2022 IEEE International Conference on Cyber Security and Resilience (CSR) | 978-1-6654-9952-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/CSR54599.2022.9850321

Auditing a Software-Defined Cross Domain Solution Architecture

Nathan Daughety*, Marcus Pendleton*, Rebeca Perez*, Shouhuai Xu†, John Franco‡
*90thCOS, United States Air Force
{nathan.daughety, marcus.pendleton.2, rebeca.perez}@us.af.mil
†University of Colorado Colorado Springs
sxu@uccs.edu
‡University of Cincinnati
franco@ucmail.uc.edu

***Abstract*—In the context of cybersecurity systems, trust is the firm belief that a system will behave as expected. Trustworthiness is the proven property of a system that is worthy of trust. Therefore, trust is ephemeral, i.e. trust can be broken; trustworthiness is perpetual, i.e. trustworthiness is verified and cannot be broken. The gap between these two concepts is one which is, alarmingly, often overlooked. In fact, the pressure to meet with the pace of operations for mission critical cross domain solution (CDS) development has resulted in a status quo of high-risk, ad hoc solutions. Trustworthiness, proven through formal verification, should be an essential property in any hardware and/or software security system. We have shown, in "vCDS: A Virtualized Cross Domain Solution Architecture", that developing a formally verified CDS is possible. *virtual CDS* (vCDS) additionally comes with security guarantees, i.e. confidentiality, integrity, and availability, through the use of a formally verified trusted computing base (TCB). In order for a system, defined by an architecture description language (ADL), to be considered trustworthy, the implemented security configuration, i.e. access control and data protection models, must be verified correct. In this paper we present the first and only security auditing tool which seeks to verify the security configuration of a CDS architecture defined through ADL description. This tool is useful in mitigating the risk of existing solutions by ensuring proper security enforcement. Furthermore, when coupled with the agile nature of vCDS, this tool significantly increases the pace of system delivery.**

***Index Terms*—Cross Domain Solution, Architecture Description Language, Trustworthiness, Configuration Security, Data Protection, Access Control, Trusted Systems, Security Analysis**

I. Introduction

In the context of security controls, a *leak* refers to the acquisition of authority over an object by a subject that did not previously have that authority. Authority refers to the privileges and/or access rights held by a subject over an object. 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 [1].

vCDS is the first 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) [2, 7]. In one instantiation of vCDS's trustworthy architecture, the seL4 microkernel/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 [17]. 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 [2].

Problem Statement. Building a system which leverages a formally verified TCB does not 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 [15].

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.

Our Contributions. In this paper, (i) we address the need to verify the implementation of a CDS system security model. (ii) We present a tool which, to the best of our knowledge, 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, to the best of our knowledge, (iii) we are the first to develop a tool to audit 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 our algorithm to check the constraints. (v) We extend the use of the ADL 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) we extend it to provide system security modeling that is far beyond the status quo.

II. Related Work

Harrison et al., in [5], present a model of protection mechanisms to determine whether or not a subject in a given state can acquire authority over an object. Additionally, they proved that safety is undecidable in general. In this paper, safety and decidability is further discussed in Section III-B. Several other protection models, such as take-grant, described in this work, have been designed and proven to be decidable in more restrictive cases [12, 13, 14, 16]. However, due to the nature of vCDS, seL4, and CAmkES which are being analyzed, this work is most closely related to the work in [1, 2].

III. Background

A. Security Models

1) 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. 1, the capability is implemented as a data structure which encapsulates an object reference and the rights conveyed to that object [11].

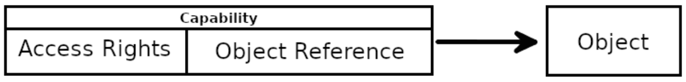

Fig. 1. A capability is an immutable object reference which encapsulates an object reference and the access rights conveyed to the object

A capability is an immutable reference that enforces the principle of least privilege (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 [7]. "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" [11]. 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" [7].

2) 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 subject or object, 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 [2, 12], 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.

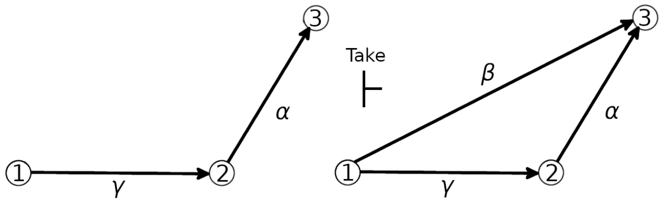

Fig. 2. Take Rule

(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* the ability to execute β operations on n_3 to n_2.

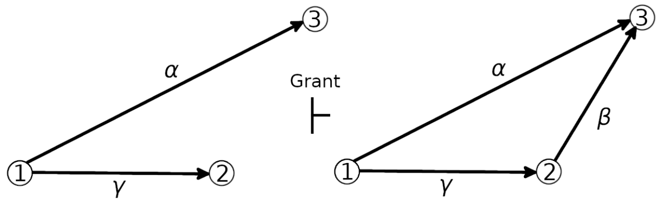

Fig. 3. Grant Rule

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

(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

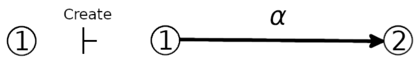

Fig. 4. Create Rule

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 α operations on n_2.

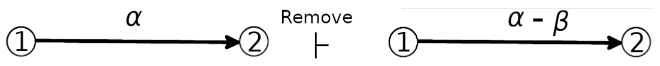

Fig. 5. Remove Rule

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.

B. Safety Model Decidability

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' [10]. In other words, if $\alpha \subseteq R$ cannot be leaked in system S, S is considered safe.

C. Analyzing Security Configurations

Now we review the conclusions of Lipton and Snyder [12] and Elkaduwe et al. [2], which 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. 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.

1) Take-Grant Decidability: Presented below are the results of the work in [12] 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 [12]. 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 [12]: 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 [12].*

Condition 2. *There exists a node, x in G and an arc from x to q, with label $\beta \subseteq \alpha$ [12]. 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 $\alpha \subseteq R$. In other words, 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 posses.

IV. Overview of vCDS Security Architecture

Daughety et al. [1] present a layered vCDS architecture which includes the optional hardware protections, the software computing base, and the components. At a high level, vCDS is a baseline system which performs CDS capabilities while providing the following: (i) trustworthy and proven reliable execution, (ii) remote deployability, (iii) accessibility to the commercial sector, and (iv) versatile applicability to a variety of use-cases and environments.

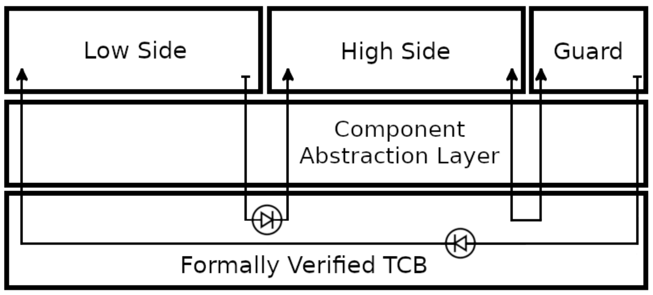

Fig. 6. vCDS stream processor architecture

Fig. 6 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 [1]: (i) Low Side, (ii) High Side, and (iii) Guard. The Low Side corresponds to a lower label and functions to retrieve the data and, if the use-case requires, calculate an integrity tag before writing the data to the High Side. The High Side corresponds to a higher label 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 [4].

A. vCDS Security Model

The vCDS TCB leverages the previously described capability model. At kernel boot time, all physical memory resources, such as *untyped* kernel objects, are allocated [3]. Contiguous memory blocks hold untyped memory objects, some of which are writable by the kernel [17]. 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" [7, 8]. The employed security protection model is essentially the take-grant protection model with the following modifications as detailed in [2]:

(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 Section III-A2, 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.

B. Access Control Model Decidability

Elkaduwe et al. [2] show that the seL4 access control model described in Section IV-A is decidable. 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 [2].

Methodology. The goal of the proofs presented by Elkaduwe et al. [2] 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 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 [2].

A *sane* state as it pertains to the introduction of new vertices, per Elkaduwe et al., 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 [2]. 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 ([2])**.** *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, i.e. $\alpha \subseteq R$, 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 [2]. 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 [2])**.** *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.*

Theorem 2, also referred to as the *Isolation Theorem*, is introduced to correct the limitations of Theorem 1. The Isolation Theorem 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 [2]. 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. Therefore, 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" [2].

C. 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 processes encapsulated by the microkernel, which, in the case of vCDS, represent the domains and filters; 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" [9].

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 labels [6]. One advantage of using CAmkES is the ability to implement 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 Section V.

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 our priority, i.e. preventing data leakage, is upheld and no leaks can occur over the link in the opposing direction.

D. vCDS Stream Processor Audit

Now we demonstrate the impact of the audit tool, described in Section V, 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, described in [1], implements four components for the stream processor: 1) Low Side, 2) High Side, 3) Guard, and 4) High Side Management Network.

Lst. 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. The `LowSide` keyword specifies a component which resides in a lower labeled space whereas the `HighSide` keyword specifies a component residing in a higher labeled space. Lst. 1 also shows the pseudocode for a connection between components. Fig. 7 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
  );

*/
```

Lst. 1. vCDS CAmkES

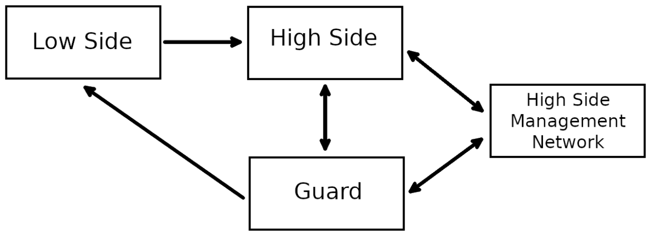

Fig. 7. vCDS stream processor security configuration

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-labeled 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 description of the security configuration, visualized in Fig. 7, is accurate to the goal of data confidentiality. The results of the audit, excluding the High Side Management Network rights, are shown in Fig. 8. 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 [1].

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

Fig. 8. vCDS Security Configuration Audit Results

V. Security Control Audit Tool

In this section, we present our methodology and the implementation of an analysis tool which seeks to verify the correctness of the security configuration of vCDS [1].

A. Application of the Isolation Theorem to capDL

As described in Section IV-C, capDL defines a system's configuration of kernel objects and capabilities, which notably includes access rights. The Isolation Theorem [2], 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.

B. 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 Fig. 9 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 sending it as input to the Collection phase. The Collection phase combines a set of functions which calculate both the intended solution as defined by the specification in the capDL and all possible propagations from each of the connections. The Audit phase follows the Collection phase where the 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 provides an alert which highlights the security control implementation error.

capDL Parser. The parser module contains the `parseCapDL` function which reads the vCDS specification from capDL and filters out unnecessary information. It then retrieves all information that is pertinent to the algorithm: the components, the connectors, the connector types, and the access rights for each of the connectors. The output of the parser is the system

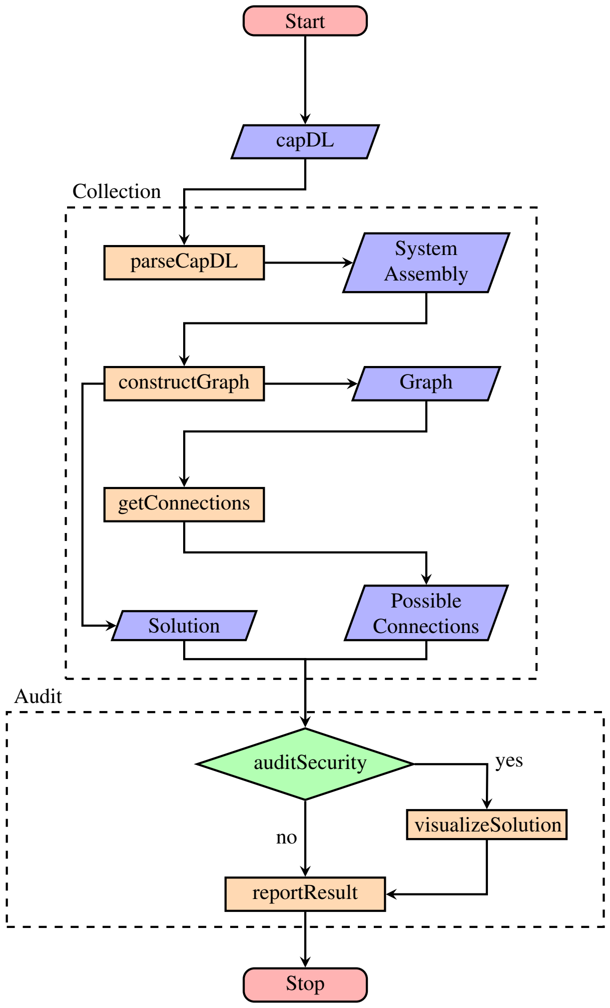

Fig. 9. Audit Tool Pipeline

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 calls the `constructGraph` function to create a graph of the system based on the provided assembly. The generated system graph is a directed graph which models the system controls for leakage prevention from higher to lower labels. The directed graph consists of: (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 the algorithm and is implemented such that the Isolation Theorem, given by [2], is deemed true. Additionally, implementations of the conditions given by [12] were added to emphasize the danger of the *take* rule, described in Sections IV-A and VI, and to show the correctness of the verification mechanism. Recall that [12] 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 permits an authority propagation and breaks the security proofs of seL4. The correct implementation of the auditor relies on the Isolation Theorem, proven formally in [2].

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.

Algorithm 1: getConnections

```
Input: digraph $G = (V, E)$, where $V$ is the set of
       vertices and $E$ is the set of edges; $\lambda_G$ labeling
       function of E
Output: array M of dimension $|V|\,x\,|V|$
1 foreach $p \in V$ do
2     foreach $q \in V$ do
3         if $p \neq q$ and (p,q) in $E$ then
4             $\alpha$ = $\lambda_G$( (p,q) )
              /* Isolation Theorem holds,
                 i.e. p can $\alpha$ q          */
5             if hasAuthority(p, q, $\alpha$) then
6                 M[p,q] = $\alpha$
7             else
8                 M[p,q] = NIL
9 return M
```

Additionally, Algorithm 2 is developed to enforce the Isolation Theorem from [2]. 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.

Algorithm 2: hasAuthority

```
Input: p, q, $\alpha$
Output: Boolean
1 if $\exists$ the authority $\alpha$, from p to q then
2     return True
3 return False
```

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 audits the graph constructed from the capDL against the graph of all possible connections and authorities provided by the connection 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 constructs 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, i.e. a diagram of the components, their respective connections and authorities, much like what is presented in Fig. 7, is generated.

C. Analysis

The analysis of this tool begins with the analysis of the take-grant security model. Recall that Lipton and Snyder [12] 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 [2]. 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 a 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 [2, 12] and can therefore be *trusted* to be correct. Further examination to prove this tool *trustworthy* is discussed in Section VI.

Fig. 10 shows the worst case execution time (WCET) for each module 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	$O(n)$

Fig. 10. WCET of Audit Tool Components

VI. Discussion

In this section we discuss the limitations of the audit tool in addition to intended future efforts concerning this work.

Versatility. One additional property of the auditor is its versatility to audit systems other than vCDS [1]. With little to no modification, 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. The tool may also be used to audit additional CDS systems described via an ADL with minor modifications to the parser module.

Theorem Examination. In the context of a vCDS audit, the take operation permits rule propagations which would otherwise be prohibited. For example, when auditing the stream processor application, additional rules, relative to Fig. 8, are permitted. The results in Fig. 11 reflect an error in the implementation of authority because the Low Side can read/write to higher components. One might prematurely conclude that the data diode link between the Low Side and the High Side has failed. However, this is not the case because the data diode comes with formal proofs of correctness. These are the results of the take operation.

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

Fig. 11. vCDS Audit Results Permitting Take Rule

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 needs to be addressed in future work as discussed below.

Future Work. 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.

VII. Conclusion

In this paper, we have addressed the need for verifying the implementation of a CDS. We have reviewed the contributions of Elkaduwe et al. to the problem of decidable object security and how their conclusions provide an important function in this work. Additionally, we have presented an algorithm and 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, to the best of our knowledge, the only algorithm which seeks to verify the correctness of a CDS. The presented tool is also the first tool which audits a CDS described by an ADL. Furthermore, we present an ADL which we have 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, we have extended this tool to generate a system security model to improve the status quo in system security modelling. Our hope is that this work inspires the further development of provably secure and trustworthy security computing systems with verified security controls.

Acknowledgement. This work was supported in part by ARO Grant #W911NF-17-1-0566, NSF Grants #2122631 (#1814825) and #2115134, and Colorado State Bill 18-086.

References

[1] Nathan Daughety et al. "vCDS: A Virtualized Cross Domain Solution Architecture". In: *IEEE MILCOM* (2021).

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

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

[4] 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.

[5] Michael Harrison, Walter Ruzzo, and Jeffrey Ullman. "Protection in Operating Systems." In: *Communications of the ACM* (Aug. 1976).

[6] 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.

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

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

[9] Gerwin Klein et al. "Formally Verified Software in the Real World". In: *Commun. ACM* (Sept. 2018).

[10] Manuel Koch, Luigi Mancini, and Francesco Parisi Presicce. "Decidability of Safety in Graph-Based Models for Access Control". In: Oct. 2002.

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

[12] R. J. Lipton and L. Snyder. "A Linear Time Algorithm for Deciding Subject Security". In: *J. ACM* (July 1977).

[13] A. Lockman and N. Minsky. "Unidirectional Transport of Rights and Take–Grant Control". In: *IEEE Transactions on Software Engineering* (1982).

[14] Naftaly H. Minsky. "Selective and Locally Controlled Transport of Privileges". In: *ACM Trans. Program. Lang. Syst.* (Oct. 1984).

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

[16] R.S. Sandhu. "The typed access matrix model". In: *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. 1992.

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