# Maia: A Language for Mandatory Integrity Controls of Structured Data

Wassnaa Al-Mawee[1], Paul J. Bonamy[2], Steve Carr[1] and Jean Mayo[3]
[1]*Department of Computer Science, Western Michigan University, 1903 W. Michigan Ave., Kalamazoo, MI 49008-5466, U.S.A.*
[2] *Department of Computer Science, Washington State University, 14204 NE Salmon Creek Ave., Vancouver, WA 98686, U.S.A.*
[3] *Department of Computer Science, Michigan Technological University, 1400 Townsend Dr., Houghton, MI 49931-1292, U.S.A.*

Keywords: Security, Structured Data Integrity, Structural Operational Semantics.

Abstract: The integrity of systems files is necessary for the secure functioning of an operating system. 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 so broad as to be useless except after the fact (e.g. backups). Often, file integrity is determined by who modifies the file or by a checksum. This paper focuses on a general model of the internal integrity of a file. 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. In this paper, we describe a language called Maia that provides a means to specify what the contents of a valid file should be. Maia can be used to specify the format and valid properties of system configuration files, PNG files and others. We give a structural operational semantics of Maia and discuss an initial implementation within a mandatory integrity system.



## 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 so broad as to be useless except after the fact (e.g. backups). 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 (Bonamy et al., 2016). 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 this paper, we give a Structural Operational Semantics (SOS) for Maia (Plotkin, 1981) and report on a preliminary implementation of a Maia compiler in the context of a mandatory integrity system (Bonamy, 2016). 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 paper is structured as follows. First, we give an overview of related work on file integrity. Then, we give an overview of Structural Operational Semantics. Next, we define Maia and give its SOS. Finally, we give a brief report on a preliminary implementation and present our conclusions.

## 2 RELATED WORK

Efforts have been made in the past to implement integrity systems using existing access control mechanisms. This includes an approximation of Clark-Wilson using Unix access controls (Polk, 1993), and



Al-Mawee, W., Bonamy, P., Carr, S. and Mayo, J.
Maia: A Language for Mandatory Integrity Controls of Structured Data.
DOI: 10.5220/0007344802570265
In *Proceedings of the 5th International Conference on Information Systems Security and Privacy (ICISSP 2019)*, pages 257-265
ISBN: 978-989-758-359-9

approaches relying on the fine-grained customizability of DTE(Ji et al., 2006). Access control can readily limit who may modify information, and may also be able to enforce restrictions on which processes can cause the changes. This provides excellent origin integrity, by restricting the source of changes. However, pure access control systems cannot directly address the problem of human error. Simply limiting who may modify information does not prevent erroneous edits. The only way to protect against such modifications would be to rely on purpose-built editors which will always make changes correctly. Thus, access control is not sufficient to address data integrity without infallible users or special software.

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 (W3C, 2008), XML Schema (W3C, 2012a) (W3C, 2012b), and RELAX NG (van der Vlist, 2003). 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 (Lesk and Schmidt, 1975) and Yacc (Johnson, 1975), and their successors Flex and Bison generate robust, fast parsers which can be embedded in C or C++ programs. ANTLR (Parr, 2015) 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 (ISO, 2004), 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 (Jim et al., 2010), are able to cope with limited context sensitivity, but still require programmer assistance to perform semantic checks.

Data description languages (Fisher et al., 2006) (Fisher et al., 2010) are designed to provide automated parsing for ad hoc data formats. Tools like PADS (Fisher and Walker, 2011) 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.

## 3 BACKGROUND

Structural Operational Semantics (SOS), introduced by G. D. Plotkin (Plotkin, 1981), 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 description 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 (Prasad and Arun-Kumar, 2003), SOS defines the semantics of a programming language from the syntax by applying the correct sequence of inference rules. Each rule has the form

$$\frac{\begin{array}{c} P_1, \\ P_2, \\ \ldots, \\ P_n \end{array}}{C}, \tag{1}$$

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*, $\gamma$, 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 $\gamma$, the syntax rule $R$ evaluates to a value $v$". This relation is understood as a transition that leaves $\gamma$ 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 MAIA

In order to ensure that data remains valid we must first validate the original file. There are a variety of special-purpose verifiers tied to particular use cases and file formats, but no general-purpose systems for verifying arbitrary file types. We solve this problem with Maia, a *single-assignment* specification language which can describe the structure of any context free language as well as semantic rules for the contents of a file.

## 4.1 Design Objectives

We have two primary objectives for the design of Maia. First, it must be able to protect a wide variety of existing files, which means we cannot restrict ourselves to only supporting certain file structures. Second, Maia specifications should be descriptions of valid files, rather than procedures for verifying files.

There are a huge number of configuration files in the average Linux system, to say nothing of all of the user-side file formats that may be on a system. While there are some repeated structures within these files, any system which focused on only one file structure would necessarily be unable to verify the remaining formats. With that in mind, we have designed Maia to be flexible with regard to the type of files it can describe. We also provide a mechanism to explicitly make use of external verifiers if necessary.

We have also designed Maia to provide implementation-independent descriptions of valid files, rather than procedures for verifying files. Any programming language could be used to write a verifier, but determining what constitutes a valid file would require not only reasoning about the rules themselves but also how they are implemented. By using a description system we can separate the meaning of rules from their implementation, which makes it easier to reason about them. It also becomes much easier to port specifications to different platforms, as all that is required is to recreate the verifier generator, not the specifications themselves.

## 4.2 Model Overview

Within Maia, we model the file verification process as two phases, corresponding to checking the file's syntax, followed by verifying the semantics. During the first phase, the file is parsed to check its structure and extract syntactic elements for processing. The second phase can then check the data in the syntactic elements without being unduly concerned about the file's structure. Using this (logically) two-phase system allows us to both mirror the way a traditional verifier would work and employ familiar constructs within the language itself.

The syntax checking component of Maia is designed to be familiar for anyone who has written a parser or perused the specification for a file or data format. The user provides an Extended Backus Naur Form (Backus, 1959)(Wirth, 1977) grammar which can then be used to break the file into pieces, verifying its structure and extracting meaningful components. We also provide some limited context sensitivity to allow syntax specifications to deal with files which contain length specifications.

The semantic portion of Maia makes use of set theory and predicate calculus to express constraints. The sets used in this phase are automatically constructed during syntax checking by grouping all occurrences of the same nonterminal (e.g. user names in the `passwd` file) into a set. It is then possible to express constraints like "user names must be unique" or "there must be a user named root" without needing to explicitly iterate over the data. This approach bears some resemblance to SETL (Dewar, 1979), though that family of languages is procedural rather than descriptive. In addition to normal set operations, we also provide a notion of ordering within sets to make it possible to express rules "root must be the first entry" or "users should be ordered by UID".

The next sections comprise a formal specification of the semantics of Maia. We have intentionally designed the syntax and semantic specification components of the language to be different from one another. This is reflective of the different underlying models for syntax and semantics, and has the advantage of making the type of a rule (syntax or semantic) obvious

with cursory inspection. The specification systems do occasionally share constructs or features, and we note those specifically. All other features are specific to either syntax or semantic specification and are not valid in the other context.

## 4.3 SOS for Maia

A Maia specification has the following basic structure:

$$M \rightarrow I^* \ (X \mid C \mid S_R \mid T)^* \tag{2}$$

where $I$ represents a file inclusion directive, $X$ is an EBNF specification of the input file syntax, $C$ represents a set 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. Maia specifications involve rules that have no meaning to present such as file inclusion and template definition. For file inclusion, we define its functionality. For templates, we refer the reader to (Bonamy, 2016). .

### 4.3.1 File Inclusion

Inclusion brings an existing specification into the current specification via the `using` keyword. 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:

$$\begin{array}{rcl} I & \rightarrow & \textbf{using } \texttt{"sysPath"} \textbf{ ;} \\ & \mid & \textbf{using } \texttt{"sysPath"} \textbf{ on } \texttt{"sysPath"} \textbf{ ;} \end{array} \tag{3}$$

where `sysPath` is the path to the specification to be imported. The path can be relative or absolute. 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. Maia supports multi-file verifiers by adding extensions via the `on` keyword as follows:

```
using "sysPath" on "filePath" ;
```

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 process, the file `/etc/group` must also be verified using the `groupfile.maia` specification.

### 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 sets that contain the strings that match that rule in the input file. Thus, syntax rules define variables that are used later in constructing sets and in verifying the properties of the constructed sets. In Maia, items in a set 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 set of variables or non-terminals, $\Sigma$ is the alphabet or set of terminals, $P$ is a set of rules and $S$ is a distinguished element of $V$ called the start symbol (Sudkamp, 2006). 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 set 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

$$\Delta(A.B) = \left\{ \delta(n) \;\middle|\; \begin{array}{l} n \in T \ \wedge \ label(n) = B \wedge \\ label(parent(B)) = A \end{array} \right\} \tag{4}$$

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

$$\frac{\begin{array}{c} \gamma \vdash N\textbf{=}\, xEy \Rightarrow_e (\Delta(N), \Delta(N.E)) \\ \gamma[N \mapsto \Delta(N), N.E \mapsto \Delta(N.E)] \vdash S_R \Rightarrow_e v \end{array}}{\gamma \vdash N\textbf{=}\, xEy\ S_R \Rightarrow_e v} \tag{5}$$

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 set 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 sets 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}
```

Sets in Maia syntax rules are constructed automatically. Each set is converted into a simple or a compound set containing the input chunks that matched the parser rule. Maia syntax phase constructs simple sets 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, $\{S_1,S_2,,S_n\}$. Then, we can define simple set $S$ as follows:

$$let\ S =^{def} \{S_1, S_2, .., S_n\} \tag{6}$$

The SOS of a simple set definition $S$ is:

$$\frac{(\gamma \vdash S_1, \gamma \vdash S_2, \ldots, \gamma \vdash S_n) \Rightarrow_e \{v_1, v_2, \ldots, v_n\}{=}v}{\gamma \vdash let\ S =^{def} \{S_1, S_2, \ldots, S_n\} \Rightarrow_d \gamma[S \mapsto v]} \tag{7}$$

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

Alternatively, Maia constructs compound sets 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 set that has $n$ simple sets $\{S_1,S_2,\ldots,S_n\}$. Each simple set $S$ 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 set $S$ as follows:

$$\begin{aligned} let\ S =^{def} \{&\{S_{1,1}, S_{2,1}, \ldots, S_{n,1}\}, \\ &\{S_{1,2}, S_{2,2}, \ldots, S_{n,2}\}, \\ &\ldots, \{S_{1,n}, S_{2,n}, \ldots, S_{n,n}\}\} \end{aligned} \tag{8}$$

The SOS of compound set $S$ is:

$$\frac{\begin{bmatrix} \{\gamma \vdash S_{1,1}, \gamma \vdash S_{2,1}, \ldots, \gamma \vdash S_{n,1}\}, \\ \{\gamma \vdash S_{1,2}, \gamma \vdash S_{2,2}, \ldots, \gamma \vdash S_{n,2}\}, \ldots, \\ \{\gamma \vdash S_{1,n}, \gamma \vdash S_{2,n}, \ldots, \gamma \vdash S_{n,n}\} \end{bmatrix} \Rightarrow_e \begin{bmatrix} \{v_{1,1}, v_{2,1}, \ldots, v_{n,1}\}, \\ \{v_{1,2}, v_{2,2}, \ldots, v_{n,2}\}, \ldots, \\ \{v_{1,n}, v_{2,n}, \ldots, v_{n,n}\} \end{bmatrix} = v}{\gamma \vdash let\ S =^{def} \begin{bmatrix} \{\{S_{1,1}, S_{2,1}, \ldots, S_{n,1}\}, \\ \{S_{1,2}, S_{2,2}, \ldots, S_{n,2}\}, \\ \ldots, \{S_{1,n}, S_{2,n}, \ldots, S_{n,n}\}\} \end{bmatrix} \Rightarrow_d \gamma[S \mapsto v]} \tag{9}$$

### 4.3.3 Set Construction

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

$$\begin{aligned} C \quad &\rightarrow \quad Var = < Str_1, \ldots, Str_n >; \\ &\mid \quad Var = < Nval_1, \ldots, Nval_n >; \end{aligned} \tag{10}$$

where $Var$ is a variable name, $Str$ is a string literal and $Nval$ is a numeric value. An example of explicit set construction is

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

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

$$\frac{\gamma \vdash Var = < Str_1, \ldots, Str_n > \Rightarrow_e \{Str_1, \ldots, Str_n\}, \quad \gamma[Var \mapsto \{Str_1, \ldots, Str_n\}] \vdash S_R \Rightarrow_e v}{\gamma \vdash Var = < Str_1, \ldots, Str_n > S_R \Rightarrow_e v} \tag{11}$$

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

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

To join sets, we reuse the angle brackets to indicate set construction, though in this case we specify how to construct an element rather than all elements in the set. 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 set `userDomain` mapping to the value {`'aD1', 'aD2', 'aD3'`}.

The SOS of set join for strings is

$$\frac{\gamma \vdash Var = < A_1 . A_2 . \ldots . A_n > \Rightarrow_e \{a_{1,1} \cdot \ldots \cdot a_{n,1}, \ldots, a_{1,m} \cdot \ldots \cdot a_{n,m}\}, \quad \gamma[Var \mapsto \{a_{1,1} \cdot \ldots \cdot a_{n,1}, \ldots, a_{1,n} \cdot \ldots \cdot a_{n,m}\}] \vdash S_R \Rightarrow_e v}{\gamma \vdash Var = < A_1 . A_2 . \ldots . A_n > S_R \Rightarrow_e v} \tag{12}$$

where $A_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,m}\}$. The SOS of set join for numeric sets 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 set *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:

$$\begin{array}{rcl} S_R & \rightarrow & E? \textbf{ forevery}? \ Var_1 \textbf{ in } Var_2 : C_n \textbf{ ;} \\ & | & E? \textbf{ exists } Var_1 \textbf{ in } Var_2 : C_n \textbf{ ;} \\ E & \rightarrow & \textbf{(require)} \mid \textbf{(warn)} \mid \textbf{(info)} \end{array} \tag{13}$$

where $E$ is an enforcement level *Var* is a set name and $C_n$ is a constraint on the set. 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.

$$\frac{\gamma \vdash Var_2 \Rightarrow_e A, \quad \forall a \in A \ \gamma[Var_1 \mapsto a] \vdash C_n \Rightarrow_e \texttt{true}}{\gamma \vdash \textbf{forevery}? \ Var_1 \textbf{ in } Var_2 : C_n \textbf{ ;} \Rightarrow \texttt{true}} \tag{14}$$

where $A$ is a set defined by a syntax rule or via set construction and $v \in B$, and

$$\frac{\gamma \vdash Var_2 \Rightarrow_e A, \quad \exists a \in A \ \gamma[Var_1 \mapsto a] \vdash C_n \Rightarrow_e \texttt{false}}{\gamma \vdash \textbf{forevery}? \ Var_1 \textbf{ in } Var_2 : C_n \textbf{ ;} \Rightarrow \texttt{false}} \tag{15}$$

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

$$\frac{\gamma \vdash Var_2 \Rightarrow_e A, \quad \exists a \in A \ \gamma[Var_1 \mapsto a] \vdash C_n \Rightarrow_e \texttt{true}}{\gamma \vdash \textbf{exists } Var_1 \textbf{ in } Var_2 : C_n \textbf{ ;} \Rightarrow \texttt{true}} \tag{16}$$

and

$$\frac{\gamma \vdash Var_2 \Rightarrow_e A, \quad \forall a \in A \ \gamma[Var_1 \mapsto a] \vdash C_n \Rightarrow_e \texttt{false}}{\gamma \vdash \textbf{exists } Var_1 \textbf{ in } Var_2 : C_n \textbf{ ;} \Rightarrow \texttt{false}} \tag{17}$$

These rules indicates that the constraint must hold for at least one member of the set $A$ in order for the file to be valid.

A constraint, $C_n$, in Maia may be a logical comparison, an expression in predicate logic, a logic constraint, a membership test *Inclusion* or a *blackbox*. Syntactically, constraints are of the form

$$\begin{array}{rcl} C_n & \rightarrow & C_n \ logic \ C_n \\ & | & \textbf{not } C_n \\ & | & \textbf{(} C_n \textbf{)} \\ & | & Inc \\ & | & Blb \\ & | & Cmpr \end{array} \tag{18}$$

$C_n$ provides one or more Boolean constraints that will be evaluated for each element in the specifying set until the rule is satisfied. For example, a rule that applies the constraint on all elements of the set, `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 set `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 set `passwdRecord`.

Set 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 set being tested. The syntax of set membership semantic rules is :

$$\begin{array}{rcl} Inc & \rightarrow & indexedName \textbf{ in } setName \\ & | & indexedName \textbf{ in} < string \ (, \ string)^* > \\ & | & indexedName \textbf{ in} < nVal \ (, \ nVal)^* > \end{array} \tag{19}$$

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

$$nVal \quad \rightarrow \quad iVal \mid fVal \tag{20}$$

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

$$\frac{\gamma \vdash iVal \Rightarrow_e iVal \ \ \gamma \vdash fVal \Rightarrow_e fVal}{\gamma \vdash nVal \Rightarrow_e v} \tag{21}$$

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:

$$\begin{array}{lcl} indexedName & \rightarrow & setName\ (\textbf{[}\ exp\ \textbf{]})?\ (\textbf{.}\ setName)? \\ & | & setName \end{array} \tag{22}$$

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

$$\frac{}{\gamma \vdash setName \Rightarrow_e \gamma(setName)} \tag{23}$$

$$\frac{}{\gamma \vdash setName.setName \Rightarrow_e \gamma(setName.setName)} \tag{24}$$

$$\frac{\gamma \vdash exp \Rightarrow_e v}{\gamma \vdash setName \Rightarrow_e \gamma(setName)[v]} \tag{25}$$

where $v \in nVal$, and $setName, setName.setName \in dom(\gamma)$.

An example of a set membership in Maia, consider the set `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 set membership is:

$$\frac{\gamma \vdash \left[\begin{array}{l} e_j \in (indexedName)\ \textbf{in}\ (setName) \\ |\ e_j \in (indexedName)\ \textbf{in}\ < string > \\ |\ e_j \in (indexedName)\ \textbf{in}\ < nVal > \end{array}\right] \Rightarrow_e v}{\gamma \vdash if[]_{i=1}^{n} \left[\begin{array}{l} e_i \in (indexedName)\ \textbf{in}\ (setName) \\ |\ e_i \in (indexedName)\ \textbf{in}\ < string > \\ |\ e_i \in (indexedName)\ \textbf{in}\ < nVal > \end{array}\right] \Rightarrow_e v} \tag{26}$$

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, setName, ...)
```

In this rule, `verifier` is the name of a black-box verifier known to the system, and the *setName* 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 which we omit for brevity. For a more thorough discussion of black boxes, comparisons and arithmetic see (Bonamy, 2016).

# 5 EXAMPLE MAIA SECIFICATION

Below is an example Maia specification for protecting the integrity of `/etc/passwd`.

```
PasswdFile = (passwdRecord Newline)+ ;
passwdRecord = name ":" password ":" uid ":"
               gid ":"
gecos ":" directory ":" shell ;
name = [a-zA-Z_][-a-zA-Z0-9_]{0,31} ;
password = "*" | "x" | CryptPassword ;
uid = StringPosDec+ ;
gid = StringPosDec+ ;
gecos = [^:\n]* ;
directory = [^:\n]+ ;
shell = [^:\n]* ;
Newline = "\n" ;
name isUnique() ;
exists name : name == "root" ;
(warn) name : name ~ /[A-Z]/ ;
uid: uid <= 65535;
gid: gid <= 65535;
directory : directory isAbsPath() and
                directory isDirectory() ;
passwdRecord : name == "root" implies uid == 0;
passwdRecord : directory isAccessibleTo(name);
passwdRecord : shell != "" implies
               ( shell is AbsPath() and
                 shell isExecutableBy(user) ) ;
```

# 6 MAIA IMPLEMENTATION

Our early approach to creating verifiers from Maia specifications involves converting the specification into input for Flex and Bison, which then generates a parser based on the spec. Flex and Bison are widely used, but require a clear separation between scanning and parsing phases (with each program handling one phase), while Maia does not make a distinction between these cases. To overcome this obstacle, the converter recognized string literals, character classes, and regular expressions in Maia syntax rules and created corresponding tokens for Flex. This approach occasionally required hand-adjustment to the Flex input, or small changes to the syntax rules themselves, but was generally successful.

While Maia's syntax rules are no more expressive than Bison's, they include a number of convenience features which Bison lacks, such as shorthand for repetition. As a result, the converter sometimes had to transform Maia syntax into Bison syntax. This was accomplished by creating Bison nonterminals which encapsulated the Maia behavior, like replacing an explicit repetition (`A = B{3} ;`) with a nonterminal containing the desired number of entries (`A : C ; C : B B B ;`). Bison's action system was

used to check semantic rules. Universal rules were checked in the action for the appropriate nonterminal, with failures causing the parser to immediately exit. Existential rules were also checked in actions, but set a flag if a rule passed. The flags can then be checked at the end of parsing to ensure compliance.

By converting Maia specifications into Flex and Bison parsers in this way, we were able to create verifier programs for the password, shadow, and groups files, which are part of the Linux login system. With light modification, we were also able to produce a single verifier that checked rules which apply across all three files to enforce constraints like "users with an entry in the password field must appear in the shadow file". The password file verifier was also used to test integrity protection with the Linux kernel module (Bonamy, 2016).

More examples, experimental result, and comparative evaluation are available in (Bonamy et al., 2016) and (Bonamy, 2016). Specifically, we have developed Maia spcecification for valid hashes by `crypt()`, linux password, shadow and group files, PNG images and `ssh` configurations.

## 7 CONCLUSIONS

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 paper, 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 are currently implementing the full Maia interpreter. Our preliminary implementation has shown that a Maia specification can be used to protect the integrity of Linux system configuration files with minimal overhead. In the future, we will build a full implementation of Maia that requires no hand modification.

## REFERENCES


Backus, J. W. (1959). The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. *Proceedings of the International Comference on Information Processing, 1959*, pages 125–132.

Bonamy, P., Carr, S., and Mayo, J. (2016). Toward a mandatory integrity protection system. In *Proceedings of the Thirty-first International Conference on Computers and Their Applications*.

Bonamy, P. J. (2016). *Maia and Mandos: Tools for Integrity Protectionon Arbitrary Files*. PhD thesis, Michigan Technological Univeristy.

Dewar, R. B. K. (1979). *The SETL Programming Langauge*. Courant Institute of Mathematical Sciences, New York University.

Fisher, K., Mandelbaum, Y., and Walker, D. (2010). The next 700 data description languages. *Journal of the ACM*, 57(2):1–51.

Fisher, K., Mandelbaum, Y., Walker, D., Fisher, K., Mandelbaum, Y., and Walker, D. (2006). *The next 700 data description languages*, volume 41. ACM.

Fisher, K. and Walker, D. (2011). The PADS project. In *the 14th International Conference*, page 11, New York, New York, USA. ACM Press.

ISO (2004). Information technology - Computer graphics and image processing - Portable Network Graphics (PNG): Functional specification. Technical Report ISO/IEC 15948:2003 (E), Geneva, Switzerland.

Ji, Q., Qing, S., and He, Y. (2006). A formal model for integrity protection based on dte technique. *Science in China Series F: Information Sciences*, (5):545 – 565.

Jim, T., Mandelbaum, Y., and Walker, D. (2010). Semantics and algorithms for data-dependent grammars. *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 45(1):417–430.

Johnson, S. C. (1975). Yacc: Yet Another Compiler-Compiler. Technical Report Computing Science Technical Report No. 32, Murray Hill, New Jersey.

Lesk, M. E. and Schmidt, E. (1975). Lex - A Lexical Analyzer Generator. Technical Report Computer Science Technical Report No. 39, Murray Hill, New Jersey.

Parr, T. (2015). *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf.

Plotkin, G. D. (1981). A structural approach to operational semantics.

Polk, W. T. (1993). Approximating Clark-Wilson"Access Triples" with Basic UNIX Controls. In *Proceedings of the UNIX Security Symposium IV*, pages 145–154.

Prasad, S. and Arun-Kumar, S. (2003). An introduction to operational semantics. http://www.cse.iitd.ernet.in/ sanjiva/opsem.ps.

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

van der Vlist, E. (2003). *RELAX NG*. O'Reilly Media.

W3C (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition). Technical report.

W3C (2012a). W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. Technical report.

W3C (2012b). W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. Technical report.

Wirth, N. (1977). What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823.