8-2020

Maia and Admonita: Mandatory Integrity Control Language and Dynamic Trust Framework for Arbitrary Structured Data

Wassnaa Al-Mawee
Western Michigan University, wassnaa.almawee@gmail.com

Maia and Admonita: Mandatory Integrity Control Language and Dynamic Trust Framework for Arbitrary Structured Data

by

Wassnaa Al-Mawee

A dissertation submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
Computer Science
Western Michigan University
August 2020

Doctoral Committee:

Steven M. Carr, Ph.D., Chair
Zijiang James Yang, Ph.D.
Li Yang, Ph.D.
Jean Mayo, Ph.D.
Said Abubakr, Ph.D.

Maia and Admonita: Mandatory Integrity Control Language and Dynamic Trust Framework for Arbitrary Structured Data

Wassnaa Al-Mawee, Ph.D.

Western Michigan University, 2020

The expansion of attacks against information systems of companies that operate nuclear power stations and other energy facilities in the United States and other countries, are noticeable with potential catastrophic real-world implications. Data integrity is a fundamental component of information security. It refers to the accuracy and the trustworthiness of data or resources. Data integrity within information systems becomes an important factor of security protection as the data becomes more integrated and crucial to decision-making. The security threats brought by human errors whether, malicious or unintentional, such as viruses, hacking, and many other cybersecurity threats, are dangerous and require mandatory integrity protection. To date, Biba and Clark-Wilson are well-known general integrity models in computer systems but they impose a number of restrictions that make them impractical to implement. Additionally, permission-based solutions are one of the popular approaches in the literature but existing solutions are designed to address the trustworthiness of who accesses the data not the trustworthiness of the data itself. To solve these problems, we propose a generally applicable system to prevent and detect compromised data integrity. The proposed work consists of two major components: Maia -the definition of mandatory integrity control language for describing integrity constraints, and Admonita -the construction of recommendation-based trust model for trustworthy data.

The integrity of systems files is necessary for the secure functioning of an operating system. Often, file integrity is determined by who modifies the file or by a checksum. Even

if a file is modified by a subject with trust or has a valid checksum, it may not meet the specification of a valid file. An example would be a password file with no user assigned a user id of 0. Maia provides a means to specify what the contents of a valid file should be. In addition, Maia can be used to specify the format and valid properties of system configuration files, PNG files and others. In this dissertation, we give a structural operational semantics of Maia to evaluate and validate our approach, and generate a Maia-verifier that supports many features of the Maia language. Additionally, we quantify Maia's impact on performance, and demonstrate that we can provide robust integrity guarantees without sacrificing usability using an implementation that has not been optimized for performance. Within the context of the Linux password file, our implementation of Maia itself is quite fast, adding only extra milliseconds overhead to the time required to compile JAVA code, load and access a single file, and construct containers. In the worst case, total processing time is 4 seconds to verify and parse 15000-record password file, and less than 2 seconds to validate the input data. Within SSH configuration file context, in the worst case, Maia requires less than 2 seconds to load, validate, and parse 10000-record SSH configuration file, and around 170 milliseconds to validate the input data.

Data integrity is critical to the secure operation of a computer system. Applications need to know that the data that they access is trustworthy. In this work, we propose a recommendation-based trust model, called Admonita, for data integrity that is applicable to any structured data in a system and provides a measure of trust to applications on-the-fly. The proposed model is based on the Biba integrity model and utilizes the concept of an Integrity Verification Procedure (IVP) proposed by Clark-Wilson.

Admonita incorporates subjective logic to maintain the trustworthiness of data and applications in a system. To prevent critical applications from losing trust, Admonita also incorporates the principle of weak tranquility to ensure that highly trusted applications can

maintain their trust levels. We develop a simple algebra around these elements and describe how it can be used to calculate the trustworthiness of system entities. By applying subjective logic, we build a powerful, artificial and reasoning trust model for implementing data integrity.

In the future, we plan to implement Admonita in a real system and measure its performance. This will involve creating a high-performance compiler for Maia that utilizes its natural parallelism. The result will be a system that measures and maintains the trust levels for the applications and data contained within it.

ACKNOWLEDGEMENTS

Since the first step until this moment in my PhD journey, Professor Steve M. Carr has been weaving for me the path of success with his professional attitude and broad, deep knowledge. Professor Carr stands like a guide in midst of computer desert; without his encouragement, support and enthusiasm, I would be missing.

Meanwhile, I would like to express my appreciation to the dissertation committee members for their insightful comments and encouragement. They have provided me an extensive personal and professional guidance and taught me a great deal about both scientific research and life in general.

For his soul that spots me as my shadow, to his voice which tolls in my ears, to his smiley face which lights my eyes with hope, to my father. I want to tell him "I did it". To her heart, the pillow I sleep on every night, to her prayers that light the dim nights, to my mother. For my siblings who support me in all my pursuits. And most of all for my loving, supportive, encouraging, and patient husband Ihab, and my beautiful daughters Lena and Sophie, who provide unending inspiration; without you I would not be able to get it done. Thank you.

Wassnaa Al-Mawee

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Integrity of data within computer systems, along with the ongoing confidentiality and availability of said data, make up the three major components of computer security. While both confidentiality and availability are subjects of frequent and ongoing study, integrity is not generally discussed in terms of complete computer systems. Instead, integrity issues tend to be either tightly coupled to a particular domain (e.g. database constraints), or else are too broad. There are few, if any, approaches to integrity which are capable of actively protecting arbitrary structured data.

Our work seeks to provide robust tools to enable general-purpose integrity protection. As part of this, we present Maia, a language to describe integrity constraints for arbitrary files [1]. In Maia, file verification is accomplished over two phases that correspond first, to checking the file syntax, and second, to checking its semantics. The user provides an Extended Backus Naur Form (EBNF) grammar to specify the file structure and extract its syntactic elements into sets for processing. Then, the sets are checked against integrity constraints in the form of predicate logic.

In addition, we design a proof of concept for Maia by using Structural Operational Semantics (SOS) [2] and report on a preliminary implementation of a Maia compiler in the context of a mandatory integrity system [3]. The semantics give precise rules for giving meaning to a Maia specification. These rules show there is no ambiguity in Maia, giving assurance that a correct implementation of Maia file verifiers is possible. This allows one to implement a compiler or interpreter and use Maia to specify integrity constraints within an integrity system.

Moreover, Integrity often is maintained by restricting access to high integrity items to only subjects that have high integrity. However, as illustrated in the first paragraph, integrity is also a property of the data itself, not just of who accesses or modifies it. How can an application know how trustworthy the data it accesses is? In addition, if an application tries to access data that is not very trustworthy, should that application be allowed to access that data and, if so, does the access affect the future trustworthiness of both the data and subject?

We address these questions by using an improvement to the trust-enhanced data integrity model of Oleshchuk [4]. We present a recommendation-based trust model for dynamic data integrity, called Admonita. Admonita is based upon subjective logic [4]–[8] and the Biba integrity model [9]; however, it incorporates the idea of an Integrity Verification Procedure (IVP) from the Clark-Wilson model [10], the principle of tranquility [11], [12] that allows integrity levels to increase or decrease, and the notion that the data itself has a measure of integrity apart from who modifies it.

In Admonita, the trust level for subjects and objects is set by a trusted authority. Admonita then incorporates the opinion of an independent observer via an IVP implemented in Maia [1], [13]. Admonita maintains the trustworthiness of both subjects and objects in a computing system via the conjunctive, consensus and recommendation operators from subjective logic. Admonita adjusts the trustworthiness of entities dynamically based upon the trust levels of subjects and the objects they access, and includes bidirectional weak tranquility to allow the trust levels to increase or decrease.

In summary, the contribution of this dissertation is five-fold:

1. We provide an extensive discussion of related research, and present three major vulnerabilities of existing data integrity approaches.

2. We design and implement a mandatory integrity language Maia, for arbitrary structured data which provides an implementation independent descriptions of valid arbitrary structured data. The proposed language is able to:

    - Tackle data integrity vulnerabilities in the existing data integrity approaches.
    - Protect arbitrary structured data in information systems.
    - Detect any attempt of modification to arbitrary structured data in information systems.

3. We design a proof of concept for Maia that shows that it can be efficiently implemented using Structural Operational Semantic SOS.
4. We evaluate Maia performance with selected system's files.
5. We design Admonita: a recommendation trust model for dynamic data integrity using subjective logic. The proposed model is able to:

    - Calculate the trustworthiness of the system entities.
    - Provide dynamic data integrity.
    - Prevent data isolation from the system resources.

In the remainder of this document, we present tools for providing general, implementable data integrity protections. The next chapter presents some general background on computer security, discretionary and mandatory system, computer language design, subjective logic, and tranquility for dynamic data integrity, We follow this with a discussion of related research. Then, we present Maia, a language for describing integrity constraints on arbitrary files types followed by a proof of concept of Maia specification using Structural Operational Semantics (SOS). Next, we present a discussion of Maia implementation performance on

selected system files. In chapter 6, we present Admonita: a recommendation based-trust model for dynamic data integrity. We conclude the dissertation as a whole with discussion of future research areas derived from this work.

CHAPTER 2

BACKGROUND

This chapter is organized within five sections. The first section provides a general background on computer security aspects including confidentiality, availability, and integrity, followed by a discussion of the differences between mandatory and discretionary systems, and how theses systems are directly related to our work. The third section address how programming languages are designed using their syntax and semantics, and how variants of language semantics can be used in specifying the behavior of programming languages. The fourth section discusses how trust models can be used to give a flexible definition of data integrity that takes into account whether the data itself can be trusted or not. In last section, we conclude this chapter with a brief discussion.

2.1. Computer Security

Confidentiality allows authorized users or processes to access sensitive and protected data. Confidentiality is enforced in a classification system such that normal users are not allowed to access another user's files by default. Sensitive information is valuable and needs to be restricted to authorized users only such as U.S. government or military documents, bank account statements, credit card numbers, and personal information. Protecting such information is a major part of information security. Two ways to protect information confidentiality include encryption and access control.

Encryption ensures that only the authorized users are able to obtain the information. Data encryption translates data into another form, so the only users or processes with access to the decryption key can read it. In this case, confidentiality is enforced even though the

stored or the transmitted form of the sensitive data is not confidential. Many data encryption techniques to ensure and enhance the confidentiality are proposed in [14], [15], and [16]. Another way to ensure information confidentiality is through enforcing file permissions and access control to restrict access to sensitive information. Controlling appropriate access levels can ensure various levels of data sensitivity. General access control contains essential services: authorization, identification and authentication, access approval, and accountability [17].

Authorization specifies what a user or a process can do, identification and authentication ensure that only a legitimate user can have access to the system' resources, access approval allows users to access associated resources based on the authentication policy, and accountability identifies what subject(s) is associated with a user. Most modern operating systems enforce an access control model on their resources such as Windows 10 and Windows server 2016. Moreover, SELinux model provides secure access modes for storing files [18], and fine-grained Type Enforcement (D-TE) that protects special classes of information by partitioning the host operating system into access control domains [19].

Availability ensures that accessing the permitted resources in a system is not denied for authorized users and that the system works promptly. When the system is non-functioning, information availability is affected. Also, when data is not secure, information security is affected. Another factor affecting data availability is time. If a system cannot retrieve or deliver the permitted information efficiently, then availability is compromised.

Availability is an important security aspect of reliability as well as of system design. An authorized user cannot access permitted resources if the desired resources are unavailable. Attempts to block availability are called denial of service attacks (i.e., handling airline reservations, air traffic control, and automated medical systems). These attacks are difficult to detect because of the complications of determining unusual access patterns that can manipulate the resources of the underlying system.

There are many approaches to ensure availability such as correctness, fault tolerance, and good backup plans. Software correctness proves that a given program operates according to some specifications. In short, correctness is the best proof against poor behavior of a compromised system. Fault tolerance allows working around crashes to parts of the system. For example, a user may be able to move to another system and continue working. Finally, availability is also the ability to recover from damaging events effectively. Using good backup plans helps in recovering damaged systems or resources but the recovery process requires significant time and resources to recover the lost data.

Integrity is concerned with whether a piece of information is trustworthy, regardless of who is accessing it. Any unintended modification to data as a result of a processing operation or retrieval, including malicious intent or by human error, is a failure of data integrity. However, confidentiality and availability are concerned with whether a given user is authorized to access some piece of information, data integrity ensures the accuracy and consistency of data at some point in its life cycle [20]. The data life cycle consists of:

1. Entering, generating and/or acquiring data,
2. Processing and/or developing data,
3. Storing, copying and distributing data,
4. Backing up and restoring data,
5. Archiving and recalling data, and
6. Deleting, eliminating and destroying data.

There are two types of integrity: origin and data. Origin integrity ensures the source of the data is acceptable. Data integrity ensures the content itself is valid or trustworthy. Origin integrity is well understood and served by the same access control mechanisms that

enforce confidentiality. Most modern operating systems provide general enforcement of origin integrity by limiting read/write access using access control mechanisms. Data validation is the pre-requisite for data integrity. In fact, data integrity is considered a complex problem such that it is possible to validate the file format, but it is not always possible to validate the data itself whether it is valid or not. In fact, it is difficult to check well-formed, incorrect data in such a system without outside help. For example, we can write a program to ensure whether an accounting ledger is well formed such that D represents today's deposits, W represents withdrawals, YB represents yesterday's balance, and TB represents today's balance. Data integrity constraint will then be D+YB-W. The issue in this example is who verifies the transactions are done correctly?

There are two different types of integrity systems for managing data integrity. Prevention systems that disallow unauthorized attempts to modify the data, and detection systems that determine data trustworthiness. Both systems can work together to prevent an incorrect process from modifying the data, and provide enough information about who caused the failure and how it happened when data integrity failure is detected.

One of the most known data integrity attacks is Heartbleed [21]. Heartbleed involves missing bounds check before a *memcpy()* call that uses non-sanitized user input as a length parameter. An attacker tricks *OpenSSL* to allocate a 64KB buffer, copying more bytes than is necessary into the buffer, and sending the 64KB buffer back at a later time. This leaks the contents of the victim's memory and causes a failure of data integrity. This attack is limited to only single or multiple Heartbleed by requesting an arbitrary number of 64KB chunks of data until secrets are released. As noted in this attack, access control mechanisms are unable to ensure origin integrity since the Heartbleed message is a part of a server-client connection. The detection system through TLS is unable to detect the semantic format of the received data.

As another example, typical double entry accounting requires that every transaction must be presented as a withdrawal from one account and an equal deposit into another account. Enforcing the integrity constraints on the ledger contains three parts:

1. Allowing certain users to write new entries into the ledger.
2. Ensuring the authorized users follow the rules of double entry accounting.
3. Auditing periodically to ensure all real transactions were recorded.

The first part provides origin integrity by allowing only permitted users to modify the ledger. The second part, works as a prevention system that enforce the rules and ensures data trustworthiness. Finally, the third part, periodic audits helps to detect integrity failures. However, there are few prevention systems that can provide general integrity protection with actual implementability.

2.2. Discretionary and Mandatory Systems

Access control is important to computer system security. To compromise a system, attackers try to gain any access to obtain restricted data or make unapproved system modifications. The Department of Defense's Trusted Computer System Evaluation Criteria [22] establishes security polices for evaluating the computer security using access control models. Access control models enforce a set of rules controlling access rights to the system resources. The most directly useful concept related to our work is the difference between discretionary and mandatory access control models. Discretionary access control policy (DAC), like the one in traditional UNIX and Linux systems, assigns access rights based on rules specified by users. In other words, the owner of an object decides who is allowed to access that object, and what privileges they have; meaning there is no central control policy. DAC has two important concepts: data ownership where every object in the system has an owner, and

access rights and permissions that are assigned by the owner to other subjects. Whereas, Mandatory Access Control policy (MAC) assigns access rights based on rules specified by a central authority; meaning all access control rules are enforced centrally. MAC is difficult to manage, and it is used to protect highly sensitive information such as government and military information. In MAC, subjects and objects are assigned sensitivity levels. A subject's sensitivity level is a level of the trust, and an object's sensitivity level specifies the level of the trust required to access the object. DAC and MAC are usually used in environments where confidentiality is most important.

2.3. Computer Language Design

Computer languages are defined by their syntax and semantics. Language syntax is a set of rules that describes how to construct a valid statement within a language. Statements that are syntactically invalid have a syntax error. Language semantics describes the meaning of syntactically valid statements. Generally, syntactic processing is handled by a parser and followed by the semantic processing that is handled by an interpreter or compiler that translates the parser's output into executable code. In some cases, and for better language analysis, both syntactic and semantic processing is done concurrently.

A syntactic metalanguage is a notation for defining the syntax of the language using a set of rules. Backus-Naur Form (BNF) is a notation for defining the syntax of a programming language formally. Extended BNF is standardized to be a precise method for specifying the syntax of many languages, and it includes the most widely adopted extensions.

A precise description of programming language semantics can be developed using an interpreter or a compiler. Building an interpreter or a compiler is a way to map the language constructs into executable codes. Variants of language semantics are used in specifying the behavior of languages:

1. Operational semantics: It is a formal operational model for specifying the meaning of a programming language by providing a comprehensive formal definition of a program in terms of its structure.
2. Denotational semantics: It is a mathematical model for specifying the meaning of a programming language by constructing mathematical objects called domains that describe the meanings of the language statements.
3. Axiomatic semantics: It is a mathematical model for specifying the meaning of a programming language by using proof rules within the language logic. In other words, it proves the language correctness with respect to logical specifications.

Operational semantics framework is used in the specification and implementation of many programming languages. Structural Operational Semantics (SOS), introduced by G. D. Plotkin [2], is used to specify a framework for describing the operational behavior of programming languages. The basic idea behind SOS is to define the behavior of a language or a system in mathematical terms, in a form that supports understanding and reasoning about the language under consideration.

SOS has been successfully applied as a formal tool to give usable semantic descriptions for real-life programming languages including Java. SOS is the preferred choice over methods based upon denotational and axiomatic semantics in the static analysis of programs and in proving compiler correctness. SOS constructs proofs from logical statements about its execution, rather than attaching mathematical reasoning to the language items. Also, it is increasingly considered in the study of the theory of concurrent process.

2.4. Trust-based Models for Data Integrity

With respect to integrity protection, all data integrity models deal with the preservation of trust. There is a need for a more flexible definition of data integrity that takes into account whether the data itself can be trusted apart from who modifies the data. Due to the lack of certainty about the degree of the trustworthiness of subjects and objects, we need to have opinions to measure the integrity of these subjects and objects. Subjective logic is an artificial reasoning framework that defines the term *opinion*, w, which expresses an opinion about the trust level of subjects/objects [5], [7]. The opinion translates into degrees of trust, distrust as well as uncertainty, which represents the absence of both trust and distrust values

Trust models are divided into two types: policy-based models and recommendation-based models. Both types use a language to express relationships about trust. Each type provides a measure of the trust in an entity, and the result of the evaluation is a complete *trust*, a complete *distrust*, or somewhere between *certain* or *uncertain*. Policy-based models require a language in which to express and analyze system policies. For example, the Keynote trust management system [23] that is based on Policy-Marker [24] is extended to support applications that use public keys. Recommendation-based models use past behavior to determine whether to trust an entity, including recommendations from other entities. For example, Abdul-Rahman and Hailes [25] base trust on the recommendations of other entities. In their model, they consider direct trust relationships and recommender trust relationships. Trust is computed based on integer values. They use -1 for direct trust as representing untrusted, values from 1 to 4 as representing the lowest to highest trust values, and 0 as the inability to make trust judgments. For recommender trust values, the integers -1 and 0 have the same meaning as with direct trust, while the values from 1 to 4 indicates how close the recommender judgment is to the entity that is being recommended. However, These models do

not consider all of the side effects of dynamic data integrity. For instance, the trust opinions of the system's subjects can keep obtaining lower trust levels when they read less trusted data but there is not mechanism in the model to raise the integrity levels, possibly resulting in isolation of the subject.

2.5. Conclusion

The classic model for computer information security defines three objectives of security: confidentiality, availability, and integrity. Each objective addresses a different characteristic of providing protection for information. Confidentiality and availability are well understood with generally implemented safeguards. There are no general-purpose integrity systems. Research on protecting arbitrary structured data integrity is limited.

In this chapter, we have presented major weaknesses in existing data integrity approaches. Most of integrity approaches deal with the trustworthiness of who accesses the data, provide a general protection for a specific data format, or do not consider data/subjects' isolation from the system resources.

CHAPTER 3

RELATED WORK

This chapter provides a discussion of related research. It is organized within seven sections. Most of the related work deals with the trustworthiness of who accesses the data, designing an integrity protection system that supports a specific data format rather than an arbitrary data format, or designing an integrity based trust model that does not guarantee dynamic data integrity.

3.1. Integrity Models

There are several security models that address secure systems for the aspect of integrity. The most directly useful and related to our work are Biba integrity model, Clark-Wilson integrity model, and Maia integrity protection language for arbitrary structured file types. Each integrity model offers a definition of data integrity and introduces their own mechanisms for preserving integrity. All of these models are addressed next.

3.1.1. Biba Integrity Model

The Biba integrity model is concerned with an unauthorized modification of data within a system by controlling who may access it. It works as a prevention system for origin integrity. The model deals with a set of subjects, a set of objects, and a set of integrity levels. Subjects may be either users or processes. Each subject and object is assigned an integrity level, denoted as I(S) and I(O), for the subject S and the object O, respectively. The integrity levels describe how subjects and objects are more or less trustworthy regarding a higher or lower integrity level. Biba model considers four access modes:

1. Modify: Allows a subject to write or update information in an object.
2. Observe: Allows a subject to read information in an object.
3. Execute: Allows a subject to execute an object.
4. Invoke: Allows a subject to communicate with another subject.

Biba model can be divided into two types of policies, mandatory and discretionary policies. Most literature on the Biba model refers to the model as being mandatory as a part of strict integrity policy [6]. The strict integrity policy is the most popular policy in the Biba model. Biba model defines a number of rules as part of the strict integrity policy:

1. Simple integrity Property: A subject can only read objects at its own integrity level or above:

$$s \in S \ \ reads \ \ o \in O \iff I(s) \leq I(o)$$

2. Integrity Star Property: A subject can only write/update objects at its own integrity level or below:

$$s \in S \ \ updates \ \ o \in O \iff I(o) \leq I(s)$$

3. Invocation property: A subject may execute another subject at its own integrity level or below:

$$s_1 \in S \ \ invokes \ \ s_2 \in S \iff I(s_2) \leq I(s_1)$$

The first property of the strict integrity policy enforces "no read down". It allows a subject to read (observe) an object only if the integrity level of the subject is less than the integrity level of the object. It ensures that high-integrity data cannot be directly contaminated by low-integrity data. For example, if the simple integrity rule is enforced, a low-integrity process may read low-integrity data, but it is prevented from contaminating a

high integrity file that stores this data. On the other hand, a high-integrity process cannot read low-integrity data and then write it in a high integrity file.

The second property of the policy enforces (no write-up). It allows a subject to write an object only if the objects integrity level is less than or equal to the subject's level. It ensures that high-integrity subjects cannot be corrupted by reading malicious low-integrity objects in the system. The last rule is the invocation property, which states that a subject at one integrity level is prohibited from invoking (send/request messages for service) a subject at a higher level of integrity.

Biba is well-known general integrity model in computer systems. The strict integrity property succeeds at enforcing origin integrity in a system, but it does not deal with the integrity of data itself; authorized users can still make improper modifications. For example, if a trusted user account is compromised, an attacker can use a trusted user's integrity level to modify high-level integrity resources.

3.1.2. Clark-Wilson Integrity Model

The most practical integrity model was proposed by David Clark and David Wilson. The Clark-Wilson integrity Model (CWM) focuses on the prevention and detection of data integrity faults using transactions. The model is based on two concepts that are used to enforce commercial security polices or constraints:

1. Well-formed transactions: A user can manipulate data using only constrained rules to ensure the integrity of data.
2. Separation of duty among users: A person who has the permission to do well-formed transaction may not have the permission to execute the constrained data.

CWM enforces integrity controls on data by separating all the data items within a system into two groups:

1. Constrained Data Items (CDIs): Data items that have associated integrity constraints.
2. Unconstrained Data Items (UDIs): Data items that do not have associated integrity constraints. For example, a simple text file.

After classifying the data items, the integrity system tests the data items through two types of procedures:

1. Integrity Verification Procedures (IVPs): Ensure that all the CDIs conform to the integrity constraints or specifications.
2. Transformation Procedures (TPs): Achieve data items transaction by changing the system's CDIs from one valid state to another.

If the system is in a valid state at the current time that means the IPVs were successfully run, and the system ensures that only TPs are allowed to manipulate the CDIs at the current time. That shows that the system will continue to be valid for the next coming states. For TP and IPV validation, an external verifier is needed to certify the correctness of TP and IVP within the system. The external verifiers impose the integrity by sets of certifications and enforcements rules. Certifications rules are the security policy restrictions inspecting the behavior of IVPs and TPs, and are done by the security officer, while enforcements rules are built-in system security mechanisms that enforce the certifications rules. The rules are as follows:

C1 (Certification) All IVPs must ensure that all CDIs are in a valid state at the same time the IVP is run.

C2 All TPs must be certified to be valid. That is, the certifier must define a set of relations specifying the set of CDIs on which a given TP is certified to run. A TP-CDIs relation has

the form (TPi, (CDIa, CDIb, CDIc, ...)), where CDIs define a list of arguments for which the TP has been certified.

E1 (Enforcement) The system must ensure that CDIs are only manipulated by certified TPs to modify them according to the relations in C2.

E2 The system must maintain and enforce a set of relations mapping a user to the set of TPs they may execute, and CDIs those TPs may modify on the user's behalf. The list of relation has the form (USERIS, TPi, (CDIa, CDIb, CDIc, ...))

C3 The list of relations in E2 must be certified to ensure separation of duty requirements.

E3 The system must authenticate the identity of each user attempting to execute a TP.

C4 All TPs must be certified to write to an append- only CDI(the log) all information that is reconstruct the nature of the operation performed.

C5 Any TP, which transforms a UDI into a CDI, must be certified in to produce either a valid CDI or nothing at all in all cases when UDI is rejected.

E4 Only the permitted agent to certify entities can change the list of associated entities with other entities. For example, the associated TPs with a CDI and the list of the users associated with a specific TP. The same permitted agent, who can certify an entity, may not be able to execute it.

Rules C1, C2, and E1 provide the core of the CWM. E2 allows for restricting access to TPs and CDIs to specific users, while C3 decides which users, E3 provides a system with mechanisms to figure out which permissions to use such that the type of authentication is not defined before use of the system, but is required before any manipulation of CDIs. C4 provides an audit trail, and can be implemented by making the log an append-only CDI. C5 handles untrusted input, so it allows data that cannot be trusted by itself to become a trusted data object. Finally, E4 makes CWM a mandatory, rather than discretionary

system, and guarantees that there in no single user can corrupt the integrity of the system by enforcing separation of duty with respect to certified and allowed relations.

Clark-Wilson provides data integrity but imposes a number of restrictions that make it impractical to implement. A transformation procedure runs into trouble if a single application is able to execute many different transformations. For example, a text editor can be used to produce HTML files, or to edit the UNIX password file. To implement Clark-Wilson, the text editor must be broken into a HTML editor, and a password file editor to be certified to produce valid HTML files, and valid UNIX password file. Additionally, an administrator needs to mange and verify all the editors and that is impractical and impossible in the worst case.

3.1.3. Maia: A Language for Integrity Protection on Arbitrary Structured File Types

Many tools exist to verify particular file formats. XML, which is widely used online and for storing configuration data, has several different verifier systems: DTD [26], XML Schema [27] [28], and RELAX NG [29]. Tools also exist for verifying HTML, and many reference implementations for image formats include sanity checking of their input. While these are powerful tools for protecting particular file types, they cannot be generalized to protecting other file formats. Instead, we need an approach that will allow us to verify a variety of file types.

Parser generators are commonly used in developing new programming languages, and can be applied to the problem of creating verifiers. Lex [30] and Yacc [31], and their successors Flex and Bison generate robust, fast parsers which can be embedded in C or C++ programs. ANTLR [32] serves a similar purpose, with a focus on emitting Java rather than C code. These tools are often sufficient to produce syntax checkers on their own, but creating

semantic checks requires detailed knowledge of the underlying parsing technology. The ties to programming language creation that makes these parser generators fast can also impact the set of languages they parse correctly. It is possible to design a programming language around the restrictions of one's chosen parser generator, but this is harder when the format to be parsed already exists. PNG images [33], for example, make use of chunk length specifiers that introduce context sensitivities which are difficult to handle in a normal parser generator. Some tools, like YAKKER [34], are able to cope with limited context sensitivity, but still require programmer assistance to perform semantic checks.

Data description languages [35] [36] are designed to provide automated parsing for ad hoc data formats. Tools like PADS [37] give programmers the ability to describe semi-structured file formats so that their programs can more readily access the contents of the file. While this approach significantly simplifies handling formats which were not designed with parsing in mind, it still requires the intervention of a programmer to describe the format in question and then perform validity checks.

Maia improves on these tools in two important ways: Maia can be used to describe any file with a context free structure, and can handle certain types of context sensitivity. Additionally, Maia specifications describe valid files, not how to validate files, meaning that no programming is required to generate a verifier. As we will demonstrate, tools can convert Maia specifications into fully functional verifier programs.

Maia is a specification language that verifies arbitrary structured file types [23]. It is designed to describe the integrity constraints for arbitrary structured data. For example, Maia within the context of Linux can be used to specify the format of system configuration files, PNG files, and others. In Maia, a file verification process is accomplished using two phases that correspond to checking the file syntax and semantics. In the first phase, the user

provides an Extended Backus Naur Form (EBNF) grammar in order to verify the file structure and extract its syntactic elements for processing. This syntax checking component of Maia is designed to work like a normal parser as generated by a parser generator. The second phase of Maia checks the constructed containers of data in the syntactic elements by using set theory and predicate calculus to express the integrity constraints. The containers used in this phase are automatically constructed during the syntax checking phase by grouping all occurrences of the same nonterminal (e.g. user names in the `passwd` file) together.

Maia has a couple of weaknesses. First, Maia does not support binding names to a container. For example, consider a rule where there is at least one user with `name` of "root", giving the container `name` that includes names for all users. Maia expresses the given rule as:

```
exists name : name == "root";
```

Also, a rule that states there is at least one user with `uid` of 0, given the container `uid` that includes uids for all users, Maia expresses the given rule as:

```
exists uid : uid == 0;
```

Another example shows a rule that applies the constraint on all elements of the container, name must be "root" that indicates the user's `uid` equal to 0. Maia expresses the given rule as:

```
forEvery passwd : name == "root" implies uid == 0;
```

From these examples, we can see that the specified names are not bound to their containers, and that leads to semantic ambiguity. A verifier cannot confirm that the `uid` (name) is an element in `uid` (container). The author of Maia uses a hand verifier to confirm that.

Second, Maia also introduces another problem, which is container bounds checking. For example, `isUnique` template can be defined within container name called `name` as:

```
(template name isUnique()) forEvery name: name[i] != name[j] ;
```

We can notice that the verifier cannot determine that i in [1..length(container)], in [i+1..length(container)] to verify:

```
( i != j  implies name[i] != name[j])
```

In addition, name on the right hand side of the rule refers to the entire container whereas name refers to an element of the container in the previous rules. In this work, we solved names binding and container bounds-checking problems, and implemented a fully automatic verifier for Maia language.

3.2. Integrity via Access Control

Access control limits who may modify information, and enforces the restrictions on which processes can cause the authorized modification. This provides origin integrity by limiting the source of changes. Origin integrity systems have been implemented using existing access control mechanisms. This includes type-based discretionary access control system (DAC), and mandatory access control system (MAC). DAC provides access control to objects by user identity. Each object has a list of users. The permission is defined on objects as a set of access modes: (read, write, and execute). An example of DAC is Unix file system, which defines the read, write, and execute permissions in each of the three bits for each user, group, and others.

MAC views computer systems as a collection of subjects (user or process) and objects. An example of a MAC system is multi-level security MLS. In MLS, a security administrator assigns sensitivity levels (Top Secret (TS), Secret (S), Confidential (C), and Unclassified (U)) to the subjects and objects. These levels are ordered hierarchically from the most to the least secure such that TS > S > C >U. Each subject is associated with clearance, and

each object is arranged within a classification. The permission is defined on objects as a set of access modes: (read, append, write, or read-write). For example, user A has a clearance secret and user B has a clearance of top secret. Both users are attempting to access an object that is classified as a top secret. Based on the level sensitivity, user B can only retrieve the data successfully.

DAC and MAC systems introduce permissions mapping difficulties when the number of files or processes grows as in a modern computer system. Badger, et al., proposed their Domain and Type Enforcement (DTE) to address these problems [19]. Under DTE system, each process executes within a domain and objects are assigned a type. The domain specifies the rights, and all process excusing within the same domain have the same set of access rights. DTE Language (DTEL) is a high level symbolic language for expressing policies in a human readable format. DTEL supports inheritance structure through a file system.

DAC and MAC systems can also introduce role management problems if permissions are assigned to a positions or roles rather than a specific user. To address this problem, Ferraiolo, et al., propose a role based access control (RBAC) [38]. Under RBAC, permissions are associated with roles, and users are assigned numbers of appropriate roles. Access control policy is expressed in different components of RBAC such as role-permission, user-role, and role-role relationships. RBAC components are able to determine whether a particular user will be allowed to access a particular piece of data in the system. Checking on user's roles rather than the user themselves, makes updating role-based permissions trivially easy, and introduces minimal complexity during normal checking.

In addition to the previous access control mechanisms, origin integrity systems have been achieved using existing access control mechanisms. This includes designing approaches for a MAC mechanism for the Unix file system [3], securing data flow and file movements using discretionary access controls [4], and an approximate Clark-Wilson model using Unix access

controls [39]. Security-Enhanced Linux (SELinux) has been integrated into Linux kernel [40]. SELinux model provides a mechanism for supporting access control policies using MLS, DTE and RBAC. However, a trustworthy entity that access an object may still violate the integrity of the data. For example, a root on a Linux system has full access and can modify anything in a manner that violates a security policy. The access control mechanisms cannot directly address such modifications and ensure data integrity without relying on users or special software. In our tool, we are looking at integrity as a property of the data, not the trustworthiness of the process modifying it.

3.3. Database Integrity

The integrity of relational database systems are concerned with the maintenance of the correctness and consistency of the data in a multi-user database environment. Database integrity violations result from many different sources such as errors in system software, typing errors, and logical errors in application programs. Many database management systems have an integrity subsystem that is responsible for monitoring transactions, detecting and reporting integrity violations, and returning the database to a consistent state.

The relational model is the conceptual basis of relational databases, and it is first proposed by Codd in 1969 [20]. The first major characteristic of the relational model is the usage of relations (tables) to store all data. Each relation consists of rows and columns, and also it must have a header and body. The header is a list of columns in the relation, and the body is the set of data organized into rows. The junction of one column and one row will result in a unique value called a tuple. The second characteristic of the relational model is the usage of keys to order or relate data to other relations. One of the most important keys in the relational databases is a primary key that is used to identify each row of data uniquely, and the foreign keys that relate data in one relation to the primary key of another

relation.

The integrity of relational database systems is widely studied and well understood. The relational model depends on a set of rules to enforce data integrity, these rules are known as integrity constraints. Integrity constraints deal with the reliability of the data stored in the database, and are divided into three categories:

1. Domain integrity rules: It is concerned with maintaining the correctness of attribute values within relations. It defines the type of the domain as a type checking in programming languages. The type of the domain must be precise to avoid violations of domain integrity.
2. Entity integrity rules: It relates to the correctness of relations among attributes of the same relation. This integrity ensures that each record in a table is unique and has a primary key that is NOT NULL.
3. Referential integrity rules: It is concerned with maintaining the correctness and consistency of relationships between relations. It requires that a foreign key must have a null entry or an entry that matches the primary key value in a table that it belongs to.

Turker and Gertz [21] provide a summary of integrity constraints enforcement in the standard SQL 1999. The standard SQL supports two powerful mechanisms of integrity: declarative and procedural constraints. Declarative constraints are applied during table creation and modification. It enforces rules that include field requirements like NOT NULL, row constraints like CHECK, table constraints like UNIQUE, and inter-table FOREIN KEY. However, procedural constraints are activated via triggers when changes are made to the database. Triggers are essential to realize effective constraint enforcing, and become useful when features supported by declarative constraints cannot meet the functional needs of the application. It can be set to run immediately before or after the database is modified

(INSERT, UPDATE, and DELETE actions). Triggered procedures are able to access the previous and new state of affected records. Also, they can be set up to create new table entities or remove old entities. Triggered procedures in the database support state change validation and record updating to be set automatically, regardless of how data modification is initiated.

Database integrity protection is implementable but it does not provide sufficient functionality to protect general data independently; the traditional relational database does not provide the integrity controls to validate individual fields in general. Although, the integrity constraints provided by SQL are very powerful but they are limited to the core type supported by the language. For example, to validate a phone number, a database user needs to provide an additional code to one that is provided by SQL.

3.4. Integrity via Fault Tolerance

Fault tolerance is the property that enables a system to continue operating correctly in the event of the hardware or software failure. A program is developed by making a series of small changes to a set of legal states (state transitions) as a basic action of assigning value to a program variable. During normal operation, the program arrives to a valid state within its legal states. When a fault occurs, the program jumps unpredictably to another state within some larger set.

A fault tolerant computer system is designed to handle hardware-related faults such as hard disk failures, input/output device failures, and software-related faults such as bugs and errors. Fault tolerant software is handled through two conditions: closure and convergence [15]. Closure requires that a program that starts in valid state will remain operating in valid states. With a Closure condition, programs will not crash or misbehave due to programming bugs or errors, while convergence requires a program to be able to return to a valid state after

faults stop occurring. Combing these two conditions will guarantee the program continue in valid states until it has failed again.

Fault tolerance and integrity protection solve the same problems: preventing and recovering from errors in the system state. However, integrity is primarily about user/process actions, while fault tolerance is a reaction to environment effects. Fault-tolerant software may recover form environmental actions, but it will not guarantee to write valid data, and cope with corrupted data (i.e. Mirroring software bugs).

3.5. Integrity via Correctness

Program correctness is achieved when a program is correct with respect to some specifications. Functional correctness refers to the input-output behavior of the program such that, for each input it produces the expected output. Hoare logic is a specific formal system for reasoning about the correctness of computer programs [16]. It defines programming language semantics through assertions known as Hoare triples: precondition: a proper condition to run the program, command: indicates the behavior of the program, and postcondition: the desired result following the program's termination. Program correctness is always described with respect to its specification precondition and postcondition.

There are two kinds of correctness proofs: partial correctness (weak): assumes all commands terminate:

```
(precondition and termination) => postcondition
```

This kind of correctness is weak since it makes no claim if a command fails to halt. Total correctness (strong): requires a command be proven to always halt, and the pre/postcondition pair being true.

```
(partial correctness and termination) => postcondition
```

Proving the correctness of a small program is useful and manageable but it does not suitable for modern software systems that have millions of lines code, representing thousands of semantics states and state transitions. These kinds of systems require a robust tool for assuring that the system behaves properly in each of its states. For example, the Java Modeling Language (JML) provides code level extensions to Java, and Spark/Ada is a proprietary system that provides the similar extension to Ada. These robust tools enable the programs to include formal specification and their enforcement at run time [17].

Ensuring data integrity through correctness introduces two major problems: first, a provably correct program is not necessarily able to defend against user invalid well-formed data even if is guarantees the total correctness. For example, a provably correct text editor is able to output a text file, which includes the wrong information in the wrong format. The other problem is about the correctness of the software system itself. For example, a text editor can modify user account information in provably correct primary user account management tools.

3.6. Integrity via Trust Models Based Subjective Logic

The following are different enhanced trust-based subjective logic models to support various organizational security policies that have been proposed. Oleshchuck proposes a trust-enhanced data integrity model that is based on the Biba integrity model using subjective logic [4]. In his model, he reformulates the rules of the Biba integrity model in terms of trust and proposes how to combine Role-Based Access Control RBAC with the introduced integrity model. Gao, et al. [8], propose a trust model by analyzing and improving subjective logic. By using subjective logic in their model [7], they can evaluate the trust relationship between peers and resolve security problems in practical computing environments. Jøsang proposes a trust management system based on subjective logic [6]. He proposes an evidence

space and opinion space that are used to evaluate and measure trust relationships.

These policy-based trust models use credentials to instantiate policy rules that determine whether to trust an entity, resource or information. The policies do not protect the system entities since the credentials themselves are information that is not protected by the model. On the other hand, trust models preserve the initial evaluation of data integrity by providing information about the trustworthiness of data and entities. These models do not consider all of the side effects of dynamic data integrity. For instance, the trust opinions of the system's subjects can keep obtaining lower trust levels when they read less trusted data but there is not mechanism in the model to raise the integrity levels, possibly resulting in isolation of the subject.

3.7. Conclusion

Computer security is comprised of the related fields of confidentiality, availability, and integrity. While confidentiality and availability are generally well-understood with widely implemented safeguards, there are no general purpose integrity systems. Existing integrity systems are either:

1. Focusing on safeguarding specific systems at the cost of generality –as with databases,
2. Providing general protection and they are not well-suited to the real-world implementation –as with Biba or Clark and Wilson integrity models,
3. Dealing with trustworthiness of who access the data –as with access control approaches, or
4. Causing data/subjects isolation from the system resources -as with trust models based subjective logic.

CHAPTER 4

STRUCTURAL OPERATIONAL SEMANTICS SOS FOR MAIA

In this chapter, we give a Structural Operational Semantics (SOS) for Maia and report on a preliminary implementation of a Maia compiler in the context of a mandatory integrity system. The semantics give precise rules for giving meaning to a Maia specification. These rules show there is no ambiguity in Maia, giving assurance that a correct implementation of Maia file verifiers is possible. This allows one to implement a compiler or interpreter and use Maia to specify integrity constraints within an integrity system.

This chapter is structured as follows: first, we give an overview of Structural Operational Semantics. Next, we define Maia and give its SOS. Finally, we give a brief conclusion.

4.1. Structural Operational Semantics (SOS)

SOS, introduced by G. D. Plotkin [2], is used to specify a framework for describing the operational behavior of programming languages. The basic idea behind SOS is to define the behavior of a program or a system in mathematical terms, in a form that supports understanding and reasoning about the program under consideration. SOS has been successfully applied as a formal tool to give usable semantics descriptions for real-life programming languages including Java. SOS is a direct approach that provides comprehensive definitions in a very simple formal mathematics. Moreover, SOS is the preferred choice over methods based upon denotational semantics in the static analysis of programs and in proving compiler correctness.

As described by Prasad and Arun-Kumar [41], SOS defines the semantics of a programming language from the syntax by applying the correct sequence of inference rules. Each

rule has the form

$$\frac{\begin{array}{c} P_1, \\ P_2, \\ \ldots, \\ P_n \end{array}}{C}, \tag{4.1}$$

where, P_i represents judgments (premises or assumptions), C is a single judgment or conclusion, and side conditions express the constraints of the rule. The inference rule states that if all of the premises are true, then the conclusion is true.

We present the SOS of Maia using big-step structural semantics that justifies a complete execution sequence using a tree-structured proof. Any semantics of a programming language involves auxiliary entities or bindings such as environments, stores, etc. We present the SOS of Maia with respect to a finite domain function called an *environment*, γ, that maps a set of variables, X, to their computed values V. The big-step transition relation $\Rightarrow_e$ is defined inductively as the smallest relation closed under the inference rules given a Maia rules specification. The SOS of Maia rules specification has the form $\gamma \mapsto R \Rightarrow_e v$ which is read "given an environment γ, the syntax rule R evaluates to a value v". This relation is understood as a transition that leaves γ unchanged. The rules can be expressed as a proof tree of why R can evaluate to a value, where the goal judgment $R \Rightarrow v$ is at the root, the internal nodes represent the rule instances with a branch for each antecedent, and the leaves are axiom instances.

4.2. Maia

In this dissertation, we present an integrity language, called Maia, that is a specification language that verifies arbitrary structured file types [1], [13]. It is designed to describe the integrity constraints for arbitrary structured data. For example, Maia within the context of Linux can be used to specify the format of system configuration files, PNG files, and others. In Maia, a file verification process is accomplished using two phases that correspond to checking the file syntax and semantics. In the first phase, the user provides an Extended Backus Naur Form (EBNF) grammar in order to verify the file structure and extract its syntactic elements for processing. This syntax checking component of Maia is designed to work like a normal parser as generated by a parser generator. The second phase of Maia checks the constructed sets of data in the syntactic elements by using set theory and predicate calculus to express the integrity constraints. The sets used in this phase are automatically constructed during the syntax checking phase by grouping all occurrence of the same nonterminal (e.g. user names in the passwd file) together.

The rest of this section provides a detailed discussion of Maia's design including syntax and semantic specification, automated container building, and other features of Maia.

4.2.1. Design Objectives

Maia has two primary objectives. First, it is must be able to protect arbitrary structured data in information systems by providing a generally applicable system to detect and prevent modifications which would compromise data integrity. Maia is designed to be as flexible as possible to support different types of structured files rather than restricting it to only support certain files structures. Second, a Maia specification is able to describe the valid format of data. In other words, Maia takes the perspective that integrity is the property of

the data. Integrity does not follow the trustworthiness of the process modifying the data. We designed Maia to provide implementation independent descriptions for valid structured files that requires not only reasoning about the verifier's rules but also how they are implemented.

4.2.2. Syntax Specification

Syntax rules represent the first phase of file verification. We built a separate verifier for this phase. The syntax phase verifier is designed to work like a normal parser as generated by a parser generator. Additionally, the syntax phase verifier is designed to exit as soon as it realizes a file is invalid. Otherwise, the syntactic elements will be constructed to be tested in the semantics phase. A Maia specification begins with a rule specifying the file to be verified using a full path specification. A Maia verifier invokes the syntax phase verifier with the specified file and its data.

Maia produces a syntax checker via the standard metalanguge Extended Backus Naur Form based on ISO/IEC 14977 specification. Some changes and extensions are added to ISO/IEC 14977 specifications in order to improve Maia syntax rules [23].

In Maia, within the syntax checking phase, nonterminal names are constrained to the following conventions:

1. Names are expressed from a common name purpose of the syntactic element.
2. Names consist of one or more words, identified from this point on by CamelCase.
3. Names that appear in the left side of the definition start with an uppercase letter.
4. Names that appear in the right side of the definition start with a lowercase letter.

A top-level nonterminal that refers to a file type specification has one nonterminal whose name starts with uppercase letter. All top-level nonterminals should receive uppercase first

letters to be reused later, even though they can be appeared on the left in definitions in other files.

4.2.3. Building Containers

Maia's syntax phase automatically constructs containers by grouping all occurrences of the same nonterminal together. In Maia, a container is a multi-set that is ordered by the order in which the elements appear in the scanned file. Each nonterminal entry in the specification will be an ordered container with the same name during the second phase. For example, consider a file consisting of one or more colons separating pairs of user names and hashed passwords, where each pair appears on its own line. The file has the form:

```
alice: 19fd01b2307d497fb174decd8bc9c121
bob: 0f68eb4c87c99c563e168cdc2cd92336
```

An example Maia syntax specification for `/etc/passwd` follows.

```
UserFile= userRecord+ ;
userRecord = name ":" password Newline ;
name=[a-z]+ ;
password = [a-z0-9];
```

The syntax checker phase compares the given file structure against file's data to obtain its syntactic elements. If the input file has proper syntax, the parser of this phase produces an abstract syntax tree. The containers are constructed in this phase by traversing the generated tree and collecting the information scattered throughout the tree.

Automated Containers Construction. Containers in Maia's first phase are constructed automatically. Each container contains the input chunks that matched the parser rule. Since each nonterminal becomes an ordered container, the example has the containers,

`userRecord`, `name`, and `password`. We consider two types of containers that are constructed within the syntax rule phase: Simple and compound containers.

Simple containers are constructed when nonterminals contain at most one other nonterminal. The elements of the constructed container appear in the order they were encountered by the parser. By considering this file,

```
alice: 19fd01b2307d497fb174decd8bc9c121
bob: 0f68eb4c87c99c563e168cdc2cd92336
```

The simple container `name` will be an array, and has the form:

```
name= (‘alice’, ‘bob’)
password= (‘19fd0...’, ‘0f68e...’)
```

Compound containers that are constructed when nonterminals contain at least two other nonterminals. Dot notation (e.g. ContainerName.ContainerMember) is used to explicitly refer to members of a compound containers. In our example, the compound container `userRecord` will be an array of simple containers `name` and `password`, and has the form:

```
userRecord.name= (‘alice’, ‘bob’)
userRecord.password= (‘19fd0...’, ‘0f68e....’)
```

By associating simple containers to their main entries, we solve the Maia's containers naming ambiguity problem. For example, given only the `passwd` file containers `name` and `uid`, there is no way to determine which correct name corresponds to which `uid` especially if there is another name container that is constructed explicitly. The main container `userRecord` provides the mechanism to define the relationship between `passwd` records, rather than using `name` and `uid` as simple containers.

Explicitly Constructed Containers. Maia's specification allows users to construct containers explicitly rather than building them from a given file. Explicit container construction in our tool is only available to the semantics phase. Elements within user-defined

containers are enclosed within curly braces and specified as a comma-separated list of either strings (enclosed by single quotation mark) or integer numbers. String and integer container types cannot be mixed. For example, both of these containers are valid in Maia:

```
class={ 'TS', 'S', 'C' , 'U'} ;
version={1,2,3,4} ;
```

In contrast, the `user` container below is invalid and causes Maia to exit and generate an error:

```
user={20, 10, 'alice1','bob85', '70R'}  ;
```

Joining Existing Containers. Maia provides a mechanism to create new containers by joining elements of two or more existing containers from the syntax phase or user-defined containers. The joining process requires containers to have the same number of elements. Attempting to join mismatched containers generates an error.

The joined containers are enclosed within curly braces as with user-defined containers and separated with one or more joining connectors. For string-based fields, the joining connector is a dot operator to indicate concatenation. For example, consider a file containing user login data, including user names and login domains:

```
user={'alice','alice', 'bob'} ;
domain={'DOMAIN1', 'DOMAIN2', 'DOMAIN2'} ;
```

The container of `user` and `domain` in Maia would have the following specification:

```
userDomain={user. domain } ;
```

This would give us the container:

```
userDomain=(‘aliceDOMAIN1’, ‘aliceDOMAIN2’, ‘bobDOMAIN3 )
```

Mathematical operators can be applied to containers for numeric types. In this case, the joining operator may be one of +, -, *, /, % and ^, which preserve their customary meaning of addition, subtraction, multiplication, division, modulus, and exponentiation.

The container-joining process generates an error if the mathematical operators are applied on containers that are string-parsed type. However, parsed numeric elements may be joined with other containers using concatenation (dot operator). This will use the raw input string rather than using the parsed value.

4.2.4. Semantic Specification

Maia's semantic rules are a straightforward adaption of predicate logic. For example, consider the rule " there must be at least one user with a `UID` of 0". Given the container `uid` that contains all the `UID`'s of all users, the formal expression for this rule is: $\exists u \in uid : u == 0$. The equivalent Maia's rule is:

```
exists u in uid: u == 0 ;
```

In Maia's semantic rules the $\exists$ is replaced by `exists` and it specifically refers to the existence of a single elements in a given container. The quantifier `u` in the right hand side refers to single elements in the containers rather than the entire container. Providing the container name in the right hand side to compare to a single element or other container variable will cause an error. Instead, containers are compared to other containers, and a container variable is compared to a single value or other container variable. For example, the following Maia semantic rule can be used to express the constraint "there must be at least one user id in `uid` and `uid1` containers such that both of them are equal", Maia semantic rule would be:

```
exists u in uid, exists v in uid1: u == v ;
```

Maia also has a rule that applies constraints on all elements of the container, for example, "`UID`'s must be in the range 0 to 32767". In predicate logic, this rule has this form:

$$\forall a \in uid : u >= 0 \wedge u <= 32767;$$

which translates into Maia as:

```
forEvery u in uid: u >= 0 and u <= 32767 ;
```

In the reminder of this section, we describe the basic form of Maia semantic rules. Then, we present the language features for building constraints between elements within containers and between containers themselves.

Rule Structure. Semantic rules have this basic form:

```
quantifiers : constraint ;
```

The quantifiers for the semantic rule specify one or more containers to which the rule applies and whether the rule must be true for all elements in the containers under consideration or at least one element. Semantic rules may apply to all constructed containers: parser-constructed, explicitly constructed, and joined containers. The rule will be false if there is at least one element does not satisfy the constraint of `forEvery` or no elements satisfy the constraint of `exists` rule.

The right side of the rule provides one or more boolean constraints that will be evaluated for each container's element or for each container until the rule is satisfied. Elements are evaluated in containers under consideration in the order they are encountered during parsing. Failure of an `exists` rule results in scanning the entire container. Comparatively, the failure of a `forEvery` rule is less expensive since evaluation can be terminated once a single case evaluates to false.

Variables, which appear in the constraint, refer to the current element of the associated container rather than a container as a whole. The variables must be compared to other variables or individual (integer or string) values. The Maia verifier generates an error if a variable is compared to a whole container. Additionally, containers' names may appear in the constraint. In this case, they must be compared to other containers to avoid generating an error.

Semantic rules are case sensitive for keywords, variables, and container names. As with syntax specification, the semicolon is used to represent the end of the rule, and white space is ignored except to separate tokens.

Logical Comparison and Connectors. Maia's constraints use any of the following logical comparisons that carry their conventional meaning: ==, !=, <, <=, >, and >=. The comparison operators are used for both numeric and string-based comparison. Elements with numeric passed types are compared as numbers, while elements with string parsed type are compared lexicographically.

For numeric parsed types, constraints in Maia also use one of any of the mathematical operators: +, -, *, /, % (module) and ^(exponentiation), that have their customary meaning and precedence. However, division (/) truncates and always returns an integer number. Maia generates an error if a mathematical operator is applied on an element that does not have a numeric parsed type. All parsed types can be concatenated using the dot operator, and the result is considered of string type.

In addition, constraints use standard logical connectors: `not`, `and`, `or`, `xor`, `implies` (implication), and `iff` (bicondition). The `not` operator can be used to negate a given constraint. The logical operators have their conventional meanings and precedence as follow:

```
not > and > or == xor > implies > iff
```

Using Quantifiers. We add variables to Maia to solve the ambiguity that is associated with container naming in the original definition [1]. A unique quantifier is added to represent each container. Maia generates an error if an existing quantifier is specified to represent other containers. Each quantifier represents all elements of the current containers. For example, the following Maia semantic rule can be used to express the constraint "there must be at least one user with a name of root", Maia semantic rule would be:

```
exists u in userRecord.name: u == 'root' ;
```

We design quantifiers in Maia to be compared to either a single value (integer or string) or other quantifiers. However, Maia generates an error if a quantifier is compared against a container. For instance, using the following rule to express the constraint " there must be at least one user id in the `uid` container that is greater than a group id in `gid` container":

```
exists u in userRecord.uid, exists x in userRecord.gid: u > gid ;
```

This rule is invalid in Maia because it attempts to compare the quantifier `u` against the container `gid`

Maia's Interpreter Values. Our Maia interpreter handles four types of values: integer, string, boolean, and list(integer, string, and boolean values). Quantifiers are compared to a single value or to other quantifiers of the same types. Containers are compared to either other containers or a list value of the same type.

In comparison constraint, Maia first applies type checking on current quantifiers, containers, and list values. Any mismatched type causes Maia to generate an error. For example, following is a semantic rule that tests the equality of the containers for user names and passwords in the `passwd` file:

```
forEvery u in userRecord.name, forEvery v in userRecord.password:
userRecord.name == userRecord.password  ;
```

4.2.5. Example Maia Specification

As examples of Maia, we first show a subset of a specification to protect the integrity of `/etc/passwd`. Then, we give a second example that is a portion of a specification to protect PNG files.

```
PasswdFile = (passwdRecord Newline)+ ;
passwdRecord = name ":" password ":" uid ":"
               gid ":"
gecos ":" directory ":" shell ;
name = [a-zA-Z_][-a-zA-Z0-9_]{0,31} ;
password = "*" | "x" ;
uid = NUMBER ;
gid = NUMBER ;
gecos = [^:\n]* ;
directory = [^:\n]+ ;
shell = [^:\n]* ;
NUMBER: DIGIT+  ;
DIGIT: [0-9] ;
Newline = "\n" ;

exists n in name : n == "root" ;
forEvery u in uid:   u <= 65535;
exists g in gid:   g <= 65535;
exists p in passwdRecord: p.name == "root" implies p.uid == 0;
forEvery p in passwdRecord : p.shell != "" ;
```

The above Maia specification ensures that there is a user named `root`, the user and group ids have a valid number, the user id of `root` is 0, and no shell is empty. The specification above the line that begins with `exists` gives the syntactic structure of a password file, and the lines beginning with `exists` through the end give what it means for a password file to be valid. The full specification of Unix password, group and shadow files can be found in [3].

As a second example, we give an excerpt of a Maia specification of PNG files.

```
ihdr = "\0\0\0\x0D" "IHDR" ihdrData ihdrCRC ;
ihdrData = width height bitDepth colourType
compMethod filterMethod interlaceMethod ;
width = UnsignedBigEndianInt{4} ;
height = UnsignedBigEndianInt{4} ;
bitDepth = BigEndianInt{1} ;
colourType = BigEndianInt{1} ;
compMethod = BigEndianInt{1} ;
filterMethod = BigEndianInt{1} ;
interlaceMethod = BigEndianInt{1} ;

forEvery i in ihdr : i.width >= 1 and i.width <= 2^31 - 1 ;
forEvery i in ihdr : i.height >= 1 and height <= 2^31 - 1 ;
forEvery i in ihdr : i.bitDepth in < 1 , 2 , 4 , 8 , 16 > ;
forEvery i in ihdr : i.colorType in < 0 , 2 , 3 , 4 , 6 > ;
forEvery i in ihdr : i.colorType in < 2 , 4 , 6 > implies
      i.bitDepth in < 8 , 16 > ;
forEvery i in ihdr : i.colorType == 3 implies
      i.bitDepth in < 1 , 2 , 4 , 8 > ;
```

```
forEvery i in ihdr : i.compMethod == 0 ;
(warn) forEvery i in ihdr : i.filterMethod != 0 ;
(warn) forEvery i in ihdr : not i.interlaceMethod
      in < 0 , 1 > ;
```

In this example, the `width` and `height` are limited to be 4 characters long using the `{4}` annotation in the respective syntax rules. The second half of the specification, beginning with `forEvery`, gives a subset of the semantic requirements for a valid PNG file. The full specification for PNG files can be found in [3].

4.3. SOS for Maia

In this section, we define a Structural Operational Semantics (SOS) for Maia. A Maia specification has the following basic structure:

$$M \rightarrow I^* \; T^* \; X^* \; C^* \; S_R^* \tag{4.2}$$

where I represents a file inclusion directive, X is an EBNF specification of the input file syntax, C represents a container construction operation, S_R represents a semantic rule and T represents a template. In the rest of this section, we focus on the semantics for X, C and S_R since they are the critical elements of Maia. The SOS of a Maia specification is

$$\frac{}{\gamma \vdash M \Rightarrow_e v} \tag{4.3}$$

where $v \in B$ and $B = \{\texttt{true}, \texttt{false}\}$.

4.3.1. File Inclusion

Inclusion brings an existing specification into the current specification via the `using` keyword that is similar to #include "file" in C. It provides both reusable definitions and the refinement of the existing specifications. Therefore, when a path is specified, and a file is included, all its syntactic specifications will be available, and all of its semantic rules are enforced. In Maia, inclusion has the form:

$$\begin{aligned} I \;\rightarrow\; & \textbf{using}\ \texttt{"sysPath"}\ \textbf{;} \\ \mid\; & \textbf{using}\ \texttt{"sysPath"}\ \textbf{on}\ \texttt{"sysPath"}\ \textbf{;} \end{aligned} \tag{4.4}$$

where `sysPath` is the path to the specification to be imported. The path can be relative or absolute. An absolute path is interpreted according to the local platform rules. Whereas, a relative path is interpreted in relation to the location of the current file. When the implementation provides standard libraries, relative paths are checked first against the location of the current file and then in the standard libraries. For example, a system loads the specification `fsobject.maia` , which is located in the current directory with the following command:

```
using "fsobject.maia" ;
```

Normal Maia specifications produce verifiers which process whatever input they are given, but this is insufficient in the event that multiple files must be parsed together. To deal with the need to process specific other files as part of verification, we provide an extension to the file inclusion system. First, specifications should be created for each of the files in the file group. Then, a new specification can be written which includes the existing specifications, links them to actual files, and provides rules connecting the specifications. Maia supports multi-file verifiers by adding extensions via the `on` keyword as follows:

```
using "sysPath1" on "sysPath2" ;
```

In the example below, the Maia specification `groupfile.maia` is linked to the file `/etc/group`:

```
using "groupfile.maia" on "/etc/group" ;
```

Thus, as part of the current verification, the file `/etc/group` must also be verified using the `groupfile.maia` specification.

The first form of a **using** statement includes other Maia rules, M, to be applied to the the same file as the rules contained the current Maia specification. The SOS for the first form of **using** is

$$\frac{\gamma \vdash \textbf{using "sysPath"} \Rightarrow_e v_1 \qquad \gamma \vdash M \Rightarrow_e v_2}{\gamma \vdash \textbf{using "sysPath"} M \Rightarrow_e v_3} \tag{4.5}$$

, where $v_3 \in \texttt{AND}(v_1, v_2)$

For the second form of Maia is similar, except that the **using** statement is evaluated in a separate environment from the the main file. This leads to the following SOS

$$\frac{\gamma_1 \vdash \textbf{using "sysPath" on "sysPath"} \Rightarrow_e v_1 \qquad \gamma_2 \vdash M \Rightarrow_e v_2}{\gamma_1, \gamma_2 \vdash \textbf{using "sysPath" on "sysPath"} M \Rightarrow_e v_3} \tag{4.6}$$

4.3.2. Syntax Rules

Maia syntax rules are an EBNF specification of input file syntax. A Maia translator can emit a specification in any parser generator system to read a file. The names that appear

on the left hand side of a syntax rule represent containers that contain the strings that match that rule in the input file. Thus, syntax rules define variables that are used later in constructing containers and in verifying the properties of the constructed containers. In Maia, items in a container are considered to be ordered based on their original order in the file being verified.

Let $G = (V, \Sigma, P, S)$ be a context-free grammar where V is a container of variables or non-terminals, Σ is the alphabet or container of terminals, P is a set of rules and S is a distinguished element of V called the start symbol [42]. Let n_i be a node in the derivation tree, T, for the derivation $S \stackrel{*}{\Rightarrow} w$, where $w \in \Sigma^*$, and $n_j, \ldots, n_k$ be the children of n_i. We denote the string derived from n_i as $\delta(n_i)$. $\delta(n_i)$ is defined recursively as

1. If n_i is a leaf node, then $\delta(n_i) = label(n_i)$
2. If n_i is an interior node, then $\delta(n_i) = \delta(n_j) \cdot \ldots \cdot \delta(n_k)$

Let $A, B \in V$. We denote the container of strings derived from A as $\Delta(A)$. $\Delta(A) = \{\delta(n) \mid n \in T \wedge label(n) = A\}$. In addition, we define $\Delta(A.B)$ as

$$\Delta(A.B) = \left\{ \delta(n) \;\middle|\; \begin{array}{l} n \in T \;\wedge\; label(n) = B \wedge \\ label(parent(B)) = A \end{array} \right\} \tag{4.7}$$

This second form is used when referring to strings derived in the context of a specific rule.

A syntax rule, X, has the form: $N = xEy$ where $N, E \in V$ and $x, y \in \Sigma^* \cup V$. The SOS of a syntax rule expressed in the context of a semantic rule, S_R, in Maia is

$$\frac{\begin{array}{c}\gamma \vdash N{=}\, xEy \Rightarrow_e (\Delta(N), \Delta(N.E)) \\ \gamma[N \mapsto \Delta(N), N.E \mapsto \Delta(N.E)] \vdash S_R \Rightarrow_e v\end{array}}{\gamma \vdash N{=}\, xEy\ S_R \Rightarrow_e v} \tag{4.8}$$

where $v \in B$ and B is a boolean value indicating a file is valid (`true`) or invalid (`false`). Essentially, syntax rules create a new mapping from the name appearing on the left hand side of an EBNF rule to the container of strings that are matched in an input file.

For example, we can state the rule `passwdRecord` to specify a record in `/etc/passwd` as:

```
passwdRecord = name ":" password ":" uid ":"
               gid ":"
```

If this rule is applied to the input:

```
alice: 19fd01b2307d497fb174decd8bc9c121:1000:1
bob: 0f68eb4c87c99c563e168cdc2cd92336:200:2
```

the constructed containers from this input are:

```
passwdRecord = {{alice,19fd..., 1000, 1 },
               {bob,0f68..., 200, 2 }}
passwdRecord.name = {alice, bob}
passwdRecord.password = {19fd..., 0f68...}
passwdRecord.uid = {1000,200}
passwdRecord.gid= {1,2}
```

Containers in Maia syntax rules are constructed automatically. Each container is converted into a simple or a compound container containing the input chunks that matched the parser rule. Maia syntax phase constructs simple containers by grouping all the occurrence of the same nonterminal together. The scope of the definition of S is limited to the occurrences

of the same variables in the expression as follow: Lets suppose that S occurs n times, $\{S_1,S_2,\ldots,S_n\}$. Then, we can define simple container S as follows:

$$let\ S =^{def} \{S_1, S_2, .., S_n\} \tag{4.9}$$

The SOS of a simple container definition S is:

$$\frac{(\gamma \vdash S_1, \gamma \vdash S_2, \ldots, \gamma \vdash S_n) \Rightarrow_e \{v_1, v_2, \ldots, v_n\}{=}v}{\gamma \vdash let\ S =^{def} \{S_1, S_2, \ldots, S_n\} \Rightarrow_d \gamma[S \mapsto v]} \tag{4.10}$$

The scoping definition of simple container $S, \Rightarrow_d$ returns a new environment with the additional mapping.

Alternatively, Maia constructs compound containers when nonterminals contain at least two other nonterminals. The scope of the definition of S is limited to the occurrences of the different variables in the expression as follows: Lets suppose that S is a compound container that has n simple cos $\{S_1,S_2,\ldots,S_n\}$. Each simple container S occurs n times such that $\{S_{1,1}, \ldots, S_{1,n}, S_{2,1}$

$, \ldots, S_{2,n}, S_{n,1}, \ldots, S_{n,n}\}$. Then, we can define the compound container S as follows:

$$\begin{aligned} let\ S =^{def} \{&\{S_{1,1}, S_{2,1}, \ldots, S_{n,1}\}, \\ &\{S_{1,2}, S_{2,2}, \ldots, S_{n,2}\}, \\ &\ldots, \{S_{1,n}, S_{2,n}, \ldots, S_{n,n}\}\} \end{aligned} \tag{4.11}$$

The SOS of compound container S is:

$$\frac{\begin{bmatrix} \{\gamma \vdash S_{1,1}, \gamma \vdash S_{2,1}, \ldots, \gamma \vdash S_{n,1}\}, \\ \{\gamma \vdash S_{1,2}, \gamma \vdash S_{2,2}, \ldots, \gamma \vdash S_{n,2}\}, \ldots, \\ \{\gamma \vdash S_{1,n}, \gamma \vdash S_{2,n}, \ldots, \gamma \vdash S_{n,n}\} \end{bmatrix} \Rightarrow_e \begin{bmatrix} \{v_{1,1}, v_{2,1}, \ldots, v_{n,1}\}, \\ \{v_{1,2}, v_{2,2}, \ldots, v_{n,2}\}, \ldots, \\ \{v_{1,n}, v_{2,n}, \ldots, v_{n,n}\} \end{bmatrix} = v}{\gamma \vdash let\ S =^{def} \begin{bmatrix} \{\{S_{1,1}, S_{2,1}, \ldots, S_{n,1}\}, \\ \{S_{1,2}, S_{2,2}, \ldots, S_{n,2}\}, \\ \ldots, \{S_{1,n}, S_{2,n}, \ldots, S_{n,n}\}\} \end{bmatrix} \Rightarrow_d \gamma[S \mapsto v]} \tag{4.12}$$

4.3.3. Container Construction

Container construction in Maia may be done explicitly. Elements are specified as a comma-separated list of either strings or numbers. Constructed containers are available to semantics rules. As in the case of syntax rules, Container construction rules create a mapping from the container name to the elements of the container. In Maia syntax explicitly constructed containers have the form:

$$\begin{aligned} C \quad &\rightarrow \quad Var \texttt{ = } < Str_1,\ \ldots,\ Str_n > \texttt{;} \\ &\mid \quad Var \texttt{ = } < Nval_1,\ \ldots,\ Nval_n > \texttt{;} \end{aligned} \tag{4.13}$$

where *Var* is a variable name, *Str* is a string literal and *nVal* is a numeric value. An example of explicit container construction is

```
classification = < "TS", "S", "C", "UC" > ;
version = < 1, 2, 3.0, 3.1, 3.2> ;
```

The SOS of the explicit construction of a container of strings is:

$$\frac{\gamma \vdash \mathit{Var} \texttt{=} < \mathit{Str}_1, \ldots, \mathit{Str}_n > \Rightarrow_e \{\mathit{Str}_1, \ldots, \mathit{Str}_n\}, \quad \gamma[\mathit{Var} \mapsto \{\mathit{Str}_1, \ldots, \mathit{Str}_n\}] \vdash S_R \Rightarrow_e v}{\gamma \vdash \mathit{Var} \texttt{=} < \mathit{Str}_1, \ldots, \mathit{Str}_n > S_R \Rightarrow_e v} \tag{4.14}$$

This rule indicates that the container name *Var* maps to the literal container elements specified when evaluating a container of semantic rules S_R. The SOS for containers of numeric values is similar.

Maia also provides a facility to create a new container by performing a per-element join operation on two or more existing containers. The containers to be joined are required to contain the same number of elements of the same type (string or numeric value). Attempting to join mismatched containers is considered an error.

To join containers, we reuse the angle brackets to indicate containers construction, though in this case we specify how to construct an element rather than all elements in the container. For string-based fields, the connector is a period to indicate concatenation, in the style of Perl's dot operator. For example,

```
 user = <'a', 'b', 'c' >
domain = <'D1', 'D2', 'D3'>
userDomain = < user . domain >
```

results in the container `userDomain` mapping to the value `{'aD1', 'aD2', 'aD3'}`.

The SOS of container join for strings is

$$\frac{\begin{array}{c}\gamma \vdash \mathit{Var} \texttt{=} < A_1 \texttt{ . } A_2 \texttt{. } \ldots \texttt{. } A_n > \Rightarrow_e \\ \{a_{1,1} \cdot \ldots \cdot a_{n,1}, \ldots, a_{1,m} \cdot \ldots \cdot a_{n,m}\}, \\ \gamma[\mathit{Var} \mapsto \{a_{1,1} \cdot \ldots \cdot a_{n,1}, \ldots, a_{1,n} \cdot \ldots \cdot a_{n,m}\}] \vdash S_R \Rightarrow_e v\end{array}}{\gamma \vdash \mathit{Var} \texttt{=} < A_1 \texttt{. } A_2 \texttt{. } \ldots \texttt{. } A_n > S_R \Rightarrow_e v} \tag{4.15}$$

where $A_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,m}\}$. The SOS of container join for numeric containers is similar.

A Maia specification is executed in two passes. The first pass consists of the rules in Sections 4.3.2 and 4.3.3 to create the environment in which the semantic rules given in the next section are evaluated. The second pass verifies the constraints placed on the file contents expressed by semantic rules.

4.3.4. Semantic Rules

Structurally, Maia semantic rules are a straightforward adaptation of predicate calculus. For example, consider the rule "there must be at least one user with a UID of 0" that may be placed on `/etc/passwd`. Given the container *uid*, which contains the UIDs of all users in `/etc/passwd`, we may express this formally as: $\exists u \in uid : u == 0$. The equivalent Maia is quite similar:

```
exists u in uid : u == 0;
```

A Maia semantic rule has the following syntax:

$$S_R \quad \rightarrow \quad quantifiers \texttt{ : } C_n \texttt{ ;} \tag{4.16}$$

$$\begin{array}{rcl} quantifiers & \rightarrow & quantifier\ ,\ quantifiers\ \textbf{;} \\ & | & quantifier\ \textbf{;} \end{array} \tag{4.17}$$

$$\begin{array}{rcl} quantifier & \rightarrow & E?\ \textbf{forEvery}\ Var_1\ \textbf{in}\ Var_2\ \textbf{:}\ C_n\ \textbf{;} \\ & | & E?\ \textbf{exists}\ Var_1\ \textbf{in}\ Var_2\ \textbf{:}\ C_n\ \textbf{;} \\ E & \rightarrow & \textbf{(require)} \mid \textbf{(warn)} \mid \textbf{(info)} \end{array} \tag{4.18}$$

where E is an enforcement level, Var is a container name and C_n is a constraint on the container. The possible enforcement levels are **(require)** which means the constraint is always checked and input file is invalid if the constraint does not hold, **(warn)** which means the constraint is always checked and a warning is issued if the constraint does not hold and **(info)** which means the constraint is only checked if requested and a warning message is given if the constraint does not hold. Below, we give the SOS for the **(require)** enforcement level, without loss of generality. The SOS for a **forEvery** rule is

$$\frac{\gamma \vdash Var_2 \Rightarrow_e A, \quad \forall a \in A\ \gamma[Var_1 \mapsto a] \vdash C_n \Rightarrow_e v}{\gamma \vdash \textbf{forEvery}\ Var_1\ \textbf{in}\ Var_2\ \textbf{:}\ C_n\ \textbf{;} \Rightarrow_e v} \tag{4.19}$$

where A is a container defined by a syntax rule or via container construction and $v \in \{\texttt{true}, \texttt{false}\}$.

This rule indicate that the constraint must hold on every element of the container A in order for the file to be valid. Similarly, the SOS for an **exists** rule is

$$\frac{\gamma \vdash \mathit{Var}_2 \Rightarrow_e A, \quad \exists a \in A\ \gamma[\mathit{Var}_1 \mapsto a] \vdash C_n \Rightarrow_e v}{\gamma \vdash \textbf{exists}\ \mathit{Var}_1\ \textbf{in}\ \mathit{Var}_2\ \textbf{:}\ C_n\ \textbf{;} \Rightarrow_e v} \tag{4.20}$$

This rule indicates that the constraint must hold for at least one member of the container A in order for the file to be valid.

A constraint, C_n, in Maia may be a logical comparison, an expression in predicate logic, or a logic constraint. Syntactically, constraints are of the form

$$\begin{aligned} C_n \quad &\rightarrow \quad C_n\ \mathit{logic}\ C_n \\ &\mid \quad \textbf{not}\ C_n \\ &\mid \quad \textbf{(}\ C_n\ \textbf{)} \\ &\mid \quad \mathit{Inc} \\ &\mid \quad \mathit{Blb} \\ &\mid \quad \mathit{Cmpr} \end{aligned} \tag{4.21}$$

C_n provides one or more Boolean constraints that will be evaluated for each element in the specifying container until the rule is satisfied. For example, a rule that applies the constraint on all elements of the container, `UIDs` must be in the range 0 to 32767. Maia semantic rule translates the given rule to:

```
forEvery u in uid : u >=0 and u <= 32767 ;
```

Maia includes the standard logical operators *and*, *or* , and *xor*. It also provides *implies* and *iff*. Logical operators allow rules like:

```
forEvery p in passwdRecord:
        p.name == "root" implies p.uid==0 ;
```

This rule is applied to the constraint `passwdRecord` to express the constraint that the root user must have `uid` equal to 0. The syntax `p.name` refers to the name field in every member of the constraint `passwdRecord`.

Since a constraint is a finite function that finally maps to a Boolean value = {`true`, `false`}. Without loss of generality, the SOS of $(C_{n_1}\ logic\ C_{n_2})$ is:

$$\frac{\gamma \vdash C_{n_1} \Rightarrow_e v_1 \quad \gamma \vdash C_{n_2} \Rightarrow_e v_2}{\gamma \vdash (C_{n_1}\ logic\ C_{n_2}) \Rightarrow_e v_3} \tag{4.22}$$

where,

$$\begin{aligned} logic \quad &\rightarrow \quad \texttt{IFF}\ (v_1, v_2) \\ &\mid \quad \texttt{IMPLIES}\ (v_1, v_2) \\ &\mid \quad \texttt{AND}\ (v_1, v_2) \\ &\mid \quad \texttt{OR}\ (v_1, v_2) \\ &\mid \quad \texttt{XOR}\ (v_1, v_2) \end{aligned} \tag{4.23}$$

and v_1, v_2, and $v_3 \in \{\texttt{true}, \texttt{false}\}$.

`not` C_n negates the constraint. For example, a constraint that requires a user `id` not greater than 32767 is expressed as follows:

```
exists u in uid: not(u > 32767) ;
```

Without loss of generality, the SOS of `not` C_n when C_n is true is:

$(\texttt{not}\ C_n) = v_1$

$$\frac{\gamma \vdash C_n \Rightarrow_e v_1}{\gamma \vdash \texttt{not}\,(v_1) \Rightarrow_e v_2} \tag{4.24}$$

where, v_1 is true and v_2 is false.

Container membership in Maia is specified with an `in` constraint. This constraint is `true` if and only if there is at least one element in a container being tested. The syntax of container membership semantic rules is :

$$\begin{aligned} Inc \;\; &\rightarrow \;\; indexedName\ \textbf{in}\ Var \\ &\mid \;\; indexedName\ \textbf{in} < string\,(,\ string)^* > \\ &\mid \;\; indexedName\ \textbf{in} < nVal\,(,\ nVal)^* > \end{aligned} \tag{4.25}$$

where $nVal$ in Maia defines numeric values and has the form:

$$nVal \;\; \rightarrow \;\; iVal \mid fVal \tag{4.26}$$

where $iVal$ is a decimal or hexadecimal integer value, and $fVal$ is a floating point value. The SOS of $nVal$ is:

$$\frac{\gamma \vdash iVal \Rightarrow_e iVal \;\; \gamma \vdash fVal \Rightarrow_e fVal}{\gamma \vdash nVal \Rightarrow_e v} \tag{4.27}$$

where $v \in \{iVal, fVal\}$

In *indexedName*, if an element has a numeric type, it can be compared it to a numeric literal, by applying a numeric operator, or concatenating it to a numeric value. In Maia ,

indexedName has the form:

$$\begin{array}{rcl} \mathit{indexedName} & \rightarrow & \mathit{Var}\ (\textbf{[}\ \mathit{exp}\ \textbf{]})?\ (\textbf{.}\ \mathit{Var})? \\ & | & \mathit{Var} \end{array} \tag{4.28}$$

For example, *indexedName* can access an element in a container name as `userDomain[i]`. The SOS of *indexedName* is defined as

$$\frac{}{\gamma \vdash\ \mathit{Var} \Rightarrow_e \gamma(\mathit{Var})} \tag{4.29}$$

$$\frac{}{\gamma \vdash\ \mathit{Var}.\mathit{Var} \Rightarrow_e \gamma(\mathit{Var}.\mathit{Var})} \tag{4.30}$$

$$\frac{\gamma \vdash\ \mathit{Var} \Rightarrow_e < u_1\ \textbf{.}\ u_2\textbf{.}\ \ldots\textbf{.}\ u_n >, \gamma \vdash \mathit{exp} \Rightarrow_e v}{\gamma \vdash\ \mathit{Var}\textbf{[}\ \mathit{exp}\ \textbf{]}\ \Rightarrow_e\ u_v} \tag{4.31}$$

where $v \in nVal$, and $\mathit{Var}, \mathit{Var}.\mathit{Var} \in dom(\gamma)$.

An example of a container membership in Maia, consider the container `disallowedCyphers` which contains `cyphers` that are not permitted under local policy. This rule can be stated as follows:

```
forEvery c in cypher:
     not (c in disallowedCyphers) ;
```

The SOS of container membership is:

$$\frac{\gamma \vdash \left[\begin{array}{c} e_j \in (indexedName) \textbf{ in } (Var) \\ \mid e_j \in (indexedName) \textbf{ in } < string > \\ \mid e_j \in (indexedName) \textbf{ in } < nVal > \end{array}\right] \Rightarrow_e v}{\gamma \vdash if[]_{i=1}^{n} \left[\begin{array}{c} e_i \in (indexedName) \textbf{ in } (Var) \\ \mid e_i \in (indexedName) \textbf{ in } < string > \\ \mid e_i \in (indexedName) \textbf{ in } < nVal > \end{array}\right] \Rightarrow_e v} \tag{4.32}$$

where i, $j \in \{1, \ldots, n\}$, and $v \in \{\texttt{true}, \texttt{false}\}$.

Black-box verifiers *Blb* are external procedures or processes that can perform tasks not expressible in Maia.

```
Blb(verifier, Var, ...)
```

In this rule, `verifier` is the name of a black-box verifier known to the system, and the *Var* is one or more elements in the current context to pass to verifier. A black-box verifier receives one or more values and returns a single value.

A constraint C_n may also include comparisons and arithmetic operators. These operations have straightforward semantics. Comparisons and arithmetic are of the form

$$\begin{array}{rcl} cmp & \rightarrow & indexedName\ cmp\ exp \\ & \mid & exp\ cmp\ indexedName \\ & \mid & indexedName\ cmp\ string \\ & \mid & string\ cmp\ indexedName \end{array} \tag{4.33}$$

For example, if we have a container of user names `name`, we can define constraint that `"root"` is the first element of the container as follow:

```
exists n in name : n == "root" ;
```

The SOS of *Cmp* has the form:

$$\frac{\left[\begin{array}{c} \gamma \vdash \mathit{indexedName} \Rightarrow_e v_1 \quad \gamma \vdash \mathit{exp} \Rightarrow_e v_2 \\ \gamma \vdash \mathit{exp} \Rightarrow_e v_1 \quad \gamma \vdash \mathit{indexedName} \Rightarrow_e v_2 \\ \gamma \vdash \mathit{indexedName} \Rightarrow_e v_1 \quad \gamma \vdash \mathit{string} \Rightarrow_e v_2 \\ \gamma \vdash \mathit{string} \Rightarrow_e v_1 \quad \gamma \vdash \mathit{indexedName} \Rightarrow_e v_2 \end{array}\right]}{\gamma \vdash \left[\begin{array}{c} \mathit{indexedName}\ \mathit{cmp}\ \mathit{exp} \\ \mathit{exp}\ \mathit{cmp}\ \mathit{indexedName} \\ \mathit{indexedName}\ \mathit{cmp}\ \mathit{string} \\ \mathit{string}\ \mathit{cmp}\ \mathit{indexedName} \end{array}\right] \Rightarrow_e v_3} \tag{4.34}$$

where,

$$\begin{array}{rcl} \mathit{cmp} & \rightarrow & \texttt{EQ}\ (v_1, v_2) \\ & | & \texttt{NEQ}\ (v_1, v_2) \\ & | & \texttt{LESS}\ (v_1, v_2) \\ & | & \texttt{LEQ}\ (v_1, v_2) \\ & | & \texttt{GTR}\ (v_1, v_2) \\ & | & \texttt{GEQ}\ (v_1, v_2) \end{array} \tag{4.35}$$

and $v_3 \in \{\texttt{true}, \texttt{false}\}$, v_1 and v_2 are either a numeric or a string value.

In Maia, expression elements, *exp*, are computed numerically using mathematical operators as presented in the nonterminal `mop`. The `dot` operator is used to concatenate the raw

input of elements or string literals. In Maia, an expression has the form:

$$\begin{aligned} exp &\rightarrow exp\ ops\ exp \\ &\mid \texttt{(}\ exp\ \texttt{)} \\ &\mid nVal \\ &\mid indexedName \\ &\mid Blb \end{aligned} \tag{4.36}$$

Maia allows three different type-values in numeric comparisons: Integer, Floating-point, or Mixed. Integer operations, *iVal ops iVal* results an integer value, except division which does not truncate and always returns a floating-point number. The semantics of floating point expression compassion (*fVal ops fVal*) operations follow the IEEE754 Standard. Finally, all operands in mixed arithmetic operations are converted to floating-point. The SOS of *exp* is:

$$\frac{\gamma \vdash exp_1 \Rightarrow_e v_1 \quad \gamma \vdash exp_2 \Rightarrow_e v_2}{\gamma \vdash (exp_1 \ \ ops \ \ exp_2) \Rightarrow_e v_3} \tag{4.37}$$

where,

$$\begin{aligned} ops &\rightarrow \texttt{EXP}\ (v_1, v_2) \\ &\mid \texttt{MULT}\ (v_1, v_2) \\ &\mid \texttt{DIV}\ (v_1, v_2) \\ &\mid \texttt{MOD}\ (v_1, v_2) \\ &\mid \texttt{ADD}\ (v_1, v_2) \\ &\mid \texttt{SUB}\ (v_1, v_2) \\ &\mid \texttt{DOT}\ (v_1, v_2) \end{aligned} \tag{4.38}$$

and v_1, v_2, and v_3 are numeric values.

4.3.5. Template Definition

Template definition aims to make Maia descriptions easier to create and maintain at the same time. Templates are inspired by C's macros, and work relatively in the same way. The template is defined with container name placeholder that will become the rule. It is required to involve one or more container name. In Maia, template definition has the form:

$$\begin{aligned} templateDef &\rightarrow \textbf{(template} \\ &\quad Var\ Var\ \textbf{(}\ Var^{*}\textbf{))} \\ &\quad replacementText\ \textbf{;} \\ \\ replacementText &\rightarrow S_R \mid C_n\ \textbf{;} \end{aligned} \tag{4.39}$$

When the template is encountered during specification evaluation, all placeholders are replaced with the provided container names, and the combination of provided names and the template name are replaced by the replacement text. For example, isUnique() template can be defined within a container called `name` as:

```
(template name isUnique())
        forEvery n in name: name[i] != name[j] ;
```

`isUnique()` template requires all user name in the container `name` are unique

```
name isUnique()
```

which becomes

```
forEvery n in name: name[i] != name[j] ;
```

when the specification is evaluated.

In semanticRule S_R template, template serves as a container, which provides context to the rule. Whereas, in constraint C_n template, the primary container name equivalents to a container provided by the current context.

In addition, templates may provide either completely quantified rules or individual, unquantified constraints. For example, isUnique() would be a complete rule, while isOwnedBy() is likely to be a constraint. For more templates, we refer the reader to [1].

The SOS of templates definitions is similar to that of syntax rules in that the introduce a new mapping in the environment. The SOS of a template definition is

$$\frac{\begin{array}{c}\gamma \vdash (\textbf{template}\ Var_0\ Var_1(Var_i^*))\ C_n \mid S_{R_1};\ \Rightarrow_e \\ (\textbf{template}\ Var_0\ Var_1(Var_i^*))\ C_n \mid S_{R_1}; \\ \gamma[Var_1 \mapsto (\textbf{template}\ Var_0\ Var_1(Var_i^*))\ C \mid S_{R_1};] \vdash \\ T^*\ C^*\ S_{R_2}^* \Rightarrow_e v\end{array}}{\begin{array}{c}\gamma \vdash (\textbf{template}\ Var_0\ Var_1(Var_i^*))\ C_n \mid S_{R_1}); \\ T^*\ C^*\ S_{R_2}^* \Rightarrow_e v\end{array}} \tag{4.40}$$

A reference to a template in a semantic rule (or container construction) requires that we introduce the small-step relation $\rightarrow_1 \subseteq Env \times Exp \times Exp$ between tow expressions, given an

environment. The small-step relation simplify one expression at a time rather than simplify to a value in one step.In addition, we need the notion of substitution. We write $e[e'/x]$ to denote the expression obtained by substituting all free occurrences of the variable x in the expression e with the expression e'.

Given these definitions, without loss of generality, we can define the SOS for template references as

$$\frac{\gamma \vdash \mathit{Var}_1 \Rightarrow_e (\textbf{template}\ \mathit{Var}_0\ \mathit{Var}_1(\mathit{Var}_i^*))\ S_{R_1};}{\gamma \vdash \mathit{Var}\ \mathit{Var}_1(\hat{Var}_i^*) \rightarrow_1 \mathit{Var}\ S_R[\hat{Var}_i/\mathit{Var}_i]} \tag{4.41}$$

4.4. Conclusion

Most integrity models deal with the trustworthiness of who accesses the data, or provide a general protection for a specific data format. We know of no general-purpose integrity systems capable of protecting the integrity of the data itself. Research on protecting arbitrary data integrity is limited. In this chapter, we have presented Maia, a language for general-purpose integrity protection. We give a formal description of the structural operational semantics for Maia using rules with simple mathematical foundations. The semantics leads to a natural interpretation of the meaning of a Maia specification.

CHAPTER 5

MAIA INTERPRETER IMPLEMENTATION

Maia Interpreter is designed to support many features of Maia language. In this chapter, we quantify Maia's impact on performance, and demonstrate that we can provide robust integrity guarantees without sacrificing usability.

This chapter is organized as follows: First, we present an overview of Maia Interpreter overall design and implementation. Then, we evaluate Maia's performance on selected structured files, and we demonstrate that we can provide robust data integrity without human intervention. Finally, we conclude this chapter by a brief discussion on Maia performance.

5.1. Benchmark Environment

Our experiments are conducted on a laptop computer equipped with an Intel Core i5 processor running at 1.4GHz, four gigabyte of RAM, and a 121.12GB hard drive. We implement Maia using ANTLR (version 4.7) and Eclipse-Mars.1 Release (4.5.1).

5.2. Maia Interpreter Overall Design

We implement an interpreter for the syntax and semantic phases of Maia without requiring human intervention. This implementation yields consistent performance as discussed in Chapter 4.

The tool relies on the availability of ANTLR [32]. Specifically, the tool is created using ANTLR (version 4.7) which generates the required files for our tool verifiers. Each phase has its own parser. We use a visitor pattern for walking the AST during each phase. In the

first phase visitor, some actions are added to construct containers to be used in the semantic phase. The semantic phase visitor has actions for verifying the given file/data.

The syntax checking phase of Maia starts by accepting a plain text file of a format that matches the existing file structure specification (`.g4`). For example, the Linux password file structure (syntax specification) in our tool might be named as `passwdFile.g4`. Then, the syntax-checking phase compares the input file `passwordFile` format against its structure as written in `passwdFile.g4`. If the `passwordFile` is syntactically correct, then the constructed containers will be passed to the second phase for semantic verification. The syntax phase generates error value when the structured file is syntactically incorrect. The semantic-checking phase validates the user input against the maia specification in a `.maia` file. If the user input is semantically correct, then Maia generates valid data, or otherwise it generates invalid data. In addition, an error value could be generated when the structure of user input does not match The Maia specification or the input is syntactically incorrect. Figure 5.1 presents the flowchart of overall design of Maia Interrupter with `passwordFile`.

5.3. Testing with the Password File

We use the Linux password file format, which is located at `/etc/passwd`, to evaluate the performance of our interpreter. Since the interpreter is implemented in Java, we are more concerned with performance relative to file size as opposed to absolute performance. The password file allows us to test a variety of Maia constraints. In fact, having a valid password file is crucial to the usability of a Linux computer system including the file's syntax specification (the file's records must be well formed), and the file's semantic specification (`uid` and `gid` must be between 0 and 65535, there must be a user named "root" that must have `uid` of 0, and handling uniqueness of usernames).

Verifier performance is tested first by adding three classes of errors in both the syntax

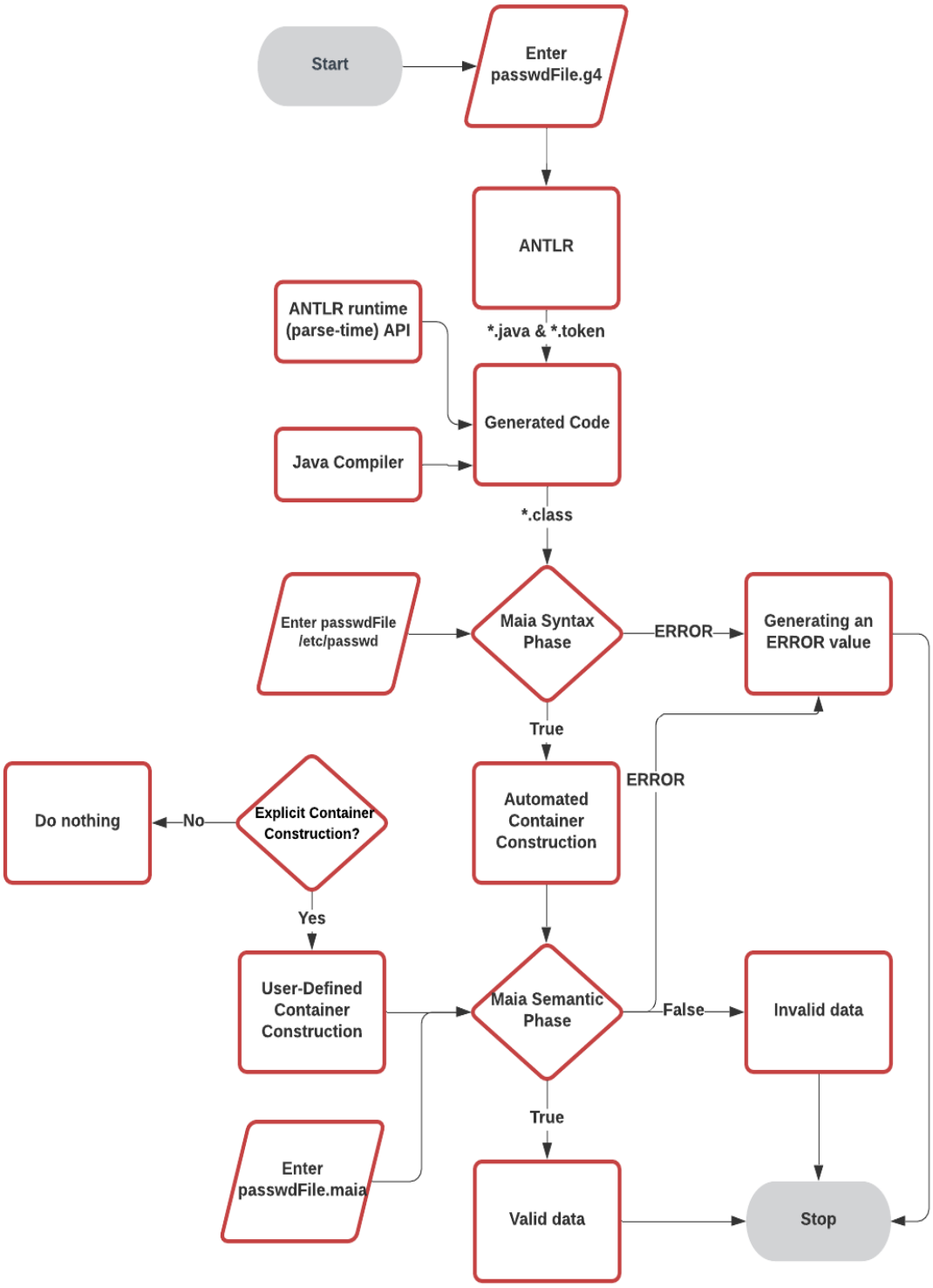

Figure 5.1: Flowchart of Maia Interpreter.

and semantic phases. In the syntax phase, a malformed record is added at the beginning of the file, in the middle, and at the end. Adding the invalid record at the beginning causes an early verification failure, while adding the malformed record in the middle causes a mid-parse verification failure. Finally, adding the invalid record at the end of the file causes an end of file verification failure. In the semantic phase, the three classes of errors are tested by causing a verification failure when the existential rule "there must be user named `root` is found to be violated with a varying number of records. We time how long it takes to open the structured file, to verify file's structure, to construct the file's containers, and to validate user input. This time is recorded at the granularity of a millisecond, and reported for further analysis.

We have obtained our source password file directly from Michigan Tech's NIS server, granting us access to 15786 total records. We generally apply our tests to the total number of records, which produce a total file size around a megabyte. The records contain usernames (full names) and all other information except password hashes.

5.3.1. Linux Password File Verifier Performance

Maia's verifier is constructed automatically by our software systems. The verifier validates the password file syntactically in the syntax phase and semantically in the semantic phase. In the syntax phase, the verifier exits upon finding an error, which may improve runtime in some cases. However, it is essential to parse the entire file to determine when it is valid so the structured file will be completely loaded to allow the syntactic elements to be constructed. Therefore, the syntax phase error free verification is likely to take longer time than file's semantic verification in most cases. We consider four verification cases for our testing: a clean, structured file (policy loaded), an invalid record at the start of the file, an invalid record in the middle, and an invalid record detected at the end of the file. Figure 5.2

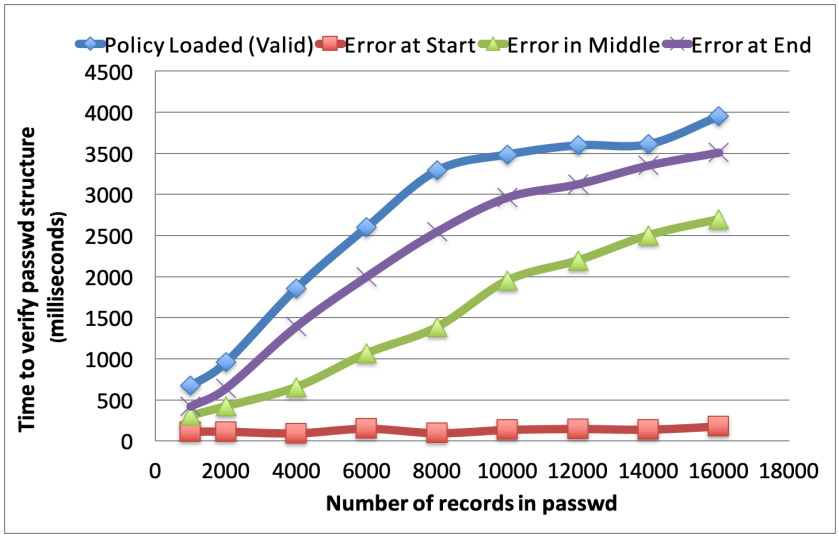

Figure 5.2: File Length versus Verification Time in Syntax Phase.

presents the syntax phase verification times for these cases.

It is obvious that Maia's syntax phase verifier is linear in file size. This follows validating the file structure and constructing containers from the syntactic elements of the `passwd` file. The syntax phase verifier is designed to exit as soon as it realizes a file is invalid. As a result, the verification time in the syntax phase varies with the error location. Loading the structured file takes the longest time because it requires scanning the entire file without any syntax error. However, an invalid record at the beginning of the file causes the Maia's syntax verifier to exit immediately, while the error at the end has a verification time close to the clean file resulting from scanning the entire file to detect the malformed record at the end of the file.

The longest verification time is observed for files that are clean or have a malformed record at the end, and is slightly less than 4000 milliseconds for a 15787 record file. When combined with the overhead from constructing containers there is an overall delay of less than 300 milliseconds.

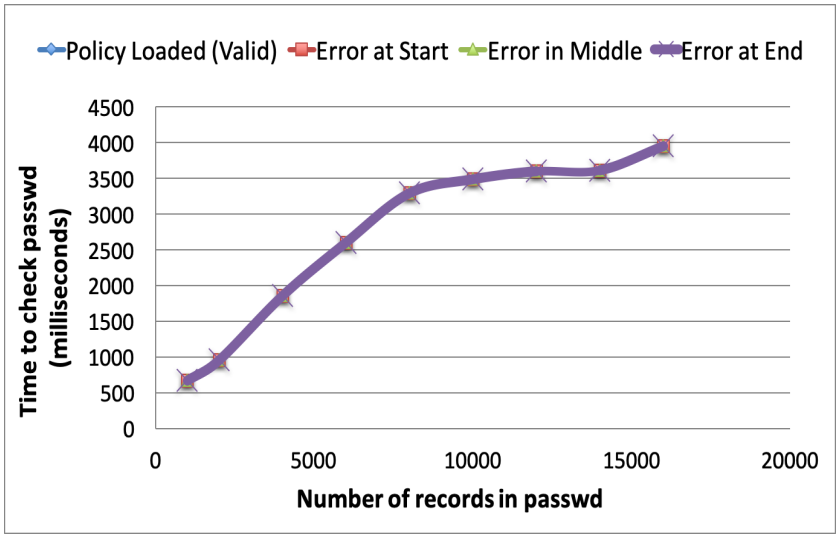

Figure 5.3: File Length versus Processing Time (Maia Verifier).

5.3.2. Baseline

Figure 5.3 presents the results of running the password verifier without the semantic phase. The valid policy is uploaded and verified for each file size. The results show that there is no variation between the different points. Only file length has an impact on verifying time, which is linear in the size of the file.

5.3.3. Linux Password File: Verifying Data Integrity

Maia's semantics phase verifier is designed to test multiple semantics rules that test the syntactic elements of the file against the semantic rules. We consider two cases. In the first verification test, we consider four cases: invalid data at start, invalid data in middle, invalid data at end, and valid data. The verification cases test the existential rule "there must be a user named `root` with a `userid` of `0`". We choose the existential rules that cause scanning the entire file in order to locate the matching data. Figure 5.4 presents the semantics phase verification times for these cases.

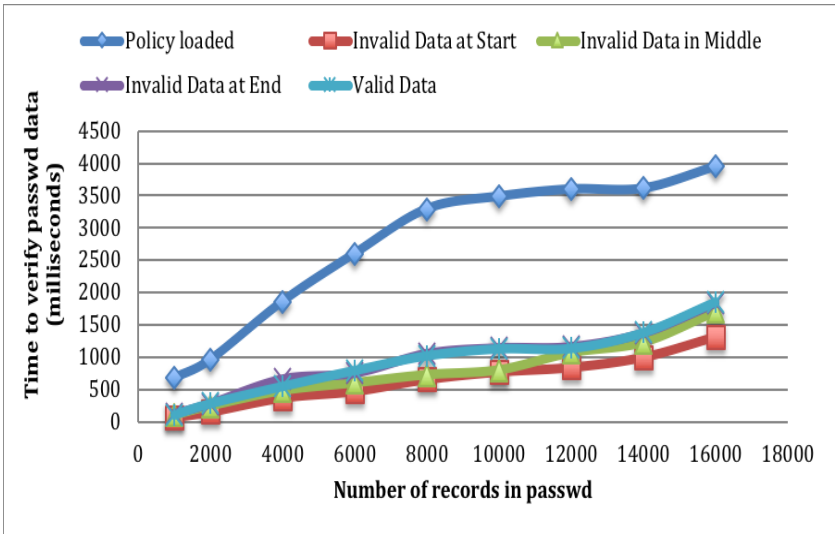

Figure 5.4: File Length versus Verification Time in Semantic Phase: Case 1.

The second verification case tests the existences of all `passwd` rules considering the last record. We consider two cases of testing, valid data and invalid data, for all password file rules. Figure 5.5 presents the semantics phase verification times for these two cases. The semantic-phase verifier is designed to test all the semantic rules in the `passwd.maia` file. For the second verification case, Maia loads the constructed containers that include the syntactic elements of the structured file. The semantics phase starts testing each semantic rules against Maia's specification and the constructed containers. The validity of the data in this test is approved when all the tested semantics rules are true; otherwise, Maia reports invalid data for the semantic rule.

Choosing the last record for different file size causes the existential rule to scan all containers for all `passwd` rules to find the violating data. Therefore, the verification times of valid and invalid data are similar.

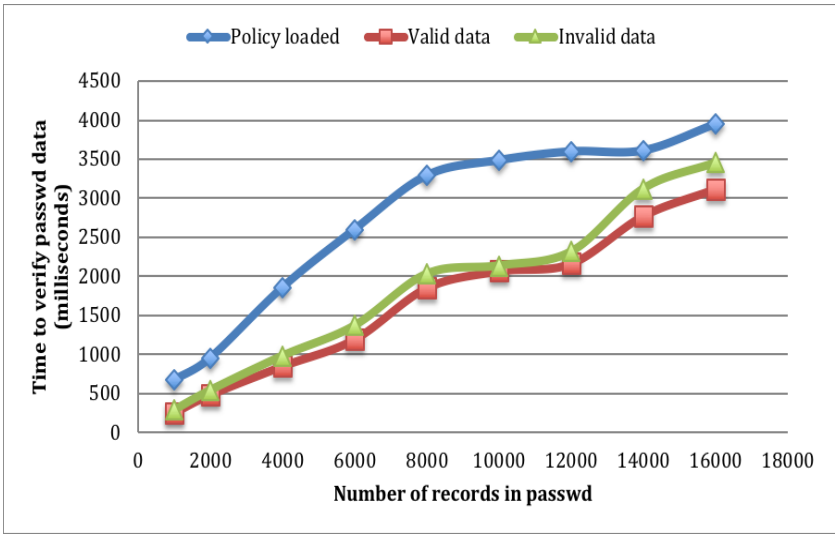

Figure 5.5: File Length versus Verification Time in Semantic Phase: Case 2.

5.4. Testing with SSH Configuration File

In addition to Password file, we test SSH Configuration file because it has a different structure. The OpenSSH SSH client configuration file provides a format that can be used at the system and per user level. The tested OpenSSH file is the default on most Linux Distributions and OS X. The format of the file includes:

1. Empty lines or comments (lines starting with '#'). Any configuration lines appearing before the first host configuration are applied globally. All the remaining configuration lines are considered part of the last host configuration.
2. Each line starts with a keyword, followed by argument(s).
3. Configuration options may be separated by white space or optional white space and exactly one.
4. Arguments may be enclosed in double quotes in order to specify arguments that contain spaces.

This specification is designed to illustrate protecting the key-value format used by SSH configurations. Configuration items can apply to any connection, or be grouped into different host configurations that can be applied in connecting to specific machines. The specification of the SSH configuration file allows us to test a variety of Maia constraints including a file's syntax specification (the file's records must be well formed), and the file's semantic specification (port must be between 0 and 65535), and (handling appropriate algorithms for ciphers and MACs). We have generated our source SSH configuration file with 10000 records that matches the specification of OpenSSH client configuration [43]. We generally apply our tests to the total number of hosts. The hosts contain all the information. We are only interested in testing valid ciphers, MACs, and ports.

Maia's verifier performance on SSH configuration file is tested first by adding three classes of errors in syntax phase as with Linux password file followed by the semantic phase. In the second phase, the three classes of errors are tested causing a verification failure when the universal rule "all hosts' ports must be between 0 and 65535". We time how long it takes to open the structured file, to verify file's structure, to construct file's containers, and to validate user input. This time is recorded at millisecond accuracy and is reported for further analysis.

5.4.1. SSH Configuration File Verifier Performance

The SSH verifier validates the SSH Configuration file syntactically in the syntax phase and semantically in the semantic phase. In the syntax phase, the verifier exits upon finding an error. The first phase is designed to parse the entire file to determine whether it is valid or not. After the validation process, the structured file will be loaded and the syntactic elements will be constructed. As with password file testing, we consider four verification cases for our testing: a clean file (policy loaded), an invalid record at the start of the file,

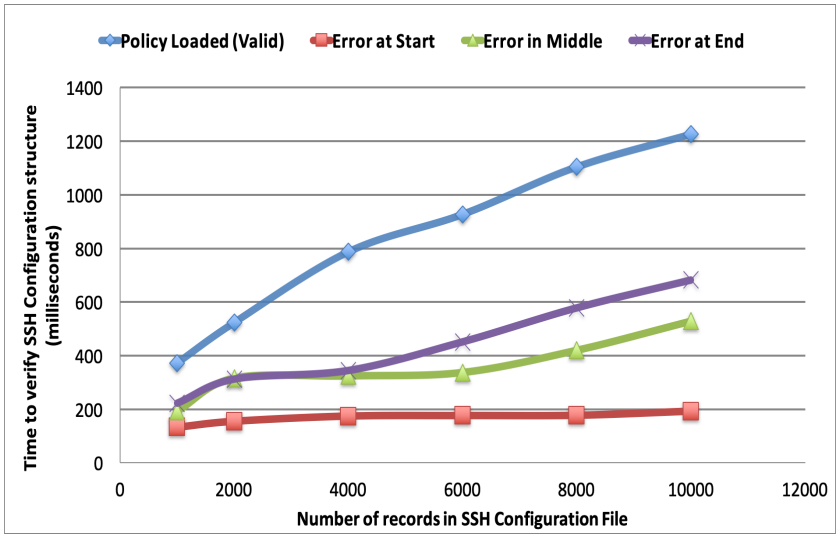

Figure 5.6: File Length versus Verification Time in Syntax Phase.

an invalid record in the middle, and an invalid record detected at the end of the file. Figure 5.6 represents the syntax phase verification times for these cases.

As with Linux password file, the verification time in syntax phase varies with error location. Loading the structured file takes longer because it requires scanning the entire file without any syntax error. Three classes of errors are added to measure the time consumed for the first phase. An invalid record at the beginning of the file causes the syntax verifier to exit immediately, while the error at the end has verification time close to the clean file resulting from scanning the entire file to detect the malformed record at the end of the file. The longest verification time is observed for files that are clean or have a malformed record at the end, and is slightly less than 1200 milliseconds for a 10000 records. When combined with the overhead from constructing containers causes an overall delay of less than 600 milliseconds.

The baseline for this file is similar to the Linux password file. It tests the valid loaded policy in the syntax phase only.

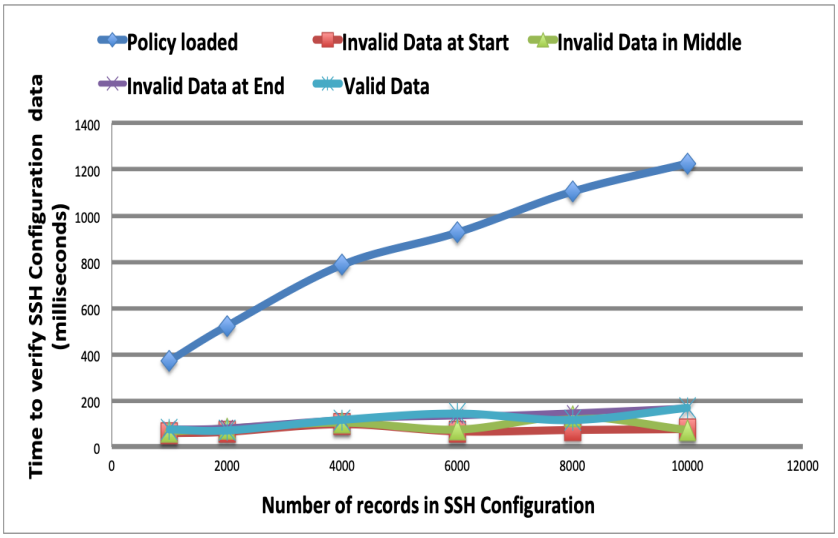

Figure 5.7: File Length versus Verification Time in Semantic Phase: Case 1

5.4.2. SSH Configuration File: Verifying Data Integrity

In the semantics phase, we consider four verification cases for our testing: invalid data at start, invalid data in middle, invalid data at end, valid data. In each case, we add a malformed record that causes data integrity violation in each file size. The verification cases test existential rule "there must be a cipher algorithm that is listed in the file specification and a port between 0 and 65535" with a data integrity violation with different numbers of records. We choose the existential rules that cause scanning the entire file size to find the matching data at specific location. Figure 5.6 presents the semantic-phase verification times for these cases.

In the second verification case, we test the existences of selected SSH Configuration rules by considering the last records in the most cases. Choosing the last record for different file size causes the existential rule scanning the entire containers for the selected SSH Configuration rules to find the matching data. Therefore, the verification times of valid and invalid data are close in the most tests. We consider two cases of testing: valid data and invalid data for

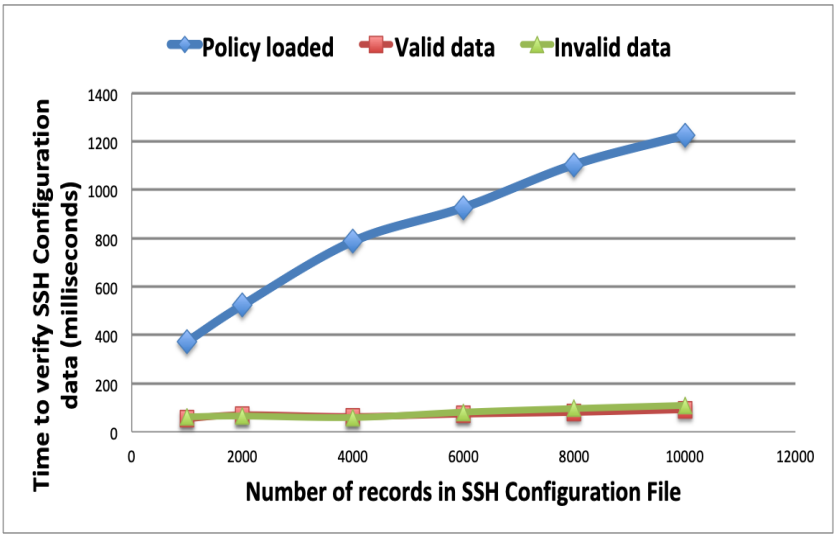

Figure 5.8: File Length versus Verification Time in Semantic Phase: Case 2

ciphers, Macs, and ports rules. Figure 5.8 presents the semantics phase verification times for these two cases.

The semantic-phase verifier for SSH Configuration file is designed to test selected semantics rules in sshConfig.maia file. The semantics phase starts testing each semantic rule against Maia's specification and the constructed containers. The validity of the data in this test is approved when all the tested semantics rules are true otherwise Maia generates invalid data for all semantics rules.

5.5. Conclusion

This chapter has presented the results of a couple of benchmarks on Maia-generated verifiers. We have also implemented an interpreter for Maia based upon the semantics that we have defined. The interpreter is fully automatic with linear performance in the size of the input data for the cases that we have tested. Maia itself is quite fast. Using Java as an implementation language adds extra milliseconds to the processing time since Java uses just-in-time (JIT) compilation. Running a program using a JIT compiler adds the cost of

the compiler producing bytecode to the actual cost of executing the program. Thus, running a Maia verifier increases processing time less than the the times reported in this chapter. In the worst case, total processing time is 4000 milliseconds to verify and parse a 15000 record password file. Constructing containers to be passed to the next phase added overhead that is more apparent on heavily loaded systems doing tight I/O loops on protected files.

CHAPTER 6

ADMONITA: A RECOMMENDATION-BASED TRUST MODEL FOR DYNAMIC DATA INTEGRITY

In this chapter, we propose a new recommendation-based trust model that is based on subjective logic and bi-directional weak tranquility. The recommendation-based trust model uses past behavior during interactions and information from other resources to determine whether to trust an entity. Our model adopts the rules from the Biba integrity model and incorporates recommendation opinions from Maia. In our trust model, integrity levels of subjects and objects are expressed as trust opinions. Since the security of the system is a subjective measure that depends on individuals who are qualified to express trust opinions, we define such trust opinions in the framework of subjective logic. We compute the recommendation values and the trust opinions with a simple algebra based on the trust metrics of our model. Also, we add a flexible definition of data integrity by using bidirectional weak tranquility.

We begin by presenting Admointa's fundamental principles. This is followed by a discussion of Admonita's design, metrics, details of its structure, and example for trustworthiness authentication. We conclude with a discussion of a trustworthiness design of using Admonita.

6.1. Subjective Logic

In this section, we use an artificial reasoning framework called subjective logic to express the levels of trust. Due to the lack of certainty about the degree of the trustworthiness of subjects and objects, we need to have opinions to measure the integrity of these subjects and objects. Subjective logic defines the term *opinion*, w, which expresses an opinion about the

trust level of subjects/objects [5], [7]. The opinion translates into degrees of trust, distrust as well as uncertainty, that represents the absence of both trust and distrust values. Let t, d, and u be *trust*, *distrust* and *uncertainty*, respectively, such that:

$$t + d + u = 1, \quad t, d, u \in [0, 1] \tag{6.1}$$

The opinion $w = \{t, d, u\}$ is a triplet satisfying (6.1). We use opinions to express trust levels. Having different levels of trust instead of a single level, such as in Biba, provides a better integrity model for real-world applications.

Subjective logic defines set logical operators that are equivalent to traditional logical operators, such as conjunction (AND), disjunction (OR), and negation (NOT), as well as some non-traditional operators that are used for combining opinions, such as recommendation and consensus. The expressed opinions are the input and output parameters for subjective logic operators. In this chapter, we define only consensus, recommendation and conjunctive operators.

Let A and B be two entities that represent observers who maintain the trust opinions of system resources, and let o be an object. When there are independent opinions about o, subjective logic defines a *consensus* operator to combine these independent opinions.

Let $w_o^A = \{t_o^A, d_o^A, u_o^A\}$ and $w_o^B = \{t_o^B, d_o^B, u_o^B\}$ be opinions held by the observers A and B, respectively, about o. According to subjective logic, the combined consensus opinion $w_o^{A,B}$

based on opinions w_o^A and w_o^B is defined as follows:

$$\begin{gathered} w_o^{A,B} = w_o^A \oplus w_o^B \\ = \{t_o^{A,B}, d_o^{A,B}, u_o^{A,B}\} \\ where, \\ \left\{ \begin{array}{c} t_o^{A,B} = (t_o^A u_o^B + t_o^B u_o^A)/(u_o^A + u_o^B - u_o^A u_o^B) \\ d_o^{A,B} = (d_o^A u_o^B + d_o^B u_o^A)/(u_o^A + u_o^B - u_o^A u_o^B), \\ u_o^{A,B} = (u_o^A u_o^B)/(u_o^A + u_o^B - u_o^A u_o^B) \end{array} \right\} \end{gathered} \tag{6.2}$$

Let A and B be two observers such that observer A invokes observer B to access an object o. Let $w_B^A = \{t_B^A, d_B^A, u_B^A\}$ be A's opinion about B's recommendation, and let $w_o^B = \{t_o^B, d_o^B, u_o^B\}$ be B's opinion about the trustworthiness of the object o. Subjective logic defines a *recommendation* operator to compute the indirect opinion w_o^{AB} based on opinions w_B^A and w_o^B as:

$$\begin{gathered} w_o^{AB} = w_B^A \otimes w_o^B \\ = \{t_o^{AB}, d_o^{AB}, u_o^{AB}\} \\ where, \\ \left\{ \begin{array}{c} t_o^{AB} = (t_B^A t_o^B) \\ d_o^{AB} = (t_B^A d_o^B), \\ u_o^{AB} = (d_B^A + u_B^A + t_B^A u_o^B) \end{array} \right\} \end{gathered} \tag{6.3}$$

Furthermore, subjective logic defines the *conjunctive* operator that expresses an opinion that is held by observer A about the trustworthiness of two distinct objects o_1 and o_2. Let $w_{o_1}^A = \{t_{o_1}^A, d_{o_1}^A, u_{o_1}^A\}$ and $w_{o_2}^A = \{t_{o_2}^A, d_{o_2}^A, u_{o_2}^A\}$ be observer A's opinions about o_1 and o_2. Then

the conjunction opinion $w^A_{o_1 \wedge o_2}$ of $w^A_{o_1}$ and $w^A_{o_2}$ is defined by:

$$\begin{gathered} w^A_{o_1 \wedge o_2} = w^A_{o_1} \wedge w^A_{o_2} \\ = \{t^A_{o_1 \wedge o_2}, d^A_{o_1 \wedge o_2}, u^A_{o_1 \wedge o_2}\} \\ where, \\ \left\{ \begin{array}{c} t^A_{o_1 \wedge o_2} = (t^A_{o_1} t^A_{o_2}) \\ d^A_{o_1 \wedge o_2} = (d^A_{o_1} + d^A_{o_2} - d^A_{o_1} d^A_{o_2}), \\ u^A_{o_1 \wedge o_2} = (t^A_{o_1} u^A_{o_2} + u^A_{o_1} t^A_{o_2} + u^A_{o_1} u^A_{o_2}) \end{array} \right\} \end{gathered} \tag{6.4}$$

In our model, we consider the opinion w_B, where $d_B < t_B$, to be more trustworthy than opinion w_A , where $d_A < t_A$, denoted $w_B \gg w_A$, if and only if $t_B > t_A$. When $t_B = t_A$, we consider higher uncertainty to be more trustworthy. In the case of $t_B = t_A \implies w_B \gg w_A \iff u_B > u_A$.

6.2. Tranquility for Dynamic Integrity Policy

In this section, we outline a tranquility principle that is trust-enhanced to protect trust levels of system entities and resources.

When a model, such as that of Oleshchuk, allows the trustworthiness of subjects to decrease due to reading low trusted data, the subject may become isolated from system resources since Biba's model incorporates only unidirectional weak tranquility. Such isolation can cause a violation of the security policy. For example, the *ls* Linux command is a command-line utility for listing the contents of a directory or directories given to it via standard input, and it writes to the standard output [44]. When *ls* accesses a corrupted directory/file, the trust level of *ls* will decrease. If *ls* continues accessing objects with low integrity levels, the trust level of *ls* may become isolated from system resources. To solve this issue, we apply the principle of weak tranquility such that the trust level may both increase

and decrease, making it bidirectional.

The tranquility principle allows controlled copying from high security levels to low security levels via trusted subjects. There are two forms of the tranquility principle: *strong tranquility* and *weak tranquility*. In strong tranquility, the security levels do not change during the normal operation of the system. In weak tranquility, the security levels may never change in such a way as to violate a defined security policy. Bidirectional weak tranquility is more desirable in our model. An entity may obtain a new low trust level due to accessing low integrity data or invoking low integrity entities. By applying bidirectional weak tranquilly, the entity can progressively accumulate higher trust levels, as actions require it. In other words, subjects and objects integrity levels will be managed within an allowable range to make the process more flexible in application. So, our model incorporates weak tranquility in a bi-directional manner where there are both maximum and minimum trusts levels that represent boundaries across which an object's integrity level may not change.

6.3. Admonita

Admonita is a recommendation-based trust model. It is based on Biba and Maia. In our proposed model, the Biba integrity model defines the subject-objects access properties, while Maia works as an Integrity Verification Procedure IVP that preserves data integrity. Basically, a Maia specification defines a set of constraints declaring what it means for data to have integrity. Maia verifies structured data when a subject writes to the file and generates a limited number of integrity levels to reflect the evaluation of the data's integrity.

The Biba integrity model is concerned with an unauthorized modification of data within a system by controlling who may access it. It works as a prevention system for data integrity. The model deals with a set of subjects, a set of objects, and a set of integrity levels. Subjects may be either users or processes. Each subject and object is assigned an integrity level,

denoted as $I(s)$ and $I(o)$, for the subject s and the object o, respectively. The integrity levels describe how subjects and objects are more or less trustworthy regarding a higher or lower integrity level.

Let $S = \{s_1, s_2, \dots\}$ be a set of subjects, and $O = \{o_1, o_2, \dots\}$ be a set of objects. According to subjective logic, the opinions about a subject and an object are expressed as $w_s = \{t_s, d_s, u_s\}$ and $w_o = \{t_o, d_o, u_o\}$ respectively, where $s \in S$ and $o \in O$. Therefore, the trust opinion about the subject w_s represents the integrity of the subject $I(s)$. Similarly, the trust opinion about the object w_o represents the integrity of the object $I(o)$.

According to [45], the definition of trust is "Anna trusts Bernard if Anna believes, with the level of subjective probability, that Bernard will perform a particular action, both before the action can be monitored (or independently of capacity of being able to monitor it) and in a context in which it affects Anna's own action." If Anna establishes trust in Bernard based on her observation and other interactions, the trust is *direct*. If it is established based on Anna's acceptance of Bernard's recommendation of other entities, then the trust is *indirect*.

Admonita combines *direct* and *indirect* opinions about the trustworthiness of subjects and objects. A security officer T expresses direct trust opinions about the subject s, denoted as w_s^T , and the trust opinions about the object o, denoted as w_o^T . Also, T maintains a list of minimum trust opinions for subjects, denoted as w_{s-min}^T, and list of maximum trust opinions for objects, denoted as w_{o-max}^T. Maia expresses the indirect subject-object trust opinion, denoted as $w_{s_o}^M$.

We incorporate the Biba model operations for both subjects and objects:

- Observe: Allows a subject s to read information in an object o, denoted as $read(s, o)$.
- Update: Allows a subject s to write or update information in an object o, denoted as $update(s, o)$.

- Invoke: Allows a subject s_1 to execute another subject s_2, denoted as $invoke(s_1, s_2, o)$.

The Biba model can be divided into two types of policies, mandatory and discretionary. Most literature on the Biba model refers to the model as being mandatory as a part of the strict integrity policy [46]. The Biba model defines a number of rules as part of the strict integrity policy. We reformulate each rule and compute the integrity level of subjects $I(s)$ and objects $I(o)$ as follows:

6.3.1. Simple Integrity Property

The Simple Integrity Property enforces *no-read-down*. It allows a subject to read (observe) an object only if the integrity level of the subject is less than the integrity level of the object.

$$s \in S \;\; reads \;\; o \in O \iff I(s) \leq I(o)$$

This ensures that high-integrity data cannot be directly contaminated by low-integrity data. For example, if the simple integrity rule is enforced, a low-integrity process may read high-integrity data, but it cannot contaminate itself by reading low-integrity data.

Our trust model adds a dynamic property to the Simple Integrity Property to allow high trust subjects to access low trust objects, however, the integrity of the subject may be lowered. The Simple Integrity Property in our model is reformulated as described below.

$$\begin{gathered} \forall s \in S, \forall o \in O: \;\; read(s, o) \iff \\ if \;\; I(s) > I(o) \;\; then \;\; I'(s) = I(s) \otimes I(o) \end{gathered} \tag{6.5}$$

When s reads o, denoted $read(s, o)$, the integrity of s may be changed by o, while the integrity of o will not be changed. If s reads less trusted data, then the integrity level of s after reading, denoted $I'(s)$, will decrease. To compute the indirect opinion $I'(s)$, denoted

as $w_{s_o}^{TM}$, let T and Maia be two observers such that the observer T invokes observer Maia to access an object o. Let w_M^T be T's opinion about Maia's recommendation, denoted as $w_{s\wedge o}^T$, and let $w_{s_o}^M$ be Maia's opinion about the trustworthiness of the object o that is accessed by the subject s. We adjust reasoning from (6.3) and (6.4) and we argue the applicability of *conjunction* and *recommendation* operators described in the previous section as follows:

$$\begin{aligned} w_{s_o}^{TM} &= w_{s\wedge o}^T \otimes w_{s_o}^M \\ &= (w_s^T \wedge w_o^T) \otimes w_{s_o}^M \end{aligned} \tag{6.6}$$

T expresses the direct trust opinions of w_s^T and w_o^T. These two opinions are combined using the conjunctive operator since they are both are assigned by the same observer. $w_{s\wedge o}^T$ represents T's opinion about Maia's recommendation. On the other hand, the Maia model expresses the indirect trust opinion about $w_{s_o}^M$. Thus, to compute the indirect opinion about the trustworthiness of $w_{s_o}^{TM}$ based on the trustworthiness of recommendation of Maia, the two opinions are combined using the *recommendation* operator. Equation (6.6) will decrease the integrity level of s when it reads less trusted o.

To prevent subject isolation, bidirectional weak tranquility is applied to ensure that the newly obtained trust level is within an allowable range. It is accomplished by comparing the new trust value $I'(s)$, denoted as t'_s, against its minimum value of trust, denoted as t_{s-min} as follows:

1. If the new trust value of the subject is greater than its minimum value such that $t'_s > t_{s-min}$, then read access will be granted.

2. If the trust values of a subject are equal such that $t'_s = t_{s-min}$, we consider higher uncertainty to be more trustworthy. In the case when $t'_s = t_{s-min}$ then $w'_s > w_{s-min}$ if $u'_s > u_{s-min}$.

3. If the new trust value of the subject is less than its minimum value such that $t'_s < t_{s-min}$ then the new integrity level of the subject violates the integrity policy. To solve this problem, we consider two cases:

   - If the subject is not allowed to have an integrity level less than its minimum integrity level, then read access will be denied and that reflects the *no-read-down* rule of the Biba integrity model.
   - If the subject is allowed to have an integrity level less than its minimum integrity level then the read access will be granted and the integrity level of the subject will be forced to go back to its previous integrity level to prevent subject isolation.

$$if \;\; t'_s < t_{s-min} \;\; then \;\; I'(s) = I(s) \tag{6.7}$$

6.3.2. Integrity Star Property

The second property of a Biba policy enforces *no-write-up*. It allows a subject to write an object only if the integrity level of the object is less than or equal to the integrity level of the subject.

$$s \in S \;\; updates \;\; o \in O \iff I(o) \leq I(s)$$

The Integrity Star Property in our model can be reformulated as follows:

$$\begin{gathered} \forall s \in S, \forall o \in O: \;\; update(s,o) \iff \\ if \;\; I(o) \leq I(s) \;\; then \;\; I'(o) = I(o) \oplus I(s) \end{gathered} \tag{6.8}$$

When s updates o, denoted $update(s,o)$, the integrity level of o, denoted as $I(o)$, with trust opinion w_o will be changed by the integrity level of s, denoted as $I(s)$, with trust

opinion w_s. If w_s is more trustworthy than w_o then the integrity level of the object after the update, denoted as $I'(o)$, will be increased and $I(s)$ will not change. In contrast, if w_s is less trustworthy than w_o then s will not be allowed to update o. This corresponds to the *no-write-up* rule of the Biba integrity model.

We consider two scenarios to enforce the *no-write-up* rule. First, when a high trust subject s updates a low trust object o invalidly, either accidentally or intentionally, Maia generates a lower recommendation opinion with a higher *distrust* value for accessing the object o. That lowers the integrity level of s by updating $I'(s)$ using (6.6). Then, $I'(s)$ is compared against $I(o)$ without updating $I'(o)$ with (6.8). With that, our model enforces *no-write-up*, and s will be denied to update o. After that, $I'(s)$ will be set using (6.7), to avoid isolation from the system resources.

The second scenario, occurs when a high trust subject s updates a low trust object in a valid format. In this case, the Maia generates a valid recommendation opinion for accessing the object o.

$$\begin{aligned} w_{s_o}^{T,M} &= w_{o\wedge s}^{T} \oplus w_{s_o}^{M} \\ &= (w_o^T \wedge w_s^T) \oplus w_{s_o}^{M} \end{aligned} \tag{6.9}$$

T expresses the direct trust opinions of w_o^T and w_s^T. These two opinions are combined using the conjunctive operator since they are both are assigned by the same observer. $w_{o\wedge s}^T$ represents T's opinion about Maia's recommendation. As with Simple Integrity Property, Maia expresses the indirect recommendation trust opinion about $w_{s_o}^M$. Since the Integrity Star Property in our model keeps s unchanged, we introduce a *conjunctive consensus* term $w_{s_o}^{T,M}$ that combines two independent opinions about accessing o. Equation (6.9) enforces increasing the integrity level of o when it is accessed by highly trustworthy s.

To prevent object isolation, bidirectional weak tranquility is applied to ensure that the

newly obtained trust level $I'(o)$ is within an allowable range. It is accomplished by comparing the new trust value of $I'(o)$, denoted as t'_o, against its maximum value of trust, denoted as t_{o-max}, as follows:

1. If the new trust value of the object is less than its maximum value such that $t'_o < t_{o-max}$ then update will be granted.

2. If the trust values of objects are equal such that $t'_o = t_{o-max}$, we consider higher uncertainty to be more trustworthy. In the case of $t'_o = t_{o-max}$ then $w'_o > w_{o-max}$ if $u'_o > u_{o-max}$.

3. If the new trust value of the object is greater than its maximum value such that $t'_o > t_{o-max}$ then the new integrity level of the object violates the integrity policy. To solve this problem, we consider two cases:

   - If the object is not allowed to have an integrity level greater than its maximum integrity level, then the update will be denied.
   - If the object is allowed to have an integrity level greater than its maximum integrity level, then the update access is granted and the integrity level of the object will be forced to go back to its previous integrity level to prevent object isolation.

$$if \;\; t'_o > t_{o-max} \;\; then \;\; I'(o) = I(o) \tag{6.10}$$

6.3.3. Invocation Property

In Biba's model, a subject may execute another subject at its own integrity level or below.

$$s_1 \in S \;\; invokes \;\; s_2 \in S \iff I(s_2) \leq I(s_1)$$

This last property states that a subject at one integrity level is prohibited from invoking (send/request messages for service) a subject at a higher level of integrity. The Invocation Property in our model is reformulated as follows:

$$\begin{aligned} &\forall s_1, s_2 \in S, \forall o \in O: \ \ read(s_1, s_2, o) \iff \\ &if \ \ I(s_1) < I(s_2) \ \ then \ \ I'(s_1) = I(s_1) \otimes (I(s_2) \otimes I(o)) \end{aligned} \tag{6.11}$$

In our trust model, when s_1 invokes s_2 to access o, it is denoted as $invoke(s_1, s_2, o)$. According to the strict integrity policy, our trust model requires $I(s_1) \geq I(s_2)$ preventing a less trustworthy s_1 using more trustworthy s_2 to update the data. This condition keeps $I(s_1)$ unchanged when s_1 reads lower integrity data o via s_2.

However, when $I(s_1) < I(s_2)$, the indirect opinion of $I'(s_1)$, denoted as $w^{TM}_{s_{1_{s_{2_o}}}}$, can be calculated using the conjunctive recommendation term as follows:

$$\begin{aligned} w^{TM}_{s_{1_{s_{2_o}}}} &= w^T_{s_1} \otimes (w^T_{s_2 \wedge o} \otimes w^M_{s_{2_o}}) \\ &= w^T_{s_1} \otimes ((w^T_{s_2} \wedge w^T_o) \otimes w^M_{s_{2_o}}) \end{aligned} \tag{6.12}$$

In Equation (6.12), the security officer T expresses the direct trust opinions of $w^T_{s_1}$, $w^T_{s_2}$ and w^T_o. The opinions of $w^T_{s_2}$ and w^T_o are combined using the conjunctive operator since they are both assigned by the same observer. Maia expresses the indirect trust opinion $w^M_{s_{2_o}}$. Then the indirect recommendation opinion of $w^{TM}_{s_{2_o}}$ is combined with $w^T_{s_1}$ using the recommendation operator. As a result, the integrity level of s_1 decreases due to using highly trusted resources.

The proposed trust opinion calculations associated with *no-read-down* and *no-write-up* operations along with a dynamic range of integrity levels give our model more flexibility and better control of integrity violations.

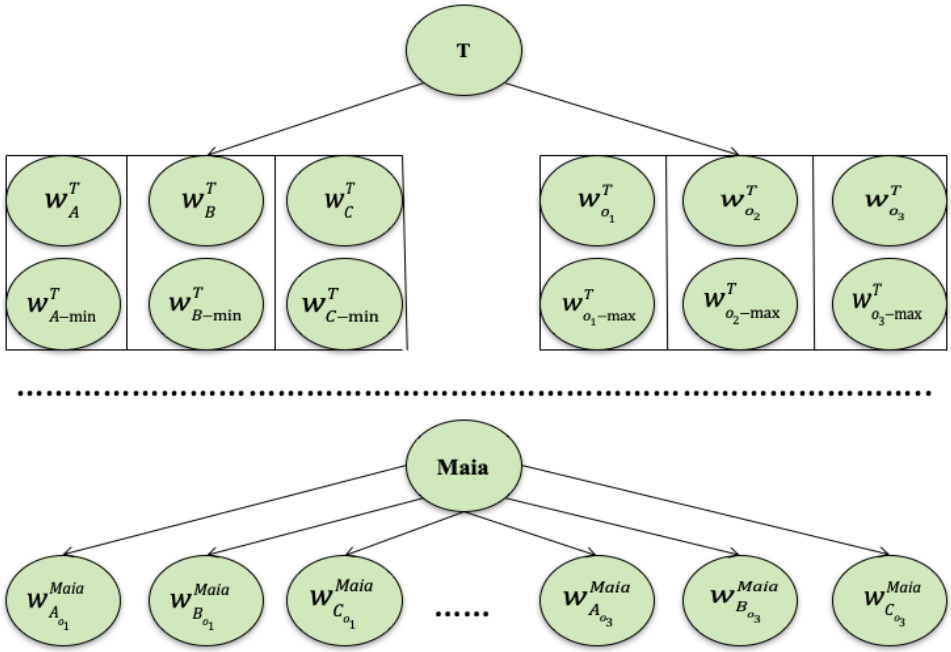

Figure 6.1: A Structure of Trustworthiness Authentication in Recommendation-Based Trust Model.

6.4. A Structure and Example for Trustworthiness Authentication

Figure 6.1 illustrates a possible structure for computing the integrity level as trust opinions about subjects/objects. The structure above the dotted line represents the opinions of the security officer T about subjects and objects as stored in $T's$ private database. Also, T maintains a list of minimum trust opinions for each subject s, denoted w^T_{s-min}, and list of maximum trust opinions for each object o, denoted w^T_{o-max}.

To ensure dynamic computation of trust opinions for system's entities, T assigns values of 0 and 1 for trusted subjects and objects, denoted T_s and T_o respectively. A value of 0 for a subject s does not allow s to obtain an integrity level less than its minimum integrity level. A value of 0 for an object o does not allow o to obtain an integrity level greater than its maximum integrity level. In contrast, a value of 1 for s allows s to have access to less

trustworthy data and return to its previous integrity level. Similarly with o, a value of 1 allows o to have an integrity level greater than its maximum, for updates, and return to its previous integrity value.

The integrity level of a subject s reflects how much T trusts s when accessing an object. It is assumed that T knows the trust levels of s. On the other hand, the integrity level of an object reflects $T's$ opinion about the trustworthiness of the data itself.

In our model, T must keep a list of her opinions, w_s^T and w_o^T, about the trustworthiness of subjects and objects, respectively. T's opinions about a subject reflects the trust level of the subject. However, T's opinions w_{s-min}^T about s ensure dynamic data integrity while the T's opinion about an object reflects the trust level about the data itself. Table 6.1 gives an example of possible opinion values.

Table 6.1: Security Officer's Opinions about Subjects' Trustworthiness

S	w_S^T	w_{S-min}^T	$T - Subject$
A	{1.00, 0.00, 0.00}	{0.99, 0.01, 0.00}	1
B	{0.98, 0.00, 0.02}	{0.85, 0.10, 0.05}	1
C	{0.88, 0.10, 0.02}	{0.80, 0.10, 0.10}	0

In order to enforce weak tranquility, T must maintain a list of her maximum opinions about objects w_{o-max}^T . Table 6.2 gives an example of possible opinion values. The structure below the dotted line represents a list of Maia trust recommendations $w_{S_o}^M$, based upon the Maia specification for that file, for each subject that wants to access the object o. Table 6.3 below gives an example of possible opinion values.

Assume subject B wants to read an object o_1, $read(B, o_1)$. Since $I(B)$ is greater than $I(o_1)$, the trust of B, denoted $I'(B)$, can now be calculated using (6.6):

Table 6.2: Security Officer's Opinions about Objects' Trustworthiness

O	w_O^T	w_{O-max}^T	$T - Object$
o_1	{0.90, 0.05, 0.05}	{1.00, 0.00, 0.00}	0
o_2	{0.96, 0.02, 0.02}	{0.96, 0.02, 0.02}	0
o_3	{0.98, 0.00, 0.02}	{0.98, 0.00, 0.02}	1

Table 6.3: Maia's Opinions about (Subject-Object) Trustworthiness

S_{o_1}	$w_{S_{o_1}}^M$
A_{o_1}	{0.95, 0.01, 0.04}
B_{o_1}	{1.00, 0.00, 0.00}
C_{o_1}	{0.89, 0.02, 0.09}

$$\begin{aligned} w_{B_{o_1}}^{TM} &= (w_B^T \wedge w_{o_1}^T) \otimes w_{B_{o_1}}^M \\ &= \{0.882, 0.00, 0.118\} \end{aligned}$$

Notice that the new integrity level of B, 0.882, is less than the old value 0.98 due to B reading object o_1 with a lower trust level than B. Also, the new $I'(B)$ satisfies (6.7) since it does not fall below its minimum trust value. Now, the integrity level of B in Table 6.1 will be replaced by the new value in order to prevent the low integrity of B from updating other objects in future interactions.

Suppose subject B wants to update object o_1, $update(B, o_1)$. Since $I(B)$ is greater than $I(o_1)$, the trust of o_1, denoted $I'(o_1)$, can now be calculated using (6.9):

$$\begin{aligned} w_{B_{o_1}}^{T,M} &= (w_{o_1}^T \wedge w_B^T) \oplus w_{B_{o_1}}^M \\ &= \{1.00, 0.00, 0.00\} \end{aligned}$$

Notice that the new integrity level of o_1, 1.00, is greater than the old value 0.90 since B has a higher trust value than o_1 . Also, $I'(o_1)$ satisfies (6.10) since it does not exceed the maximum trust opinion. Now, the integrity level of o_1 in Table 6.2 will be replaced by the new value.

Consider the case when a subject invokes another subject to access an object o. Assume B invokes A to access o_1. We need to modify the trust level of B for two reasons. First, the integrity level of B is lower than the integrity level of A, so the trust model will prevent B from using A. Second, subject A accesses less trusted data o_1. This decreases the integrity level of A.

To calculate the trustworthiness of B, $I'(B)$, first $I'(A)$ is calculated using (6.6) and (6.7) to let A obtain back its trust opinion since it is a trusted subject. Then $I'(B)$ is calculated using (6.12):

$$\begin{aligned} w^{TM}_{B_{A_{o_1}}} &= w^T_B \otimes ((w^T_A \wedge w^T_{o_1}) \otimes w^M_{A_{o_1}}) \\ &= \{0.8379, 0.00882, 0.15328\} \end{aligned}$$

The new integrity level of B, 0.8379, is less than the old value 0.98 and that is due to the Invocation Property. In addition, the new trust value of B violates the integrity policy since it is less than its minimum trusted value. However, B is a trusted subject. Therefore, our trust model allows B to read the less trusted object and obtain back its trust opinion. Now, the trustworthiness of B, denoted as $I'(B)$, can be calculated using (6.7):

$$if \;\; t'(B) < t_{B-min} \;\; then \;\; I'(B) = I(B)$$

$$I'(B) = \{0.98,\, 0.00,\, 0.02\}$$

If B is not a trusted subject, then the read access will be denied.

6.5. Conclusion

This chapter presents a new recommendation-based trust model for data integrity called Admonita. Admonita incorporates subjective logic, the Biba integrity model, the Clark-Wilson integrity model and the principle of bidirectional weak tranquility. Compared to previous models, our model adds the opinion of an IVP from Clark-Wilson of the integrity of the data that is a property of the data itself rather than the opinion of a trusted user. In addition, our model uses bidirectional weak tranquility to allow opinions about the integrity of data to change dynamically within a restricted range. The result is a model that determines the integrity of subjects and objects in a system that is not based solely on the integrity of the users in the system.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

The classic model for computer information security defines three objectives of security: confidentiality, availability, and integrity. Each objective addresses a different characteristic of providing protection for information. Confidentiality and availability are well understood with generally implemented safeguards. There are no general-purpose integrity systems. Research on protecting arbitrary structured data integrity is limited. In this work, we present the major weaknesses in existing data integrity approaches. Most integrity approaches deal with the trustworthiness of who accesses the data, provide a general protection for a specific data format, or cause data/subjects' isolation form the system resources.

To solve these problems, we present Maia, a language for general-purpose integrity protection. We give a formal description of the structural operational semantics for Maia using rules with simple mathematical foundations. The semantics leads to a natural interpretation of the meaning of a Maia specification. We have also implemented an interpreter for Maia based upon the semantics that we have defined. The interpreter is fully automatic with linear performance in the size of the input data for the cases that we have tested. In the future, we plan to move a full implementation of the Maia interpreter into MandOS [1] to verify file integrity in a complete system. Moreover, We plan to analyze the risk factors in Maia using logistic regression. The analysis can develop strategies and highlight problems, which are important issues to manage, control and reduce the risks of error.

In addition to Maia, we propose a new recommendation-based trust model for data integrity called Admonita. Admonita incorporates subjective logic, the Biba integrity model,

the Clark-Wilson integrity model and the principle of bidirectional weak tranquility. Compared to previous models, our model adds the opinion of an IVP from Clark-Wilson of the integrity of the data that is a property of the data itself rather than the opinion of a trusted user. In addition, our model uses bidirectional weak tranquility to allow opinions about the integrity of data to change dynamically within a restricted range. The result is a model that determines the integrity of subjects and objects in a system that is not based solely on the integrity of the users in the system.

In the future, we plan to implement Admonita in a real system and measure its performance. This will involve creating a high-performance compiler for Maia that utilizes its natural parallelism. The result will be a system that measures and maintains the trust levels for the applications and data contained within it, where the integrity of the data can be a range of values rather just 0 or 1.

BIBLIOGRAPHY

[1] P. Bonamy, S. Carr, and J. Mayo, "Toward a mandatory integrity protection system", in *Proceedings of the Thirty-first International Conference on Computers and Their Applications*, 2016.

[2] G. D. Plotkin, *A structural approach to operational semantics*, 1981.

[3] P. J. Bonamy, "Maia and mandos: Tools for integrity protectionon arbitrary files", PhD thesis, Michigan Technological Univeristy, 2016.

[4] V. Oleshchuk, "Trust-enhanced data integrity model", in *2012 IEEE 1st International Symposium on Wireless Systems (IDAACS-SWS)*, Offenburg, Germany, 2012, pp. 109 –112.

[5] A. Jøsang, "An algebra for assessing trust in certification chains", in *Proceedings of the Networks and Distributed Systems Security (NDSS'99)*, San Diego, CA, 1999.

[6] A. Jøsang, "A logic for uncertain probabilities", *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 9, no. 3, 279 — 311, Jun. 2001.

[7] A. Jøsang, "The consensus operator for combining beliefs", *Artificial Intelligence Journal*, vol. 142, no. 1-2, pp. 157 –170, Oct. 2002.

[8] W. Gao, G. Zhang, W. Chen, and Y. Li, "A trust model based on subjective logic", in *Proceedings of the Fourth International Conference on Internet Computing for Science and Engineering*, Harbin, China, 2009, pp. 272 –276.

[9] K. J. Biba, "Integrity considerations for secure computer systems", The MITRE Corporation, Tech. Rep., 1977.

[10] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies", in *IEEE Symposium of Security and Privacy*, 1987, pp. 184 –194.

[11] D. E. Bell and L. J. L. Padula, "Secure computer systems: Mathematical foundations", The MITRE Corporation, Bedford, MA, Tech. Rep. MTR-2547, Nov. 1973.

[12] M. Bishop, *Computer Security: Art and Science*, Second. Pearson Education Inc., 2019.

[13] Removed, "To protect the anonymity of the author", in *A conference*.

[14] S. N. R. R. D. Puthal X. Wu and J. Chen, "Seen: A selective encryption method to ensure confidentiality for big sensing data streams", in *in IEEE Transactions on Big Data*, vol. PP.

[15] J. R. Y. Liu and S. Rizvi, "Ensuring data confidentiality in cloud computing: an encryption and trust-based solution", *23rd Wireless and Optical Communication Conference (WOCC)*, pp. 1–6, 2014.

[16] R. Karimi and M. Kalantari, “Enhancing security and confidentiality on mobile devices by location-based data encryption”, in *17th IEEE International Conference on Networks*, 2011, pp. 241–245.

[17] R. S. Sandhu and P. Samarati, “Access control: Principle and practice," in ieee communications magazine”, 9, vol. 32, 1994, pp. 40–48.

[18] S. S. E. Nemeth G. Snyder and T. Hein, “Unix system administration handbook,third ed.”, Prentice Hall, 2000.

[19] D. L. S. K. M. W. L. Badger D. F. Sterne and S. A. Haghighat, “Practical domain and type enforcement for unix”, CA, 1995, pp. 66–77.

[20] E. Gelbstein, *Data Integrity-Information Security's Poor Relation*. Information Systems Audit and Control Association Journal, 2011, pp. 20–25.

[21] J. Banks, “The heartbleed bug: Insecurity repackaged, rebranded and resold”, vol. 11, no. 3, 259–279, [Online]. Available: `https://doi.org/10.1177/1741659015592792`.

[22] *Trusted computer system evaluation criteria department of defense, computer security center*, 1985.

[23] M. Blaze, J. Feigenbaum, and A. D. Keromytis, “Trust management for public –key infrastructures”, in *Proceedings of the Ninth International Workshop on Services Computing (Lecture Notes in Computer Science 1550)*, Apr. 1998, pp. 59 –63.

[24] M. Blaze, J. Feigenbaum, and J. Lacy, “Decentralized trust management”, in *Proceeding of 1996 IEEE Symposium on Security and Privacy*, May 1996, pp. 164 –173.

[25] A. Abdul-Rahman and S. Hailes, “A distributed trust model”, in *Proceeding of the 1997 Workshop on New Security Paradigms*, Sep. (1997, pp. 48 –60.

[26] W3C, “Extensible Markup Language (XML) 1.0 (Fifth Edition)”, Tech. Rep., Oct. 2008. [Online]. Available: `http://www.w3.org/TR/REC-xml/`.

[27] ——, “W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures”, Tech. Rep., Apr. 2012. [Online]. Available: `http://www.w3.org/TR/xmlschema11-1/`.

[28] ——, “W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes”, Tech. Rep., 2012. [Online]. Available: `http://www.w3.org/TR/xmlschema11-2/`.

[29] E. van der Vlist, *RELAX NG*. O'Reilly Media, Dec. 2003, ISBN: 978-0-596-00421-7.

[30] M. E. Lesk and E Schmidt, “Lex - A Lexical Analyzer Generator”, Murray Hill, New Jersey, Tech. Rep. Computer Science Technical Report No. 39, Oct. 1975.

[31] S. C. Johnson, “Yacc: Yet Another Compiler-Compiler”, Murray Hill, New Jersey, Tech. Rep. Computing Science Technical Report No. 32, Jul. 1975.

[32] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2015.

[33] ISO, "Information technology - Computer graphics and image processing - Portable Network Graphics (PNG): Functional specification", Geneva, Switzerland, Tech. Rep. ISO/IEC 15948:2003 (E), Mar. 2004.

[34] T. Jim, Y. Mandelbaum, and D. Walker, "Semantics and algorithms for data-dependent grammars", *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, vol. 45, no. 1, pp. 417–430, Jan. 2010.

[35] K. Fisher, Y. Mandelbaum, D. Walker, K. Fisher, Y. Mandelbaum, and D. Walker, *The next 700 data description languages*. ACM, Jan. 2006, vol. 41.

[36] K. Fisher, Y. Mandelbaum, and D. Walker, "The next 700 data description languages", *Journal of the ACM*, vol. 57, no. 2, pp. 1–51, Jan. 2010.

[37] K. Fisher and D. Walker, "The PADS project", in *the 14th International Conference*, New York, New York, USA: ACM Press, 2011, p. 11.

[38] D. F. Ferraiolo and D. r. Kuhn, "Role-based access controls", 1992, pp. 554–563.

[39] W. T. Polk, "Approximating Clark-Wilson"Access Triples" with Basic UNIX Controls", in *Proceedings of the UNIX Security Symposium IV*, Oct. 1993, pp. 145–154.

[40] P. Loscoco and S. Samalley, "Integrating flexible support for security policies into linux operating system", 2001.

[41] S. Prasad and S. Arun-Kumar, *An introduction to operational semantics*, http://www.cse.iitd.ernet.in jiva/opsem.ps, 2003.

[42] T. A. Sudkamp, *Languages and Machines: An Introduction to the Theory of Computer Science*. Pearson Education, 2006.

[43] S. Prasad and S. Arun-Kumar, *Ssh config file*, https://www.ssh.com/ssh/config.

[44] J. William E. Shotts, *The linux command line,* `https://wiki.lib.sun.ac.za/images/c/ca/TLCL-13.07.pdf`.

[45] D. Gambette, "Trust: Making and breaking cooperative relations", Basil Blackwell Ltd., Tech. Rep., 1988.

[46] N. Balon and I. Thabet, *The biba security model,* `https://pdfs.semanticscholar.org/7360/c680906617622f27ef2596c7efcc902795db.pdf`.