Michigan Technological University
Digital Commons @ Michigan Tech

Dissertations, Master's Theses and Master's Reports

2016

Maia and Mandos: Tools for Integrity Protection on Arbitrary Files

Paul J. Bonamy
Michigan Technological University, pjbonamy@mtu.edu

Copyright 2016 Paul J. Bonamy

Recommended Citation

Bonamy, Paul J., "Maia and Mandos: Tools for Integrity Protection on Arbitrary Files", Open Access Dissertation, Michigan Technological University, 2016.
http://digitalcommons.mtu.edu/etdr/122

Follow this and additional works at: http://digitalcommons.mtu.edu/etdr

Part of the Information Security Commons, and the OS and Networks Commons

Maia and Mandos:
Tools for Integrity Protection on Arbitrary Files

By

Paul J. Bonamy

A Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

In Computer Science

Michigan Technological University

2016

© 2016 Paul J. Bonamy

This dissertation has been approved in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science.

Department of Computer Science

Dissertation Co-Advisor: *Jean Mayo*

Dissertation Co-Advisor: *Steve Carr*

Committee Member: *Ali Ebnenasir*

Committee Member: *Xinli Wang*

Committee Member: *Zijiang Yang*

Department Chair: *Min Song*

For Richard and Ethel,
Lenore, and Marianne

omnia mutantur, nihil interit

Contents

List of Figures **11**

List of Tables **11**

Preface **13**

Acknowledgements **15**

Abstract **17**

1 Introduction **19**
- 1.1 Background 19
  - 1.1.1 Computer Security 19
  - 1.1.2 Confidentiality 20
  - 1.1.3 Availability 20
  - 1.1.4 Integrity 21
  - 1.1.5 Mandatory and Discretionary Systems 22
  - 1.1.6 Language Semantics 22
- 1.2 Related Work 23
  - 1.2.1 Biba Integrity Model 23
  - 1.2.2 Clark-Wilson Integrity Model 24
  - 1.2.3 Database Integrity 25
  - 1.2.4 Access Control Systems 26
  - 1.2.5 Integrity via Access Control 27
  - 1.2.6 Linux Security Modules 27
  - 1.2.7 Fault Tolerance 28
  - 1.2.8 Correctness 28
- 1.3 Motivation 28
  - 1.3.1 Implementability and Generality 29
  - 1.3.2 Correctness v Integrity 30
  - 1.3.3 Fault Tolerance v Integrity 30
- 1.4 Conclusion 30

2 Maia **31**
- 2.1 Related Work 31
- 2.2 Design Objectives 32
- 2.3 Model Overview 32
- 2.4 Syntax Specification 33

2.4.1 Nonterminal Names . . . 33
2.4.2 Concatenation . . . 34
2.4.3 Optional . . . 34
2.4.4 Repetition (0+) . . . 34
2.4.5 Repetition (1+) . . . 34
2.4.6 Repetition (Exactly N) . . . 34
2.4.7 Repetition (Range) . . . 35
2.4.8 Comments . . . 35
2.4.9 Special Sequences . . . 35
2.4.10 Character Classes . . . 35
2.4.11 Wildcard Match (Dot) . . . 35
2.4.12 Regular Expressions . . . 36
2.4.13 Numeric Parsing . . . 36
2.4.14 Limited Context Awareness . . . 37
2.5 Building Sets . . . 38
2.5.1 Automated Set Construction . . . 39
2.5.2 Explicitly Constructed Sets . . . 40
2.5.3 Joining Existing Sets . . . 40
2.6 Semantic Specification . . . 41
2.6.1 Rule Structure . . . 42
2.6.2 Logical Comparisons and Connectors . . . 42
2.6.3 Grouping . . . 43
2.6.4 Regular Expressions . . . 43
2.6.5 Element Length . . . 43
2.6.6 Set Inclusion . . . 44
2.6.7 Count-Based Constraints . . . 44
2.6.8 Using Compound Sets . . . 45
2.6.9 Index-based Constraints . . . 45
2.6.10 Enforcement Levels . . . 46
2.7 Other Language Features . . . 47
2.7.1 Semantic Templates . . . 47
2.7.2 Black-box Verifiers . . . 48
2.7.3 Inclusions and Libraries . . . 49
2.7.4 Multi-file Verifiers . . . 50
2.8 Proof of Concept Implementation . . . 51
2.8.1 Overall Design . . . 51
2.8.2 Required Software . . . 51
2.8.3 File Naming . . . 51
2.8.4 Identifying Tokens . . . 52
2.8.5 Generating Bison ENBF . . . 52
2.8.6 Providing Semantic Checks . . . 54
2.8.7 Creating Multi-file Verifiers . . . 55
2.9 Conclusion . . . 55

3 Mandos . . . 57
3.1 Related Work . . . 57
3.2 Integrity Model . . . 58
3.2.1 Determining Validity . . . 58

3.2.2 Integrity Guarantees . . . 58
3.2.3 On Files and Names . . . 59
3.3 Implementing Integrity Protection . . . 59
3.3.1 Implementation Overview . . . 59
3.3.2 Setup and Cleanup Helpers . . . 60
3.3.3 Enforcement Options . . . 60
3.3.4 Failure Cases . . . 62
3.3.5 Readers-Writer Locking . . . 62
3.3.6 Linux Security Modules Interface . . . 63
3.3.7 SecurityFS Interface . . . 64
3.3.8 Module Breakdown . . . 64
3.4 Using Mandos . . . 65
3.4.1 Compiling and Installing Mandos . . . 65
3.4.2 Configuration Interface . . . 65
3.4.3 Protecting `/etc/passwd` with Mandos . . . 68
3.5 Conclusion . . . 69

4 System Performance . . . 71
4.1 Benchmark Environment . . . 71
4.2 Testing with the Password File . . . 71
4.3 Piecemeal Analysis . . . 72
4.3.1 File Handling . . . 72
4.3.2 Password Verifier Performance . . . 72
4.4 Single File Performance . . . 74
4.4.1 Baseline . . . 75
4.4.2 Mandos Overhead . . . 75
4.4.3 Mandos Performance . . . 76
4.5 Concurrent Verification . . . 78
4.6 Conclusion . . . 79

5 Future Work . . . 81
5.1 Maia . . . 81
5.1.1 Reference Implementation . . . 81
5.1.2 Additional Specifications . . . 81
5.1.3 Other Applications . . . 82
5.2 Mandos . . . 82
5.2.1 Link Mode Protections . . . 82
5.2.2 Interface for Userspace Software . . . 82
5.2.3 Awareness of File Groups . . . 83
5.2.4 File Redirection . . . 83
5.2.5 Mandos for Remote File Systems . . . 84

A Maia Specifications . . . 85
A.1 `crypt()` Hashes . . . 86
A.2 File System Objects . . . 87
A.3 Templates for Set Ordering . . . 88
A.4 Linux Password File . . . 89
A.5 Linux Shadow File . . . 91
A.6 Linux Group File . . . 92

A.7 Linux Login Set 93
A.8 PNG Images 94
A.9 SSH Configuration 101

B EBNF Maia Spec **105**

Bibliography **109**

List of Figures

2.1 Parse Tree for UserFile Example 39

3.1 Overview of Mandos Integrity Protection System 60
3.2 Flowcharts for Mandos File Access Hooks 63
3.3 Mandos' Configuration Directory Structure 66
3.4 Maia Specification for Mandos Policy Files 67
3.5 Sample Status File for `/etc/passwd` 68
3.6 Sample Mandos Policy for `/etc/passwd` 69

4.1 File Length versus Setup/Cleanup Time 73
4.2 File Length versus Verification Time 74
4.3 File Length versus Processing Time (Stock Kernel) 75
4.4 Source Code of Verifier for Always-Reject Tests 76
4.5 File Length versus Mandos Overhead 76
4.6 File Length versus Processing Time (Mandos Kernel) 77
4.7 Simultaneous Processes versus Processing Time 79

List of Tables

3.1 File Access Rules for Readers / Writer Locking 62
3.2 Lines of Code in Mandos Source Files 65
3.3 Potential Status of Mandos Protected Files 68

Preface

Chapter 3 is a significantly expanded version of the paper "Toward a Mandatory Integrity Protection System" by Bonamy, Carr, and Mayo, presented at the International Society for Computers and Their Applications' (ISCA) 31st International Conference on Computers and Their Applications (CATA 2016) in Las Vegas, NV on 4 April 2016. The paper is also published in the Proceedings of same. Bonamy wrote the paper, in addition to performing all of the data collection and analysis. Mayo and Carr provided advice as to venue selection, formatting, and places where the document was unclear or could be improved.

The chapter goes into much greater detail as to the actual implementation and use of Mandos, and formally describes many formats not presented in the original paper. Figure 4.7 is derived from experimental data also used in the paper, but was created specifically for this document.

Acknowledgements

This work would not have been possible without the contributions of a vast support network. Here, then, in no particular order, are selection of those individuals whose help was especially important, and who have not slipped beyond the recall of my writing-addled brain.

My family...
who supported this mad endeavor and put up with me when things went poorly

My sister...
who turned her trained eye to making this document look its best, and helped decrease the bandit population of Pandora in search of vaults full of epic loot

My advisors...
for guidance in finding questions worth asking and presenting the answers well

My committee...
for agreeing to lend me their expertise and verify the quality of my work

All of my students over the years...
who introduced me to the joys of teaching and indirectly inspired Maia and Mandos

My fellow graduate students...
who helped find problems and make improvements, and generally kept me from being a hermit

Dr. Michael Bowler...
who helped me to see what Maia is not, shining light on what it is

Nick Kuras...
who did his best to cause distraction and in so doing kept me sane(-ish)

Ben DePew...
who entered the graduate program with me and provided a place for me to live long after he moved on. At least one of us got the degree, and hopefully now both of us can have real jobs

The faculty of the School of Mathematics and Computer Science at LSSU...
who told me it would be a crime if I didn't do to grad school. I'm glad I listened.

Thank you all for your guidance and support, and my thanks as well to everyone else who aided me in this journey. For all the bumps along the way, I'm grateful to have reached this milestone.

Abstract

We present the results of our dissertation research, which focuses on practical means of protecting system data integrity. In particular, we present Maia, a language for describing integrity constraints on arbitrary file types, and Mandos, a Linux Security Module which uses verify-on-close to enforce mandatory integrity guarantees. We also provide details of a Maia-based verifier generator, demonstrate that Maia and Mandos introduce minimal delay in performing their tasks, and include a selection of sample Maia specifications.

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. However, while both confidentiality and availability are subject to 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 so broad as to be useless except after the fact (e.g. backups). There are few, if any, approaches to integrity which are both capable of actively protecting data and actually implemented in a form that is directly useful.

Our work seeks to provide robust tools to enable general-purpose integrity protection. This is accomplished by creating a language to describe integrity constraints for arbitrary files. We have also designed a proof of concept system that implements our language, and a Linux kernel module which uses verifiers to provide strong integrity guarantees. Combined with performance analysis, we can demonstrate both theoretical safeguards for system integrity and the practical implications of putting those safeguards into effect.

The remainder of this chapter presents some general background on computer security and language design (Section 1.1), followed by a discussion of related research (Section 1.2), and the deficiencies of same (Section 1.3). We then present details of our specification language (Chapter 2), kernel module (Chapter 3), and performance analysis (Chapter 4), before concluding with some promising areas for future work (Chapter 5). We also provide a pair of appendices containing sample file specifications with explanations (Appendix A) and a grammar for part of the language (Appendix B).

1.1 Background

1.1.1 Computer Security

Broadly speaking, computer security is focused on what users – authorized or otherwise – are allowed to do on a computer system. Security is specifically interested in the activities of users and the software that works on their behalf, rather than issues caused by, for example, hardware failures. This is not to say that computer security cannot help deal with hardware faults and the like, nor that mitigation techniques for other failures cannot reinforce a security system. Security simply tends to be more concerned with intentional actions (even those that are unintentionally wrong) than with things happening outside of the computer system's control.

Computer security is typically broken down into three inter-related aspects: confidentiality, availability, and integrity. (Bishop [14], provides a reasonable, though not definitive, overview of these as well.) These aspects tend to blend into one another as technologies in one area provide benefits in another, but they provide a good overall model for what it means for a computer system to be secure. For the purposes of this discussion, we will present each aspect in terms of a computer system, including not only the software on the machine itself, but also in terms of features external to the physical machinery, like its immediate environment, and the infrastructure that serves the

computer. It would, however, be equally appropriate to consider the aspects of computer security without these externalities.

1.1.2 Confidentiality

Confidentiality is probably the most intuitively obvious aspect of computer security. At its core, confidentiality is simply keeping users or processes from accessing things they are not allowed to access. Normal users probably should not be allowed to access one another's files by default, and a random Joe on the street almost certainly should not be able to get their hands on the nuclear launch codes. If a bank vault or safe is being used as a security metaphor, confidentiality is almost certainly the core concept under discussion.

Confidentiality generally has access control as a front-line defense. Only certain people are allowed to possess the keys to a vault, or the combination to a safe. Likewise, if we can prevent people who should not access certain resources or information from gaining access, we protect the confidentiality of those resources. Most modern operating systems enforce an access control model, though the granularity (and general utility) of the models tends to vary. This ranges from the complete lack of access control in FAT16 and FAT32 file systems [48], and consequently early Windows, through the rather coarse but ubiquitous Unix model [51], to the incredibly fine-grained DTE [9]. Placing access terminals in physically secured locations, with guards and lists of permitted personnel, would also provide confidentiality via access control. Air gaps between public networks and confidential data serve a similar purpose.

Unfortunately, there are times when access control systems cannot be relied upon to protect confidentiality, either because there is no good way to enforce access control (as with packets traversing the internet), or because the access control system itself has been subverted. In these cases, confidentiality can be enforced via encryption, thereby requiring a user to not only gain access to the encrypted form but also to possess an ability to decrypt the data. In this way, confidentiality of information can be assured, even though the stored or transmitted form of the information is not necessarily confidential.

Confidentiality can also be invoked to protect not only the access to a resource or piece of information, but to hide its very existence. Many access control mechanisms support this sort of functionality as a matter of course. (Denying read access to a folder on a Unix system is a classic example.) Steganography [63] [34] [24], which focuses on hiding data within other, innocuous, information, can also be used to cloak the existence of data, either during storage or transmission, in much the same way encryption can make it unusable to any but the intended recipients.

1.1.3 Availability

Where confidentiality focuses on keeping unauthorized users away from system resources, availability (which is closely related to reliability) seeks to ensure that authorized users are always able to access permitted resources. A user cannot get anything done if they cannot access their own files, and an unopenable bank vault is as big a problem for the bank as it is for potential thieves. Availability is why denial of service attacks are considered a security issue, even if the attack has no purpose other than knocking a resource offline: an offline service does its users no good at the best of times, and significant harm in the worst case.

There are many approaches to maintaining availability, which depend to a great extent on the particular ways a system is intended to be accessed and how it might fail. Redundant servers and network connections are a popular solution for making sure a hardware failure doesn't bring down a service. Paired with elastic provisioning, redundancy can also provide a check against the aforementioned (distributed) denial of service attacks, by scaling up the amount of computing power

and bandwidth available to a service faster than the attackers can generate requests. Firewalls can also be employed to selectively filter traffic, weeding out the worst of an attack before it can reach the service itself.

Provable correctness and fault tolerance can also help with availability. Software which is demonstrably correct is, at worst, much harder to perturb into behaving badly, and at best proof against poor behavior. Building in fault tolerance can also help a service survive unforeseen failure cases, by allowing parts of the system to remain online during failures, or even getting misbehaving systems back to work automatically.

The ability to recover from problems quickly and effectively is also a component of availability. All systems will eventually fail, and the response to that failure is often critical to how quickly a service can come back online. Good backups play a roll here, but so do provisions to prevent hot backups from mirroring software bugs.

1.1.4 Integrity

Whereas confidentiality and availability are concerned with whether a given user can gain access to some piece of information, integrity is entirely concerned with whether a piece of information is trustworthy, regardless of who is accessing it. Typically we consider two overall types of integrity: data integrity (the content itself is valid) and origin integrity (the source of the data is acceptable).

In practice, origin integrity is reasonably well-served by the same access control mechanisms that enforce confidentiality. Starting from a system that prevents read access according to some criteria, it is straight-forward to add a facility to block write accesses. In fact, most, if not all, access control mechanisms for modern operating systems provide more or less equivalent facilities for limiting reads and writes, which provides for general enforcement of origin integrity.

Data integrity is a more complex problem. It is certainly possible in some cases to validate the format of a file, but it is not necessarily possible to verify that the data itself is valid. For example, we could write a program to decide whether an accounting ledger is correctly formed, but determining whether the transactions it represents should have happened is beyond the scope of most software. Further, it is possible for the data to be well formed but wrong, which can be difficult to check without outside help.

Two general classes of integrity systems exist for handling data integrity. Prevention systems are geared toward disallowing unauthorized attempts to modify the data (overlapping with origin integrity), or attempts to modify it in unauthorized ways. Detection systems seek to determine whether data is no longer trustworthy, and, ideally, how it got that way. Used together, these systems can prevent blatantly incorrect operations, such as corrupting a file's format, and provide enough information after the fact to detect when data integrity failed, and what or who caused the failure.

To return to our accounting example, typical double-entry accounting requires that every transaction be represented as a withdrawal from one account and an equal-sized deposit into another. Enforcing the integrity of the ledger requires three parts: we must only allow certain people to write new entries into the ledger, we must ensure that they have obeyed the rules of double-entry accounting, and we must periodically check to make sure that all real transactions were recorded, and that no transactions were recorded that didn't happen – that is, preform an audit. Limiting who may modify the ledger helps to ensure origin integrity. Enforcing the rules of double-entry accounting gives us a reasonable prevention mechanism, assuring we can trust that the accounting data was modified correctly. Finally, periodic audits, and the logs that make them possible, gives us a way to detect places where the ledger doesn't match reality and (hopefully) when and how those discrepancies crept into the data.

As noted above, access control systems generally do a good job as a first defense of origin integrity. Routine auditing of system logs can also help to detect integrity failures after the fact, which makes recovery more feasible if good backups are available. However, there are precious few prevention systems that combine good, general integrity safeguards with actual implementability. Our work aims to correct this by providing a generally applicable system to detect and prevent modifications which would compromise data integrity.

1.1.5 Mandatory and Discretionary Systems

The Department of Defense's Trusted Computer System Evaluation Criteria [2] presented the first major methodology for evaluating computer security. While the specification provides a whole range of interesting evaluation metrics, the most directly useful to this discussion is the difference between mandatory and discretionary access control.

All access control systems work by enforcing a set of rules controlling who is permitted access to system resources. Under a discretionary model, like the one used by the Unix file system, there is no central access control policy, and users have direct control over who may access their files. By contrast, under a mandatory model, access control rules are defined by some central authority – an organizational policy, legal requirement, or the like – and enforced for all users. Mandatory models, then, make it much easier to ensure that access control rules are enforced correctly, while making it harder for an individual user to share data they should reasonably be able to share.

While the notions of mandatory and discretionary systems were proposed in the context of access control, they can be reasonably applied to any security system that could be controlled by a central authority or individual users. For example, a system administrator could decide to enable full-disk encryption (mandatory encryption), or a user could decide to encrypt some of their own files (discretionary encryption). Mandatory and discretionary controls can also be used together, with the administrator specifying a baseline mandatory policy which users may the further restrict or relax at their discretion, within the bounds of the mandatory guidelines.

1.1.6 Language Semantics

At the most fundamental level, (computer) languages are defined by their syntax – which describes how to construct valid statements within the language – and their semantics – which attach meaning to those statements. It is possible to reason about a language given only its syntax or semantics, but actually using the language to communicate an idea requires both. For programming languages, syntax is handled by the parser and semantics by the interpreter or compiler that converts the parser's output to executable code.

The use of formal notation to specify language syntax goes back at least as far as the development of ALGOL 58 [7]. The metalanguage used in that case has since been standardized into Backus-Naur Form (BNF) [8] [36], which has the convenient feature of being mechanically transformable into a working parser. This makes describing and interpreting a novel syntax reasonably easy, assuming the syntax can be expressed in BNF.

Accurately describing language semantics is a larger problem, about which entire books have been written without covering all of the possibilities. For example, Schmidt [58] gives detailed coverage of denotational semantics but only a brief introduction to other semantic systems. For our purposes, two varieties of semantics are sufficient: operational semantics and denotational semantics.

The simplest possible way to specify the semantics of a language is to create an interpreter for it. Building an interpreter or compiler is one way to provide operational semantics, mapping language constructs to meaning by converting the language to executable code. While operational semantics are convenient for testing language designs, they present problems when a language is ported to a

different architecture, or a new interpreter is implemented. The semantics of the language are tied to the details of the existing implementation, so any issue not dealt with in that implementation (say, a difference in bit shifting behavior on different hardware) is left effectively undefined. This can lead to significant problems with interoperability between implementations or across platforms.

The denotational approach (Tennant [60] provides a good overview) is to tie the meaning of a language construct directly to the construct itself. By defining the language semantics in such a way that the meaning of a statement is derived solely from the meaning of its components, we can separate the semantics from their implementation. This has two major advantages, the first of which is that it overcomes the reimplementation problem of operational semantics. Because the semantics are not tied to a particular implementation, there is no need to worry that the original implementation may lack vital semantic details.

The second advantage of denotational semantics is provability. Under operational semantics, any reasoning about a program must keep in mind the action of the implementation on the program. Possessing denotational semantics for a language allows us to reason directly about the program itself. Any conclusions drawn from that reasoning will hold up wherever the program is run, assuming the implementation does a faithful job translating the program. This is an incredibly powerful feature, and makes proving the behavior of a program a significantly more tractable undertaking.

1.2 Related Work

1.2.1 Biba Integrity Model

Biba's integrity model [13] is one of two highly influential models of data integrity. His model is geared toward ensuring the trustworthiness of data within a system by controlling who may access it. In the language outlined above, this would be a prevention system for origin integrity, granting some enforcement of data integrity as well.

The model consists of a set of subjects, a set of objects, and a set of integrity levels. Subjects may be either users or processes working on their behalf. Each subject or object is assigned an integrity level within the system, which effectively describes how much confidence one has in a program to work correctly, or a piece of data to be accurate. Finally, we assume that integrity levels are orderable in some way, so that we can speak of something having higher or lower integrity – being more or less trustworthy – than something else.

Using these definitions, Biba defines three rules as part of the Strict Integrity Policy:

1. A subject may not read from an object which is less trustworthy than itself.

2. A subject may not write to an object which is more trustworthy than itself.

3. A subject may not execute another subject which is more trustworthy than itself.

The first rule serves two purposes. First, it make sure that subjects can still access information, even if they are, themselves, untrustworthy. This solves a problem with preliminary versions of the model. Secondly, it ensures that no one inadvertently picks up untrustworthy information and then carries it to somewhere more trustworthy. If this were not the case, a trusted subject could read something which is not trustworthy (say, a rumor on the Internet) and then use their write access to make it appear more trustworthy than is actually the case (by, for example, writing it into the Encyclopedia Britannica).

Rule two serves a similar purpose: preventing writes above one's own integrity level ensures that high-integrity data is not polluted by low integrity information. This is notionally equivalent to

preventing people outside of the accounting office from modifying the ledgers. Together with the first rule, this ensures that high-integrity data cannot be directly polluted with low-integrity data, though it does result in something of an information blackout at the top level, as someone with highest-level integrity would only be able to read other highest-integrity sources.

The third rule prevents circumvention of the second by means of an additional, more trustworthy, process. Without this rule, it would be possible to use a trusted process to modify high-integrity data, even though the subject initiating the change would not be able to make it themselves. Note, however, that this rule has no effect on reading data, as a higher-integrity subject would actually have less access to information than a lower-integrity one.

It is interesting to note that Biba's integrity model is effectively a military-style system for data classification set up to work in reverse. Where Biba allows reading from higher integrity, and writing to lower, a typical data classification system disallows reading from higher security, but permits writing things that will end up with higher clearance.

1.2.2 Clark-Wilson Integrity Model

David Clark and David Wilson [15] take a significantly different approach to integrity. Their model focuses on prevention and detection of data integrity faults using transactions. This is a reasonably close match to commercial needs for data integrity, whereas Biba is closely tied to origin integrity.

Clark and Wilson separate all data items within a system into two categories: those upon which integrity constraints should be enforced, called "Constrained Data Items" (CDIs), and "Unconstrained Data Items" (UDIs), which do not have associated integrity constraints. CDIs and UDIs can be thought of as embodying the notions of levels of trust from Biba, where the former is fully trusted and the latter is untrusted. Note, though, that there are only these two levels. There is no intermediate partially-trusted state.

The integrity system itself, then, consists of two types of procedures: "Integrity Verification Procedures" (IVPs) and "Transformation Procedures" (TPs). Integrity Verifications Procedures exist to ensure that all CDIs conform to the integrity specifications in place on the system when the IVP is executed. This is akin to an auditing function in an accounting system. Transformation Procedures model well-formed transactions, changing the system's CDIs from one valid state to another. A double entry transaction in an accounting system would be an example of a TP. If we can assume the system was in a valid state at some time in the past because the IVPs were successfully run, and the system ensures that only TPs are allowed to manipulate CDIs, it is then straightforward to show that the system will continue to be valid going forward.

Unfortunately, there is no way to prove that a TP or IVP is valid within the system itself. Instead, an external observer must certify the correctness of the procedures. Once they are verified, however, TPs and IVPs can be enforced by the system. For this reason, Clark and Wilson propose sets of certification and enforcement rules, with the former governing the inspection of procedures for flaws, and the latter governing how the system implements and constraints procedures. The five certification, and four enforcement, rules are presented below, intermixed in the same order Clark and Wilson present them:

- C1 (Certification) All IVPs must properly ensure that all CDIs are in a valid state at the time the IVP is run.
- C2 All TPs must be certified to be valid. Further, the certifier must define a set of relations specifying the set of CDIs on which a given TP is certified to run.
- E1 (Enforcement) The system must ensure that CDIs are only manipulated by TPs certified 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.

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

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

C4 All TPs must write enough information to a log to reconstruct the nature of the operation performed.

C5 Any TP which transforms a UDI into a CDI much be certified in to produce either a valid CDI or nothing at all in all cases.

E4 An entity that can certify a TP or IVP may not execute any TPs or IVPs they certify.

Rules C1, C2, and E1 provide the core the Clark-Wilson model outlined above. E2 allows for restricting access to TPs (and CDIs) to certain users, with C3 serving to decide which users, and E3 providing the system a way to figure out which permissions to use. C4 is primarily to provide an audit trail, and can be implemented by making the log an append-only CDI. C5 allows data that cannot be trusted by itself (e.g. data being typed in by a user) to become a trusted data object. Note that this would correspond to writing to a higher integrity level in Biba, and would therefore be explicitly disallowed. Finally, E4 makes Clark-Wilson a mandatory, rather than discretionary, system, and guarantees that no single user can corrupt the integrity of the system.

1.2.3 Database Integrity

While generally applicable integrity protection systems are relatively rare, integrity of (relational) database systems is widely studied and well understood. It's useful to consider the state of database systems, both in terms of fundamental models and subsequent integrity studies, to gain insight into general integrity.

Codd [16] provides the first real definition of a relational model for databases, at a time when most databases required extensive manual traversal. Of particular note are a formalization of relationships, along with the basics of treating querying as transformation of relationships. Codd's relationship model provides an excellent framework for considering structured data, including sets of records in text files.

Türker and Gertz [64] give a summary of integrity features provided in the SQL 1999 standard, as compared to the commercial databases available at the time. The authors provide a useful taxonomy of integrity constraints in databases, which generalizes nicely to systems-level integrity. They note three general types of integrity constraints related to the contents of database tables, and three additional levels based on the number of database states required to verify the constraints.

Row constraints exist within a single database row, or record, and can be evaluated independently for each row. For example, a customer record must contain a name. Table constraints involve at least two rows in the same table, and generally correspond to uniqueness requirements. Finally, inter-table constraints act between records in multiple tables. Beyond this, state constraints can be validated independently of state changes, while state transition constraints require knowledge of how data has changed most recently. Finally, temporal (or state sequence) constraints rely on two or more states at different times, such as a requirement that a user not reuse old passwords.

As described by Türker and Gertz, the SQL standard supports two major varieties of integrity: descriptive constraints, applied during table creation and modification, and procedural constraints activated via triggers. Descriptive constraints are part of the table structure itself, and provide rules that must always be met. This includes field requirements like `NOT NULL`, row constraints like `CHECK`,

table constraints like `UNIQUE`, and the inter-table `FOREIGN KEY` system. All of these constraints are directly enforceable by the database system itself, and can be revalidated at basically any time, being, essentially, state constraints.

Procedural constraints are triggered when changes are made to the database. Procedures can be set to run immediately before the database is modified, or immediately after. Triggered procedures are given access to the previous and new state of affected records, which allows them to enforce temporal constraints on how records may change over time. Triggers can also be set up to automatically modify other records in response to a change. For example, a triggered procedure might detect that a product's identifier has changed, and update existing order records to reflect the new identifier. A triggered procedure could also create new table entries or remove old entries, for example removing all access rights when an employee is removed from the database. Embedding triggered procedures in the database allow state change validation and record updating to happen automatically, regardless of how the data change is initiated.

1.2.4 Access Control Systems

As discussed in the background section, access control systems provide a first line of defense for both confidentiality and integrity of data. The vast majority of operating systems include at least discretionary access control, and extensive research has gone into providing strong, fine-grained protection. While most of that research is beyond the scope of this discussion, it is worth discussing a selection of projects.

Type-based mandatory access control systems view computer systems as a collection of active entities (users or processes) called subjects, and a collection of passive entities referred to as objects. Subjects are then grouped into domains, and objects into types, so access control rules can be expressed as a table. When a subject wishes to access an object, the domain of the subject is matched to the type of the object in the table, revealing the relevant access permissions. A similar lookup is performed if one subject wishes to access another for some reason.

Unfortunately, while these systems work well in theory, a table-based view of permissions does not map well to file or process hierarchies commonly used in operating systems. This means that they generally cannot exploit features like inheritance through a file system or process tree. Authoring permissions tables is further complicated by the vast and ever-growing number of files and processes present on a modern computer system.

Badger, et al., proposed their Domain and Type Enforcement (DTE) [9] [10] system specifically to address these difficulties. They provide the Domain and Type Enforcement Language (DTEL) which provides a symbolic, high-level, and, most importantly, human-friendly means of expressing DTE rules. Because their language is designed to be symbolic, reusing rules in different contexts, or even on different machines, is straightforward and works largely independent of the enforcement system itself. Their enforcement system is also flexible enough to allow access controls to be enforced even if the underlying file system or applications are unaware of DTE itself.

User-focused mandatory and discretionary access control systems can also introduce management trouble if permissions should be tied to a position or role rather than a particular user. For example, a departmental webmaster needs full access to the website, but hiring a new webmaster requires that editing rights be taken from the old webmaster and given to the new one. Ferraiolo, et al., propose a role-based access control (RBAC) system to account for this [20]. Under an RBAC arrangement, users are assigned one or more roles, only one of which may be active at a time. Each role has an associated set of permitted operations. When an access control check needs to be made, the user's active role is used to determine whether the operation is permitted. Checking based on the user's role, rather than the user themselves, introduces minimal overhead during a normal check,

and makes updating role-based permissions trivially easy.

In 2001, the National Security Agency started presenting its Security Enhanced Linux project [46]. SELinux – which has been integrated into Linux kernel since the the kernel's 2.6 release in 2003 – is designed to provide a flexible framework for adding security policies to a Linux system. It supports both DTE and RBAC systems, as well as a variety of other functionality that can be used to attain incredibly fine-grained access control policies.

1.2.5 Integrity via Access Control

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 [54], and approaches relying on the fine-grained customizability of DTE [31]. 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.

1.2.6 Linux Security Modules

A sizable number of security projects for Linux were started around 2000, including SELinux. Most of these projects required patching the kernel to introduce their functionality, which meant that it was difficult to integrate more than one into the official kernel. Rather than select a single project to integrate, or force all projects to remain disparate patches, a standard API was created to allow security systems to be treated like modules which could be built into the kernel and then selectively enabled.

The Linux Security Modules system [70] was designed to provide the hooks required to (re-)implement most of the access control systems under development at the time. It also had a specific design goal of being able to provide all of the functionality required to replicate the kernel's existing support for the POSIX.1e capabilities system. For the most part, this entailed allowing the security module to further restrict a permission check (that is, disallow an access the kernel would have allowed under discretionary access control), though supporting capabilities also required a means to grant permission that would have otherwise been denied.

Building the desired support consisted of two main changes to the kernel itself. First, an additional, opaque field was added to nine of the core kernel structures, to allow a security module to attach persistent information to objects under its control. This field is expected to be managed by the module itself, though the kernel ensures that it is passed around correctly.

Additionally, hooks were added to many of the internal functions to facilitate fully transparent intervention by the security module. In many cases, two hooks are included in a given kernel function. The first allows the security module to perform a permissions check of its own, and the second allows for updating the security data maintained by the module. Interestingly, hooks were inserted into a number of functions which would not otherwise perform access control validation, such as the call used to write data to a file. This is to allows modules to, for example, check whether permissions have changed since a file was opened, and potentially offers an opportunity to process file integrity every time data is written.

Although the functionality provided by the Linux Security Modules system is not sufficient for all security modules, a number of systems have switched to using it. This includes shifting the

aforementioned POSIX.1e capabilities system into LSM, as well as SELinux and an implementation of DTE for Linux. The SELinux team, in particular, have provided extensive documentation of porting their system onto LSM [59].

1.2.7 Fault Tolerance

Broadly speaking, where computer security is concerned with issues arising from user's activities, fault tolerance seeks to mitigate problems arising from hardware failures, the environment, and other external sources. We can think of the state space of a program as the set of all possible combinations of values of a program's variables, with state transitions provided by the code that changes the variables. Any state that the program could arrive at during normal operation is considered a valid state. The remainder are invalid. When a fault occurs, the program jumps unpredictably to another state within its state space. This destination state may be a valid one, though it may also be an invalid state.

Fault tolerant software possesses two properties: closure and convergence [5] [4]. Closure requires that a program starting at a valid state will remain in valid states as it operates. This effectively means that the program will not crash, or otherwise misbehave, due to programming errors or the like. Ideally, all software should possess the closure property, though this is of course hard to guarantee in practice.

A fault could be transient (e.g. a cosmic ray flipping a bit) or ongoing (e.g. line noise affecting stability). Convergence requires that a program be able to return to a valid state (and therefore correct operation) once it is no longer receiving faults. Combined with closure, this will allow the program to continue in valid states until it is perturbed again. It is not, strictly speaking, required that a fault tolerant program return to completely normal operation. It could, for example, shut itself down in a controlled fashion. It is, however, preferable that the program resume normal operation if possible.

1.2.8 Correctness

Program correctness is focused on the goal of proving that a program will always operate according to some specification. The structure of Hoare logic [27] allows program behavior to be analyzed in terms of triples of preconditions, commands, and postconditions. For each Hoare triple, one generates a proof that, given the precondition, once the command completes the postcondition will be true. If all possible inputs to the precondition meet with valid postconditions, the triple is said to be correct. By chaining together correct triples, it is possible to prove the correctness of a program in a way that mirrors the program's structure.

There are two general flavors of correctness proof. "Partial" correctness stipulates that a program will behave according to its specification assuming all commands terminate, but makes no claims if a command fails to halt. The stronger "total" correctness [47] requires that a command be proven to always halt, in addition to the pre- and postconditions being true.

Some programming languages are better suited to proving correctness than others. This imposes certain limits on the types of programs that are generally proven, though proofs of algorithm correctness independent of actual implementation are both common and useful.

1.3 Motivation

In the preceding sections, we laid out the highlights of computer security, and discussed a number of integrity models and other research ventures relevant to security in general and integrity in particular.

We will now discuss some of the reasons those research endeavors do not provide workable integrity protection throughout a computer system.

1.3.1 Implementability and Generality

While Biba and Clark-Wilson are well known as starting points for discussing integrity in computer systems, they possess a number of restrictions that prevent general implementation. Biba's model provides for origin integrity, but only safeguards data integrity insofar as subjects can be trusted to do the right thing. The model assumes that subjects will always behave in a manner befitting their assigned integrity level. This is reasonable if we consider subjects in the abstract, but falls apart when confronted by real world. The rules preventing a subject from reading below their integrity level cannot be enforced outside of the computer system. Therefore, it is entirely possible for a user acting in good faith to encounter low-integrity data in their daily life and then inadvertently carry it into a high-integrity data store. Further, if a trusted user account is compromised, an attacker can use the trusted user's integrity level to modify high integrity resources, actively subverting the trustworthiness of the data. This makes a practical implementation of Biba problematic, as the access restrictions would make life harder on high-integrity subjects without actually protecting the integrity of the data store when it matters.

Clark-Wilson manages to protect data integrity in the face of subverted user accounts, but it imposes other difficulties that make it impractical to implement as stated. Specifically, the notion of a transformation procedure works well in theory, but runs into trouble if a single application is capable of performing many different transformations. For example, a text editor could be used to produce HTML files, or to edit the Unix password file. To comply with Clark-Wilson, the editor would have to be certified to always produce valid HTML, and to always produce a valid password file. This is, however, impossible, as these two formats are radically different. Therefore, in order to have any hope of implementing Clark-Wilson, the text editor would have to be broken into an HTML editor, a password file editor, and so on for every other plain text format. This is clearly an untenable requirement. Further, administrators would have to certify each and every one of those editors, which would be time consuming at best, and may prove to be deeply impractical or outright impossible in the worst case. Generating vast numbers of single-purpose executables, certifying their correctness, and then managing access control rules for them is an incredibly high cost to pay, even if it could guarantee data integrity in most cases.

Whereas Biba and Clark-Wilson are general but not implementable, database integrity protection is in active use in the wild, but does not generalize well. The constraints provided by SQL are quite powerful, but are limited to the core types already given by the language. In order to validate, for example, a phone number, a database user must craft additional code beyond that provided in SQL. True, some databases support embedding this code within the database, but this doesn't add the new type to the database itself. Altering descriptive constraints also requires modifying the table definitions themselves, which is not to be undertaken lightly. Taken together, while SQL databases provide excellent integrity protection as far as they go, they do not natively provide sufficient functionality to protect general data on their own, or to make refining integrity rules on the fly feasible.

Additionally, while database-driven file systems have been implemented (see, for example, BeOS's BFS [25], or Microsoft's to-be-released-someday WinFS [56]), they do not solve the general integrity problem. Database techniques help to preserve the integrity of the file system itself, but we once again run into a generality problem for protecting individual files. Traditional relational databases simply do not provide the sort of fine-grained integrity controls required to validate individual fields in general, let alone arbitrary files.

1.3.2 Correctness v Integrity

On the face of it, a system comprising only provably correct software would seem not to need separate integrity protections. After all, how could a correct program output incorrect data? However, relying on correctness for integrity runs into two major problems. First, not even a provably correct program will necessarily be able to defend against a user providing well-formed but erroneous data. After all, a person intent on embezzling money can produce transactions that look correct, so some sort of detection system will be required even if the software is completely correct. Similarly, a provably correct text editor could still output a text file which contains the wrong information in the wrong format. This does not actually contradict the correctness of the editor itself, so long as its proven function is "open files, accept changes to them, and faithfully output those changes" rather than, say, "always produce a valid HTML file".

The other problem with correctness as a full defense is that it relies on all of the software on a system being correct. It doesn't matter whether the primary user account management tools are provably correct if other programs (such as a text editor) can also modify user account information. This makes transitioning to an integrity-through-correctness system problematic, especially given the difficulty of proving software correctness. Such a transition would necessarily require either waiting until the entire system has proven counterparts or else putting up with missing functionality until all components are online.

1.3.3 Fault Tolerance v Integrity

Fault tolerance and integrity are, to a very large extent, two aspects of the same overall problem: preventing and recovering from errors in the system state. A system with robust integrity protection will also end up being more resilient in the face of faults which would otherwise have damaged data, and fault-tolerant software is less likely to end up corrupting data. That said integrity is primarily about user actions, whereas fault tolerance is responding to environment actions. A fault tolerant program may well be able to recover from environmental effects, but it isn't really guaranteed to always write valid data. Likewise, a fault tolerant program may be about to cope with corrupted data being received from the system, but that does not necessarily start the process of fixing the data. Fault tolerance and integrity work together well, but neither can replace the other.

1.4 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 focused on safeguarding specific systems at the cost of generality – as with databases – or provide for general protection but are not well-suited to real-world implementation – as with models of Biba or Clark and Wilson.

In the remainder of this document, we present tools for providing general, implementable data integrity protections. The next chapter presents Maia, a language for describing integrity constraints on arbitrary file types. We follow this with a discussion of Mandos, a Linux Security Module which provides mandatory integrity protection by verifying files after they are modified to ensure they are still valid. We also present a selection of benchmarks which show that, while Maia and Mandos necessarily impose a performance penalty, this penalty is quite small. We conclude the dissertation as a whole with discussion of future research areas derived from this work, as well as sample Maia specifications and a formal specification of Maia's syntax.

Chapter 2: 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[1] (MY-uh), a specification language which can describe the structure of any context free language as well as semantic rules for the contents of a file. Using Flex and Bison, we then demonstrate that it is possible to transform a Maia specification into a working verifier program with minimal human intervention.

We begin by presenting a variety of special purpose tools and parsers which do not meet our needs. We then present the governing principles of Maia, which is designed to allow users to specify what constitutes a valid file without specifying how to actually verify the file. We follow up with a detailed discussion of syntax and semantic specification, automated set building, and other features of Maia. We conclude with a discussion of a prototype implementation of a Maia verifier generator. A selection of Maia specifications, both complete and adapted for our verifier generator, can be found in Appendix A.

2.1 Related Work

Many tools exist to verify particular file formats. XML, which is widely used online and for storing configuration data, has no fewer than three different verifier systems: DTD [66], XML Schema [67] [68], and RELAX NG [65]. 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 protected particular file types, they cannot be generalized to protecting other file formats. Instead, we need an approach which will allow us to generate new verifiers easily.

Parser generators are commonly used in developing new programming languages, and can be applied to the problem of creating verifiers. Lex [38] and Yacc [33], and their successors Flex and Bison generate robust, fast parsers which can be embedded in C or C++ programs. ANTLR [52] 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 [30], 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 [32], are able to cope with limited context sensitivity, but still require programmer assistance to perform semantic checks.

[1] "Then those of the Ainur who desired it arose and entered into the World at the beginning of Time; and it was their task to achieve it, and by their labours to fulfil the vision which they had seen. [...] The Great among these spirits the Elves name the Valar, the Powers of Arda, and Men have often called them gods. [...] With the Valar came other spirits whose being also began before the World, of the same order as the Valar but of less degree. These are the Maiar, the people of the Valar, and their servants and helpers." [61]

Data description languages [22] [21] are designed to provide automated parsing for ad hoc data formats. Tools like PADS [23] 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 with minimal human intervention.

2.2 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 configurations 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 as flexible as possible with regard to the sort 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.

2.3 Model Overview

Within Maia, we model the file verification process as two phases, corresponding to checking the file's syntax and 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 [7][69] 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 the structure of formal logic 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) together.

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 [18], 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 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.

2.4 Syntax Specification

Maia's syntax checker is designed to work like a normal parser as generated by a parser generator. Backus Naur Form has the feature of being able to represent precisely the set of context free languages, which is desirable for parsers. Extensions are normally employed to make specifications more concise and easier to work with, leaving the correspondence to context free languages. By adopting an Extended BNF for Maia, we gain this correspondence to context free languages.

There are many EBNF variants in use, so we choose to base ours on the ISO/IEC 14977 [29] standard. This standard provides robust semantics, but we feel that the syntax could be improved to more closely match other tools administrators are familiar with, such as regular expressions. We also provide some constructs which are not directly supported, but can be built from existing constructs, and an extension that makes Maia grammars context sensitive in certain narrow cases. The following is a summary of our changes to ISO/IEC 14977, along with examples comparing the standard's syntax to its Maia counterpart. Anything left unspecified follows directly from the ISO/IEC specification.

2.4.1 Nonterminal Names

The standard requires that nonterminal names consist of a letter, optionally followed by additional letters or numbers. We retain this requirement for all nonterminal, variable, and template names in Maia, along with a restriction that names may not conflict with keywords. While names are otherwise unconstrained, we apply the following conventions throughout this document:

- Names are drawn from the common name or purpose of the syntactic element.
- Names consist of one or more words, presented in camelCase.
- Names which appear only on the left side of a definition begin with an uppercase letter.
- Names which appear on the right side of a definition begin with a lowercase letter.

A specification for a file type will generally only have one nonterminal whose name begins with an uppercase letter (the "top level nonterminal"), corresponding to the file as a whole. However, a file containing specifications for helper types (e.g. filesystem objects or cryptographic hash formats) may have multiple top level nonterminals. All of these top-level nonterminals should receive uppercase first letters as a way to signal that they are intended for reuse, even though they will appear on the left in definitions in other files.

2.4.2 Concatenation

The standard calls for concatenation to be expressed with a comma. We choose, instead, to indicate concatenation with whitespace. This is in keeping with ANTLR's syntax, and makes definitions cleaner looking.

ISO	`foo = bar , baz ;`
Maia	`foo = bar baz ;`

2.4.3 Optional

While the square brackets ([ ]) used by the standard are conventional for manual pages, we choose to use an appended question mark instead. The question mark is normal for regular expressions, and we can still make groups of symbols optional using grouping operators in conjunction with a question mark. Changing this symbol also makes the square brackets available for other things.

ISO	`[foo]`
Maia	`foo?`

2.4.4 Repetition (0+)

The standard calls for the use of curly braces ({ }) to indicate a case where symbols may be repeated zero or more times. We replace this with an appended asterisk (*), which is typical for a regular expression. Again, grouping can be handled explicitly, if needed.

ISO	`{foo}`
Maia	`foo*`

2.4.5 Repetition (1+)

While the standard provides support for zero-or-more repetitions, and exactly N repetitions, it does not directly support having one-or-more repetitions of a symbol. Maia provides this functionality using the + operator.

ISO	`foo, {foo}`
Maia	`foo+`

2.4.6 Repetition (Exactly N)

In Maia, we express a requirement for exactly N repetitions of some symbol by following the symbol with {N}. Once again, this is similar to a regular expression approach. It also extends nicely into specifying ranges.

ISO	`N * foo`
Maia	`foo{N}`

2.4.7 Repetition (Range)

ISO/IEC 14977 does not include a way to directly express that a symbol should be repeated between M and N times. In Maia, we accomplish this by following the symbol with $\{M,\ N\}$. Either M or N, but not both, may be omitted. If no starting value is provided, the symbol may be present between 0 and N times. If no end value is provided, the value must appear at least M times. In either case, the comma must be provided.

ISO	`M * foo, N-M * [foo]`
Maia	`foo{M, N}`

ISO	`ISO: M * foo, {foo}`
Maia	`foo{M, }`

ISO	`ISO: N * [foo]`
Maia	`foo{ ,N}`

2.4.8 Comments

We choose to use C-style // and /* */ comments, with their customary meanings, within both the syntax and semantic sections of Maia. This has the advantage of being familiar and eliminating the need for end-of-comment marks on single-line comments.

ISO	`(* this is a comment *)`
Maia	`/* this is also a comment */` `//this, too`

2.4.9 Special Sequences

The standard includes syntax for adding other sequences to the language. As we are already modifying the standard, we omit the special sequence syntax and add our own syntax directly.

2.4.10 Character Classes

Character classes are normal components of regular expression parsers, and are even present in some shells. While they do not provide functionality that cannot be had using alternation, we choose to provide them to help make descriptions more compact and readable.

ISO	`'a' | 'b' | 'c' | ... | 'z'`
Maia	`[a-z]`

2.4.11 Wildcard Match (Dot)

As a special case of character classes, we also provide the dot (".") which matches any single byte, including line breaks or whitespace. When used with the repetition facilities, this allows for extracting data whose value need not be examined further.

2.4.12 Regular Expressions

While the regular languages are a subset of the context free languages, and therefore can be expression using EBNF, we provide the option to explicitly use regular expressions. Regular expressions much be enclosed in slashes (/), and are required to use Perl 5 regular expression syntax [53]. All regular expressions are evaluated as though the `s` modifier was applied, causing "." to match all characters, including whitespace.

2.4.13 Numeric Parsing

In addition to its library and inclusion system (see Section 2.7.3), Maia provides a small set of nonterminals specifically geared toward parsing numeric values. These nonterminals exist primarily to allow specifications to make use of parsed numeric values later on, either within semantic rules or as part of context aware parsing.

To make the numeric types as simple as possible, the base types correspond to a single character or byte within the number. The repetition system can then be applied to provide additional characters or bytes as needed. This allows us to avoid the need to predefine the lengths of all the different numeric representations that may appear in a file.

Maia provides capabilities for parsing both string-encoded and binary-encoded values. The repetition system is the same in either case, though string-based encodings impose some added complications on repetition that will be described where relevant.

String-Encoded Numbers

Maia's string parsing facilities resemble the capabilities of C's `scanf()` family of functions. The `StringDec` nonterminal matches integers with decimal encoding, in much the same way as the conversion specifier `%d`. Values may be preceded by leading zeroes, which must occur after the "-" for negative values. When dealing with negative values, the leading "-" counts toward the length of the value, so a `StringDec{4}` would match values between -999 and 9999. We also provide the nonterminals `StringPosDec` and `StringNegDec`, which work similarly to `StringDec` but match only positive or negative values, respectively. `StringNegDec` values must begin with a "-", which is still counted toward the length of the value for repetition. A parsed `StringNegDec`'s value will also be negative for the purposes of semantic rules.

In addition to parsing decimal-encoded values, Maia can parse hexadecimal values via the `StringHex` type, which is equivalent to `%x`. The value is considered to be unsigned, and will be matched case-insensitively. A leading "0x" or "0X" is optional, but counts toward the total length if present. Thus, `StringHex{4}` will match 0xFF or FFFF, but not 0xFFFF. As with decimal encodings, leading zeroes are permitted though they must follow the "0x", if present.

We also provide the `StringInt` nonterminal, which matches either a `StringDec` or a `StringHex` with some restrictions. The parsed value will be assumed to be decimal unless it is preceded by "0x" or "0X". Thus, the value "0A83" would be valid for a `StringHex` but not for a `StringInt`. This restriction is enforced to remove ambiguity in the case of a value consisting of only the digits 0 - 9. Such a value could be either decimal encoded or hexadecimal encoded, so we choose to always interpret it as decimal. Leading "-", "0x", and "0X" strings still count toward the length of the value, so `StringInt{4}` would match decimal values from -999 to 9999, or hexadecimal values up to 0xFF. This behavior is equivalent to C's `%i`, except that Maia does not parse octal values and will treat numbers beginning with a "0" (but not "0x" or "0X") as decimal.

The `StringReal` is equivalent to `%g` and matches text representations of real numbers. It will accept positive or negative real values encoded in decimal. Leading zeroes are supported, but not

required, and the integer portion of the value may be omitted if it is zero. Likewise, the decimal point and trailing zero may be omitted if the value is an integer. `StringReal` will also accept values containing exponents marked by "e" or "E". For the purposes of repetition, "-", ".", "e", and "E" count toward the length of the value. The internal representation will be at least double precision, but no other guarantees are made.

Binary-Encoded Numbers

We provide multiple types for matching binary-encoded integers. The underlying grammar for all of these types match a single byte, so the added complexity comes from the need to handle signed-ness and byte ordering. The basic types are `BigEndianInt`, `LittleEndianInt`, and `HostInt`, which match, respectively, integers with big-endian encodings, little-endian encodings, and host byte ordering. All three types assume the underlying value is signed. If this is not the case, the Unsigned variants (`UnsignedBigEndianInt`, `UnsignedLittleEndianInt`, and `UnsignedHostInt`) may be used instead. Thus, in order to match a 32-bit signed integer in host-byte order, the repetition `HostInt{4}` would be used.

We also provide the types `BigEndianReal`, `LitleEndianReal`, and `HostReal` which work analogously to the binary integer types. These types support IEEE 754 [28] binary encodings for floating point values. Other real number encodings are not supported. Because we require IEEE 754 encodings, only certain repetition values (2, 4, 8, and 16) correspond to valid representations. Other repetition values are allowed, but their parsed value is undefined.

As a convenience, the Maia standard library should include host-specific definitions of common variable types based on the binary integer types. Thus, a file type that relies on local integer sizes can be defined in a way that requires no architecture-specific knowledge in the specification itself. For example, a 32-bit machine might have the following definitions:

```
 Long = HostInt{4} ;
  Int = HostInt{4} ;
Short = HostInt{2} ;
```

Likewise, the conventional names for binary reals should be provided:

```
     Float = HostReal{4} ;
    Double = HostReal{8} ;
LongDouble = HostReal{16} ;
```

2.4.14 Limited Context Awareness

A number of data types, including PNG images [30] and IMAP messages [17] include a field which specifies the length of a subsequent block. This is a convenient way to handle variable length fields if the parser expects it, but it cannot be represented in a context-free form. Thus, if we wish to be able to verify these formats, we must include functionality that would be impossible in a strictly context-free parser.

We accomplish this by introducing a single context-sensitivity into Maia's syntax specification system. We allow a specification to replace the constant in an exactly-N-times repetition with the name of a nonterminal present in the specification. This is subject to two restrictions: the designated nonterminal must be guaranteed to be present before the repeated symbol, and it must have a definition consisting only of a parsed (string or binary) integer type. In the event that the length-specifying nonterminal is encountered multiple times, the most recent occurrence is used. A syntax specification that fails to meet these conditions is considered invalid.

For example, consider a file which consists of a single integer followed by exactly that many user records. The first part of a Maia specification for this file might look like this:

```
SampleFile = numRecords record{numRecords} ;
numRecords = StringInt+ ;
```

We could verify this file without context sensitivity, using a semantic check on the number of record elements that are present. However, such a check would require parsing the entire file before an error could be detected, which would increase verification time if there are more entries than expected.

Additional complications present themselves if there are multiple fields with variable lengths. The PNG image format makes use of many chunks, each of which is preceded by a length specification. In this case, we cannot simply rely on semantic rules to check the overall length, because any given chunk might be the wrong length. Thus, it is necessary to include our limited context sensitivity to correctly parse some file types.

We also support performing basic mathematical operations as part of the repetition, along with a limited form of C's ternary operator for decision making. This allows us to account for formats like PNG which include a fixed-size chunk in their length specifications. Basic logic is required for formats like the PPM image, where the number of bytes present in the image encoding relies in part on the maximum value of a color channel. These operations may also include other nonterminals, subject to the restrictions outlined above. However, none of these operations is permitted to actually change the value held in any nonterminal.

The ability to account for limited context sensitivity is not unique to Maia. ANTLR's semantic predicates [52] provide a vast toolkit for injecting semantic information back into the parser. Likewise, the combination of a bound variable and a data-dependent constraint in YAKKER (referred to as "attribute-directed parsing" [32]) can produce an equivalent result. Maia's syntax is derived from YAKKER's, but it is deliberately simpler than the other systems' approaches, both to make it more readable and to reduce the number of possible context sensitivities.

2.5 Building Sets

Consider a file consisting of one or more colon separated pairs of user names and hashed passwords, where each pair appears on its own line. The file might begin like this:

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

A Maia syntax specification for the file might look like this:

```
UserFile = userRecord+ ;
userRecord = name ":" password Newline ;
name = [a-z]+ ;
password = [a-z0-9]{32} ;
```

Given this file and description, a traditional parser would produce an abstract syntax tree similar to Figure 2.1. However, this representation is not ideal if we wish to reason about all user names in the file, or the way names and passwords are related. Such checks would require collecting information scattered throughout the tree, either through traversals or manual collection during the parse. To overcome this difficulty, Maia semantic specifications abstract related data into ordered sets which can be automatically generated or manually specified.

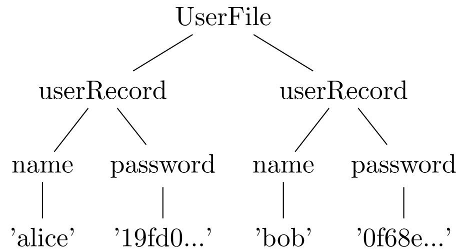


Figure 2.1: Parse Tree for UserFile Example

2.5.1 Automated Set Construction

Maia's syntax checking phase automatically constructs and populates sets based on the syntax specification. Each nonterminal entry in the specification becomes an ordered set with the same name during semantic analysis. Thus, our example would have the sets *UserFile*, *userRecord*, *name*, and *password*. While this set building will always take place for nonterminals appearing in semantic rules, we do not require that Maia implementations construct sets if they are not being used. Implementations are also free to implement the sets however they wish, so long as they provide the correct behavior.

Nonterminals which contain at most one other nonterminal (like *name* and *password*) are converted into simple sets containing the input chunks that matched the parser rule. Elements always appear in the order they were encountered by the parser. A simple set can be thought of an as array, though many types of constraints can be expressed without referring to array indices at all. Thus, the semantic system would have access to a set containing all of the parsed names when validating our sample file, which we could visualize as a Perl-style array:

```
name = ( 'alice', 'bob' )
```

Nonterminals which contain at least two other nonterminals (like *userRecord*) are also converted into sets, which we refer to as "compound sets". While it is possible to examine the raw input corresponding to a compound set element, this is not the only form for the data. Each compound set element also contains what is, essentially, an associative array or map using the included nonterminal names as keys, and their corresponding input chunks as values. Thus, a single record in the sample file might contain the following (expressed in the style of a Perl hash, for clarity):

```
{
    'name' => 'alice',
    'password' => '19fd0...',
}
```

By extension, the set *userRecord* would be, in essence, an array of maps:

```
userRecord = (
    {   'name' => 'alice',
        'password' => '19fd0...' }
    ,
    {   'name' => 'bob',
        'password' => '0f68e...' }
)
```

This conversion from nested nonterminals to maps can happen as many times as necessary, allowing arbitrarily complex files to be accessed at any level of granularity.

The primary advantage of these nested nonterminals is that they provide a means of associating entries in otherwise independent sets. Given only the `passwd` file sets *name* and *uid* and the constraint that the sets are ordered, there is no way to determine which name corresponds to which UID. Thus, we could not specify the rule "the user 'root' must have UID 0". The set *passwdRecord*, however, naturally provides a mechanism to pair name and UID, allowing us to express the requirement as a function of passwd records, rather than of the *name* and *uid* sets.

2.5.2 Explicitly Constructed Sets

Sometimes it is desirable to fully define a set's contents, rather than having it built from a file. Explicit set construction makes use of the angle brackets to enclose an element list. Elements are specified as a comma-separated list of either strings (enclosed by single or double quotation marks) or numbers (unenclosed). Explicitly constructed sets may not mix strings and numbers. Thus, both of these are valid:

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

but this is not:

```
model = < 60, 70, "70D", 85, "85D", "P85D" > ;
```

Constructed sets preserve the ordering of their members, so we could have the integrity system consider the ranking of classifications as part of its rules. Constructed sets are only available to semantic rules, and cannot be used as part of a syntax specification.

2.5.3 Joining Existing Sets

Occasionally we wish to reason about the properties of two or more sets taken as a unit, but do not have a suitable nonterminal in the syntax specification. Consider a file containing user login data, including user names and login domains. User names can be reused across domains, and each domain can have multiple users, but each user name / domain pair may occur at most once. If the name and domain fields are adjacent, we can create a nonterminal which will have only those fields, providing a convenient way to perform uniqueness testing. However, if additional fields are between the name and domain (e.g. a password hash), we would end up with a set of users and a set of domains, but no way to connect them without including the intervening fields, like so:

```
user = ( 'alice', 'alice', 'bob' )
domain = ( 'DOMAIN1', 'DOMAIN2', 'DOMAIN2' )
```

To get around this problem, Maia provides a facility to create a new set by performing a per-element join operation (often referred to as a "zip" in functional languages) on two or more existing sets. The sets to be joined are required to contain the same number of elements. Attempting to join mismatched sets is considered an error.

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.

Thus, we might have:

```
userDomain = < user .  domain > ;
```

which would give us this set:

```
userDomain = ( 'aliceDOMAIN1', 'aliceDOMAIN2', 'bobDOMAIN2' )
```

Set names in a join may be replaced by string literals. In that case, the literal will be present in each new element, at the specified location. Thus, we might specify

```
userAtDomain = < user .  '@' .  domain > ;
```

which produces

```
userAtDomain = ( 'alice@DOMAIN1', 'alice@DOMAIN2', 'bob@DOMAIN2' )
```

If the sets to be joined are all parsed numeric types, mathematical operations may be used in place of concatenation. In that case, the connector may be one of +, -, *, /, and %, which retain their customary meanings of addition, subtraction, multiplication, division, and modulus. Numeric literals may be supplied in place of set names, and all values will be considered real (rather than integers) for the purposes of arithmetic. No non-numeric value is associated with set elements generated by a numeric join operation.

Parsed numeric values may be joined with other sets using concatenation. This will use the raw input chunk, rather than the parsed value. It is considered a syntax error to apply mathematical operators to a set which does not use a parsed numeric type.

2.6 Semantic Specification

Structurally, Maia semantic rules are a straightforward adaptation of first-order logic. For example, consider the rule "there must be at least one user with a UID of 0". Given the set *uid*, which contains the UIDs of all users, we could express this formally as:

$$\exists u \in uid : u == 0$$

The equivalent Maia is quite similar:

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

Aside from replacing ∃ with `exists`, the only significant change is that there is no need to specifically refer to a single element in a set as part of a Maia rule. Set names on the right sides of rules are automatically understood to refer to single elements in the sets, rather than the entire set.

We might have also rules which place a requirement on all elements of a set, for example "UIDs must be in the range 0 to 32767, inclusive". In formal logic, we would employ a universal quantifier:

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

which translates into Maia as:

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

While Maia semantic rules are meant to be similar to first-order logic, they differ in important ways. First, whether a logical statement is true depends only on whether it can be derived from underlying axioms via the rules of formal logic. The truth of a Maia rule, by contrast, is dependent on the input file. Second, Maia rules may use information which is not available in formal logic. For example, our `count()` operator allows rules to use the number of elements which have a property, while formal logic is limited to specifying only that all, some, or no elements have a property.

In the remainder of this section, we describe the basic form of a Maia semantic rule. We then present language features for building constraints on individual set elements and using compound sets to group related sets. This is followed by a discussion of constraints between elements within a set, count-based constraints, and Maia enforcement levels.

2.6.1 Rule Structure

Semantic rules have this basic form:

```
context :  constraint ;
```

The context for the rule specifies a set to which the rule applies and whether the rule must be true for all set elements or at least one element, along with the set under consideration. Semantic rules may apply to any parser-constructed or joined set, but not to sets whose contents are explicitly specified. A rule is considered to have been broken if at least one set element does not satisfy the constraint of a `forEvery` rule, or no elements satisfy the constraint of `exists` rule. This maps directly to the normal meanings of the universal (∀) and existential (∃) quantifiers in formal logic. As universal rules are more common than existential ones, the `forEvery` keyword may be omitted. Note that a verifier must scan the entire file to determine that an `exists` rule has been violated, but can detect a `forEvery` violation at the element that violated the rule.

The right side of the rule provides one or more boolean constraints which will be evaluated for each set element until the rule is satisfied. Elements are evaluated one at a time, in the order they are encountered during parsing. Whether evaluation happens during the parse or in a separate pass is up to the Maia implementation. An implementation is permitted, but not required, to stop evaluating `exists` rules after an element has satisfied the constraint. Failure to satisfy the constraint of a `forEvery` rule may terminate verification, depending on the rule's enforcement level (Section 2.6.10).

Except in certain cases (see Section 2.6.6 and Section 2.6.7), set names which appear in a constraint refer to the current element of that set, rather than the set as a whole. Thus, constraints may only refer to elements of the set specified in the context or its subcomponents for compound sets, as these are the only elements guaranteed to be available when the element is evaluated.

As with syntax specifications, the semicolon is used to denote the end of a rule, and whitespace is ignored except to separate tokens. Semantic specifications are case sensitive, both for keywords and set names.

2.6.2 Logical Comparisons and Connectors

Constraints are allowed to make use of any of the following logical comparisons, which carry their conventional meanings: `==`, `!=`, `<`, `<=`, `>`, and `>=`. These operators are used for both numeric and string-based comparisons. Elements with parsed numeric types are compared as numbers, exactly as they would be in C. All other elements are compared lexicographically, in the style of `strcmp(3)`.

Elements may be compared to other elements or to numeric or string literals. Numeric comparisons may use the mathematical operators `+`, `-`, `*`, `/`, `%` (modulo), and `^` (exponentiation), which

have their customary meaning and precedence. Division (`/`) does not truncate, and always returns a real number. The dot operator (`.`) may be used to concatenate the raw input of elements or string literals. If an element does not have a parsed numeric type, it is considered an error to compare it to a numeric literal or apply a numeric operator. Parsed types may be concatenated, the result of which is no longer considered numeric.

The standard logical connectors `and`, `or`, and `xor` are provided and have their conventional meanings. We also provide the connectors `implies` and `iff` which correspond to implication and bicondition (if-and-only-if) operations. Implication, in particular, can be used to apply constraints only where certain requirements are met. The logical operator `not` may be used to negate a constraint, but cannot connect constraints.

Logical operators have lower precedence than any other operator, and obey the following relationship:

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

2.6.3 Grouping

The parentheses may be used to explicitly group constraints, and provide the highest precedence. Parentheses may appear as part of a comparison, and work as expected. The parentheses used in the `length()`, `count()`, and `blackbox()` operators, and in templates, serve only to collect arguments for the operators, and do not otherwise affect precedence.

2.6.4 Regular Expressions

Constraints are allowed to involve regular expression matching using the operator ∼ to mean "matches following the regular expression" and !∼ to mean "does not match the following regular expression". Using a regular expression on a parsed numeric element will perform the match on the raw input for the element. Regular expressions much be enclosed in slashes (/), and are required to use Perl 5 regular expression syntax [53]. All regular expressions are evaluated as though the `s` modifier was applied, causing "." to match all characters, including whitespace. Back-references and captures may be used, but their values are not accessible outside of the regular expression.

2.6.5 Element Length

It is possible to determine the length of an element in a set using the `length()` operator. When applied to a non-numeric element, `length()` returns the number of characters in the underlying input chunk, as though a call were made to the C library function `strlen()`. When applied to a parsed numeric value, `length()` returns the size of the underlying representation, not necessarily the most compact form. Thus, a `StringInt` that matched "01" would have a different length from one that matched "1", even though both have a parsed value of 1. The binary numeric types will necessarily have constant length, so `HostReal`{4} will have length four regardless of its value.

When applied to a compound set, `length()` still returns the length of the entire input chunk. This will include any terminals which are matched as part of the chunk, and thus the length of a compound set may be greater than the sum of the lengths of its component nonterminals. The length of an element created by joining sets with concatenation is the sum of the lengths of the components, plus the lengths of any literals that were added. Taking the length of a numerically joined element is an error.

2.6.6 Set Inclusion

It is possible to determine whether a set element is also a member of another set using the keyword `in`, which has the following structure:

```
element in setName
```

where `element` is the name of some set which may be constrained in the current rule and `setName` is the name of some other set. The set name may be replaced by a set explicitly constructed in the style of Section 2.5.2. In the case of compound sets one may ask whether an element in one subset is also an element of a different subset (e.g. `uid in gid`). Asking whether an element is a member of its own set is disallowed, as that constraint would always be vacuously true.

The Maia implementation may perform set membership checking however it wishes. However, it must always be the case that an `in` constraint is true if and only if there is at least one element,e, in *setName* for which the comparison `element == e` is true. If *setName* contains parsed numeric values, the constrained element must also belong to a parsed numeric type and the comparison will be performed numerically. If *setName* is not numeric, the comparison will be performed lexicographically, even if the constrained element has a parsed numeric value.

In some cases, as with the permitted cypher types in the SSH configuration file [45], it is possible to handle set inclusion with either a syntactic or semantic rule. We recommend performing the check at the syntax level, as this will definitely happen during initial parsing rather than potentially being delayed until semantic checking. If a file format permits options which are disallowed by local policy (e.g. weak cyphers in SSH configurations), we advise structuring the Maia specification so that the syntax rules accept any valid option and undesirable options are subsequently eliminated via semantic rules. For example, one could create a *disallowedCyphers* set which contains the cyphers which are not permitted under local policy and then include a rule like

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

This structure makes the fact that certain options are disallowed explicitly, rather than relying on documentation to explain why they are absent. It also makes tailoring existing rules to local needs easier when paired with Maia's inclusion system (Section 2.7.3). Baseline rules can encode validity from the perspective of software using the file, and then local policy can be layered on via additional semantic rules.

2.6.7 Count-Based Constraints

It may be desirable to express constraints about the number of times a particular condition is met. Some constraints, like "there must be exactly N consecutive occurrences of this nonterminal," can be verified as a function of syntax checking. However, the parser cannot reasonably verify requirements like "a user may belong to no more than four groups", which requires a semantic count facility.

The counting facility, then, is intended to return the number of times a particular condition is satisfied within a set. This number can then be used as part of a standard semantic constraint. The count operator takes the following form:

```
count(setName, constraint)
```

where `setName` may be any set and constraint is any valid semantic constraint that would be valid for the named set. If no constraint is specified, the total number of set elements is returned. Otherwise, `count()` will return the number of elements which satisfy the constraint, which will be in the range $[0, |setName|]$.

Constraints may employ any set names that may be used as elements in the current rule, in addition to those which are valid for the named set. Thus, we can express the requirement that a user belong to no more than four groups as follows:

```
forEvery userName :  count(groupUser, groupUser == userName) <= 4 ;
```

Note that even though the rule as a whole applies to *userName*, `count()` is being applied to the *groupUser*. Thus, counts need not be performed on the context set of a rule.

2.6.8 Using Compound Sets

As described in Section 2.5.1, nonterminals which contain multiple nonterminals are converted into compound sets. Rules for these compound sets may refer to included nonterminals as needed. When employing compound sets in this way, all included nonterminal values will be drawn from the same syntactic element, allowing us to associate related values from different simple sets. Thus, we can use a rule like

```
passwdRecord :  name == "root" implies uid == 0 ;
```

to express the requirement that the root user must have UID 0.

Dot notation (i.e. `setName.setMember`) may be used to explicitly refer to members of a compound set if the set is being used in a rule with a different context, or it is otherwise unclear which set member is intended.

2.6.9 Index-based Constraints

Section 2.5.1 notes that set elements retain information about the order in which they were encountered during parsing. Maia allows rules to access order information using an indexing operator. Using this operator – denoted by square brackets following a set name – it is possible to place explicit or implicit constraints on the order of set elements.

Explicit Indexing

Under explicit indexing, a specification requires that an element appear at a particular place in its set. Sets are always 0-indexed. Thus, if we have a set of usernames called *name*, we can require that "root" be the first element of the set as follows:

```
forEvery name :  name[0] == "root" ;
```

In this case, the rule could also be written as an `exists`, because it relies on a property of only one set element.

Explicit indexing may use parsed-numeric sets for indexes, which may be acted upon by any of the operators allowed in numeric comparisons. For example, the EXT4 file system [1] contains both inodes, which locate data on the disk, and directory entries which map file names to inodes. Inodes are stored in what amounts to an array, so an inode number is an index into that array. Directory entries contain both an inode number and a copy of the file's type, which is also stored in the inode. Let *dirEntry* be the compound set of directory entries, which includes an *inodeNumber* and *fileType*. Further, let *inode* be the compound set of inode entries, including its own *fileType*. Using these, we can express a requirement that the file type reported in a directory entry match the type in the corresponding inode:

```
forEvery dirEntry :  inode[inodeNumber].fileType == fileType ;
```

Implicit Indexing

It is also possible to construct indexed rules which use locally-defined index variables to compare set elements to one another. In this case, the numeric literal or set name in an explicitly indexed rule is replaced with a new variable name. For example, we can use `name[i]` to refer to the `i`th element of *name*. Any unknown name appearing within an indexing operation is taken to be a new index variable. We prefer i, j, k, etc for index variable names, as they are traditional in this sort of application.

Maia allows implicitly indexed rules to have one or more index variables. Assuming *heap* contains an array-like implementation of a max-heap, we can impose the parental dominance requirement using a single index:

```
forEvery heap :  heap[i] >= heap[2*i] and heap[i] >= heap[2*i + 1] ;
```

Rules can express constraints between arbitrary groups of elements by using multiple index variables. For example, the following rule requires that all elements of *name* be unique:

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

In essence, the rule is processed by generating every possible pairing of `i` and `j` (where both variables run from 0 to `count(name) - 1`), and then comparing the values of `name[i]` and `name[j]`.

Rules may also refer directly to the index variables, in addition to using them as indices into the sets. Thus, the following rule can be used to require that priorities are non-decreasing:

```
forEvery priority :  priority[i] <= priority[j] iff i < j ;
```

Similar rules can be derived for other ordering constraints. Maia automatically enforces a requirement that index variables not be equal during comparisons, so there is no need to explicitly include `iff i != j` in our uniqueness test. Any number of indices may be used within a rule, though as a practical matter the number of combinations to check grows very quickly with the number of indices.

2.6.10 Enforcement Levels

While most rules are intended to be enforced at all times, there are occasions when rule violations aren't actually a problem. For example, the login system on some Linux distributions will accept user names which contain capital letters, even though the manual pages for `passwd` specify that names should not contain capitals. Thus, the presence of a capital letter might not be a full violation, but could merit a warning. There could also be rules which are useful in certain rare cases, but not in general. To meet these needs, we provide support for variable enforcement levels.

The level is specified at the beginning of a rule, as follows:

```
(level) rule
```

where `rule` is any valid semantic enforcement rule, and `level` is one of the following:

`require` This rule enforces a fundamental property of the underlying data and must always be processed. A violation is considered a failure of the file overall and should halt processing. `require` is the default, and need not be specified explicitly.

`warn` This rule corresponds to a tenant of good style or standard practice, but is not actually required to hold. The rule will be checked by default, but a violation does not necessarily invalidate the file or halt processing.

`info` This rule is largely informative or only relevant in rare circumstances. It is not processed by default, but may be activated by the verifier. A failure does not necessarily invalidate the file or halt processing.

In the context of a compiler, these rules would correspond, respectively, to an error, a warning, and a warning that is only reported if additional output is requested (e.g. via gcc or clang's `-Wall`). As with compilers, an implementation may provide a facility to change the default behavior of the rule levels. For example, informative rules could become warnings, warnings could be elevated to full errors or suppressed, or required rules could be adjusted to not halt processing.

2.7 Other Language Features

Thus far, we have described the core functionality of Maia. In this section, we present some additional facilities aimed at making Maia descriptions easier to create and maintain.

2.7.1 Semantic Templates

Some rule constructs are common across specifications, or even within the same specification. For example, we may wish to require that names be unique or that data be sorted in some way. Maia provides the ability to create semantic templates, which provide for the reuse of rules and can make specifications more concise or readable.

Templates are inspired by C's macros, and work in approximately the same way. The template is defined with set name placeholders in what will become the rule. When the specification is processed, template invocations are replaced with the contents of the template, swapping placeholders for actual set names.

For example, we can define an "isUnique()" template rule like this:

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

We can then invoke the template to require that all user names in the set *name* are unique

```
name isUnique() ;
```

which becomes

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

when the specification is evaluated.

Templates are required to involve at least one set name, but may work on additional sets if desired. To define a template, we signal that it is a template, provide a placeholder for the required set name, give the template a name, provide additional placeholder names if necessary, and finally specify the replacement text.

The placeholder set name provided between `template` and the template name ("set" in the "isUnique()" template, above) is the primary set for the template. In a fully quantified (rule) template, this will be the set which provides context to the rule. In a constraint template, the primary set name corresponds to a set provided by the current context – either the context set itself or a component of a compound set. Additional set placeholders may be provided in the parentheses, and may be useful if a template will be used with a compound set.

For templates which have only a primary placeholder, something like “set” is sufficient. For templates with multiple placeholders, we advise providing generic but descriptive names, like “directory” or “username”. All instances of placeholders within the replacement text will be replaced with the given set names before the rule is evaluated, so there is no need to worry about conflicts.

Template names should be reflective of the purpose of the template. The template name will always appear immediately after the name of its primary set when the template is invoked. Thus, as a matter of style, we prefer to construct template names so that they read as statements in that context. Thus, we would use names like “isUnique()” or “isOwnedBy()” rather than “unique()” or “owner()”.

Templates may provide either completely quantified rules or individual, unquantified constraints. (“isUnique()” would be a complete rule, while “isOwnedBy()” is likely to be a constraint.) The replacement text for a template may use essentially any functionality available to a normal semantic rule, including other templates so long as this does not result in circular dependencies. The only other restriction is that a template may never define another template. Replacement text generally should not refer to sets which are not provided via placeholders, but this is not disallowed.

When the template is encountered during specification evaluation, all placeholders are replaced with the provided set names, and the combination of provided names and the template name are replaced by the replacement text. However, the semicolon is not included in the replacement, so an end-of-rule marker must be provided when the template is invoked. This is fundamentally the same approach as a function-like C macro, except that one placeholder in a Maia template appears before the name of the template.

2.7.2 Black-box Verifiers

While Maia is intended to be able to verify a wide range of file types, we have deliberately omitted some functionality. For example, Maia does not include models for interacting with the file system or performing complex calculations like checksumming. Providing support for all every possible construct would require either trying to anticipate all use cases so we can build in language support or adding full Turing-completeness. In the former case, there will always be new functionality which we must add, in the latter a lot of existing software would have to rewritten and debugged in Maia. Instead, we provide the ability to invoke black-box verifiers – external procedures or processes that can perform tasks not expressible in Maia itself.

Black-box verifiers are intended to be invoked as part of the semantic component of a specification, and make use of the `blackbox` keyword, like so:

```
blackbox(verifier, setName)
```

where `verifier` is the name of a black-box verifier known to the system and the `setName`s are one or more elements in the current context to pass to the verifier. Using the `blackbox` keyword together with a verifier name allows us to keep the language namespace uncluttered and makes it clear that the verifier works outside of the bounds of the normal verification system.

From the perspective of a Maia specification, a black box verifier receives one or more values and returns a single value. These values may be operated on by any of the mathematical operators or by concatenation. Black boxes may return a boolean, numeric, or string value, which can then be used as normal within a constraint. For example, a black box could return the hash of a value (string), a score according to some model (numeric), or whether something is a path to a directory (boolean). Results from boolean black boxes may be used directly, while numeric and string values will require a comparison.

Black box verifiers may be invoked from templates, which can be used to simplify calling the black box. For example, if we have a verifier which determines whether a path is a directory, and want to guarantee that all users' home directories are, in fact, directories, we could invoke black box directly:

```
forEvery homeDir :  blackbox(fsobj_isDirectory, homeDir) ;
```

or via a template:

```
(template path isDirectory()) blackbox(fsobj_isDirectory, path) ;
                              ...
forEvery homeDir :  homeDir isDirectory();
```

Black boxes must be registered with the Maia implementation in order to be used, and consist of two parts: a specification-side interface, and an implementation. The interface defines the name of the black box, details of its parameters, and its return type as seen from a Maia specification. These should be portable across platforms and Maia implementations.

A black box implementation provides instructions for actually performing the black box operation, and are tied to the particular platform and Maia implementation. Implementations may be functions in an appropriate language which will be inserted into a generated verifier to provide the necessary functionality. For example, a Maia implementation that outputs C-based verifiers for Linux could implement the `fsobj_isDirectory` black box by making a call to the C function `stat()` and interpreting the results. Black boxes may also invoke system binaries, for example calling the `md5sum` utility to compute hashes.

2.7.3 Inclusions and Libraries

Maia supports breaking specifications into multiple files, which provides the customary benefits to readability and reuse. Included specifications may provide syntax definitions, semantic rules, or both. This way, inclusions can provide both reusable definitions and allow for the refinement of existing specifications.

Existing specifications are brought into the current specification via the `using` keyword:

```
using "specPath";
```

Here, `specPath` is the path to the specification to be imported, which may be relative or absolute. An absolute path is interpreted according to the rules of the local platform. Relative paths are interpreted from the location of the current file. If the implementation provides standard libraries, relative paths are checked first against the location of the current file and then in the standard libraries. This behavior is equivalent to `#include "file"` in C. For example, we can load the specification `fsobject.maia`, located in the current directory, like so:

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

When a file is included, all of its syntactic specifications become available, and all of its semantic rules are enforced. Nonterminals are added directly to the current namespace. Top-level nonterminals – that is, ones which appear only on the left side in the files in which they were defined – are not permitted to share names with any nonterminals in either the importing or imported file. A namespace collision involving a top-level nonterminal is considered an error with the Maia specification.

Sub-nonterminals – ones which appear on the right side of a syntactic statement – may share names across files. In this case, the nonterminal defined in the importing file may be referred to without qualification. Imported sub-nonterminals must always be referred to by a fully-qualified name, which has the form `TLNonterminal.subNonterminal`. `TLNonterminal` is the name of a top-level nonterminal which is defined in the same file as the sub-nonterminal and makes use of the sub-nonterminal, directly or indirectly. If Maia specifications hold to our style rule that top-level nonterminal names begin with a capital letter while non-top-level names beginning with lower case, there is no way for a namespace collision to involve both a top-level nonterminal and a sub-nonterminal.

Maia implementations should include a collection of common or useful specifications and templates. For example, it would be helpful to include an "FSObject" library containing a syntax specification for a valid path, along with templates like "isDirectory()", "isFile()", "isOwnedBy()", or "isAccessibleBy()". Templates for rules like "isUnique()", "isDescending()", or "isNonDescending()" are also valuable. The inclusion of standard libraries of this sort simplifies the process of writing rules, and can help hide system-specific details.

In addition to importing useful file components like paths or hash formats, inclusions may be used to add additional constraints to an existing specification. Adding constraints is handled by including an specification and then applying new semantic rules to it. For example, an administrator might start with a password file specification designed for their system and then add a rule that requires names to be no more than eight characters long. Inclusions may only add constraints to an existing specification. They cannot remove constraints.

2.7.4 Multi-file Verifiers

In some cases, we care about whether files are valid relative to one other in addition to being valid of themselves. For example, the validity of the Linux login files `/etc/passwd`, `/etc/group`, and `/etc/shadow` depends, to some extent, on the contents of the other files. If a user's record in `passwd` indicates that they should have a password hash in the shadow file, but no hash is present, the overall group is invalid even though the individual files may be correct.

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.

The mechanism to include a specification and link it to a file is a modification of the `using` keyword:

```
using "specPath" on "filePath" ;
```

In this construction, specPath behaves exactly as before. filePath is the path to a file to which to connect the specification. It may be an absolute or relative path, with relative paths processed relative to the current (runtime) directory. Thus,

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

would cause the system to import the existing specification `groupfile.maia` and link it to the file `/etc/group`.

Using this mechanism, syntactic and semantic rules are included exactly as they would be without a linked file. When the verifier is run, in addition to verifying the input file, the verifier will also access

the file specified by filePath and verify it according to the specification. All sets made available as part of the specification can then be used for cross-file rules. For example, if we include specifications for both the password and group files, we could then have a rule like:

```
forEvery PasswdFile.gid :  PasswdFile.gid in GroupFile.gid ;
```

which would require that all uses have primary groups which actually exist.

When processing file groups in this way, the overall verification fails if any individual file triggers a verification failure. It is also considered a failure if linked files do not exist or are otherwise inaccessible.

2.8 Proof of Concept Implementation

We have developed a proof of concept Maia verifier generator which supports many of the features of the language. In this section, we present an overview of the tool, including its design goals and implementation.

2.8.1 Overall Design

Our goal is to implement the majority of Maia's most interesting features, rather than providing a complete reference implementation. To that end, we use existing tools wherever possible, and implement only enough functionality to support other needs. For example, some generated parsers require hand editing before they can be used successfully. These changes are generally quite small (e.g. reordering regular expressions) and could be performed automatically in a complete implementation.

The tool as a whole is a Flex-and-Bison-based parser which emits Flex and Bison files which can then be compiled into working verifiers. We essentially rewrite Maia syntax specifications into Flex and Bison and then add semantic rules as actions within the Bison parser. This structure allows us to provide the majority of Maia's functionality without needing Maia-specific support facilities when compiling a generated verifier.

2.8.2 Required Software

As noted above, our tool relies on the availability of Flex and Bison. Specifically, the tool was created using Flex 2.5 and Bison 3.0, and requires both programs for itself and for generated verifiers. We also rely on the uthash library (version 1.9.7) to provide hash table functionality, and the Perl Compatible Regular Expression Library (PRCE, version 8.35) for regular expression support in semantic rules. Runtime libraries for these programs are required to use the verifier generator. Additionally, the development files for the libraries must be installed in order to compile the tool's output.

2.8.3 File Naming

Our tool assumes Maia specifications will be presented in plain text with a file extension of ".maia". For example, a password file specification might be named `passwdFile.maia`. The input file name is used, with file extension, as the basis for the direct output of the tool and the eventual, executable verifier. The input file name is also used, without extension, in the generated files to support multi-file verifiers. While we assume a ".maia" extension, using a different naming convention will not cause problems aside from potentially resulting is slightly odd file and variable names.

Starting from the provided file name, the tool outputs Flex and Bison specifications named *filename.l* and *filename.y*, respectively. We also generate a makefile which is named *Makefile.filename* and will produce a binary named *filenamer*. Thus, if we provide the input file `passwdFile.maia`, the tool will create `passFile.maia.l`, `passwdFile.maia.y`, and `Makefile.passwdFile.maia`, and the executable verifier will be named `passwdFile.maiar`.

2.8.4 Identifying Tokens

Maia is designed to use a lexerless parser for syntax checking, so its specifications do not make a distinction between lexer rules and parser rules. Flex and Bison, however, are designed to use a separate lexer phase (provided by Flex), so it is necessary to extract tokens from the Maia syntax specification.

As our tool parses Maia specifications, it makes note of each unique string literal, character class (including "."), or regular expression it encounters. These automatically receive token names of the form *filename_string_NN*, *filename_cclass_NN*, or *filename_regex_NN*, respectively, where "filename" is the name of the input file without extension, and "NN" is a running count of how many tokens of that type have been encountered. Once the file is entirely parsed, we know that any string, character class, or regular expression that occurred as part of a syntax rule should be treated as a token that will be included in the Flex specification.

String-based tokens are always written before character classes or regular expressions, to ensure that they are matched first. Other tokens are written out in the order they were encountered. This may cause problems in some cases, as Flex will decide between two matching patterns on the basis of which is closer to the beginning of the specification. Thus, if a broad pattern is placed before a more specific one, the broad pattern will always win. To remedy a problem of this type – which is reported by Flex during compilation – a user can simply reorder the patterns in the generated Flex specification. The question of whether one regular expression matches a subset of another is decidable, so it would be feasible for a more refined verifier generator to perform this reordering automatically.

Flex-generated lexers do not maintain state information about where in a parse they are. This can lead to additional token mis-selection problems in some cases which cannot be remedied by reordering patterns. For example, in the password file we might specify that a username will match the character class `[-azAZ0-9_]{1,32}`. That is, a username consists of between one and thirty-two digits, letters, underscores, or hyphens. The GECOS field, later in a password record, can contain essentially any character except a colon or newline. Thus, it can be represented by the character class `[^:\n]+`.

Clearly, the username character class is a subset of the GECOS field character class, and thus must appear first in the Flex specification in order to be matched at all. However, if the first "word" of the GECOS field happens to consist only of letters and digits, it will be matched as a username, which causes a parsing failure when the lexer returns a username token rather than the expected GECOS token. The only general solution to the problem would be to introduce some context into lexical analysis, either by pushing data from the parser back into the lexer to by eliminating lexing as a separate phase. It is possible to work around the problem by using the more general pattern for both (e.g. having usernames also matched by `[^:\n]+`) and then using a semantic rule to restrict the general pattern.

2.8.5 Generating Bison ENBF

Maia provides direct support for a number of constructs which lack direct counterparts in Bison's EBNF. To work around this, our tool creates new non-terminals as needed, and uses them to provide

the necessary functionality. For example, a Maia specification might include an optional nonterminal:

```
foo = bar?  baz ;
```

This is not supported in Bison, so we generate a new nonterminal which uses alternates and replace the original, optional nonterminal:

```
foo :  quz baz ;
quz :  bar | /* empty */ ;
```

Similar techniques can be applied to a variety of other Maia constructs. Generated nonterminal names all have the form *filename_nonterm_NN*, where "filename" is once again the extension-less name of the input file, and "NN" is a running count of the number of generated nonterminal. We elect to use simpler names in these examples, for clarity.

Inline Alternatives

Maia supports inline alternatives. That is, an expression of the form `A = B C | D ;` which matches nonterminal B followed by either nonterminal C or D. In Bison, we move the alternate to its own nonterminal.

Maia	`A = B C | D ;`
Bison	`A : B E ;`
	`E : C | D ;`

Optional Nonterminals

As described above, Maia supports the use of optional nonterminals, which may or may not appear as part of a larger expression. In Bison, we replace the optional nonterminal with a new nonterminal which matches either the original nonterminal or nothing.

Maia	`A = B? C ;`
Bison	`A : D C ;`
	`D : B | /* empty */ ;`

Zero or More Repetitions

For zero or more repetitions, we replace the existing nonterminal with a new one which matches either the empty string or itself followed by the original nonterminal.

Maia	`A = B* ;`
Bison	`A : C ;`
	`C : C B | /* empty */ ;`

One or More Repetitions

The case of one or more repetitions is equivalent to the zero-or-more case, except that the new nonterminal matches either one of the old nonterminal or itself followed by the old nonterminal.

Maia	`A = B+ ;`
Bison	`A : C ;`
	`C : C B | B ;`

Exactly N Repetitions

Cases where we must match exactly N occurrences of a nonterminal are handled by creating a new nonterminal which is a concatenation of N instances of the repeated nonterminal.

Maia	`A = B{3} ;`
Bison	`A : C ;`
	`C : B B B ;`

Between M and N Repetitions

In order to match between M and N occurrences of a nonterminal, we provide alternates of M occurrences, $M + 1$ occurrences, and so on, up to N occurrences.

Maia	`A = B{2, 4} ;`
Bison	`A : C ;`
	`C : B B | B B B | B B B B ;`

At Most N Repetitions

For at most N repetitions, we provide alternates of all numbers of occurrences from 1 to N.

Maia	`A = B{ , 3} ;`
Bison	`A : C ;`
	`C : B | B B | B B B ;`

Grouping

Groups in Maia specifications are converted into their own nonterminals. This process can be conducted more than once and happens before other modifiers, like optional or repetition, are applied.

Maia	`A = (B C)? ;`
Bison	`A : E ;`
	`E : D | /* empty */ ;`
	`D : B C ;`

2.8.6 Providing Semantic Checks

We do essentially no explicit set building to support Maia semantic rules. Instead, we take advantage of Bison actions, which are executed whenever a nonterminal is matched. This allows us to provide a wide range of semantic constraints with little additional overhead.

We support a sizable subset of Maia's semantic functionality, including logical comparisons, grouping, regular expression matching, arithmetic, and operations on compound sets. At present we do not support length or count constraints, indexing, enforcement levels, black boxes, or templates, with the exception of providing "isUnique()". This limits the type of rules which can be expressed, though we find that it is sufficient to expression constraints on files like the login set of password, shadow, and group.

Universal rules (those either implicitly or explicitly marked with `forEvery`) are evaluated during the action. If the rule does not hold for a particular match, we immediately exit with an error. Otherwise, parsing continues as normal.

Existential rules are processed with the aid of flag variables, which are initially set to 0. During an action, if an existential rule is true the corresponding flag variable is incremented, otherwise it is left unmodified. Once parsing has been finished, the flags are checked. Any rule whose flag has a nonzero value is satisfied. Is any flags remain at zero, their rule was not satisfied and an error is reported.

While we support neither indexed constraints nor templates, the "isUnique()" template is sufficiently useful to merit direct implementation. This is accomplished by creating a uthash-based hash table for each value which must be held unique. When a new value is encountered, we consult the hash table to see whether it already exists. If so, we report an error and halt processing. Otherwise, we add the new value to the hash table and continue.

2.8.7 Creating Multi-file Verifiers

Our tool provides no direct facilities for building multi-file verifiers. However, it does provide some functionality to simplify the process of combining its verifiers into a single, multi-file executable.

By default, all Bison parsers use the same function and variable names, which makes it difficult to use two different parsers in the same program. Bison 3 provides the "%define api.prefix" directive, which causes Bison-generated functions and variables to use the specified prefix, rather than the default value of "yy". Our tool uses this directive to set the prefix for all Bison functions and variables to the extension-less name of the input specification file. Combined with naming convention used for generated tokens, nonterminals, and other variables, this means that parsers generated by our tool are not subject to namespace collisions when combined.

Constructing a multi-file verifier with our tool requires four major steps. First, use the tool to produce Flex and Bison files for each individual file. Then, construct a new C file which will replace the generated `main()` methods and invoke of the sub-verifiers. This new file must provide all of the required initialization for each sub-verifier (found in their generated files), as well as initializing anything required to implement the cross-file constraints. Next, remove initialization from the individual sub-verifiers and add checks for the cross-file constraints. This will likely take the form of new constraint checks added to the existing parsers. Finally, build the new verifier by handing the sub-verifier specifications to Flex and Bison, and then compiling together all of the various C files this produces, along with the manually created one.

2.9 Conclusion

In this chapter we presented Maia, a language for describing integrity constraints on arbitrary files. Maia differs from other parsers and verifiers in that it is designed to produce complete verifiers for any file type, rather than being tied to a particular format or needing additional programming. Maia specifications begin with a description of the syntactic structure of a format, expressed in extended BNF. We then use named nonterminals in the file's structure to construct sets of data to which semantic rules can be applied. This allows the data within a file to be constrained without the need to explicitly write a parse.

We also presented a partial implementation of a Maia verifier generator. This tool converts specifications written in a subset of Maia into Flex and Bison specifications which can be compiled into verifiers with minimal human intervention.

Chapter 3: Mandos

In order to provide mandatory data integrity protection, we must be able to protect data regardless of where it is stored in the system or how it is accessed. Further, in order to maximize the chances of adoption, an integrity system must interfere with the day to day operation of the computer system as little as possible. In this chapter, we present Mandos[1] (MAN-dose), a Linux Security Module which uses a validate-after-change model to provide robust data integrity guarantees regardless of who modifies protected data or what tools they employ.

We begin by presenting some tools which operate in a similar space to our own. This is followed with a discussion of Mandos' fundamental principles, and details of our implementation. We conclude with topics related to using Mandos, ranging from compilation and installation through Mandos' configuration interface to a sample configuration.

3.1 Related Work

A number of security modules have been developed for Linux over the years, including five which have been integrated into the mainline kernel: SELinux [46] was originally developed at the National Security Agency, and focuses on providing robust, extremely fine-grained mandatory access control, though this can come at the cost of complicated configurations. SMACK (Simplified Mandatory Access Control Kernel) [57] focuses on simplicity in expressing rules, and works best on file systems with extended attributes. TOMOYO Linux [26] takes a declarative approach, and can be used as a systems analysis tool. AppArmor [12] [11] takes an application-centric approach, constraining what resources a running program may access. Finally, YAMA [62, `documentation/security/Yama.txt`] is a very small module whose primary purpose is to restrict `ptrace()` system calls.

Some research has gone into using MAC systems to enforce data integrity [31] [50], but these systems tend to rely on the assumption that applications which are allowed to modify data will do so correctly. Configuring systems to provide these integrity guarantees can also be quite complicated.

The Filesystem in Userspace (FUSE) mechanism [62, `documentation/filesystems/fuse.txt`] allows users to create their own filesystems without modifying the kernel itself. This mechanism could be used to provide file systems which will protect the integrity of anything they store. However, while such a system would provide strong guarantees within its borders, it could do nothing to protect files on other file systems. Thus, it would be extremely difficult to protect system configuration files which often have to be stored in specific locations.

Mandos is designed to to provide strong integrity guarantees without the need to certify editing programs in advance. It is straightforward to configure, and does not require that the entire file system have protections assigned before it can go to work. Mandos is also capable of providing its

[0] This chapter expands on "Toward a Mandatory Integrity Protection System" published in *CATA 2016*.

[1] "Námo the elder dwells in Mandos, which is westward in Valinor. He is the keeper of the Houses of the Dead, and the summoner of the spirits of the slain. He forgets nothing; and he knows all things that shall be, save only those that lie still in the freedom of Ilúvatar. He is the Doomsman of the Valar; but he pronounces his dooms and his Judgements only at the bidding of Manwë." [61]

integrity guarantees on any directly-attached storage medium regardless of filesystem, so long as files can be created on the filesystem.

3.2 Integrity Model

Our work is intended to provide protections similar to those offered by Clark and Wilson's model by verifying files when they are modified rather than relying on programs which have been proven correct. Under Mandos, we automatically detect when files are opened for modification, flagging them for verification and possible rollback later. When the file is closed, we verify its contents. If the new contents prove to be valid, the changes are made available to anything that might access the file in the future. If a problem is detected, the changes are rolled back, leaving the file as it was when it was opened.

3.2.1 Determining Validity

Mandos does not provide, or require, a particular system for determining whether a file is valid. Maia verifiers are an excellent fit, but by no means the only option. As a practical matter, a verifier must be able to perform its task quickly, as it must finish before a file can be made available. Beyond this, verifiers might be entirely self-contained, or could rely on complex provers or validation tools. Any verifier will do, so long as it reflects the needs of the administrator and can decide whether a file is valid in a tolerable length of time.

Separating verification from enforcement provides a number of advantages. First, it simplifies the integrity system by allowing it to hand off correctness testing. Second, it reduces setup work by eliminating the need to recreate existing verifiers. This is especially important for complex formats like XML, which are hard to verify correctly. Third, it allows administrators to select verifiers that work well with their existing toolchains.

3.2.2 Integrity Guarantees

Fundamentally, we provide two guarantees to any process interacting with integrity-protected files. These guarantees are held regardless of what (or who) accesses the files, for any files on directly-attached storage.

G1: A protected file shall be valid when first opened by a process.

When a process opens a protected file, the file shall be valid regardless of changes made by any past process. Past changes which were valid will be reflected by the current state of the file, but invalid changes will have been completely removed.

This guarantee applies whenever a process opens a file which it is not currently accessing. Thus, if process P opens file F for writing, closes F, and reopens F for reading at a later time, F is guaranteed to be valid at both opens, though its contents may have changed in the interrim. However, if P opens F for writing and then opens F for reading as a separate action, F may not be valid at the second open, subject to the second guarantee. This allows P to maintain multiple handles to a file it is modifying without the need to verify the file after every change.

G2: A process shall not see changes to open, protected files that it did not cause.

If processes P_1 and P_2 are both accessing protected file F, neither may change F in such a way that the other sees the change. We make this guarantee to prevent two major classes of errors introduced by allowing processes to see unvalidated changes: First, P_1 might introduce data which breaks F regardless of P_2's view of the file, such as writing an invalid record in the password file. Second, P_1 might change the file in such a way that the version on disk is correct but the version P_2

sees is incorrect, as with moving a still-needed close tag from one side of P_2's current parse location to the other. In both cases, the concurrent access allows (apparently) invalid data to appear in a file which should be guaranteed correct under G1.

While changes are not presented during inter-process concurrent accesses, they do appear during intra-process concurrency. For example, if process P opens a protected file for writing (F_W) and subsequently for reading (F_R), reads using F_R will see any changes made using F_W. This behavior is exactly the same as what is seen without Mandos: a process with multiple handles to a file will see its changes reflected across all of the handles. Once P closes both F_W and F_R, the file will be verified and subsequent file accesses will find the file valid under G1.

Applying these guarantees to the password file, we are assured of two things: `/etc/passwd` will never contain invalid records, and nothing will ever be allowed to read the `/etc/passwd` file while it is open for writing. This is a significant, and we feel beneficial, change from the baseline system, where the password file is probably valid, but we do not know that until someone tries to log in.

3.2.3 On Files and Names

For many programs, the fact that data exists in a certain place in the file system is as important as the fact that the data is valid. For example, the standard Linux login system requires not only that user records be properly formatted but also that they exist in the files `/etc/passwd`, `/etc/shadow`, and `/etc/group`. Renaming the files would constitute an integrity violation even if their data is still valid, because the data would no longer be where the login system expects it. Thus, a system-level data integrity system must provide at least the option to guarantee that a file have the correct name, in addition to having good data.

However, the need to protect a file's name does not necessarily extend to disallowing access to a file's contents in other ways. For example, while an administrator may require that `/etc/passwd` have exactly that name, they may also create links to the file elsewhere in the file system. This allows integrity protected files to be reused by linking to them wherever they are needed, rather than requiring separate copies of the file which must then be kept in sync.

3.3 Implementing Integrity Protection

Having discussed the overall guarantees provided by our integrity model, we now turn our attention to an implementation of that model. Mandos is an experimental Linux Security Module designed to show that our integrity guarantees can be implemented on a running system. Software does not need to be aware of Mandos to take advantage of its protections, and protections apply regardless of the user accessing the files, rendering Mandos entirely transparent to userspace software.

3.3.1 Implementation Overview

In order to provide our guarantees, Mandos must be able to preserve a copy of the known-good contents of a file which can be restored, if need be, after the file is modified. When a process requests (allowed) write access to a protected file, Mandos automatically performs setup work, creating a backup copy of the file and marking it for later validation. The process may then access the file however it wishes without interference. When the process' last open handle to a file is closed, Mandos then invokes the file's verifier and cleans up the backup, either deleting the backup if the file is valid or using the backup to restore the file if the the verifier rejected the file. Figure 3.1 presents an overview of this behavior.

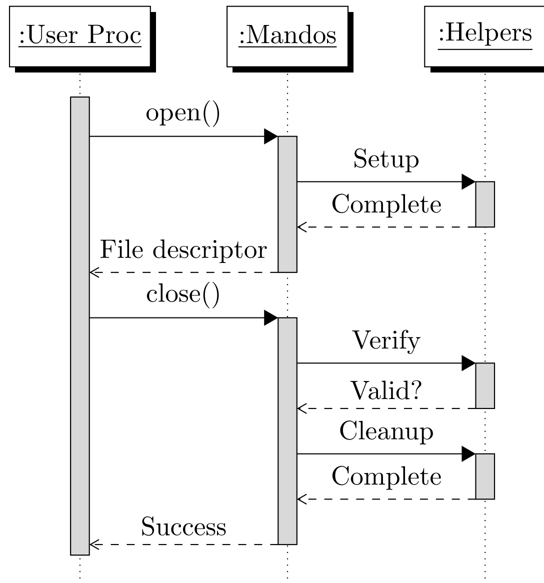


Figure 3.1: Overview of Mandos Integrity Protection System

The setup and cleanup / verify operations happen during the `open()` and `close()` system calls, and thus appear atomic from userspace. Because Mandos intercepts all calls to open and close files, and its operations are guaranteed to finish before return is controlled to the user program, there is no way to avoid Mandos protections.

3.3.2 Setup and Cleanup Helpers

Setup and cleanup operations are facilitated by a pair of small userspace programs. The helpers are very simple: setup receives the full path of a protected file and a unique identifier (currently the number of microseconds since the start of the epoch) and copies the protected file into a new file with a name of the form *full_path.unique_id*. Cleanup also receives the path of a protected file, along with the assigned unique identifier, information about the file's protection mode, and whether the current version is valid. If the current version is valid, the backup is removed. Otherwise, the current file is copied, if need be, and the backup is restored.

Using userspace programs for these tasks allows us to move policy decisions about where backups should live out of the kernel. For example, it would be possible to switch to storing all backups in the same place without modifying the kernel module itself. Using normal programs, which are run as root, also helps to minimize permission-related issues. The system will automatically do the right thing, so there is no worry that Mandos will introduce confidentiality concerns.

3.3.3 Enforcement Options

Internally, Mandos provides configurable modes for protecting a file's data, it name, and whether it may be hard-linked. The data and name mode protections are fully implemented. Link mode configuration is stored and reported, but is not currently enforced. (See Section 5.2.1 for some of

the issues surrounding link modes.) Protection modes are independent of one another, allowing for configurations like optional files with data protection and required files with arbitrary contents.

The following data protection modes can be used to safeguard the contents of a file:

No Protect The contents of the file are not constrained in any way. No verification is performed, and the integrity guarantees are not in effect. This mode is selectable as part of a file's integrity policy, and is also automatically applied to any file not covered by a policy.

Rollback The contents of the file are subject to our integrity guarantees. Selecting this data mode requires that a verifier be specified as part of the configuration. File are automatically verified after they have been written to and closed, and are rolled back to a known-good state if they are found to be invalid. This is a reasonable default for files which should be under integrity protection.

Branch Like Rollback, except that a copy of the invalid version of a file is created when the file is reset. This mode is helpful in that it allows after-the-fact analysis to see what changes were attempted and why they failed.

No Change No processes will be allowed to change the file. When configured for No Change, requests to open a file for writing are disallowed by Mandos, even if they would normally be permitted under the system's access control scheme.

Along with the data protection modes we also provide two name modes:

Require Require that a file with the specified name exist. Files whose names are Required may not be renamed or deleted, nor may their containing directories be renamed. Require name mode is considered the default for data modes other than No Protect, though this is not enforced.

Optional The file may exist, but is not required. This mode is intended for use cases where a file is optional, but should be under integrity protection if it exists, as would be the case with, e.g., a user's personal SSH configuration. The file may be renamed or deleted, and its containing folder(s) may also be renamed.

Finally, we provide two (as yet unimplemented) options for controlling hard linking:

Disallow Hard links to this file are not permitted. This protection mode may not be applied to files which are already hard linked.

Allow Hard links to this file are permitted. By default, all hard links receive the same data mode, though the name mode only applies to the name specified in the configuration. It is permissible to apply protections to multiple links to the same file (e.g. if one location is required and another is optional), but only if the following requirements are observed:

- If any hard link is set to No Change mode, all must be set to No Change mode.
- If any hard link is set to Rollback or Branch, all must be set to either Rollback or Branch.
- If any hard link has a verifier specified, all must specify the same verifier path.

When protections are added for a file which has a data mode of Rollback or Branch, Mandos automatically protects the specified verifier. Verifiers protected in this way are configured for No Change data mode, Require name mode, and Allow link mode. Mandos maintains protections for verifiers until all files using the verifier are removed from active policy, after which the verifier's protections are automatically removed.

3.3.4 Failure Cases

Mandos is designed to fail in a way which preserves system integrity whenever possible. In the event that the setup utility cannot be run or reports an error, Mandos will disallow write access to protected files until it is fixed. Likewise, if a verifier cannot be run, crashes, or is killed, Mandos will assume that the file is invalid, though it will force a branch copy to be made even if the file is in Rollback mode. If the cleanup helper fails, the integrity guarantees cannot be enforced until an administrator manually corrects the protected file and fixes the helper. Any error results in a console message being sent to the system log.

If setup, or verify / cleanup runs longer than about two minutes a kernel watchdog automatically reports that a system call is taking too long. Setup, cleanup, and the verifiers can be killed by root, which Mandos recognizes as an error with the program. Modified files will remain locked until an administrator kills the misbehaving process, but the rest of the system continues to operate normally.

3.3.5 Readers-Writer Locking

We supplement our backup/restore system with readers-writer locking on protected files to support concurrent accesses. Under this system, a file which is currently closed may be accessed however a process desires, subject to discretionary access control. Likewise, if only a single process is accessing a file, it may open whatever additional handles it desires, exactly as would normally be the case. Our system varies from normal behavior only when two processes both seek to access the same file at the same time. If a file is currently open for reading, any other process that requests read access will be granted it. However, requests for write access will be denied (returning an error to the caller) until there are no other readers. Similarly, if a file is currently open for writing, any other process requesting read access will be denied until the writer closes all of its active handles. Table 3.1 summarizes this behavior.

Table 3.1: File Access Rules for Readers / Writer Locking

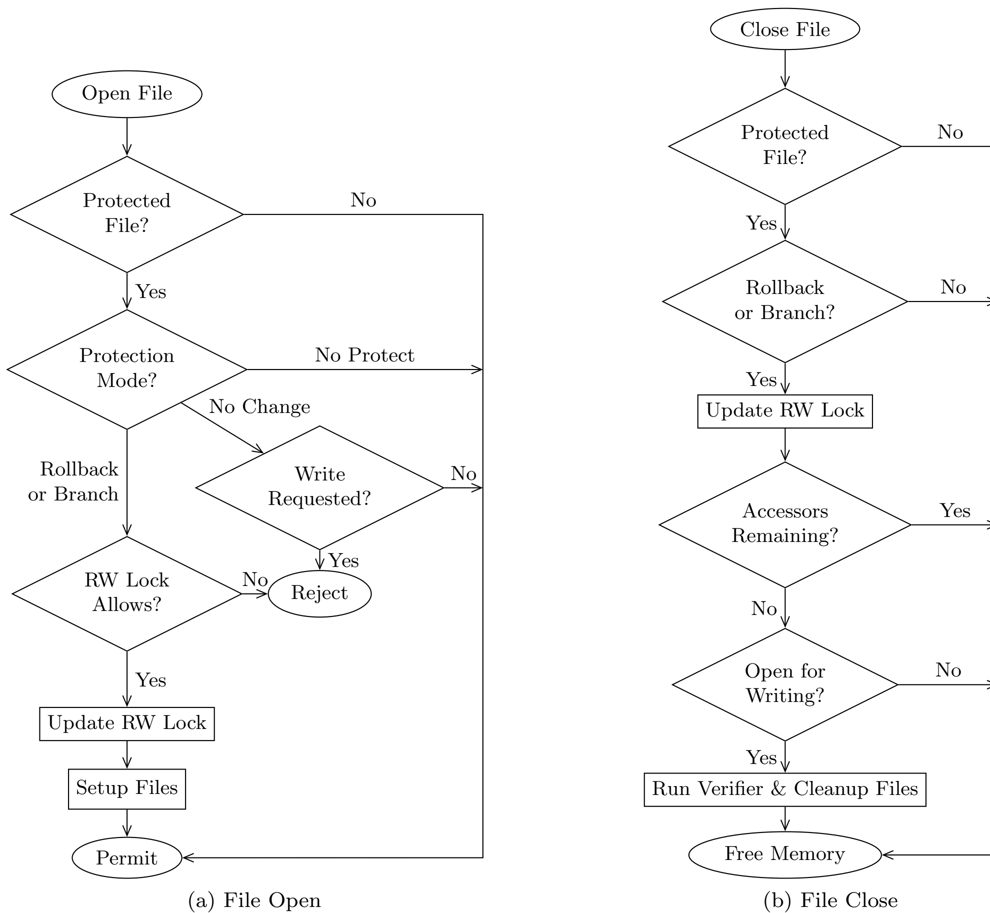
Readers	Writers	Permit Read?	Permit Write?
0	0	Yes	No
1	0	Yes	(reader == writer)
> 1	0	Yes	No
*	1	(reader == writer)	(new writer == old writer)

Internally, we track readers-writer locking using a simple structure which records the PID of the file's current writer and a list of PIDs of all current readers. We also count how many handles each PID has to the file, so we can determine when a process has finished with a file.

The combination of backup making and readers-writer locking provides our integrity guarantees and offers a number of advantages: Backup copies may be kept close to their originals, making them easy to find if they are needed. Close proximity also means that we need not cross file system boundaries, making access characteristics more uniform. Our approach is also entirely file system neutral, and will work regardless of the underlying file system so long as it is possible to create new files. By comparison, file system specific approaches like the use of transactions would provide excellent support for our guarantees, but only when that file system is available. Relying on the file system to provide our integrity guarantees means that simply moving a file to a different file system eliminates any chance at integrity protection.

3.3.6 Linux Security Modules Interface

The Linux Security Modules interface provides a total of 194 hooks into the 3.18 kernel, five of which are used by Mandos. The majority of the module's implementation is located in the file open and file free security hooks, invoked respectively when a process requests that a file be opened and when the file's kernel representation is freed after a close. During an open, Mandos checks whether there is currently a policy related to the requested file. Files with No Protect or No Change protections can be permitted or denied immediately. For Rollback or Branch files, Mandos checks whether the access is permitted under the readers / writer lock, and updates the lock structure as appropriate. If a new write is to be permitted, Mandos will then create the backup copy of the file before returning control to the program. If at any time a check fails, Mandos returns access denied, preventing the calling process from accessing the requested file. Figure 3.2a presents this behavior as a flowchart.



(a) File Open

(b) File Close

Figure 3.2: Flowcharts for Mandos File Access Hooks

The free security hook is invoked when a process' last open handle to a file is closed and is intended to free any memory the security module associated with the file. Once again, Mandos checks to see whether the file has an associated policy, and, if so, whether it has Rollback or Branch

protections. For Rollback and Branch files, we then update the readers / writer lock and determine whether the last accessor has closed the file. If the file was open for writing, Mandos then invokes the appropriate verifier and cleans up the file based on the results of verification. Finally, Mandos frees any remaining memory it had associated with this particular file handle. Figure 3.2b gives a flowchart of this behavior.

We use the inode unlink and inode rename hooks to provide name protections. Files with name mode Require may not be renamed or removed, so we check protection modes whenever a rename or unlink request is made. If the file is unprotected, or has name mode Optional, the operation is permitted, otherwise we return access denied. We also deny requests to rename directories which contain files with required names. No extra effort is required to prevent directory deletion, as it is not possible to unlink a directory which contains files.

Mandos also uses the file allocate security hook, but no policy decisions are made there. The hook is only used to allocate a small amount of memory to associate with files which are being opened. This memory is freed as part of the file free security hook, which is the intended purpose of that hook.

Because of the hooks Mandos uses, most kernel accesses to files are also protected automatically. Specifically, any file accessed using `filp_open()`, directly to indirectly, will be subject to Mandos' integrity guarantees. It is certainly possible for kernel code to avoid Mandos protections, but doing so would require taking specific steps to circumvent the module.

3.3.7 SecurityFS Interface

The kernel provides a number of special-purpose file systems to allow modules to communicate with userspace. ProcFS [62, `documentation/filesystems/proc.txt`] is the oldest, and was originally intended to provide information about individual processes, but has since been used for general purpose configuration. SysFS [62, `documentation/filesystems/sysfs.txt`] reflects kernel data structures directly in a virtual file system, making them easier to explore and manipulate.

The SecurityFS [62, `security/inode.c`] interface is mounted under the SysFS directory (at `/sys/kernel/security`), and provides a free-form interface for LSM modules. Modules specify the names of files, alongs with relative paths and a set of functions to handle access to the files. For example, the module would provide a function which receives data when a userspace process writes to the file. User processes may then interact with the files with using standard system calls according to normal file permissions.

Mandos relies on SecurityFS to provide configuration and status information. At present, this includes being able to query whether the module is enforcing policy, turning the module on, reading current policy, adding new policy, and determining the status of protected files. Section 3.4.2 provides detailed coverage of the use of these files.

3.3.8 Module Breakdown

Mandos' implementation is spread across four C source files, with two additional files for configuring the kernel build system. As much as possible, we try to separate Mandos' policy enforcement functionality from its userspace interface. To that end, all of the LSM and SecurityFS functions are isolated in their own files (`mandos_lsm.c` and `mandos_fs.c`, respectively). All of the functionality shared by the modules – primarily functions and definitions required for maintaining and accessing integrity policy – is distributed between `mandos_common.c` and `mandos_common.h` Separating module functionality in this way promotes separation of concerns and helps to encourage a well-defined interface between policy enforcement and configuration.

Table 3.2 gives line counts for each of the files. We present two different values for `mandos_lsm.c`, as a sizable portion of that file is made up of stub functions or LSM boilerplate.

Table 3.2: Lines of Code in Mandos Source Files

File	LoC
`mandos_common.h`	332
`mandos_common.c`	799
`mandos_fs.c`	1185
`mandos_lsm.c`	1172
`mandos_lsm.c` (Only Mandos)	730
Total	**3488**

3.4 Using Mandos

Thus far we have described our model for integrity protection, and how we implement that model within the kernel. In this section, we present details of using Mandos, including installation and configuration.

3.4.1 Compiling and Installing Mandos

The LSM interface requires that modules be built into the kernel at compile time, rather than allowing run-time loading. Thus, installing Mandos requires recompiling the kernel.

Mandos itself is reasonably self-contained. The module's source code is contained in a single folder which needs to be placed in the security directory of the kernel source tree. Once that is accomplished, the `Makefile` and `Kconfig` files in the security directory need to be adjusted so they are aware of Mandos. No other changes are required to make Mandos known to the build system. Once that is done, Mandos can be selected as part of kernel configuration, in the same way AppArmor, SELinux, or any other security module is activated.

Mandos also requires that its helpers be installed in order to function correctly. The setup and cleanup programs are expected to be located in `/mandos`. The programs must be runnable by root, but no other permissions are required. We typically store verifiers used by Mandos policies in `/mandos/verifiers`, but this is simply for convenience and is in no way required by the module.

3.4.2 Configuration Interface

Mandos' configuration interface is normally located at `/sys/kernel/security/mandos`, which is made available by mounting the SecurityFS filesystem. (SecurityFS is not mounted by default on all platforms, but can be easily added to `fstab`.) The interface is primarily spread over three files, with a directory containing status information about protected files. Any user may read active policy and status files, but only root may load policy or enable the module. Figure 3.3 presents the directory structure with a few status files.

The `enabled` file is both used to report the current state of the module and to switch it on. When read, the file will contain either "0" or "1", indicating, respectively, that the module is currently either disabled or enforcing policy. Writing a "1" to the file causes the module to check that the helper programs are present. If so, the module is enabled. Otherwise, an error is reported and the

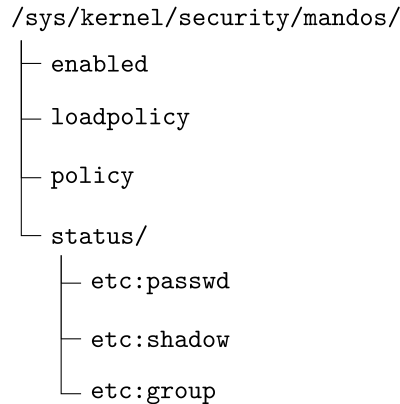


Figure 3.3: Mandos' Configuration Directory Structure

module remains disabled. Mandos cannot be switched off by writing a "0" to `enabled`. Attempting to do so will print an error noting that the module can only be turned off by rebooting. Writing any value other than "1" or "0" will display an error.

The `loadpolicy` file is used to add new policy items. Policy may be added at any time, regardless of whether Mandos is currently enabled, and administrators may add one or more policy items at a time. However, there is currently a requirement that a single set of policy items occupy no more than 2048 bytes. This limit only applies to a single load, so more policy may be added using subsequent accesses to `loadpolicy`.

Figure 3.4 presents a Maia specification for the policy format. Each policy written to `loadpolicy` must begin with a version specifier, which is currently required to be "v1". Following this, the administrator provides individual policy records for each file to be protected. Policy records always begin with the string "pol". This is followed by a tag, which provides logical grouping for the policy item (e.g. that a file is part of the login system, or a helper program). Tags are provided as an administrative convenience, and are not processed internally. The remainder of the policy specifies the complete path to the file, data, name, and link modes, and a verifier, if needed, followed by the string "lop", which indicates the end of the record. All items are newline delimited, so paths should not be quoted. A verifier must be specified for Rollback and Branch data modes, and must not be specified for NoProtect and NoChange modes. A single hash mark ("#") on a line indicates the end of policy records and the beginning of the comments sections. Nothing after the "#" is processed, though it does count toward the 2048 byte limit.

Newly added policy items are automatically checked before they are put into effect. Protected files must be regular, and must exist if they are required. If a verifier is specified, the verifier must execute and return no errors with the file. Verifiers are automatically assigned No Change and Required protections. These protections are removed when the verifier is no longer associated with any files. If a file has name mode Require, all parent directories of the file are also assigned Require name mode. Failing any of the checks results in the policy item not being added, though the system will continue to process other new policy items if they were provided.

Active policy can be found in the `policy` file. This file uses the same format as `loadpolicy`, and contains all active policy items. Policy items added by an administrator are presented first, in arbitrary order. Automatically generated policy items, such as those created for verifiers or protected directories, are presented in the comment section of the file. These policy items have

the same format as the other policy items, but will always have the tag "AUTO" (for verifiers or protected directories) or "HELPER" (for the Mandos helper programs). This format was chosen so that the complete policy can be reviewed and loaded from the review copy without changing the data. Mandos does not commit policy to disk automatically, so administrators must either keep track of individual policy files or copy `policy` and restore it later via `loadpolicy`.

```
1  using "fsobject.maia" ;
2
3  MandosConfig = version Newline policy+ (commentStart .*)? ;
4
5  version = "v1" ;
6
7  policy = polStart Newline
8           tag Newline
9           filePath Newline
10          dataMode Newline
11          nameMode Newline
12          linkMode Newline
13          (verifier Newline)?
14          polStop Newline ;
15
16 commentStart = "#" ;
17
18 polStart = "pol" ;
19 filePath = FSObject ;
20 tag      = [-a-zA-Z0-9_]{1, 32} ;
21 dataMode = "NoProtect" | "Rollback" | "Branch" | "NoChange" ;
22 nameMode = "Require" | "Optional" ;
23 linkMode = "Allow" | "Disallow" ;
24 verifier = FSObject ;
25 polStop  = "lop" ;
26
27 Newline = "\n" ;
28
29 policy : verifier != "" iff
30         ((dataMode == "Rollback") or (dataMode == "Branch")) ;
```

Figure 3.4: Maia Specification for Mandos Policy Files

We choose not to modify the result of the `close()` system call for a number of reasons. Doing so would alter the semantics of the call in a way user programs do not expect. It is also unlikely that a process would check the result of closing a file, and the LSM interface does not provide a mechanism to return a status code via close. Instead, Mandos provides status files for each file with Rollback or Branch protection. These status files provide information about the last write access to a file and are located in the directory `/sys/kernel/security/mandos/status`.

We construct the name of a status file from the full path and name of a protected file as follows:

1. Remove the leading slash ("/") corresponding to the root directory
2. Replace all colons (":") with two colons ("::")
3. Replace all slashes ("/") with colons (":")

For example, "/etc/passwd" would become "etc:passwd". This replacement guarantees unique names for the status files, though names can be quite long if the protected file is deep in the file system.

Status files are three lines long, and intended to be readily understood by both software and humans. The first line of the file contains the complete, unmodified path to the file. The second line contains a single digit, drawn from Table 3.3, indicating the result of the last write. The last line contains a timestamp for the last access, followed by the PID of the accessor and a one or two work status matching the numeric value on line two. If a file is found to be invalid, its status will normally reflect its data mode. However, if a file's verifier fails to run for some reason, Mandos will always branch the file, which is reflected in the status file. Figure 3.5 presents a sample status file.

Table 3.3: Potential Status of Mandos Protected Files

Code	Status	Meaning
0	Unchanged	File has not been modified since it was protected
1	Committed	Changes were made and accepted
2	Rolled Back	Changes were found to be invalid, and rolled back
3	Branched	Changes were found to be invalid and branched

```
1 /etc/passwd
2 0
3 2016-01-26T15:15:32 0 Unchanged
```

Figure 3.5: Sample Status File for `/etc/passwd`

Files which have not been modified will present the time they were placed under protection, and give a modifier PID of 0. Otherwise, the timestamp reflects when the file was verified after it was last opened for writing. A process wishing to know whether its changes to a protected file were accepted can check the corresponding status file. Line two provides an easy to find readout of whether the changes were committed. A process can then use the PID on line three to determine whether its changes are being reflected by that status, or something else made changes in the interim.

3.4.3 Protecting `/etc/passwd` with Mandos

To wrap up this chapter, it is illustrative to consider how one might place the file `/etc/password` under integrity protection. Tags are intended to group policy items by the purpose of the file, to make policy items a bit easier to find. "login" is a reasonable tag, as the file is part of the login system.

It must be possible to modify the password file, but we want it to have strong integrity guarantees, so Rollback or Branch mode is indicated. Rollback provides robust protection, but is not particular

helpful for understanding why a change failed. By selecting Branch mode, we can examine invalid files after the fact without jeopardizing system integrity.

The login system looks for user data at `/etc/passwd`, so it is reasonable to use Require mode for the name of the file. There is no readily apparent reason to allow hardlinking the password file, so Disallow link mode is sufficient. If hardlinking was desired, this can be switched to Allow mode.

Since we specified Branch mode, we must provide a verifier. (The same would be true in Rollback mode.) A Maia-based verifier can process the password file without difficulty, but any equivalent tool would suffice. We elect to keep verifiers near the Mandos helpers, but they may reside anywhere in the file system. Figure 3.6 presents this policy as it would be provided to Mandos via `loadpolicy`.

```
1 v1
2 pol
3 login
4 /etc/passwd
5 Branch
6 Require
7 Disallow
8 /mandos/verifiers/passwdFile.maiar
9 lop
```

Figure 3.6: Sample Mandos Policy for `/etc/passwd`

By loading this policy, `/etc/passwd` itself would be placed under protection. Because it has a required name, Mandos would then automatically grant Require mode to `/etc`. The verifier would also be placed under Require and No Change protection. As a result, the directories `/mandos` and `/mandos/verifiers` would also receive Require name mode. Thus, adding a single policy item results in protecting a total of five files or directories.

3.5 Conclusion

In this chapter we have presented Mandos, a Linux module which provides robust data integrity guarantees. Mandos guarantees that protected files will be valid the first time a program opens them, and that they will be free of external changes for the duration of a program's access to the file. These guarantees are enforced using the Linux Security Modules interface, which is capable of intercepting all userspace accesses to files. Configuration is provided via a SecurityFS interface which supports querying and updating policy at any time, and provides status about protected files which may be modified.

Chapter 4: System Performance

Mandos must, necessarily impose a penalty on system performance to provide file protection and validation. In this chapter, we quantify Mandos' impact on performance, and demonstrate that we can provide robust integrity guarantees without sacrificing usability.

4.1 Benchmark Environment

Our benchmarks are conducted on a desktop computer equipped with an Intel Core 2 Duo E6420 processor running at 2.16GHz, one gigabyte of RAM, and a 160GB, 7200 RPM hard drive with 8MB of cache. The benchmarking machine runs Debian Wheezy (i.e. Debian 7) using the standard desktop installation.

We use custom-compiled kernels for all benchmarks. The initial configuration was created by running `make localmodconfig` while the machine was booted into its default kernel, which configures the kernel build system to include only the modules which are currently in use. This configuration was then used, unmodified, to compile version 3.18-RC2 of the kernel, which became our baseline environment. We use the same configuration, but with Mandos compiled in and enabled, for all Mandos testing.

4.2 Testing with the Password File

All of our benchmarks use files with the format of the password file, normally located at `\etc\passwd`. Having a valid password file is essential to the usability of a Linux computer system, so it is natural to protect the integrity of the file. The password file allows us to exercise a variety of Maia constraints, including syntax specification (records must be well-formed), universal rules (gid must be between 0 and 65535), existential rules (there must be a user named "root"), rules on compound sets (the user named "root" must have uid 0), and uniqueness constraints (usernames must be unique).

We do not operate on the copy of the file used by the login system, but instead make use of copies elsewhere in the file system. This way we can generate intentionally broken files as part of our testing without breaking the ability to log into the system if the file is unprotected. We can also generate many files alongside one another without unduly cluttering `\etc`.

We have developed tools which can produce new password-formatted files on demand, including a variable number of records and, optionally, a variety of errors. The tools work by reading in an existing password file, shuffling the entries, and selecting the desired number of entries from the beginning of the shuffled list. They can then introduce three classes of error into the file. An invalid record (literally "INVALID") can be added at the beginning of the file, causing an early verification failure due to a malformed record. The first (non-invalid) record in the file can be duplicated into the middle of the file, causing a mid-parse verification failure due to a duplicate user name. It is also possible to have the tools omit the login entry for root, causing an end of file verification failure when the existential rule "there must be a user called root" is found to have been violated. We can also shuffle the resulting file's entries after errors have been added. Once the file contents have been

constructed in memory, the tools time how long it takes to open a designated file, write the contents to disk, and close the file. This time, recorded with microsecond accuracy, is then reported so it can be analyzed later.

We have drawn our source password file directly from Michigan Tech's NIS server, giving us access to 15786 total records. The records contain usernames and (approximate) full names, but no password hashes or other user information. We generally constrain our tests to 15000 records, which produces a total file size around a megabyte. This is probably quite large for a login file, as most organizations with that many users (including Tech) use other systems to manage login data. However, a megabyte is not an unreasonable size for other types of data that might require protection.

4.3 Piecemeal Analysis

We begin our investigation of Mandos' performance by considering each of its major components in isolation. This way we can estimate a minimum value for the overhead added by Mandos, and gain some understanding of how that overhead is distributed over the different operations. Piecemeal analysis can also indicate likely targets for optimization, should that be desired in the future.

Using the password file generation tools, we produce new files which can then be handed to the setup and cleanup utilities or a verifier. We perform 10000 individual trials (setup, cleanup, or verify) for each file size. To account for outliers caused by other system processes – e.g. log rotation or the like – we eliminate any individual trial more than three standard deviations from the mean. We assume that trial times will be approximately normally distributed, so 99.7% of values should fall within three standard deviations of the mean. In practice, we never eliminated more than 188 trials from any given test, and averaged about 60 trials removed (standard deviation: approximately 56, median: 30).

4.3.1 File Handling

As currently implemented, Mandos must always make one copy of a file – the backup, created at `open()` – but may end up performing a total of three copies over the course of a run. (The initial backup, a branch copy, and then rolling back the backup.) The setup and cleanup procedures are invoked for all protected files, so we expect their runtime to contribute to overall performance regardless of verification time. Figure 4.1 presents a summary of timing results.

All of the operations are generally linear, which follows from the fact that they are, at worst, copying files. Cleanup for a valid file is the fastest operation, requiring only a call to `unlink()`. Setup and rollback involve one copy each and have generally similar performance, except for the addition of an `unlink()` in rollback. Branch operations are more costly than rollbacks, but only two to three times more costly. Thus, while there is some extra overhead associated with a branch it should not be a significant factor in overall runtime unless the files being branched are quite large. Combining setup and cleanup time – as would be the case in a running system – gives a worst case delay of less than 13 milliseconds to set up and branch megabyte-sized files.

4.3.2 Password Verifier Performance

While the file handling operations require only copying a file, a verifier must perform additional processing on the file to determine whether it is valid. Maia verifiers exit immediately upon finding an error, which can improve runtime in some cases. However, it is still necessary to parse the entire file to determine whether it is correct, so verification is likely to take longer than file backup and

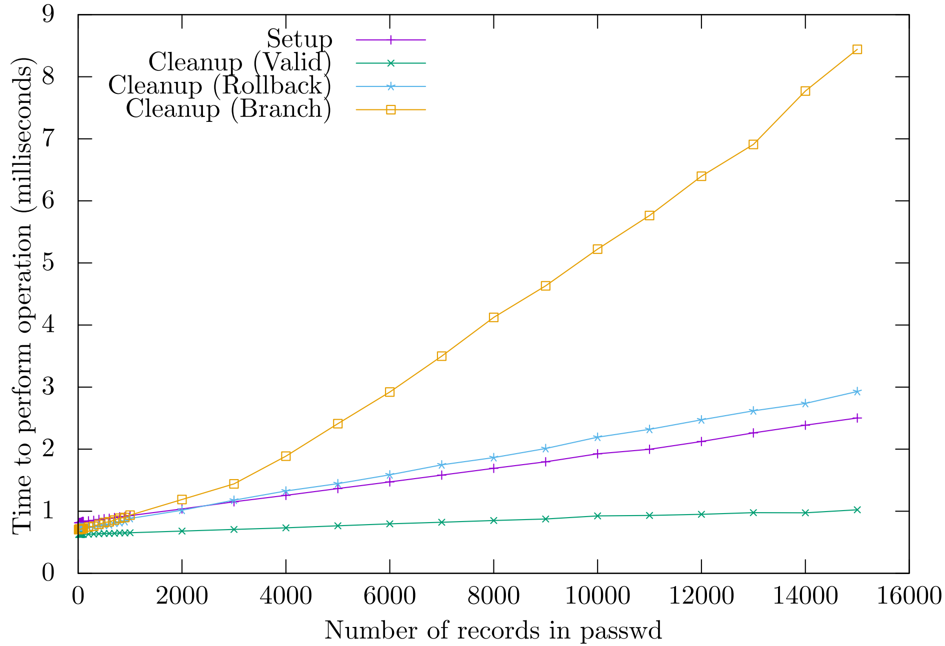


Figure 4.1: File Length versus Setup/Cleanup Time

restore in most cases. We consider four verification cases for our testing: a clean file, an invalid record at the start of the file, a duplicate record in the middle, and a missing root entry, detected at the end of scanning. Figure 4.2 presents average verification times for these cases.

It is immediately apparent that our verifier is linear in the number of records (and therefore file size), regardless of whether the file contains errors. This follows from the overall linear structure of the file, and our use of Flex and Bison to actually produce the parser for Maia verifier. We expect that many files protected by Mandos will also have linear verifiers, as most system files – especially configuration files – are simply read from beginning to end, rather than containing heavily linked structures.

Our verifiers are designed to exit as soon as they know a file is invalid, which is reflected by the way verification time varies with error location. Verifying that a file is clean requires scanning the entire file, and cannot be short circuited. By contrast, an invalid record at the beginning of the file causes the verifier to exit immediately, resulting in essentially constant time to failure. Likewise, an error in the middle exits in about half the time required to verify a clean file.

The error at the end case relies on the violation of an existential rule, namely that a record for root must exist. While it is possible to detect that an existential rule has been satisfied without processing an entire file – finding root's entry satisfies the rule – we cannot conclude that the rule has been violated without scanning the entire file. Thus, existential rules cannot be used to end verification more quickly than scanning a clean file.

The longest verification time is observed for files which are clean or have existential constraint violations, and is slightly less than 74 milliseconds for a 15000 record file. When combined with overhead from file handling, this gives of an overall delay of less than 87 milliseconds if the file must be branched due to an error. More reasonable file sizes – up to 5000 records – have verification times

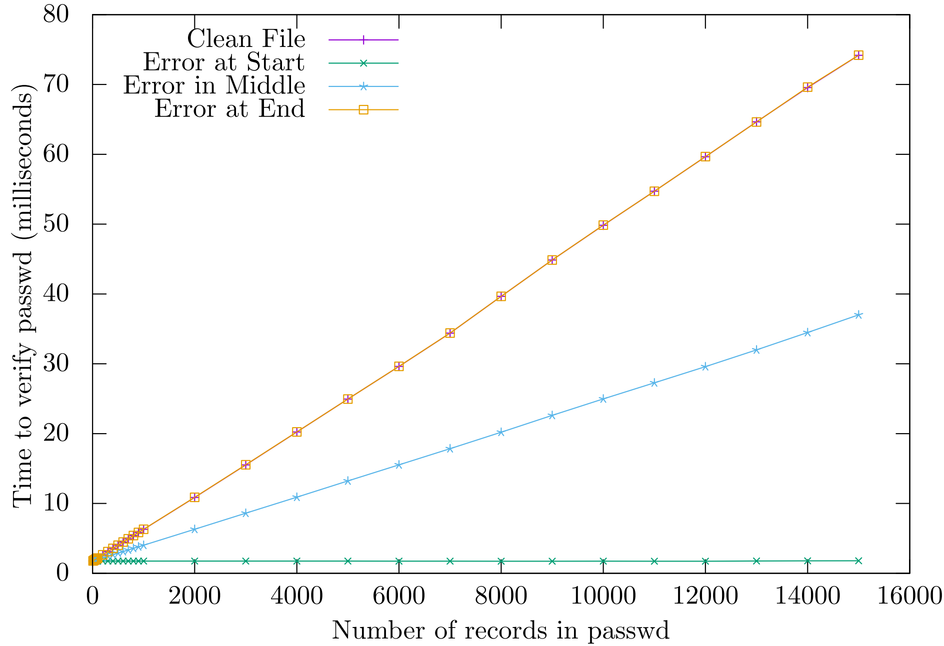


Figure 4.2: File Length versus Verification Time

less than 25 milliseconds, or less than 32 milliseconds including set up and a branch. Conventional wisdom holds that computer systems should respond in less than 100ms [49], though more recent research indicates that images can be recognized in as little as 13ms [55]. Thus, Mandos' overhead should not affect overall usability, though it may be noticeable in some cases.

4.4 Single File Performance

Having demonstrated that Mandos' file handlers and Maia verifiers are reasonably fast, we turn our attention to the performance of Mandos as a whole. For these tests, we consider a single process modifying a single protected file of varying size and correctness. This reflects what we expect to be the normal use-case of Mandos: protecting files which are infrequently modified.

We reuse our password file generating tools, creating valid files along with files with errors at the beginning, middle, and end of the file. As before, we time individual open / write / close sequences with microsecond accuracy, and perform 10000 trials for each configuration, eliminating individual trials more than three standard deviations from the mean. Fewer than 150 trials were eliminated for any configuration. Before each set of tests we create a new password file of the size to be tested next (e.g. creating a 20-record file just before testing 20-record files) to minimize variability caused by differing sizes between the old and new files. File sizes do change slightly as a result of variation in the selected records, but these differences are quite small compared to variability introduced by changes in record count.

We begin our examination by considering the baseline of an unmodified kernel. We then evaluate Mandos' performance without policy loaded and with verifiers which always accept or reject to

examine the impact of the file handling routines. We then introduce verification, to examine Mandos' performance in a real-life scenario.

4.4.1 Baseline

Figure 4.3 presents the results of running our benchmarks on a stock kernel. There is no noticeable variation between the different error types, which follows naturally from a stock kernel not doing any validity checking. Instead, only file length has an impact on write time, which is linear in the size of the file.

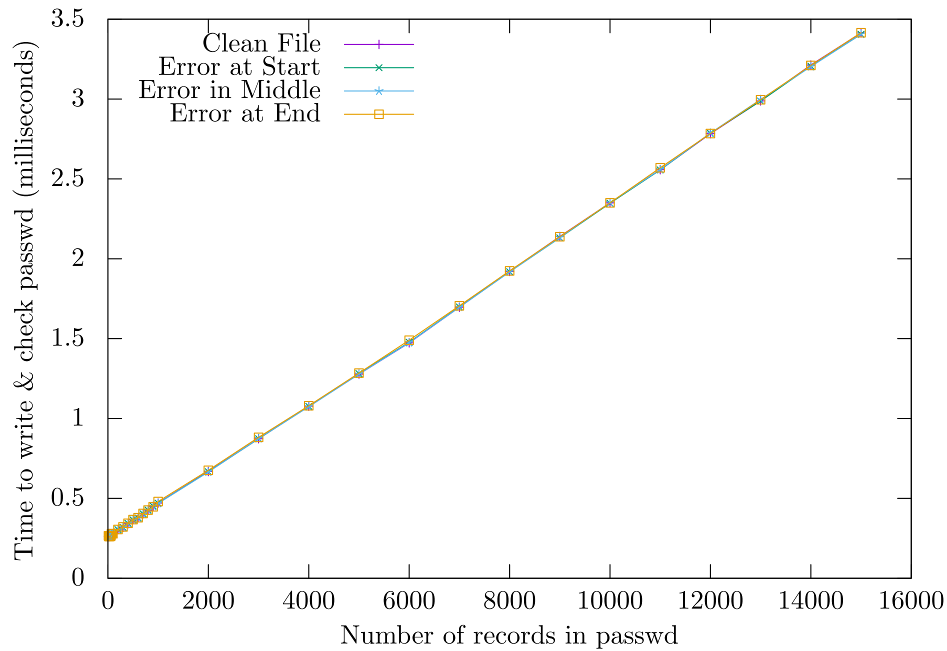


Figure 4.3: File Length versus Processing Time (Stock Kernel)

4.4.2 Mandos Overhead

We now consider the overhead imposed by Mandos' policy considerations, and its helper processes. To examine these parameters, we run tests in four configurations: no policy loaded, using `/bin/true` as the verifier (always accept), and using a custom verifier which always rejects in both branch and rollback configurations. Because Mandos will only enforce policy on files which were valid when the policy was added, we cannot simply use `/bin/false`. Instead, we use a custom-build program which returns false if a particular file is present in the file system, allowing us to switch failure on and off easily. File system caching should minimize the added overhead of this check. Figure 4.4 presents the entire source code of the always-reject program.

Figure 4.5 presents a summary of these test results, along with the unmodified-kernel baseline. There is no appreciable difference between the baseline and running Mandos without policy, indicating that activating Mandos adds negligible time for unprotected files. Always accepting a file adds

```c
#include <unistd.h>

int main() {
        return !access("/home/pjbonamy/VERFAIL", F_OK);
}
```

Figure 4.4: Source Code of Verifier for Always-Reject Tests

about four milliseconds for a 15000 record file, which follows from the time required for the setup and cleanup programs to run for a file of that size. Oddly, rollback and branch seem to take the same amount of time, even though testing the tools individually indicates that branch should be slower. The twelve millisecond difference between rollback / branch and the baseline is about what we would expect to see for a setup followed by a branch.

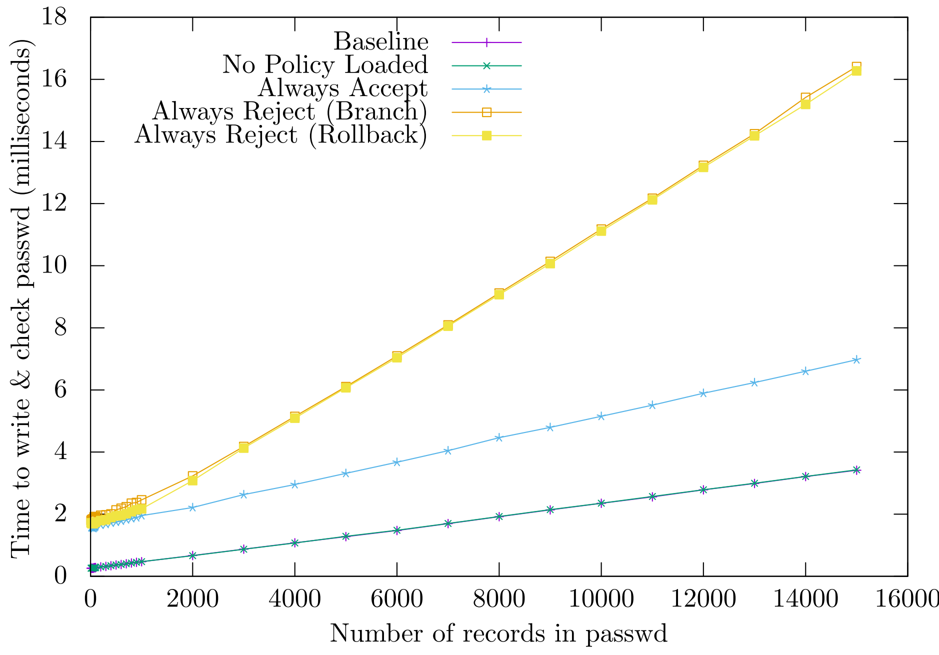


Figure 4.5: File Length versus Mandos Overhead

4.4.3 Mandos Performance

For our final tests in this category, we run a complete Mandos policy, including our password file verifier. We expect that most files protected by Mandos will be modified infrequently by a single process at a time, so these tests are reflective of the general case for protected files. Figure 4.6 summarizes the results.

As with the verifier benchmarks, there is a marked difference in performance between clean files, and files with errors at the beginning, middle, and end of the file. Overall processing time in all

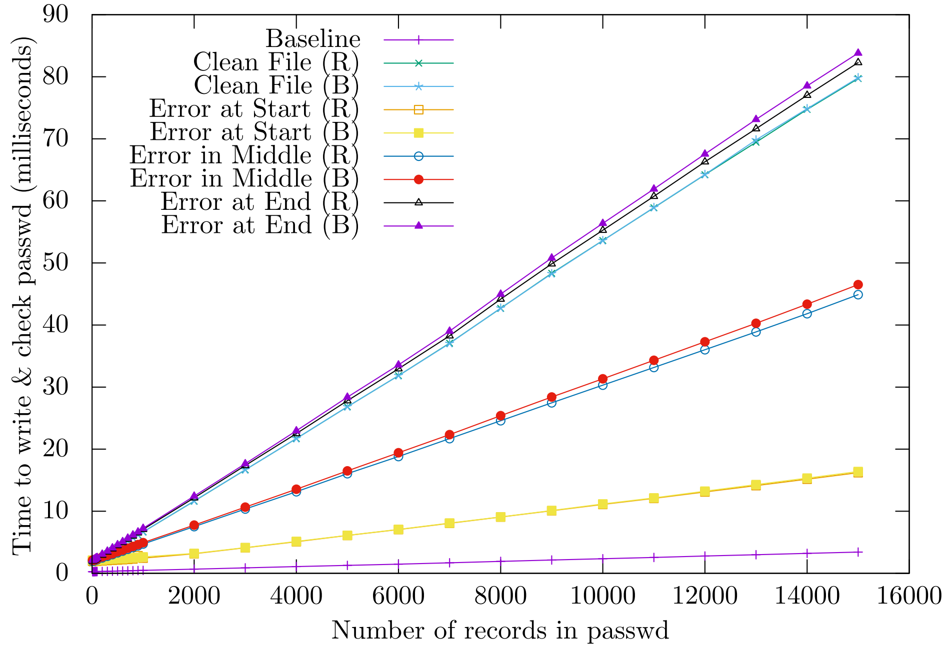


Figure 4.6: File Length versus Processing Time (Mandos Kernel)

cases is close to what we would expect, given our piecemeal results. In the worst case – where we discover an error at the end of a 15000 record file and must then generate a branch – we see a total time of about 85 milliseconds, which is about 80 milliseconds longer than the baseline case without Mandos. For more typical password files containing at most a few thousand users, Mandos adds under 50 milliseconds to total write time.

Interestingly, the error-at-start case shows no performance difference between branch and rollback, which corresponds to our results in Figure 4.5. However, if the error is in the middle of the file or at the end, we do observe a slight timing variation between branch and rollback, which corresponds to Figure 4.1. Clean files are expected to take the same amount of time, as they do not actually have to be rolled back or branched.

It is unclear why we would only observe a difference between branch and rollback time in cases where verification time is negligible. We suspect caching might be involved, however disabling the hard drive's write cache or flushing the page cache during testing did not cause rollback and branch to take different amounts of time for the always-fail verifier. When we ran our benchmarks in a virtual machine – thereby adding additional caching provided by the VM software and the host operating system – the always-fail / error at start case showed a difference in time between branch and rollback. Thus, it is likely the case that the (lack of) delay is being caused by some aspect of the benchmarking machine, rather than Mandos itself. While a definitive explanation for this behavior may point to potential optimizations, the delay itself is quite small and does not detract from the overall value of Mandos.

4.5 Concurrent Verification

Although we expect Mandos to be used primarily to protect files which are infrequently modified, it is entirely possible that multiple protected files might be accessed simultaneously. For example, an administrator might be updating a group of related files (like the login set of password, shadow, and group), or a number of separate users might each be working on one or two protected files. In order to quantify Mandos' performance in this case, we have run benchmarks on a pathological case for the system, thereby hopefully establishing worst-case performance.

For this benchmark, we simulate a variable number of processes all running tight I/O loops on protected files. In particular, we run processes that each loop as quickly as possible, generating password file contents, opening its own file, dumping the password records to disk, and then closing the file, triggering a verify. We hold the length of the files constant at 15000 records, which is the worst case we can readily produce and results in files of about a megabyte. This test protocol allows us to both test a system under heavy I/O load and to see what happens when many processes are causing Mandos verifications at once.

All of the files used in this test are located in the directory. While it is possible to have Mandos-protected files anywhere in the file system, we expect it to be most useful for configuration files which tend to reside in `/etc`. Thus, it is reasonable to expect many protected files to be in the same overall location. Mandos could also be used to protect system software from corruption, which could result in updates happening throughout the file system during a software upgrade. In this case, while many files in many places will need to be processed during the upgrade, we expect that only one or two packages will be updated at a time. Thus, the total number of simultaneous accesses should be relatively low.

The benchmark launches a C program which opens the existing password file and reads it into memory. The program then forks off the desired number of processes, each of which enters a tight loop during which it selects entries from the existing password file, decides randomly whether to include an error and, if so, where, and then writes its new password file to an assigned file which is used throughout the test. This is functionally similar to our single-file tests, though we rely on the pre-fork reading of the input file to reduce our memory overhead.

Generated file contents will be valid approximately half of the time, with errors split evenly between beginning and end. Each process times how long it takes to open its file, write its new contents, and then close the file. Rather than running a set number of iterations, we instead have each test run for five minutes. No new processes are launched directly by the benchmark once it forks off the specified number of processes, though Mandos does still invoke its setup, cleanup, and verifier processes as usual. We run the benchmark on an unmodified kernel, and on Mandos configured to rollback on an error.

Figure 4.7 presents the timing results for these benchmarks, as the number of processes varies from 1 to 100. As expected, in the baseline case of an unmodified kernel, whether the file has errors has no impact on overall write time, which peaks at about 2.5 seconds per file written with 100 processes. Interestingly, while there is some variability in Mandos, for the most part the file state does not seem to impact performance significantly. Write times peak at around 5 seconds per file with 100 processes, regardless of the presence of errors. Once we get beyond five or ten processes in tight loops working on files, the difference in verification time is completely swamped by the time spent waiting for the system to serve each individual process. Clearly, having many processes doing file I/O in tight loops is bad for overall performance, but while Mandos makes this worse it is only about twice as bad up to 100 processes.

While the overall trend looks generally linear in the figure, write times explode as the number of processes increase. At 1000 processes – the highest number the benchmarking machine could

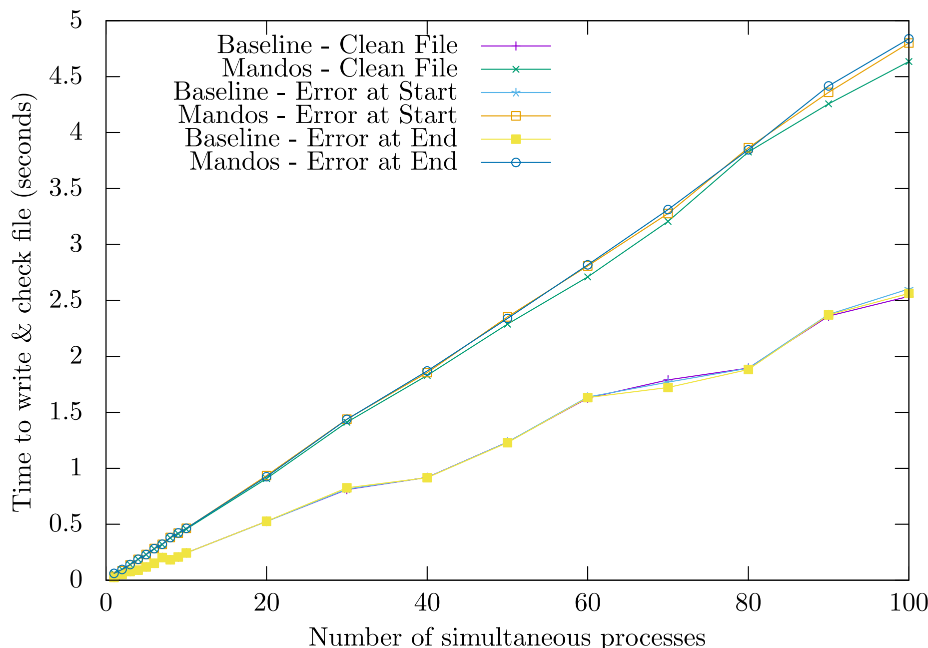


Figure 4.7: Simultaneous Processes versus Processing Time

reliably handle – each write cycle took approximate two minutes on the Mandos kernel. The stock kernel faired better, needing 4500 processes to cross the two minute mark.

Assuming that Mandos-protected files are only modified infrequently, it is unlikely that too many would every be open at once. Thus, we think it is reasonable to expect that there will never be more than 5-10 processes simultaneously causing verifications. In that case, we can see that Mandos only adds about a quarter of a second to output times in this synthetic worst case. On a system that is not doing tight loops on protected files, Mandos may well be faster even with more simultaneous verifications.

4.6 Conclusion

This chapter presented the results of a series of benchmarks on both Maia-generated verifiers and the Mandos integrity module. Mandos itself is quite fast, adding only a few milliseconds to the time required to access a single file. Verification increases processing time, but only by about 60 milliseconds in the worst case. Total processing time was less than 90 milliseconds to verify and branch 15000 record password files.

Mandos' added overhead is more apparent on heavily loaded systems doing tight I/O loops on protected files. However, in these cases Mandos takes only about twice as long as a stock kernel, and variability related to a file's validity is much less apparently. Overall, Mandos performs quite well in its intended use case of infrequently accessed single files, and is serviceable for heavily concurrent accesses.

Chapter 5: Future Work

Thus far, we have presented our work in developing tools for mandatory integrity protection. Maia is a language for describing integrity constraints on arbitrary files, for which we have created a proof of concept verifier generator. Mandos is a Linux module which uses a verify on exit model to enforce straight, Clark-Wilson-like integrity guarantees. In this chapter, we present a selection of future areas of research derived from our existing work.

5.1 Maia

Maia is a flexible tool for describing constraints on a wide variety of file structures. While the language itself seems reasonably complete, it presents opportunities for further exploration.

5.1.1 Reference Implementation

Our proof of concept implementation is incomplete. As we described in Section 2.8, our tool does not support features like indexing, black boxes, or templates. Producing a feature-complete implementation prompts research in one of two areas: either we must come up with automated ways to handle the complexity of using a separate lexer, or we need to employ lexerless parsing.

Our existing verifier generator identifies terminals like string literals, character classes, and regular expressions and uses them as tokens to be recognized by a lexer. This strategy could be improved by automatically reordering character classes and regular expressions from most to least strict, thereby preventing some masking issues. It may also be possible to automatically detect cases where mismatches cannot be fixed by reordering expressions and instead rewriting the expressions, but this runs the risk of changing the overall semantics of a specification.

Maia is designed to describe any file with a context free structure, but most parser generators restrict the set of languages they can parse to make their parsers faster. Tools like Earley parsers [19], which can parse any context free language, have been around since the 1970s but have historically been avoided because they typically require $O(n^3)$ time in the worst case. Research since then, such as the work of Leo [37] and Aycock and Horspool [6], improves the efficiency of Early-style parsers, enabling tools like Marpa [35] to run in $O(n)$ time for many common grammars while parsing all other grammars no more slowly than $O(n^3)$. We could adapt this work for Maia, enabling us to produce verifiers which work for the full range of languages Maia can describe without needing to work around the limitations of our parser generator.

5.1.2 Additional Specifications

We have already created Maia specifications for a variety of formats, including the login files, SSH configuration files, and PNG images (see Appendix A). In addition to producing a fully functional Maia implementation, we would also like to create a library of additional Maia specifications, including reusable component specifications and templates. This could serve both as a collection of sample specifications and as a toolkit for describing new file formats.

Creating new specifications, especially for common file types, makes it more likely that prospective Maia users will be able to find an existing specification that fits their needs. Providing plenty of templates and component specifications – like for `crypt()` password hashes – also makes it easier to create new specifications by building on past work. Both of these should encourage new users and new specifications.

5.1.3 Other Applications

We have designed Maia to work specifically with files, but it should be possible to adapt it to any format which can be broken into discrete chunks for verification. For example, it may be possible to build a firewall which uses Maia descriptions of valid packets to filter network traffic. Maia could also be adapted to provide partial or ongoing verification, checking files as their contents are written, rather than waiting for a complete file to arrive.

Maia's parsing framework could potentially also be adapted for use as a library. A Maia verifier must be able to extract file components so they can be checked against semantic constraints. This facility could be adapted for use by other software, allowing the Maia verifier to provide a reusable parsing backend. Providing a Maia-based, general purpose parser generator also has the advantage that the parsing specification used to develop the software can be directory converted into a Maia integrity specification. Thus, the process of creating a parser for a new file format creates a Maia specification with minimal additional work.

5.2 Mandos

Mandos demonstrates that it is possible to provide robust, Clark-Wilson style integrity guarantees without requiring provably correct programs to modify data. Our system is transparent to user programs, is completely file-system agnostic, and imposes minimal overhead on file operations. We see a number of avenues of future research, which may provide benefits to Mandos itself and to other, unrelated software.

5.2.1 Link Mode Protections

As discussed in Section 3.3.3, Mandos has configuration options to control whether a file may be hard linked, but these are not currently enforced. At present, Mandos relies on file names, rather than inode numbers, to keep track of protected files. In order to support enforcing hard link requirements, we would need to ensure that any directory entry pointing to a protected file's inode triggers protections. This could be accomplished either by scanning the file system to find hard links when files are added, or by attaching protections to inodes rather than file names. The later solution would require more extensive changes to Mandos, but also potentially offers more flexibilty, as then we can attach data protections to inodes, name protections to directory entries, and link protections at the connection points between inodes and directories.

5.2.2 Interface for Userspace Software

At present, the only way a process can determine whether a file it is accessing is protected by Mandos is to consult either the Mandos policy or check for a status file. This makes Mandos completely transparent to userspace processes, but also limits ways the module could be used. We would like to investigate whether there is a benefit to providing userspace processes with the ability to interact with Mandos directly.

For example, a process may wish to request verification before its file is closed. This could be useful in the event that the process believes a transient fault has affected data integrity. It could also be used to allow processes to verify the data they have already written while waiting for more data to arrive. Supporting partial verification could also permit large files to be checked as they are written, removing some of the overhead of verification-on-close.

Mandos-aware processes could also request different behaviors when attempting to open a protected file. The current behavior is to deny access to files which are currently open for writing, which can lead to a process needing to repeatedly try to open a file. Instead, a process could request that it block when it cannot open a file because it needs to be verified. This would allow these processes to go to sleep until the file is ready to be accessed, rather than needing to busy-wait in userspace if they want to access the file as soon as possible.

5.2.3 Awareness of File Groups

At present, Mandos treats all protected files as though they are completely independent of one another, but this is not necessarily the case. For example, the Linux password, shadow, and group files together control who may log into a system and are all interrelated. Under the present system, these files could all share a verifier which examines not only the individual files but the relationships between them, but this can lead to ordering constraints when it comes time to add or remove users. (See the discussion in Section A.7 for details.)

Under a file group structure, the existing protection model could be extended to cover sets of files. When files in a set are modified, verification is not triggered until the last handle to the last file in the set is closed. Then, the entire set is validated and committed or rolled back as a unit. This would require careful management of the readers/writer locks, and backup files, but has the significant advantage of simplifying the requirements for editing groups of files. So long as processes are careful to make sure their accesses in a set overlap, there is no need to worry about the order in which the files are processed.

5.2.4 File Redirection

Mandos currently relies on locally made backup copies, and access control, to facilitate its integrity guarantees. A different strategy could involve being able to redirect files accesses to other places in the file system. For example, if a process requests write access to a file, it could be redirected to a copy located elsewhere in the file system, leaving the original file unmodified, and accessible, until the changes have been verified. Thus, being able to redirect files would reduce file handling overhead (a cleanup no longer needs to restore a backup) and allow processes to read the old, good version of a file while a new version is being written. File redirection could also be useful without Mandos, especially if it were used alongside a file system firewall.

To make redirection work properly, we would need more than the ability to redirect a single open request. It would be necessary to ensure that all concurrent open requests by a process refer to the same file. Thus, we have to keep track of which processes have active redirections, and guarantee that new opens go to the redirected file. Adding to this, we would also need to be able to deal with the case where a file is already open for reading, but a subsequent open for write by the same process would cause a redirect. In this case, the existing read handles should be moved to the new file to guarantee that changes are reflected properly. Additional care would also be required if there are open read handles to the original version of a file when a new one is put into place. Processes with a handle to the old data should have new opens pointed at the old file, lest they be presented with two different versions of the data.

5.2.5 Mandos for Remote File Systems

At present, Mandos only provides protection guarantees for files on directly-attached file systems. Many environments rely on network shares, so it would be worthwhile to investigate protecting files on remote file systems with Mandos. A local copy of Mandos cannot defend remote files from changes originating on other systems (either the server itself, or another connected host), reducing our integrity guarantees. It would be reasonably straightforward to install Mandos on the file server, but this makes it difficult for processes on connected clients to access status information about files they access. Possible approaches for a complete solution might involve having local and remote Mandos installations coordinate their efforts, or providing local fallback behaviors to verify on open and then either deny access or issue a warning.

Appendix A: Maia Specifications

In this appendix, we present a collection of example Maia specifications, including templates, files for the login systems, PNG images, and the SSH configuration file. We present each specification in its entirety first, followed by a description of how the specification works. We intend the specifications to be representative of what might be used in a production environment, though in some cases we include additional restrictions which are illustrative if not actually required.

A.1 `crypt()` Hashes

```
CryptPassword = posixCrypt | glibcCrypt ;

posixCrypt = [a-zA-Z0-9./]{13} ;

glibcCrypt = "$" id "$" salt "$" encrypted ;

id = "1" | "2a" | "5" | "6" ;

salt = [a-zA-Z0-9./]{ , 16} ;

encrypted = [a-zA-Z0-9./]+ ;

glibcCrypt : id == "1" implies length(encrypted) == 22 ;
glibcCrypt : id == "5" implies length(encrypted) == 43 ;
glibcCrypt : id == "6" implies length(encrypted) == 86 ;
```

Description

This specification is intended to be included as part of a standard library, rather than corresponding to a particular file format. The `crypt()` [39] function is provided as part of the standard C library. It is primarily intended to create hashes of passwords for storage and its results may appear in all of three login-related files (`/etc/passwd`, `/etc/shadow`, and `/etc/group` as well as being used by other software. The official standard for `crypt()` only provides one hashing algorithm, but the GNU C Library (glibc) version, used on Linux, provides multiple algorithms. This specification provides support for both the POSIX and glibc version of `crypt()`'s output.

Discussion

As noted above, the glibc version of `crypt()` provides functionality not available in the POSIX version. As a result, the function's output will differ depending on which version was used. We account for this by providing `CryptPassword` as an alternate of a POSIX `crypt()` password and the glibc version. This has the added advantage that a specification could use either of those nonterminals if it only needed POSIX or glibc hashes.

This specification does not provide an ability to verify that something was correctly hashed (which would require making the password available to the verifier), so a POSIX hash is simply thirteen characters. We do not make a distinction between salt and hash in this case because it does not enable additional testing. A glibc hash has more internal structure, so we break it out into its component parts. The ID field identifies the algorithm used to generate the hash, which will determine the length of the encrypted field. The salt is permitted to have variable length under glibc, though the character set is the same. Finally, we leave the encrypted field's length unconstrained at the syntactic level, relying on semantic rules to ensure it contains the expected number of characters. (Algorithm 2a is Blowfish, which is not provided in mainline glibc, but is added by some Linux distributions.)

A.2 File System Objects

```
1 (template fso exists())
2                  blackbox(fsobj_exists, fso) ;
3
4 (template fso isFile())
5                  blackbox(fsobj_isFile, fso) ;
6
7 (template fso isBinaryExec())
8                  blackbox(fsobj_isBinaryExec, fso) ;
9
10 (template fso isDirectory())
11                 blackbox(fsobj_isDirectory, fso) ;
12
13 (template fso isAbsPath())
14                 blackbox(fsobj_isAbsPath, fso) ;
15
16 (template fso isRelPath())
17                 blackbox(fsobj_isRelPath, fso) ;
18
19 (template fso isAccessibleTo(user))
20                 blackbox(fsobj_isAccessibleTo, fso, user) ;
21
22 (template fso isExecutableBy(user))
23                 blackbox(fsobj_isExecutableBy, fso, user) ;
24
25 (template fso isOwnedBy(user))
26                 blackbox(fsobj_isOwnedBy, fso, user) ;
```

Description

The file system object specification is intended to be included as part of a standard library, and provides mechanisms to access the properties of objects in the file system. We rely heavily on black box verifiers to provide access to the file system, and wrap the black boxes in templates to make them easier to interact with.

Discussion

This specification provides only semantic rules, without providing any syntax definitions. As a result, a specification wishing to use file system objects will still need to provide its own syntax rules to identify a path or file name. We take this approach because Unix paths are essentially unconstrained, but most file formats do include some limitations. For example, a path may include colons, but the password file (see Section A.4) uses colons as field separators. Thus, if we provided a syntax specification other specifications would either need to limit it to match their own format requirements or extend it to include valid paths we omitted.

A.3 Templates for Set Ordering

```
1 (template set isUnique())
2                 forEvery set : set[i] != set[j] ;
3
4 (template set isAscending())
5                 forEvery set : i < j implies set[i] < set[j] ;
6
7 (template set isNonDescending())
8                 forEvery set : i < j implies set[i] <= set[j] ;
9
10 (template set isNonAscending())
11                forEvery set : i < j implies set[i] >= set[j] ;
12
13 (template set isDescending())
14                forEvery set : i < j implies set[i] > set[j] ;
```

Description

Once again, this specification is intended for inclusion in a standard library. It uses the indexing system (see Section 2.6.9) to provide templates for various common set orderings.

Discussion

The implicit indexing system is Maia in primarily intended for enforcing ordering constraints, though it could be used for other things. Uniqueness testing does not require directly comparing indices, because the constraint needs to hold for every pair of set elements. Enforcing ordering does require comparing indices, as we need to know about both the values of the elements and their relative location in the files.

Implicit indexing can have interesting side effects for what data needs to be collected. In an all-pairs case, like `isUnique()`, it is necessary to keep track of all of the values which are seen over the course of parsing a file requiring $O(n)$ storage. In our proof-of-concept implementation, uniqueness testing uses a hash for storage though that may not work for other all-pairs rules. The other indexed rules here could also be implemented by retaining a collection of all encountered values, along with ordering information. However, with these rules at least it would be possible to implement the rule with $O(1)$ additional memory by simply storing the last value we encountered. For example, the `isAscending()` rule can be processed as follows:

1. Assume that all previous elements are in order (trivially true for the first element)
2. Compare current element to previous element
   (a) If the current element is greater than the last, continue
   (b) Otherwise, fail

It is unclear whether this strategy would hold in general, but it should be possible to detect cases where it is valid – perhaps by intercepting these templates – and substitute the $O(1)$ check.

A.4 Linux Password File

```
1  using "CryptPassword.maia" ;
2  using "FSObject.maia" ;
3  using "SetTemplates.maia" ;
4
5  PasswdFile = (passwdRecord Newline)+ ;
6
7  passwdRecord = name ":" password ":" uid ":" gid ":"
8                 gecos ":" directory ":" shell ;
9
10 name = [a-zA-Z_][-a-zA-Z0-9_]{0,31} ;
11
12 password = "*" | "x" | CryptPassword ;
13
14 uid = StringPosDec+ ;
15
16 gid = StringPosDec+ ;
17
18 gecos = [^:\n]* ;
19
20 directory = [^:\n]+ ;
21
22 shell = [^:\n]* ;
23
24 Newline = "\n" ;
25
26 name isUnique() ;
27
28 exists name : name == "root" ;
29
30 (warn) name : name ~ /[A-Z]/ ;
31
32 uid: uid <= 65535;
33
34 gid: gid <= 65535;
35
36 directory : directory isAbsPath() and directory isDirectory() ;
37
38 passwdRecord : name == "root" implies uid == 0;
39
40 passwdRecord : directory isAccessibleTo(name);
41
42 passwdRecord : shell != "" implies
43                ( shell is AbsPath() and shell isExecutableBy(user) ) ;
```

Description

The password file (typically stored at `/etc/passwd`) uses a colon-separated format to store user login information [43]. This includes a username, numeric user and default group identifiers, a full name, and information about the user's home directory and default shell. The password file may also store a hash of the user's password, though the file is world-readable, so password hashes are generally stored in the shadow file.

Discussion

This specification is the first in this collection designed to be tied to a particular file (`/etc/passwd`). It makes use of the preceding three specifications to streamline the overall specification.

Structurally, we build the file as newline-separated password records. This could be handled equivalently by including the newline as part of the password record. We separate them primarily to draw a distinction between the login records themselves and how they are stored in the file.

Relatively few restrictions are actually placed on the contents of the password file. This specification is designed to enforce the explicit restrictions along with a number of common-sense additions. For example, there is no particular requirement that usernames be unique. The login system will simply use the first record it encounters. Likewise, it is not strictly required that there be a user named "root", though this is the traditional name for the superuser account on Linux.

The manual page for the password file specifies that user names should not contain capital letters, but some distributions, including Debian, automatically create user accounts which have capitals. We use a warning to check usernames for capital letters, as that provides greater flexibility. We permit implementations to either elevate or squash warnings, so an administrator who wishes to enforce the restriction can it required while an administrator who wants or needs capitals can disable the warning.

The manual page for the password file does specifically require UIDs and GIDs be in the range 0 to 65535. Using the `StringPosDec` parsed numeric type automatically requires that the the value be non-negative, so the bottom end of the range is provided for free. Thus, we only need to restrict the upper end of the range to provide the overall restriction.

It's reasonable to require that home directories are specified using absolute paths (the current working directory is not well-defined during login), and that the specified thing actually be a directory. We need not perform an explicit existence check, as existence is implied by being a directory.

The last rules use password records as a compound set to perform sanity checks for given users. There is no specific requirement that "root" be the superuser account, but it is reasonable to assume that if an account with that name exists it should be a superuser account. The rule could be strengthened by changing `implies` to `iff`, which would guarantee that only root is a superuser account.

A user needs to be able to access their own home directory to log in successfully. Likewise, while a shell need not be specified, it is important that a user be able to execute their shell if one is specified. Being executable by the user requires that the shell actually exist, so no separate existential check is required.

A.5 Linux Shadow File

```
1  using "CryptPassword.maia" ;
2  using "SetTemplates.maia" ;
3
4  shadowFile = (shadowRecord Newline)+ ;
5
6  shadowRecord = name ":" password ":" lastPasswordChange ":"
7                 minimumPasswordAge ":" maximumPasswordAge ":"
8                 passwordWarningPeriod ":" passwordInactivityPeriod ":"
9                 accountExpirationDate ":" reserved ;
10
11 name = [a-zA-Z_][-a-zA-Z0-9_]{0,31} ;
12
13 password = CryptPassword ;
14
15 lastPasswordChange = StringPosDec* ;
16
17 minimumPasswordAge = StringPosDec* ;
18
19 maximumPasswordAge = StringPosDec* ;
20
21 passwordWarningPeriod = StringPosDec* ;
22
23 passwordInactivityPeriod = StringPosDec* ;
24
25 accountExpirationDate = StringPosDec* ;
26
27 reserved = [^:\n]* ;
28
29 name isUnique();
30
31 Newline = "\n" ;
```

Description

The shadow file (typically stored at `/etc/shadow`)is a colon separated file used to store password hashes and other, password-adjacent information [44]. The file was created so that access to password hashes could be restricted without disallowing access to other password file information. (Processes routinely access the password file to map UIDs to usernames when listing file ownership, for example.)

Discussion

Once again, we rely on libraries to provide additional functionality in this specification. We also rely on the `StringPosDec` type to ensure that, if we have additional information or restrictions they have sane values.

A.6 Linux Group File

```
1  using "CryptPassword.maia" ;
2  using "SetTemplates.maia" ;
3
4  GroupFile = (groupRecord Newline)+ ;
5
6  groupRecord = groupName ":" password ":" gid ":" userList? ;
7
8  groupName = [a-zA-Z_][-a-zA-Z0-9_]{0,31} ;
9
10 password = CryptPassword | x ;
11
12 gid = StringPosDec+ ;
13
14 userList = userName ("," userName)* ;
15
16 userName = [a-zA-Z_][-a-zA-Z0-9_]{0,31} ;
17
18 Newline = "\n" ;
19
20 groupName isUnique();
21
22 gid isUnique();
```

Description

The group file (typically stored at `/etc/group`) is a colon separated file listing all user groups known to the system and providing details of which users are members of each group [41]. It defines the name of each group, along with a numeric identifier and optional list of users associated with the group. A group may also have an associated password, which must be supplied by a non-group-member if they wish to use the `newgrp` command to switch to the group [40] [42]. (A user may not switch to a group of which they are not a member if that group does not have a password.)

Discussion

For this specification, we want to ensure that group names and group identifiers are unique, so we do not accidentally end up with overlapping or contradictory definitions. Beyond that, we are primarily concerned with making sure the structure of the file is correct. We need a multi-file verifier (like the one in the next section) to ensure that specified usernames actually map onto real users.

A.7 Linux Login Set

```
1 using "PasswdFile.maia" on "/etc/passwd" ;
2 using "ShadowFile.maia" on "/etc/shadow" ;
3 using "GroupFile.maia"  on "/etc/group"  ;
4
5 passwdRecord : password == 'x' implies name in shadowRecord.name ;
6
7 PasswdFile.gid : gid in GroupFile.gid ;
8
9 GroupFile.userName : userName in PasswdFile.name ;
```

Description

The password, shadow, and group files can be verified individually, but there are also constraints between files that we might wish to enforce. For example, we might wish to require that a user's default group actually appear in the group file. This specification uses the specifications for the individual files to create a multi-file verifier for the login set.

Discussion

To create a multi-file verifier, we need to specify not only which single-file specifications to include but to which files they should be bound. In this case, we give full paths to the protected files, as they do not move around and absolute paths mean we can put the verifier anywhere in the file system.

We enforce two major restrictions to protect the sanity of the login system. First, we require that any user whose password hash should be in the shadow file (indicated by an "x" in the password file password hash) actually appear in the shadow file. We then require that all group IDs specified in the password file be in the group file. Between these restrictions,we guarantee that users who should have a password will actually have a password, and that users will log in with a valid group.

In this specification, we also provide the sensible restriction that users named in the group file should also be present in the password file. This is not necessary, as a user who is not in the password file cannot log in, and therefore would have no access to groups. A similar restriction could be added, requiring that users in the shadow file have password file records. However, this sort of restriction makes adding users more complicated.

Assume that all three files – password, shadow, and group – are protected by something like Mandos, which will revalidate the entire set on close. Without the requirement that users in the group file be listed in password, adding a new user only requires that group and shadow are updated, if need be, before the user record appears in password. This is a reasonable ordering, as, again, a user cannot log in without a record in password. However, adding the requirement that users in the group file be in password means that the group entry has to exist before the new user is added to password, but it cannot reference the new user. If a group is being created with the user (which is standard practice for desktop installations descended from Debian), this would require two open / write / close cycles on group: one to add the group so the password file is valid, and one to add the reference to the new user. A similar problem occurs if we enforce a restriction that shadow entries must correspond to a password entry, though now the password entry must be written twice: once to add the user and a second time to mark the user as having a password in shadow.

Note that these ordering restrictions only apply if the files are going to be verified while they are still in the process of being updated. A single check performed after all updates are complete could rule on the validity of the set as a whole without any ordering issues.

A.8 PNG Images

```
1 PNGImage = sig ihdr (prePTLE | preIDAT | anyTime)* plte?
2                 (postPLTE | preIDAT | anyTime)*  idat+ anyTime* iend;
3
4 sig = "\x89PNG\r\n\x1A\n" ;
5
6 prePTLE = chrm | gama | iccp | sbit | srgb ;
7
8 postPLTE = bkgd | hist | trns ;
9
10 preIDAT = phys | splt ;
11
12 anyTime = time | itxt | text | ztxt | other ;
13
14 ihdr = "\0\0\0\x0D" "IHDR" ihdrData ihdrCRC ;
15 ihdrData = width height bitDepth colourType
16                  compMethod filterMethod interlaceMethod ;
17 width = UnsignedBigEndianInt{4} ;
18 height = UnsignedBigEndianInt{4} ;
19 bitDepth = BigEndianInt{1} ;
20 colourType = BigEndianInt{1} ;
21 compMethod = BigEndianInt{1} ;
22 filterMethod = BigEndianInt{1} ;
23 interlaceMethod = BigEndianInt{1} ;
24
25 plte = plteLen "PLTE" plteData plteCRC ;
26 plteLen = UnsignedBigEndianInt{4} ;
27 plteData = plteEntry{plteLen / 3} ;
28 plteEntry = plteRed plteGreen plteBlue ;
29 plteRed = BigEndianInt{1} ;
30 plteGreen = BigEndianInt{1} ;
31 plteBlue = BigEndianInt{1} ;
32 plteCRC = BigEndianInt{4} ;
33
34 idat = idatLen "IDAT" idatData idatCRC ;
35 idatLen = UnsignedBigEndianInt{4} ;
36 idatData = .{idatLen} ;
37 idatCRC = BigEndianInt{4} ;
38
39 iend = "\0\0\0\0" "IEND" "\xae\x42\x60\x82" ;
40
41 chrm = "\0\0\0\x10" "cHRM" chrmData chrmCRC ;
42 chrmData = whiteX whiteY redX redY greenX greenY blueX blueY ;
43 whiteX = UnsignedBigEndianInt{4} ;
44 whiteY = UnsignedBigEndianInt{4} ;
45 redX = UnsignedBigEndianInt{4} ;
46 redY = UnsignedBigEndianInt{4} ;
```

```
47 greenX = UnsignedBigEndianInt{4} ;
48 greenY = UnsignedBigEndianInt{4} ;
49 blueX = UnsignedBigEndianInt{4} ;
50 blueY = UnsignedBigEndianInt{4} ;
51 chrmCRC = BigEndianInt{4} ;
52
53 gama = "\0\0\0\x04" "gAMA" gamaData gamaCRC ;
54 gamaData = UnsignedBigEndianInt{4} ;
55 gamaCRC = BigEndianInt{4} ;
56
57 iccp = iccpLen "iCCP" iccpData iccpCRC ;
58 iccpLen = UnsignedBigEndianInt{4} ;
59 iccpData = .{iccpLen} ;
60 iccpCRC = BigEndianInt{4} ;
61
62 sbit = sbitLen "sBIT" sbitData sbitCRC ;
63 sbitLen = UnsignedBigEndianInt{4} ;
64 sbitData = .{sbitLen} ;
65 sbitCRC = BigEndianInt{4} ;
66
67 srgb = "\0\0\0\x01" "sRGB" srgbData srgbCRC ;
68 srgbData = srgbRenderIntent ;
69 srgbRenderIntent = BigEndianInt{1} ;
70 srgbCRC = BigEndianInt{4} ;
71
72 bkgd = bkgdLen "bKGD" bkgdData bkgdCRC ;
73 bkgdLen = UnsignedBigEndianInt{4} ;
74 bkgdData = .{bkgdLen} ;
75 bkgdCRC = BigEndianInt{4} ;
76
77 hist = histLen "hIST" histData histCRC ;
78 histLen = UnsignedBigEndianInt{4} ;
79 histData = .{histLen} ;
80 histCRC = BigEndianInt{4} ;
81
82 trns = trnsLen "tRNS" trnsData trnsCRC ;
83 trnsLen = UnsignedBigEndianInt{4} ;
84 trnsData = .{trnsLen} ;
85 trnsCRC = BigEndianInt{4} ;
86
87 phys = "\0\0\0\x09" "pHYs" physData physCRC ;
88 physData = pixelsPerUnitX pixelsPerUnitY unitSpecifier ;
89 pizelsPerUnitX = UnsignedBigEndianInt{4} ;
90 pizelsPerUnitY = UnsignedBigEndianInt{4} ;
91 unitSpecifier = BigEndianInt{1} ;
92 physCRC = BigEndianInt{4} ;
93
94 splt = spltLen "sPLT" spltData spltCRC ;
```

```
95 spltLen = UnsignedBigEndianInt{4} ;
96 spltData = .{spltLen} ;
97 spltCRC = BigEndianInt{4} ;
98
99 time = "\0\0\0\x07" "tIME" timeData timeCRC ;
100 timeData = year month day hour minute second ;
101 year = BigEndianInt{2} ;
102 month = BigEndianInt{1} ;
103 day = BigEndianInt{1} ;
104 hour = BigEndianInt{1} ;
105 minute = BigEndianInt{1} ;
106 second = BigEndianInt{1} ;
107 timeCRC = BigEndianInt{4} ;
108
109 itxt = itxtLen "iTXt" itxtData itxtCRC ;
110 itxtLen = UnsignedBigEndianInt{4} ;
111 itxtData = .{itxtLen} ;
112 itxtCRC = BigEndianInt{4} ;
113
114 text = textLen "tEXt" textData textCRC ;
115 textLen = UnsignedBigEndianInt{4} ;
116 textData = .{textLen} ;
117 textCRC = BigEndianInt{4} ;
118
119 ztxt = ztxtLen "zTXt" ztxtData ztxtCRC ;
120 ztxtLen = UnsignedBigEndianInt{4} ;
121 ztxtData = .{ztxtLen} ;
122 ztxtCRC = BigEndianInt{4} ;
123
124 other = otherLen otherName otherData otherCRC ;
125 otherLen = UnsignedBigEndianInt{4} ;
126 otherName = [a-zA-Z]{4} ;
127 otherData = .{otherLen} ;
128 otherCRC = BigEndianInt{4} ;
129
130 ihdr : ihdrCRC == blackbox(CRC-32, "IHDR" . ihdrData) ;
131 idat : idatCRC == blackbox(CRC-32, "IDAT" . idatData) ;
132 plte : plteCRC == blackbox(CRC-32, "PLTE" . plteData) ;
133 chrm : chrmCRC == blackbox(CRC-32, "cHRM" . chrmData) ;
134 gama : gamaCRC == blackbox(CRC-32, "gAMA" . gamaData) ;
135 iccp : iccpCRC == blackbox(CRC-32, "iCCP" . iccpData) ;
136 sbit : sbitCRC == blackbox(CRC-32, "sBIT" . sbitData) ;
137 srgb : srgbCRC == blackbox(CRC-32, "sRGB" . srgbData) ;
138 bkgd : bkgdCRC == blackbox(CRC-32, "bKGD" . bkgdData) ;
139 hist : histCRC == blackbox(CRC-32, "hIST" . histData) ;
140 trns : trnsCRC == blackbox(CRC-32, "tRNS" . trnsData) ;
141 phys : physCRC == blackbox(CRC-32, "pHYs" . physData) ;
142 splt : spltCRC == blackbox(CRC-32, "sPLT" . spltData) ;
```

```
143 time : timeCRC == blackbox(CRC-32, "tIME" . timeData) ;
144 itxt : itxtCRC == blackbox(CRC-32, "iTXt" . itxtData) ;
145 text : textCRC == blackbox(CRC-32, "tEXt" . textData) ;
146 ztxt : ztxtCRC == blackbox(CRC-32, "zTXt" . ztxtData) ;
147 other : otherCRC == blackbox(CRC-32, otherName . otherData) ;
148
149 PNGImage : count(chrm) <= 1 ;
150 PNGImage : count(gama) <= 1 ;
151 PNGImage : count(iccp) <= 1 ;
152 PNGImage : count(sbit) <= 1 ;
153 PNGImage : count(srgb) <= 1 ;
154 PNGImage : count(bkgd) <= 1 ;
155 PNGImage : count(hist) <= 1 ;
156 PNGImage : count(trns) <= 1 ;
157 PNGImage : count(phys) <= 1 ;
158 PNGImage : count(time) <= 1 ;
159
160 PNGImage : count(iccp) == 1 implies count(srgb) == 0 ;
161 PNGImage : count(srgb) == 1 implies count(iccp) == 0 ;
162
163 ihdr : width >= 1 and width <= 2^31 - 1 ;
164 ihdr : height >= 1 and  height <= 2^31 - 1 ;
165 ihdr : bitDepth in < 1, 2, 4, 8, 16 > ;
166 ihdr : colorType in < 0, 2, 3, 4, 6 > ;
167 ihdr : colorType in < 2, 4, 6 > implies bitDepth in < 8, 16 > ;
168 ihdr : colorType == 3 implies bitDepth in < 1, 2, 4, 8 > ;
169 ihdr : compMethod == 0 ;
170 (warn) ihdr : filterMethod != 0 ;
171 (warn) ihdr : not interlaceMethod in < 0, 1 > ;
172
173 PNGImage : colourType == 3 implies count(PLTE) == 1 ;
174 PNGImage : colorType in < 0, 4 > implies count(PLTE) == 0 ;
175 plte : plteLen % 3 == 0 ;
176 PNGImage : (plteLen / 3) <= 2 ^ bitDepth ;
177
178 sbit : colourType == 0 implies sbitLen = 1 ;
179 sbit : colourType in < 2, 3 > implies sbitLen = 3 ;
180 sbit : colourType == 4 implies sbitLen = 2 ;
181 sbit : colourType == 6 implies sbitLen = 4 ;
182
183 srgb : srgbRenderIntent in < 0, 1, 2, 3 > ;
184 (warn) PNGImage : count(srgb) == 1 implies count(gama) == 1 ;
185 (warn) PNGImage : count(srgb) == 1 implies gamaData == 45455 ;
186 (warn) PNGImage : count(srgb) == 1 implies count(chrm) == 1 ;
187 (warn) PNGImage : count(srgb) == 1 implies whiteX = 31270 ;
188 (warn) PNGImage : count(srgb) == 1 implies whiteY = 32900 ;
189 (warn) PNGImage : count(srgb) == 1 implies redX = 64000 ;
190 (warn) PNGImage : count(srgb) == 1 implies redY = 33000 ;
```

```
191 (warn) PNGImage : count(srgb) == 1 implies greenX = 30000 ;
192 (warn) PNGImage : count(srgb) == 1 implies greenY = 60000 ;
193 (warn) PNGImage : count(srgb) == 1 implies blueX = 15000 ;
194 (warn) PNGImage : count(srgb) == 1 implies blueY = 6000 ;
195
196 bkgd : colorType in < 0, 4 > implies bkgdLen == 2 ;
197 bkgd : colorType in < 2, 6 > implies bkdgLen == 6 ;
198 bkgd : colorType == 3 implies bkgdLen == 1 ;
199
200 hist : histLen == plteLen ;
201
202 trns : colourType == 0 implies trnsLen = 2 ;
203 trns : colourType == 2 implies trnsLen = 6 ;
204 trns : colourType == 3 implies trnsLen <= plteLen ;
205 PNGImage : colorType in < 4, 6 > implies count(trns) == 0 ;
206
207 phys : pixelsPerUnitX >= 1 and pixelsPerUnitX <= 2^31 - 1 ;
208 phys : pixelsPerUnitY >= 1 and pixelsPerUnitY <= 2^31 - 1 ;
209 phys : unitSpecifer in < 0, 1 > ;
210
211 time : year >= 1995 ;
212 time : month in [1, 12] ;
213 time : day in [1, 31] ;
214 time : hour in [0, 23] ;
215 time : minute in [0, 59] ;
216 time : second in [0, 60] ;
217
218 other : not otherName in < "IHDR", "PLTE", "IDAT", "IEND",
219                            "cHRM", "gAMA", "iCCP", "sBIT",
220                            "sRGB", "bKGD", "hIST", "tRNS",
221                            "pHYs", "sPLT", "tIME", "iTXt",
222                            "tEXt", "zTXt" > ;
```

Description

The PNG (Portable Network Graphics) image format [30] is a raster format that supports lossless compression. It was originally developed to work around the patent restrictions, and other issues, with the GIF format. The first version of the standard was published as a W3C Recommendation in October of 1996. The current version of the standard is from 2003, and was released as both a W3C Recommendation and an ISO/IEC International Standard (ISO/IEC 15948:2003).

This Maia specification is designed to be able to verify the internal consistency of a PNG image. That is, it checks that image chunks are the right size, and performs other sanity checks. However, it neither guarantees that an image will have the dimensions specified in its header nor makes any attempt to verify that the encoded image is correct. The former class of restriction could be added with additional semantic rules. The latter would require machine vision techniques or the like, and is thus beyond the scope of Maia-based verification.

Discussion

The PNG specification makes use of ASCII character codes in some places, to aid readability of the raw code. However, it explicitly states that any letters present in identifiers must correspond to ASCII codes and not equivalent glyphs from unicode or the like. For this specification, we assume that strings will be interpreted as ASCII values, rather than a different character encoding. If this approach is not taken by the implementation, all string literals should be changed to explicit values.

A PNG image begins with an eight-byte signature or file header, which serves both to identify the file type and to detect potential transmission-related errors. This signature must be present, in exactly the given format, for the image to be considered valid. (The signature includes testing for transmission systems which do not support 8-bit data, systems which automatically convert DOS line endings to Unix, and systems which convert Unix line endings to DOS.)

The remainder of the image is composed of chunks, all of which share the same general format: four bytes specifying the length of the data field, four bytes giving the type of the chunk, length bytes of data, and four bytes giving a CRC value calculated for the type and data fields, but not the length. The specification requires that all multi-byte data values be given in network (i.e. big-endian) byte order. The length field may be zero, in which case no data field is present. The type field shall consist of four bytes with decimal values of 65 to 90 or 97 to 122, corresponding to the ASCII encoding of upper- and lower-case letters. The case of the letter carries an advisory meaning when defining additional chunks.

The standard defines eighteen chuck types, and permits additional types to be created. Four of the chunks are considered critical, must be correctly interpreted by all implementations, and must appear in a specified order. (The critical chunks are an image header, optional palette, the actual image data, and an end-of-file marker.) The remaining fourteen chunks are termed ancillary chunks and have locations that are only constrained relative to the critical chunks. We replicate this structure by defining a PNG using fixed locations for the critical chunks and grouping the remainder according to their relative locations. PNG images support adding additional ancillary chunks. These must have the same general structure as any other chunk, but are otherwise largely unconstrained.

Some chunks have a fixed size, so we hard code these values and the contents of their data blocks. For the remaining chunks, we use Maia's limited context sensitivity to define the data field to be *length* single bytes. This allows us to find chunk boundaries – assuming the file is correctly structured – but can make it a bit more difficult to access the contents of the data field.

Aside from the challenges associated with variable-length data fields, a PNG image is reasonably straightforward, structurally. The semantic rules for the format are potentially more complex, and begin around line 130 of this specification.

Our first semantic requirement is simply that all of the chunks have valid CRC values. For this, we assume the existence of a blackbox that computes and returns the four-byte (32-bit) CRC value of some provided data. We need not perform a CRC check on the `iend` chuck because its CRC value is necessarily constant and can thus be checked by the syntax specification.

Most of the ancillary chunks are not allowed to appear more than once. This would be a straightforward restriction if they had a fixed ordering, but is complicated by the fact that most chunks can appear in multiple locations. We work around this by enforcing count restrictions on the chunks which do not permit duplicates. It is also excepted that a PNG will provide either an ICCP or SRGB color space profile, so we enforce a require that only one be present.

The remainder of the specification is divided into groups for each chunk. The image header has the most constraints applied to it, as it describes how the remainder of the file should be processed. In some cases, the specification only provides one or two methods to accomplish a task, even though a full byte is provided to indicate chosen method. In the case of compression, choosing an algorithm other than the one provided is considered an error. For filter and interlace method, however, it is

not an error to use different methods, so we only issue a warning.

The specification recommends that images which include an sRGB chunk also include gamma and chroma data, in case the decoder does not implement sRGB color spaces. However, the specification requires that, if gamma and chroma are provided for an sRGB image, they be given particular values. We deal with this by providing warnings if the gamma or chroma chunks are missing or have the wrong values.

The time chunk indicates when the image was last modified. It requires that the complete year be specified, which we express as a simple, numeric comparison. The PNG format did not exist before 1995, so it is reasonable to require that images be last modified after that. We also allow a second value of 60 because the specification specifically allows for leap seconds.

In this specification, the `other` chunk is used to catch any chunk type included in an image but not provided by the standard. Any number of these may be included, and they have essentially any type specifier. We simply require that they not use a type specifier that is provided by the ISO specification.

A.9 SSH Configuration

```
using "NetworkAddress.maia" ;

SSHConfig = configLine* hostConfig* ;

configLine = ( comment | configKeyword ) Newline ;

hostConfig = host Newline configLine+ ;

comment = "" | "#" .* ;

configKeyword = addressFamily | ciphers | compression |
                compressionLevel | hashKnownHosts | hostName |
                localForward ;

sep = WS+ | ( WS* "=" WS* ) ;

WS = [\n\t ] ;

Newline = "\n" ;

host = [hH][oO][sS][tT] sep
                ( hostOpt | '"' hostOpt '"' )
                ( WS+ ( hostOpt | '"' hostOpt '"' ) )* ;

hostOpt = pattern | "*" ;

pattern = [a-zA-Z0-9.?*!]+ ;

addressFamily =
        [aA][dD][dD][rR][eE][sS][sS][fF][aA][mM][iI][lL][yY] sep
        ( addrFamilyOpt | '"' addrFamilyOpt '"' ) ;

addrFamilyOpt = "any" | "inet" | "inet6" ;

ciphers = [cC][iI][pP][hH][eE][rR][sS] sep
                ( ciphersOpt | '"' ciphersOpt '"' )
                (WS* "," WS* ( ciphersOpt | '"' ciphersOpt '"' ) )* ;

ciphersOpt = "3des-cbc" | "aes128-cbc" | "aes192-cbc" |
                "aes256-cbc" | "aes128-ctr" | "aes192-ctr" |
                "aes256-ctr" | "arcfour128" | "arcfour256" |
                "arcfour" | "blowfish-cbc" | "cast128-cbc" ;

compression = [cC][oO][mM][pP][rR][eE][sS][sS][iI][oO][nN] sep
                ( compressionOpt | '"' compressionOpt '"' ) ;

```

```
47 compressionOpt = "yes" | "no" ;
48
49 compressionLevel =
50 [cC][oO][mM][pP][rR][eE][sS][sS][iI][oO][nN][lL][eE][vV][eE][lL] sep
51          ( compLevel | '"' compLevel '"' ) ;
52
53 compressionLevelOpt = StringPosDec ;
54
55 hashKnownHosts =
56          [hH][aA][sS][hH][kK][nN][oO][wW][nN][hH][oO][sS][tT][sS] sep
57          ( hashKnownHostsOpt | '"' hashKnownHostsOpt '"' ) ;
58
59 hashKnownHostsOpt = "yes" | "no" ;
60
61 hostName = [hH][oO][sS][tT][nN][aA][mM][eE]
62                   sep ( hostNameOpt | '"' hostNameOpt '"' ) ;
63
64 hostNameOpt = IPv4 | IPv6 | ValidHostName ;
65
66 localForward = [lL][oO][cC][aA][lL][fF][oO][rR][wW][aA][rR][dD] sep
67                   localForwardOpt ;
68
69 localForwardOpt =
70          ( ( address ":" )? localPort WS address ":" hostPort ) |
71          ( ( IPv6 "/" )?localPort WS IPv6 "/" hostPort );
72
73 address = IPv4 | ValidHostName | "[" IPv6 "]" ;
74
75 localPort = StringPosDec+ ;
76
77 hostPort = StringPosDec+ ;
78
79 ciphers: ciphersOpt[i] != ciphersOpt[j] ;
80
81 compressionLevelOpt : compressionLevelOpt >= 1 and
82                   compressionLevelOpt <= 9 ;
83
84 localPort : localPort >= 0 and localPort <= 65535 ;
85
86 hostPort : hostPort >= 0 and hostPort <= 65535 ;
```

Description

The OpenSSH SSH client – which is the default on most Linux distributions and OS X – provides a configuration file format which can be used both at the system and per-user level. Configurations can provide global configuration (either to the entire system, or across all of a user's connection) as well as configuration tied to particular remote hosts. This specification is designed to illustrate protecting the key-value format used by SSH configurations, though it does not include support for all keywords supported by the file.

Discussion

For this specification, we assume the existence of a library which provides specification for network names. We are only interested in standard encodings for IPv4 and IPv6 addresses, along with valid host names. Such a library could, however, also provide black boxes or templates to do things like check whether IP addresses are in private ranges or verify that remote hosts are reachable or present in DNS.

SSH configuration options take the form of key-value pairs. Configuration items can apply to any connection, or be grouped into host configurations which only apply when connecting to certain machines. The configuration files also support comments, which it defines as any line which either begins with "#" or is empty. Any configuration lines appearing before the first host configuration are applied globally. All remaining configuration lines are considered part of the last host configuration. Configuration lines behave the same way regardless of whether they are global or applied to a particular host.

The SSH configuration file is case insensitive with regard to configuration keywords. To account for this, we use character classes to construct the syntax specification. This results in some long definitions, but is not otherwise particularly complicated. It is also reasonably easy to script generating such a character class.

It is worth nothing that the `ciphers` option expects that a single cypher list not contain duplicates, but duplicates are permitted between different instances of the option. (Hosts are generally independent of one another, so it would not make sense for specifying a preference for the aes256-cbc cypher on one host to make it unavailable on another.) Because we require only local, rather than global, uniqueness, we cannot use the `isUnique()` template in this case. Instead, we write a similar rule (line 79), but rely on the context to guarantee that we only consider one option at a time.

Appendix B: EBNF Maia Spec

In this appendix, we present an EBNF specification for the majority of Maia. Specifically, we omit a specification of Maia's EBNF format, and define a few items by reference to other specifications.

This specification is written using Maia's EBNF (see Section 2.4). Whitespace is not significant in Maia specifications, except to separate tokens. Thus, to aid readability, we assume that whitespace may be present between any pair of nonterminal or terminal values in this specification unless otherwise noted.

```
MaiaSpecification = inclusion*
        ( syntaxRule* setConst* semanticRule* templateDef* )* ;

inclusion = "using" qm sysPath qm sc
        | "using" qm sysPath qm "on" sysPath qm sc
        ;

sysPath = /* a system appropriate path, relative or absolute */ ;

syntaxRule = /* a Maia EBNF statement */ ;

/* set construction */

setConst = name def oa string (cm string)* ca sc
        | name def oa nVal (cm nVal)* ca sc
        | name def oa strJoin dot strJoin (dot strJoin) sc
        | name def oa numJoin mop numJoin (mop numJoin) sc
        ;

strJoin = string | setName ;
numJoin = nVal | setName;

/* semantic rules */

semanticRule = context cn constraint sc ;

context = enfLevel? "forEvery"? setName
        | enfLevel? "exists" setName
        ;

enfLevel = "(require)" | "(warn)" | "(info)" ;

constraint = comparison
        | constraint logic constraint
        | "not" constraint
        | op constraint cp
        | inclusion
        | blackbox
        ;

inclusion = indexedName "in" setName
        | indexedName "in" oa string (cm string)* ca sc
        | indexedName "in" oa nVal (cm nVal)* ca
        ;

blackbox = "blackbox" op name cm expression (cm expression)* cp ;

comparison = indexedName cmp expression
```

```
49          | expression cmd indexedName
50          | indexedName cmp string
51          | string cmp indexedName
52          | indexedName t regex
53          | indexedName nt regex
54          ;
55
56 /* Note: No whitespace is permitted around the dot */
57 indexedName = setName os expression cs (dot setName)
58          | setName
59          ;
60
61 expression = expression ops expression
62          | op expression cp
63          | nVal
64          | indexedName
65          | blackbox
66          ;
67
68 /* semantic template definitions */
69
70 templateDef = op "template" name name op (name (cm name)*)? cp cp
71          replacementText sc ;
72
73 replacementText = semanticRule | constraint ;
74
75 /* semantic template invocation */
76
77 setName name op (setName (cm setName)*)? cp ;
78
79 /* token-like definitions */
80
81 /* a nonterm, variable, template, or blackbox name */
82 name    = [a-zA-Z][a-zA-Z0-9]* ;
83 setName = name (dot name)* ; /* no whitespace around dot */
84
85 /* a string literal */
86 string = /"(\\.|[^\\"\n])*"/ | /'(\\.|[^\\'\n])*'/ ;
87
88 regex  = /* a perl-5-compatible regular expression */
89
90 iVal = /* a decimal or hexadecimal value recognized by scanf's %i */
91 fVal = /* a floating point value recognized by scanf's %g */
92 nVal = iVal | fVal ; /* a numeric constant of some sort */
93
94 /* comparison operators */
95 cmp = "==" | "!=" | "<" | "<=" | ">=" | ">" ;
96
```

```
97  /* logical connectives */
98  logic = "iff" | "implies" | "and" | "or" | "xor" ;
99
100 /* mathematical operators */
101 mop = "+" | "-" | "*" | "/" | "%" | "^" | "." ;
102
103 /* all binary operators */
104 ops = mop | dot ;
105
106 /* single characters */
107 def = "=" ;      dot = "." ;
108 cn  = ":" ;      sc  = ";" ;
109 op  = "(" ;      cp  = ")" ;
110 os  = "[" ;      cs  = "]" ;
111 oa  = "<" ;      ca  = ">" ;
112 cm  = "," ;      qm  = '"' ;
113 t   = "~" ;      nt  = "!~" ;
```

Bibliography

[1] Ext4 Disk Layout - Ext4 (and Ext2/Ext3) Wiki. `https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout`. Accessed 20 Jan 2016.

[2] *Trusted Computer System Evaluation Criteria, DoD 5200.28-STD.* Department of Defense, Computer Security Center, 1985.

[3] Anderson, R. J., Ed. *Information Hiding, First International Workshop, Cambridge, U.K., May 30 - June 1, 1996, Proceedings* (1996), vol. 1174 of *Lecture Notes in Computer Science*, Springer.

[4] Arora, A., and Gouda, M. Closure and convergence: a foundation of fault-tolerant computing. *Software Engineering, IEEE Transactions on 19*, 11 (Nov. 1993), 1015–1027.

[5] Arora, A. K. *A foundation of fault-tolerant computing.* PhD thesis, University of Texas at Austin, Austin, TX, USA, 1992.

[6] Aycock, J. Practical Earley Parsing. *The Computer Journal 45*, 6 (June 2002), 620–630.

[7] Backus, J. W. 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* (1959), 125–132.

[8] Backus, J. W., Wegstein, J. H., van Wijngaarden, A., Woodger, M., Nauer, P., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., and Vauquois, B. Revised report on the algorithm language ALGOL 60. *Communications of the ACM 6*, 1 (Jan. 1963), 1–17.

[9] Badger, L., Sterne, D. F., Sherman, D. L., Walker, K. M., and Haghighat, S. A. Practical Domain and Type Enforcement for UNIX. In *Security and Privacy, 1995. Proceedings., 1995 IEEE Symposium on* (1995), pp. 66–77.

[10] Badger, L., Sterne, D. F., Sherman, D. L., Walker, K. M., and Haghighat, S. A. A domain and type enforcement UNIX prototype. *Computing Systems 9*, 1 (1996), 47–83.

[11] Bauer, M. Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal 2006*, 148 (Aug. 2006), 13.

[12] Bauer, M. Paranoid penguin: security features in SUSE 10.0. *Linux Journal 2006*, 144 (Apr. 2006), 12.

[13] Biba, K. J. Integrity Considerations for Secure Computer Systems. *MITRE Corporation*, Technical Report MTR-3153 (Apr. 1977).

[14] Bishop, M. A. *The Art and Science of Computer Security.* Addison-Wesley Longman Publishing Co., Inc., Dec. 2002.

[15] Clark, D. D., and Wilson, D. R. A Comparison of Commercial and Military Computer Security Policies. *1987 IEEE Symposium on Security and Privacy 0* (Apr. 1987), 184.

[16] Codd, E. F. A relational model of data for large shared data banks. *Communications of the ACM 13*, 6 (June 1970), 377–387.

[17] Crispin, M. Internet Message Access Protocol - Version 4rev1. Tech. Rep. RFC 3501, Mar. 2003.

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

[19] Earley, J. An efficient context-free parsing algorithm. *Communications of the ACM 13*, 2 (Feb. 1970), 94–102.

[20] Ferraiolo, D. F., and Kuhn, D. R. Role-Based Access Controls. *15th National Computer Security Conference* (Oct. 1992), 554–563.

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

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

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

[24] Franz, E., Jerichow, A., Möller, S., Pfitzmann, A., and Stierand, I. Computer based steganography: How it works and why therefore any restrictions on cryptography are nonsense, at best. In Anderson [3], pp. 7–21.

[25] Giampaolo, D. *Practical File System Design with the Be File System*, 1 ed. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1999.

[26] Harada, T., and Handa, T. TOMOYO Linux: A Lightweight and Manageable Security System for PC and Embedded Linux. In *Proceedings of Ottawa Linux Symposium* (2007), pp. 27–30.

[27] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (Oct. 1969), 576–580.

[28] IEEE Computer Society. IEEE Std 754-2008 - IEEE Standard for Floating-Point Arithmetic. IEEE, Aug. 2008.

[29] ISO. Information technology - Syntactic metalanguage - Extended BNF. Tech. Rep. ISO/IEC 14977:1996(E), Geneva, Switzerland, Dec. 1996.

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

[31] Ji, Q., Qing, S., and He, Y. A formal model for integrity protection based on DTE technique. *Science in China Series F: Information Sciences 49*, 5 (Oct. 2006), 545–565.

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

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

[34] Kahn, D. The history of steganography. In Anderson [3], pp. 1–5.

[35] Kegler, J. Marpa, A Practical General Parser: The Recognizer, June 2013.

[36] Knuth, D. E. Backus Normal Form vs. Backus Naur form. *Communications of the ACM 7*, 12 (Dec. 1964), 735–736.

[37] Leo, J. M. I. M. A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead. *Theoretical Computer Science 82*, 1 (May 1991), 165–176.

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

[39] Linux Programmers Manual. crypt(3). `http://man7.org/linux/man-pages/man3/crypt.3.html`. Accessed: 5 Feb 2013.

[40] Linux Programmers Manual. gpasswd(1). `http://linux.die.net/man/1/gpasswd`. Accessed: 22 Feg 2016.

[41] Linux Programmers Manual. group(5). `http://man7.org/linux/man-pages/man5/group.5.html`. Accessed: 5 Feb 2013.

[42] Linux Programmers Manual. newgrp(1). `http://linux.die.net/man/1/newgrp`. Accessed: 22 Feg 2016.

[43] Linux Programmers Manual. passwd(5). `http://man7.org/linux/man-pages/man5/passwd.5.html`. Accessed: 5 Feb 2013.

[44] Linux Programmers Manual. shadow(5). `http://linux.die.net/man/5/shadow`. Accessed: 5 Feb 2013.

[45] Linux Programmer's Manual. ssh_config(5). `http://linux.die.net/man/5/ssh_config`. Accessed 19 Jan 2014.

[46] Loscocco, P., and Smalley, S. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 ...* (2001).

[47] Manna, Z., and Pnueli, A. Axiomatic approach to total correctness of programs. *Acta Informatica 3*, 3 (1974), 243–263.

[48] Microsoft TechNet. Controlling Access to Files and Folders. `http://technet.microsoft.com/en-us/library/cc938434.aspx`. Accessed: 28 Feb 2013.

[49] Miller, R. B. Response time in man-computer conversational transactions. In *the December 9-11, 1968, fall joint computer conference, part I* (New York, New York, USA, 1968), ACM Press, pp. 267–277.

[50] Muthukumaran, D., Rueda, S., Talele, N., Vijayakumar, H., Teutsch, J., and Jaeger, T. Transforming commodity security policies to enforce Clark-Wilson integrity. In *the 28th Annual Computer Security Applications Conference* (New York, New York, USA, 2012), ACM Press, p. 269.

[51] Nemeth, E., Snyder, G., Seebass, S., and Hein, T. *UNIX System Administration Handbook*, third ed. Prentice Hall, Aug. 2000.

[52] Parr, T. *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, LLC, Jan. 2013.

[53] Perl Programming Documentation. perlre - Perl regular expressions. `http://perldoc.perl.org/perlre.html`. Accessed 20 Jan 2016.

[54] Polk, W. T. Approximating Clark-Wilson "Access Triples" with Basic UNIX Controls. In *UNIX Security Symposium IV Proceedings* (Oct. 1993), pp. 145–154.

[55] Potter, M. C., Wyble, B., Hagmann, C. E., and McCourt, E. S. Detecting meaning in RSVP at 13 ms per picture. *Attention, Perception, & Psychophysics 76*, 2 (Dec. 2013), 270–279.

[56] Rizzo, Thomas. WinFS 101: Introducing the New Windows File System. `http://msdn.microsoft.com/en-US/library/aa480687.aspx`. Published: 17 March 2004, Accessed: 25 Feb 2013.

[57] Schaufler, C. *The simplified mandatory access control kernel*. White Paper, 2008.

[58] Schmidt, D. A. *Denotational Semantics - A Methodology for Language Development*. 1997.

[59] Smalley, S., Vance, C., and Salamon, W. Implementing SELinux as a Linux security module. *NAI Labs Report 1* (2001), 43.

[60] Tennent, R. D. The denotational semantics of programming languages. *Communications of the ACM 19*, 8 (Aug. 1976), 437–453.

[61] Tolkien, J.R.R. *The Silmarillion*. George Allen & Unwin, 1977.

[62] Torvalds, Linus. Linux Kernel version 3.18rc2. `https://www.kernel.org`.

[63] Trithemius, J. *Steganographia*. Jan. 1499.

[64] Türker, C., and Gertz, M. Semantic integrity support in SQL: 1999 and commercial (object-) relational database management systems. *The VLDB Journal 10*, 4 (2001), 241–269.

[65] van der Vlist, E. *RELAX NG*. O'Reilly Media, Dec. 2003.

[66] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition). Tech. rep., Oct. 2008.

[67] W3C. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. Tech. rep., Apr. 2012.

[68] W3C. W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. Tech. rep., Apr. 2012.

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

[70] Wright, C., Cowan, C., Morris, J., Smalley, S., and Kroah-Hartman, G. Linux Security Modules: General Security Support for the Linux Kernel. *USENIX Security Symposium* (Aug. 2002).