FPU=y
CONFIG_UNWINDER_FRAME_POINTER=y
# end of Kernel hacking
```

Listing A.6: Kernel Build Config

Appendix B

Appendix B: Auditor Code

B.1 Lipton and Snyder and Elkaduwe et al.: Condition 1/Theorem 1

Essentially, Condition 1 of Lipton and Snyder, [91], and Theorem 1 of Elkaduwe et al., [44], are the same. Lipton and Snyder's Condition 1 states that a node, p, and a node, q, are connected in graph $G = (V, E)$, i.e. a path exists between p and q independent of directionality. Elkaduwe et al. state that if two existing entities are not connected, then they will never be able to leak authority to one another. While the former focuses on establishing the existence of a connection, the later states that the absence of a connection means leakage cannot occur. Listing B.1 shows the implementation of Condition 1/Theorem 1.

```
1 def condition1(graph, p, q):
2     if not has_path(graph.to_undirected(), p, q):
3         print('Failed: Condition 1.')
4         return False
5 
6     return True
```

Listing B.1: Condition 1/Theorem 1

B.2 Lipton and Snyder: Condition 2

Condition 2 states that there exists a node, x, in graph, $G = (V, E)$, with some arc to q. A problem is manifested in the following example pertaining to vCDS: Given a node, *Low*, with W (write) authority to a node, *High*, and a node, *Guard* where there exists a path between *High* and *Guard* with RW (read/write) authority, then *Low* is able to *take* the ability to RW to the *Guard* from the *High* component. This results in *Low* being able to read from a high component, the *Guard*. This is why vCDS does not allow the use of the take rule. Listing B.2 shows the implementation of Condition 2.

```
def lipton_cond2(graph, p, q, access):
    in_edges = list(graph.in_edges(q)) # if in_edges == [], q has no incoming
    edges

    for i in range(0, len(in_edges)):
        a = graph.get_edge_data(in_edges[i][0], q)[i]['label']

        if access in graph.get_edge_data(in_edges[i][0], q)[i]['label'].lower():
            return True

    return False
```

Listing B.2: Condition 2

B.3 Elkaduwe et al.: Theorem 2

Theorem 2, presented by Elkaduwe et al., [44], states: if there exists a nonempty set of outgoing edges from a node, and there is a labeled arc along an edge, then the authority given by the label along that arc will not be exceeded in any future graph state. Listing B.3 shows the implementation of Theorem 2.

```
def elkaduwe_theorem2(graph, p, q, access):
    for i in range(0, graph.number_of_edges(p, q)):
        if list(graph.out_edges(p)) != [] and \
           graph.get_edge_data(p, q) and \
           access in graph.get_edge_data(p, q)[i]['label'].lower():
```

```
6             return True
7
8     return False
```

Listing B.3: Theorem 2

Appendix C

Appendix C: Towards Trustworthy Cross-Domain Technologies

Below is an article which I have prepared for SIGNAL magazine in response to an article written by Shaun Waterman on August 1, 2021: *Assured Cross-Domain Access Through Hardware-Based Security: Sponsored Content* [129].

C.1 The Problem of Cross Domain Technology

The DoD is continually battling for faster and more secure CDS technology and recent cloud-based security operations have reinforced the need for reliable and secure cross domain solutions (CDS). Furthermore, the NSA expressly desires more rigorously designed CDS solutions (NSA, 2021). These facts are common knowledge in spaces where immediate, and often remote, access to highly classified information is required. In this article, we appeal to the best interests of the DoD by claiming that any system composed of hardware and/or software which has not been formally verified should not claim to provide high assurance. In other words, any system that is not trustworthy cannot and should not be relied upon. All security components are required to be relied upon, yet the status quo security systems and technologies often do not provide formal verification proofs. The gap between reliability and reality is marked by the difference between trust and trustworthiness, which we describe and clarify below. Additionally in this article, we address the pious claims of previous work regarding hardware and software CDS technology. Our purpose

is to show the necessity of trustworthy CDS technology including the long-term time-to-market benefits, while fulfilling the best interests of the DoD.

C.1.1 Status Quo

The status quo in CDS technology can be described as being ad hoc solutions which lack provable assurance (U.S. Army, 2018; Farroha et al, 2009). CDSs are generally physically isolated solutions, not lending themselves to use in offsite computing environments which are commonplace (Looking Glass, 2019). When it comes to evaluation, vendors are limited by government requirements and processes (Daughety, 2021). Specifically, usability, i.e. the service provided by the system, is ignored when it should be the central focus; the paperwork is evaluated rather than the product; and the evaluation system squeezes "a very volatile and competitive industry into a bureaucratic straightjacket, in order to provide purchasers with the illusion of stability" (Anderson, 2008).

C.1.2 Definitions/Background

In order to better understand the goal of this article, we will first review the differences between trust and trustworthiness in the context of CDSs. A trusted CDS is a solution that is assumed to be reliable and effective during a security event. On the other hand, a trustworthy CDS is a solution which has been formally and mathematically proven reliable and effective during a security event. Therefore, trustworthiness requires comprehensive formal verification. In short, "formal verification is the use of mathematical techniques to ensure that a design conforms to some precisely expressed notion of functional correctness" (Sanghavi, 2010). In other words, a formally verified software program is one that is proven not to violate the specification for which the software was intended to fulfill, i.e. the software is functionally correct and free from vulnerabilities. Furthermore, software and hardware can be formally verified to enforce security properties of confidentiality, integrity, and availability for example. One such evaluation system is the Common Criteria's (CC) Evaluation Assurance Level (EAL). The EAL is a grade assigned to a hardware or software system based on a security evaluation. The level ranges from EAL1, "functionally tested", to EAL7, "formally verified design and tested" (Common Criteria, 2009). Knowing the definitions of trust and trustworthiness, we present the following claim: any system less than EAL6 should not be trusted. More specifically, no system evaluated below EAL6 should be trusted, and no system below EAL7 is trustworthy. It

should additionally be noted that, due to the faults associated with the CC's evaluation system, just because a system is evaluated to EAL7 does not necessarily mean it is in fact trustworthy. We will discuss ways to improve the reliability of the evaluation later in this article.

The importance of evaluation is clear when functionality and security is seen through the lens of trust and trustworthiness. However, as we previously addressed, the evaluation system itself leaves much to be desired. Contrary to the concept of security through obscurity, we have Shannon's Maxim (Shannon, 1949). Shannon's Maxim is closely related to Kerckoff's principle of cryptography, and, in the context of a trusted computing base (TCB), can be formulated as follows: a system should remain secure even when the enemy has access to the source code. After all, if a system is completely secure, an attacker would not be able to find a vulnerability if he/she were provided with the source code. Given that, in many real-world cases, the security and tactics of a use-case instance should be closely guarded, the extent to which Shannon's Maxim should apply is to the TCB, i.e. the smallest set of services necessary to provide the security properties of a CDS. We further discuss this point below.

C.2 Misconception 1: Software-based security CDSs are inadequate while hardware-based CDSs are more secure

Vendors have claimed that software-based security CDSs are inadequate while arguing that hardware-based CDSs are more secure. However, all-in approaches, i.e. either hardware-only or software-only approaches, to CDS systems are cause for ad hoc, high risk solutions (B. M. Thomas and N. L. Ziring, 2014; R. J. Anderson, 2008). In reality, security must be architected into the hardware and software in order to build secure, trustworthy CDS systems. As established, the only way a system can be proven secure and trustworthy is through comprehensive formal verification of functional correctness and of security. In contrast, vendors often do not provide any mathematical proofs or mention of formal verification for their systems. This includes vendors who produce hardware-only CDS solutions and claim that hardware is secure, once again, without providing mathematical proofs. Furthermore, claims which have been made, specifically about the vulnerability of all software and that software security cannot be guaranteed, assume that some subset of all software, current and future, will never be comprehensively formally verified for functional correctness or se-

curity guarantees. This is obviously incorrect as the following sources present such solutions which are additionally available for independent verification: (Daughety, 2021; Heiser, 2020; Klein, 2014).

C.3 Misconception 2: Formal verification is too hard and expensive

One of the arguments against formal verification is that the complexity of software makes it difficult and expensive to formally verify security-critical solutions. While this reasoning is sound, a formally verified software-based CDS exists, and, once again, is available for independent verification (Daughety, 2021). Furthermore, the seL4 microkernel is a comprehensively formally verified (for functional correctness and security guarantees, i.e. confidentiality, integrity, and availability) microkernel and hypervisor which is also available for independent formal verification. We will further discuss the need for independent formal verification shortly.

The DoD is aware that the pace of operations and high demand of cross domain technology pressures engineers into high risk, ad hoc solutions (B. M. Thomas and N. L. Ziring, 2014; R. J. Anderson, 2008). Furthermore, the time to market is at odds with the goal of achieving the highest level of security using formal verification, i.e. time to market is not aligned with the DoD's best interest. Because the formal verification process is difficult and expensive, these factors should be considered when developing systems which process and transmit classified information, considering the devastating impact of such information being leaked.

To counter the cost and difficulty of the verification process, the experts, (Heiser, 2020; Klein, 2018), see the value in pressing on and winning the battle for proving the correctness of security and functionality of software. Furthermore, once a system has been formally verified, reusing formally verified building blocks (verify once, reuse many) would mitigate both cost and time to market while making security verification faster. To this point, there exists a CDS auditing tool which verifies the security configuration of vCDS, a CDS built upon a formally verified TCB (Daughety, 2021).

C.4 Misconception 3: It is sufficient for vendors to self-promote the security of their own products

Vendors often self-promote the security of their own products without undergoing rigorous independent verification. These overreaching claims are not the only issue, however. Recall from the review of the status quo that the evaluation system and processes are also flawed; in order for a system to be evaluated, vendors can effectively "game" the evaluation system by exploiting vulnerabilities in the evaluation requirements and processes.

Independent verification means that the proofs are publicly available for confirmation such that any entity can verify the security of the system. As discussed, the EAL alone does not necessarily prove trustworthiness, however, the independent evaluation of a system can verify a system to be trustworthy. The point of contention here is that, because the DoD has not historically been proficient at keeping secrets, they now have few artifacts openly available for independent evaluation.

A better way to combat the bias of self-promotion and lack of independent verification is through the adherence to Shannon's Maxim. Specifically, the TCB should be available for independent verification while the tactics of a particular use-case instance, including data and computations, should remain closed. Transparency should be embraced about artifact security and limitations so that decision makers can calculate better risk analysis.

C.5 Time for Disruption

Due to latest advances in formally verified TCBs, a cyber operations paradigm shift has begun and vCDS is leading the way as a step in the right direction (Daughety, 2021). vCDS leverages the latest advances in trustworthy systems while remaining open to independent, formal verification. Furthermore, all risks and limitations of the system are known so that better risk analysis may be calculated. The need is clear – for any CDS to be relied upon, it must be comprehensively proven trustworthy. Furthermore, for any CDS to be provably secure, it must be available for independent verification. vCDS checks both boxes and serves as proof to counter previous claims that software-based security CDSs are inadequate while hardware-based CDSs are more secure. Additionally,

vCDS shows that the common belief that formal verification is too hard and expensive is simply an excuse resulting in high risk, ad hoc solutions (U.S. Army, 2018; Farroha et al, 2009).

Appendix D

Appendix D: Correspondence

D.1 seL4 Developer Correspondence

The following is an email to the seL4 developers with specific questions I had relating to Dataports as they pertain to access rights and data diode implementations. Matthew Fernandez confirmed my suspicions and answered my questions.

D.1.1 Inquiry to seL4 Developers

To whom it may concern,

I am learning about CAmkES Dataports and potentially using it as a data diode to allow one component to write to shared memory, and a second component to read from the shared memory.

Question 1: I noticed that in aeroplage.camkes, the KeyboardDriver component has the dataport char_out while the Switch has char_in. Char_in is given read-only access, but, based on the CAmkES manual, char_out would have rwx privileges (if I understand correctly). So, how is the dataport with rwx (keyboarddriver) and r (switch) a data diode when the keyboarddriver has read, write, and execute privileges (the data can be read back by the keyboarddriver?)?

It seems to me that giving the switch read-only access does make the connection "act" as a data diode but it is not a strict data diode? The CAmkES manual (ref below) does not explicitly say

that write only is allowed but does suggest it can be done. The Intel manual (ref below, section 5.11) says that pages are either read-only or read/write-only (no write-only). In fact, the page setting for access is only one bit.

Question 2: Is write-only possible with dataports and what would *_access = "W" do in CAmkES? Could the "data diode" be affected by the hardware read/write if the memory is write-only?

Currently, I have not thought of a case where the read/write access of the keyboarddriver would compromise the function of a data diode if the switch is read-only.

Thanks, Nathan

References:
https://github.com/seL4/camkes/blob/master/apps/aeroplage/aeroplage.camkes
https://docs.sel4.systems/projects/camkes/manual.html Port Privileges Section
https://software.intel.com/sites/default/files/managed/7c/f1/253668-sdm-vol-3a.pdf Section 5.11

D.1.2 Response from Matthew Fernandez

Hi Nathan,

I can give some historical context about this, but you're probably better off asking the current maintainers on the seL4 mailing list.

For question 1, yes, I think you're correct that the char_out port has RWX access. When we were implementing (or rather re-implementing) CAmkES, I made the dataport permissions map directly to the seL4 page mapping flags. It didn't occur to me until later that "write-only" has no equivalent in the paging models of most hardware. I then started to wonder what was actually going on. Looking into the kernel source I found seL4 was silently downgrading these to kernel-only mappings.

This isn't a bug, but it is unexpected to CAmkES users. We had some discussion about how to resolve this and ended up working around this in the loader [0]. This resulted in the least surprise to users, but it is still not quite intuitive.

[0]: https://github.com/seL4/capdl/blob/master/capdl-loader-app/src/main.c#L1457-L1463

For question 2, the answer for most practical purposes is no. I'm not aware of any standard hardware that accepts write-only mappings. You sort of have this with some types of I/O memory, but your compiler needs to be explicitly aware of this (i.e. volatile pointers). To achieve true write-only, you could have an arbitrator process that unmaps memory from the writer once it's done and then maps it into the reader. But then you obviously have the overheads of coordination and remapping operations.

As you've said, in this single-writer case, giving the writer R as well does not seem to convey any extra abilities.

Thanks, Matt