Toward a General, Implementable Framework for Integrity Protection on Arbitrary Files

Paul J. Bonamy
Michigan Technological University, Houghton, MI 49931 USA
{pjb}@mtu.edu

Abstract. We propose, as dissertation work, the creation of a system for protecting the integrity of arbitrary files present on a (Linux) computer. The proposed work will consist of three major components: creation of a language for describing integrity constraints, implementation of a system which can enforce that language, and construction of integrity rules to demonstrate the utility of that system in a practical application. We also discuss background requisite to the topic and demonstrate its general desirability.

1 Introduction

Integrity of data within computer systems, along with the ongoing confidentiality and accessibility of said data, make up the three major components of computer security. However, while both confidentiality and accessibility are subject to frequent and ongoing study, integrity is not, in general, discussed in terms of computer systems in any great detail. 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.

We propose to rectify this lack of general-purpose integrity protection by designing a language to describe integrity constraints for arbitrary files. We further propose to implement a system which can interpret the language and enforce it at runtime, and finally deploy both within a running computer system. In this way, we can demonstrate both theoretical safeguards for system integrity and the practical implications of putting those safeguards into effect.

The remainder of this document is structured as follows: We present some general background on computer security and language design in Section 2 followed by a discussion of related research in Section 3 and the deficiencies of same in Section 4. We will then present an outline of our proposed work in Section 5. Appendix A presents an illustrative set of sample integrity rules.

2 Background

2.1 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.

Typically, computer security is broken down into three inter-related aspects: confidentiality, availability, and integrity. (Bishop[1], provides a reasonable, though not definitive, overview of these as well.) These aspects tend to get a bit blurry around the edges, with technologies in one area providing 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.

2.2 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[2], and consequently early Windows, through the rather coarse but ubiquitous Unix model[3], to the incredibly fine-grained DTE model[4]. Placing access terminals in physically secured locations, with guards and lists of permitted personnel, would also serve 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 also 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[5][6][7], 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.

2.3 Accessibility

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. Accessibility 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 accessibility, 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 fiber lines) 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.

Notions of correctness and fault tolerance can also help with accessibility. 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 accessibility. 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.

2.4 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 straightforward 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 sort of 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 determine whether an accounting ledger is correctly formed, but determining whether the transactions it represents are correct is generally beyond the scope of a piece of software. Further, it is possible for the data to be well formed but wrong, even if we believe we can trust the data’s origin.

Two general classes of integrity systems exist for handling data integrity. Prevention systems are geared toward preventing 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 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, perform 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 accounts were 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 proposed dissertation work aims at correcting this by providing a generally applicable system to detect and prevent modifications which would compromise data integrity.

2.5 Mandatory and Discretionary Systems

The Department of Defense’s Trusted Computer System Evaluation Criteria 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 real 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 assure 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). Note that mandatory and discretionary controls can be employed 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.

2.6 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 [9]. The metalanguage used in that case has since been standardized into Backus-Naur Form (BNF)[10][11], 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 rather larger problem, about which entire books have been written without covering all of the possibilities. For example, Schmidt[12] 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 the aforementioned denotational semantics.

The simplest possible way to specify the semantics of a language is to implement an interpreter for it. This gives a direct mapping from language constructs to meaning by converting the language into executable code, and provides what is known as operational semantics. While operational semantics are convenient for testing language designs, they present problems when a language is to be 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[13] 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 interpreter 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 interpreter 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.
3 Related Work

3.1 Biba Integrity Model

Biba’s integrity model\cite{14} 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 so-called 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 assures 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 assures 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.

3.2 Clark-Wilson Integrity Model

David Clark and David Wilson\cite{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, and systems, 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 the system ensures that only TPs are allowed to manipulate CDIs, and we can assume the system was in a valid state at some time in the past (because the IVPs were successfully run), 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, for that matter) 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 though 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 must 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 about. 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 of 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.

3.3 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 (read: tables), 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 [17] provide 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.

3.4 Access Control Systems

As discussed in the background section, access control systems provide a first line of defense for both conﬁdi-
entiality 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) system speciﬁcally to address
these difﬁculties. They proposed 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, the person in
charge of administering a departmental website needs full access to all of its ﬁles, but that access needs to be
transferred when someone else takes over as webmaster. Ferraiolo, et al., propose a role-based access control (RBAC) system to account for this[19]. 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, doesn’t introduce a lot of extra effort during a normal check, and makes updating role-based permissions trivially easy.

In 2001, the National Security Agency started presenting it’s Security Enhanced Linux project.[20] SELinux, which has been integrated into Linux kernel since the the kernel’s 2.6 released 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 whole host of other functionality that can be used to attain incredibly fine-grained access control policies.

3.5 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[21], and approaches relying on the fine-grained customizability of DTE[22]. These systems can generally provide excellent origin integrity, though enforcing data integrity normally requires modification to the existing access control framework.

3.6 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[23] 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. Officially this is to allow finer-grained access control checks, though it potentially also offers an opportunity to process file integrity every time data is written.

While the security module system is designed to allow a module to work properly with programs that are unaware of the existence of the security system, it also provides a system call to allow security-aware programs to interact with a running module directly. This provides a number of interesting opportunities, including the ability to request information from the security system at runtime, independent of normal module operation.

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.[21]

3.7 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 the like. 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[25] [26]. Closure requires that a program starting at a valid state will remain in valid states as it operates. This effectively means that the program isn’t going to 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.

3.8 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 so-called 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[28] 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.

4 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.

4.1 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 origins 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 passwd file. To comply with Clark-Wilson, the editor would have to be certified to always produce valid HTML, and to always produce a valid passwd 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 passwd 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 be impossible, or at the very least deeply impractical. 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, and 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, the 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[29], or Microsoft’s to-be-released-someday WinFS[30]), 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 along arbitrary files.

4.2 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. A person intent on embezzling money can produce transactions that look correct, after all, 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, of course, 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 configuration. 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.

4.3 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, as we discussed above, 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 fix the data being corrupt, or necessarily start the process of fixing the data. Fault tolerance and integrity work together well, but neither is really capable of serving for the other.
5 Proposed Project

We propose the creation of a general-purpose, mandatory data integrity protection system for Linux. This system is patterned after the Clark-Wilson model, with some significant differences we feel will make general implementability feasible. We retain the notion of constrained and unconstrained data items, as this permits gradual adoption of our system. However, we propose to generalize integrity verification procedures into a single, OS-level subsystem that uses descriptions of data formats for CDIs to perform verification on demand. We further propose to eliminate the need for certified transformation procedures by revalidating constrained files any time they are modified, regardless of the modifying process. In this way we can maintain a trusted data store even if we do not entirely trust the processes working on the data. We feel that these modifications will allow us to express the strengths of Clark-Wilson without running into the implementational pitfalls described above.

We conceive of this dissertation project as three distinct but interrelated components, which serve as stepping stones toward a working system. First, we propose developing a language with which to describe integrity rules for constrained files. Once we have a language prototype, we will build a runtime system capable of monitoring accesses to protected files and revalidating the files according to our integrity rules. Finally, we will perform extensive testing to profile the runtime characteristics of our system and demonstrate that it can provide protection in a production environment. These components end up providing us with strong theory-, implementation-, and experimentation-related research to perform. We discuss each of these components in detail after a brief description of the Unix passwd system, which serves as a motivational example.

5.1 Unix and the passwd File

Operating systems which support having multiple users and restricting access to resources based on the identity of the user must have a facility to store login credentials and related information. Historically, Unix has employed a pair of plain text files, named passwd and group, to store, respectively, information mapping a username/password to a (potentially) unique user identifier (UID), default group membership, etc, and a listing of all user groups on the system, along with the membership of the groups. These files are arranged with a single line, containing colon-separated fields, per user or group, and have, historically, been world-readable to allow things like UID to username conversion by user processes such as ls.

The Linux passwd file structures its entries as follows[31]:

```
account:password:UID:GID:GECOS:directory:shell
```

with individual fields corresponding, respectively, to user name, a hash of the user’s password, a numeric user ID, the numeric ID of the user’s primary group, an optional field which generally contains the user’s ‘real’ name (e.g. “Paul J Bonamy” rather than “pjbonamy”), the user’s home directory, and the shell to spawn on login. BSD-derived platforms use a slightly different format[32], but the general idea holds.

The group file is conceptually similar, using the following format[33]:

```
group_name:password:GID:user_list
```

where we have the name of the group, the hash of an optional password required to switch to the group, the group’s unique identifier, and a comma-separated list of usernames indicating the membership of the group. Generally speaking, usernames in the user_list must correspond to entries in the passwd file, and a user must be a member of their default group.

As computers have gotten faster, storing password hashes in the main passwd file has gone from a non-issue to a potential security vulnerability if an attacker is able to brute-force or otherwise break the hash, revealing user passwords. Because many standard programs assume passwd will be readable, and rely on that to provide functionality, only allowing access by the login system is infeasible. Instead, the login system has been modified to allow for storing actual passwords elsewhere, in an access-limited file named shadow.
This way, processes that need it can still access the passwd file, but the password hashes are hidden from everything but the login system.

The shadow file\cite{shadow} shares the colon-separated field layout of the passwd and group files. Each record contains nine fields, the first two of which specify a username and password, with the remaining seven providing optional controls for password changing or expiration policies.

Linux machines also support a variety of other user authentication schemes, including relying on a central server to provide the requisite data. However, the classic passwd file and its associates are still commonly found on individual installs, and are often present even if another authentication scheme is used to provide for a local root account. The ubiquity of the passwd file, combined with its straight-forward structure and general importance, make it an excellent motivating example for the remainder of our discussion.

### 5.2 Theory - A Description Language for Integrity Protection

The first step in maintaining the integrity of a file is being able to describe what a valid file should look like. Without a solid, decidable description, there is no way to know whether file integrity has been violated, making actual enforcement impossible. That being the case, we propose the creation of a language for describing a valid file and denoting how the file may change over time. This language is, within our system, analogous to the Domain and Type Enforcement Language, in that it is intended to provide a concise, human-friendly means of expressing rules which govern the system.

While we do not yet have an exhaustive list of the features the description language must possess, we have established a number of features which appear desirable in order to protect things like the passwd files. That selection of features includes:

**Application Independence (aka Black Box Principle)** Unlike Clark-Wilson’s model, our language should not include bindings to specific programs when determining whether integrity has been violated. Treating applications as black boxes that modify system state, and only concerning ourselves with the way state is changed, allows integrity to be preserved without foreseeing all possible valid means of changing a file. For example, there are utilities to manage the passwd file, but an admin might decide to write new scripts to perform specific transformations. Restricting this sort of behavior will only serve to annoy users, and possibly force them to circumvent the integrity system. Treating programs as black boxes allows our system to work even in the face of changing access models. Existing access control mechanisms like DTE or SELinux can restrict process access rights, if administrators wish to impose such restrictions.

**Description, not procedure** The language should be structured to provide a mechanism for describing a valid file, rather than a procedure for validating a file according to its description. This allows for greater clarity in the description, and helps prevent inadvertent platform dependence in the rules. It is also far easier to utilize and trust a human-readable description of a file than some opaque blob of executable code. As a side benefit, our description language could also be used to power stand-alone validators for discretionary, on-demand integrity checking.

**Provable semantics** We wish to provide denotational semantics for our description language. This allows us to directly prove features of the language, and should also aid in constructing, and reasoning about, integrity rules. Being able to show that a rule set will always result in a correctly validated file would be a major benefit to our system. Providing robust semantics independent of our implementation of the enforcement system also makes implementing other validators based on our language easier, and will hopefully pave the way to adoption in other contexts.

In the event that providing fully denotational semantics proves to be infeasible, we will still provide denotational semantics wherever possible and fill in the remaining semantics operationally. This provides a complete set of semantics to support independent implementation, even if it reduces provability.
Support for varying file types There is no standard structure for files which we may wish to place under integrity protection. Even within a single general style of file, there may be significant differences in implementation. Therefore, the description language must be able to express rules for a variety of file structures, as well as being able to cope with binary and text files. Minimally, we wish to support record-based files such as the passwd file. Well-structured single record files – like images – or key-value-pair-based files – such as the ssh configuration file – are also likely targets.

Ability to process files by record Record-based files like the passwd file can be reasonably thought of as consisting of many separate records which are all bound by the same set of restrictions. That being the case, it makes sense to be able to define what a record looks like, and what is required for each of its fields to be valid, and then allow the system to verify that all records match the definition. This would likely consist of a specification for how to separate records (e.g., they are newline delimited), how to identify fields within a record (they are colon separated), and then individual rules for determining field validity, singly or relative to one another. These record-centric rules correspond to Tü rker and Gertz’ row constraints.

It is important that record-based rules be able to handle cases where the validity of a single record depends in part on the combined state of two different fields in the same record. For example, it is technically permitted for any user on a Linux system to have UID 0, even though that UID is typically reserved for the root user and carries enhanced privileges. An administrator might wish, however, to add a rule like “only root may have UID 0” specifically to prevent non-root users being able to run with root privileges.

Collective rules across records In some multi-record files, records are assumed to be completely independent of one another, as with log entries in the wtmp system log. However, examples such as the passwd file can also be found where it is desirable to express constraints between records, as well as within them. This could, for example, be used to enforce a rule that all users must have unique usernames. Tü rker and Gertz refer to this sort of rule as a table constraint[17].

Support for linking multiple (record-based) files A single logical set of data may be physically stored over multiple files. In that case, there must be a way to specify integrity constraints not only within the individual files but also across files. For example, there must be a way to express the fact that all group IDs in the passwd file must also appear in the group file. This is loosely analogous to the notion of referential integrity in a relational database (or to Tü rker and Gertz’ inter-table constraints[17]), and is a natural extension of supporting files containing multiple records.

Field Naming Fields in record-based files tend to have human-friendly names, either provided by a structure definition for binary files or convention for text-based files. The description language should allow rule sets to name fields and then refer to the fields by name both within the specification and as part of failure logging. This way, users are freed from the need to remember the absolute ordering of fields, making rule authoring less error-prone. Debugging and error logs can also return more helpful information.

Reusable validation rules / components It should be possible to construct validation primitives that can then be reused to construct larger rules. For example, phone numbers might be present in many places within a record or file. Rather than needing to maintain an independent set of rules for each, it is sensible to define a phone number once, and reuse that definition. This reduces the number of places implementation errors may occur, and makes modifying what constitutes a valid phone number easier. The field naming system should be able to be extended to provide useful names for component rules.

A (small) Collection of Primitive Types Given that we wish to support reusable validation rules anyway, it makes sense to provide a small set of existing primitives to make rule construction easier. We propose including primitives for, minimally, Integers, Real (i.e., floating-point) numbers, and Strings. By explicitly
supporting numeric types, we both make certain sorts of recognition easier, and provide the ability to use mathematical comparisons in rules. For example, we can say “UID >= 0”, rather than needing to develop a regular expression or similar to recognize only positive integers. It may also be beneficial to recognize the key-value pair as a primitive type.

Layering of Requirements It is infeasible to try and specify integrity rules for an entire system at once. Instead, we propose allowing rules to be added over time, with new rules providing additional integrity constraints. This way, a system can be brought online with a “permit everything” default rule, with additional rules adding integrity constraints for specific file types as needs.

An additional benefit of a layering approach is that multiple sets of rules could be applied to the same resources. This way the set of rules which specify what constitutes a valid passwd file set from the perspective of the login system could be supplemented by organization-specific rules with additional constraints. Users could also add supplemental, discretionary rules to enforce requirements on their files without needing to involve an administrator.

Ability to control relaxing of requirements In many (most?) cases, successive layers of integrity rules should serve to further restrict what counts as ‘valid’. It would, after all, make no sense to allow an administrator to specify passwd records the login system cannot parse. However, in some cases it may be desirable to allow relaxing a restriction, as, for example, in the case of a transitional section of a website that cannot yet pass strict validation. It may be worth investigating a system for relaxing integrity constraints, though this may be a feature better left to a later revision of the description language, when it is more thoroughly understood.

File-class-based binding of integrity rules Certain file types, like images, appear throughout the filesystem. Rather than trying to specify integrity rules for these files in all of the places they may occur, it makes more sense to provide a mechanism that allows the integrity system to match files to integrity rules wherever they occur. This also helps support the black-box principle by allowing accesses by any process to be constrained by the same rules.

Selectable behavior on failure Detecting an integrity violation does no good without a mechanism for responding to the violation. In some cases, as with the passwd file, violations should be strictly disallowed, with any changes that would cause a problem rolled back to prevent harm. However, in other circumstances it may be desirable to allow the change to take place, but annotate the modification or trigger secondary actions. For example, a programmer may make an incremental change to a source file which leaves the file uncompilable, but this change should still be allowed as they are likely to fix the problem later. As with relaxation, a sensible default for failure state should be selected, with a mechanism to allow deviating from that default as necessary.

Reporting of updates, violations Logging is an integral part in many integrity systems, including Clark-Wilson, and provides a way of tracking, and responses to, changes as the system is used. Integrity rules should be able to specify which sorts of actions result in logging, what to log, and, to some extent, control where that log is maintained. For example, modifications to the passwd file should be recorded, providing a record of how, and by whom, the file changed over time. Failed changes should also be noted, to permit debugging applications or identifying potential attacks.

Modifications to files under administrator control (e.g. the passwd file) should likely be logged in a system log. However, admin logs should not be swamped by entries about user-controlled files, so a mechanism to partition the logs is indicated. Combined with layering, this could also allow users to specify personal logging locations in addition to the system log when dealing with files they can reasonably control.
5.3 Implementation - Runtime Enforcement System

Description languages are interesting from a theoretical perspective, but it is entirely possible to construct a language that cannot be meaningfully implemented. This could be due to ambiguity in the language specification, or general non-decidability. For that reason, in addition to creating a language for describing integrity protections, we also propose implementing a system which can enforce rules specified in our language. Here are some of the design principles we feel will benefit the system:

Plain Text Configuration Storage Historically, Unix operating systems prefer storing configuration data in readily editable plain text files, rather than relying on user-unfriendly binary files. We prefer to continue this tradition and employ a plain text data storage format if at all possible. This will make it easier to author rulesets, and allow maximum flexibility for administrators. The use of a compiled, binary representation internally would not conflict with this principle.

Possibility of General Adoption We wish to build our enforcement system so that its general adoption is feasible. Such adoption is, of course, generally beyond our control, but we feel that constructing the system to permit a wide range of users to adopt it increases its overall worth. This will likely require a modular build, as well as the ability to bring the system online without requiring integrity specifications for the entire system up front.

Development for Modern Linux While it would certainly be possible to implement this integrity system on one of the research-focused operating systems, we prefer to target a modern Linux kernel for two reasons. First, targeting Linux makes general adoption much more likely than it would be for a research kernel. More significantly, using a mature platform with significant third-party software will allow for much more robust compatibility testing, and will hopefully reveal any unanticipated edge cases. For the moment, the 2.6.x family of kernels is a likely target, as it is well represented in the major distributions, but a 3.x kernel is not out of the question.

Minimally Invasive Implementation We could implement this system by extensively modifying the kernel itself to support our new functionality. This strategy, however, makes it more likely that implementation will break existing functionality and limits the potential for widespread adoption. Instead, we intend to target the existing systems for adding kernel functionality, which will both reduce the chances of breaking the kernel and make integration into other systems easier.

The Linux Security Modules interface already provides a suite of hooks for trapping accesses to files. Of particular use for us, LSM has hooks for adding policy enforcement to file open and write, and has a call-back invoked when a file is closed. When a file is opened for writing, we can immediately check to see if it should be integrity controlled. If so, we add the requisite metadata to LSM’s security field so our system will be able to track the file. If the particular file format supports it, we can then run some validation after individual writes or sets of write operations. We can then perform final validation once the file is closed, as part of the call-back provided by LSM. This way we’re certain that the file is in a completed state and ready to check.

Transparent Operation The integrity control system must continue to enforce integrity rules even if the software acting on the system is unaware of integrity control. Without this mandatory inclusion in integrity control, there can be no guarantee that a file which should be protected is still valid if a non-aware program accesses it. To this end, we wish to construct our system to transparently hook existing system calls, rather than require new, integrity-aware versions. It is conceivable that software which is aware of the integrity system could interact with it directly, but we wish to avoid making protection contingent on awareness.
Minimal Performance Impact  Given that the integrity system will need to validate the contents of integrity-controlled files, some impact on performance is inevitable. However, we wish to minimize that impact to the greatest extent possible, as a system which safeguards integrity is useless if no one is willing to use it. Specifically, we wish to develop the system to minimize the system resources that must be dedicated to integrity protection.

Reasonable (Re-)Validation Timing  We must guarantee that any file which is supposed to be under integrity control will be valid when a process attempts to access it. In all likelihood, this will require a final integrity check when the file is closed following writing. However, we would also like to provide support for piecemeal validation as the file is modified, so we can respond to malformed data as quickly as possible.

Concurrent Access Awareness  There are cases where multiple processes rely on simultaneous access to shared files to transfer state data. In the event that our system is called upon to enforce integrity on those shared files, we must be able to handle simultaneous access in a consistent, if not precisely optimal, fashion.

Careful Handling of File Sets  File sets introduce special considerations for validating interrelated files. For example, creating a new user, and corresponding group, within the passwd set would require creating the group, creating the user, and then assigning the user to the group’s user_list. If these operations are not done in precisely the right order, an overly strict integrity system might disallow individual modifications even though the overall change would be valid once completed. Therefore, we need to provide a mechanism that will allow file set integrity to be enforced meaningfully without making actually modifying the file set unduly difficult.

5.4 Experimental - Practical System Protection

Generating a language to describe integrity rules, and a runtime system to enforce the rules, is all well and good, but on their own neither of these demonstrate that the pair can actually be used to provide real-world protection. Specifically, we must also demonstrate that the description language can be used to craft rulesets that correctly specify real file types, and that the runtime system both enforces those rules correctly and does so in a way that minimally impacts correct operation of the computer system as a whole. In order to demonstrate these requirements, we propose a third component of this project: protecting a real system from integrity failures.

The best way to test a description language is by using it to actually describe things. If we can successfully describe the format of every file type we test, we can be reasonably sure that our language is a good start for a general purpose description language. (Assuming, of course, that we have a reasonably representative set of files types.) If, on the other hand, we discover file types that cannot be well described, we not only know that the language is lacking, but also gain some insight into where the language needs improvement.

Actual file type selection is a problem for a later date, but we wish to select a wide variety of record-based (and potentially key-value) file types to describe. This will almost certainly include one-off system configuration files like the passwd file, where file validity is essential to the system remaining functional. We also intend to describe some file types more commonly used by user processes. While user-space files aren’t as critical to overall stability (and security) as their system counterparts, their corruption will be more obvious to users, and likely result in significant loss of productivity. For this reason we feel that providing good integrity protection to both system and user files is essential.

Selecting a wide variety of files also provides plenty of opportunity to ensure that the runtime system can properly interpose itself into normal operations to actually enforce the rules without causing undesirable side effects. Failures will likely take one of two forms: failure to maintain integrity in files that should be protected, and interference with files that shouldn’t be under integrity control. Relying too heavily on a single file type, or small set thereof, for testing purposes could mask either type of problem. If certain file types are accessed in an unusual fashion, the runtime system may not be able to protect them, or may interpret
accesses incorrectly. This would be invisible if those types are not included in the testing set. Likewise, if protected files are restricted to locations well away from their unprotected counterparts, we are less likely to notice interference in normal operations.

As with the description language, finding cases where the runtime system can’t provide adequate protection (or is overzealous in ‘protecting’ things) indicates flaws in our system. We must be able to verify that our runtime system will interpose itself in every case where it is required, but never interfere with unconstrained files. If this first requirement is not met, the system does not provide fully mandatory protection. A failure in the latter case will end up annoying users, reducing general acceptance of the software.

We intend to run language and runtime testing in a variety of environments. Initial testing will be run on our development machine, which will enable us to identify and correct major flaws without unduly impacting other responsibilities. Once the system has proven to be fairly reliable in isolation, we will set it up on a computer which sees frequent, single-user activity. This way, we can conduct more comprehensive testing in the presence of real-world use cases including teaching duties and grading. Ideally, we would like to run large-scale tests in one of the publicly accessible computer labs once our system has matured. This would, however, require support from lab administrators, and may prove to be impractical. Testing in a production server environment would also be desirable, though it encounters similar potential restrictions.

There is a third potential failure case for our proposed integrity system: it could operate correctly, but degrade overall performance to such an extent that the computer becomes unusable. This is a potentially worse failure than being insufficiently expressive or not offering protection in all cases, because at least in those cases we gain some utility from the use of the system. Basic testing will reveal glaring inefficiencies, but cannot quantify actual performance changes caused by our system.

In light of this, we propose one final set of experiments, wherein we will quantify the impact our software has on system performance. This will take the form of running timing experiments in three configurations: runtime system disabled, runtime system enabled but not active, and runtime system actively enforcing rules. An active runtime system cannot help but impose at least some performance penalty on a running process accessing constrained files - it must, by necessity, perform validity checks on the files, which cannot be a zero-cost operation. By comparing performance when the system is disabled, idle, and active, we can assess how much of a penalty is being introduced, and whether it comes from the simple existence of the runtime system, or specifically from enforcing rules. We may also be able to gain insights into where to introduce optimizations, allowing us to further reduce the impact of using our system.

It is our hope that by attempting to apply our integrity system to a real computer system, and then rigorously testing how it affects that system, we can demonstrate that our system provides real integrity protection in a way that could be widely adopted. Even if we discover that our system cannot function correctly in all cases, we could at least provide a reasonable understanding of when it works and how it fails, which will provide an opportunity to guide further research in this field.

5.5 Timeline

Research and software development are both subject to unforeseen delays. Even so, it helps to have timeline for completing the project, even if that timeline will be subject to change. Here is a sketch of our intended development timeline:

<table>
<thead>
<tr>
<th>Timeframe</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remainder of this semester</td>
<td>Set up development machine, initial testing with LSM</td>
</tr>
<tr>
<td>Summer 2013</td>
<td>Major language design, construction of stand-alone testing verifier</td>
</tr>
<tr>
<td>Fall 2013</td>
<td>Construction of integrity control module, refinements to language design</td>
</tr>
<tr>
<td>Spring 2014</td>
<td>Ongoing development work, testing in single-user production</td>
</tr>
<tr>
<td>Summer/Fall 2014</td>
<td>Performance testing, testing in production lab / server environment</td>
</tr>
</tbody>
</table>
6 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.

We propose a dissertation project in three parts to address the lack of general-purpose integrity protection. First, we propose the creation of a special purpose language which can be used to express integrity constraints on arbitrary files. We then propose to implement a runtime system which can enforce constraints described in our language. Finally, we will extensively test our runtime system to ensure that it can provide robust protect for system integrity without sacrificing system performance or usability. We feel that this project will aid system security by helping to ensure that integrity of system resources is maintained.
A Sample Integrity Rules

This section contains a number of sample validity rules for a variety of files and use cases. It is intended to provide a concrete example of the sorts of thing we wish the description language to cover, so the examples are written to be illustrative, rather than being a precise representation of our eventual syntax.

To demonstrate the protection of (groups of) record-based file, we provide rules for each of the files making up the passwd set (/etc/passwd, /etc/group, and /etc/shadow), along with file-set rules for checking relationships between the files. We also present examples of restricting an existing ruleset, and a sample ruleset for validating key-value-based files, specifically the per-user SSH configuration file.

At the highest level, our descriptions provide a grammar for parsing the structure of a file, starting from the file level and working down to individual components. We employ a lightly modified version of Backus Naur Form to express that grammar, but switch to regular expressions in cases where that makes the representation more compact without affecting readability. In addition to BNF-like grammar descriptions, we also provide additional information and constraints. This take the form of data types and additional rules, which make expressing certain constraints easier, and human-friendly field descriptions which are intended to make the rules more accessible and also aid debugging in the event of a validation failure.

Throughout the rules, we use indentation to indication grouping or ownership for properties. Thus, we when we define the new shared type Crypt_Password (on the next page), everything indented one level from that definition either describes the Crypt_Password (as with its type, description, or rules), or is a subtype of the Crypt_Password (like the Crypt_Password_ID). Definitions will generally have a type, a description, and a set of rules, though these are optional. Some types of definition gain additional data, which will be described as it arises.

Each of the following examples is preceded by a brief description, along with general comments on assumptions we make in the course of the example. We have also added the occasional comment (indicated by C-style /* */ notation) within the examples themselves, where we felt additional clarification was indicated.
A.1 The Crypt_Password Type

Passwords used by the passwd set are all generated by the crypt() function. The output of this function has a well-defined structure[35] that allows validation of both its overall structure and its components. This, then, is a reusable type that can be used throughout the passwd set (and elsewhere), and includes extra checking to make sure hash lengths are appropriate for the selected algorithm.

The intention with rules blocks is that a type must satisfy all of its rules in order to be considered valid. Rules may include a BNF-like description, one or more testable statements, or both. We use ::= to indicate a BNF-like description, where the name of the type is assumed to appear on the lhs. A BNF description provides for validation of a type’s structure and the structure of all subcomponents.

We use two forms of testable statement in this example: the implication and the match. An implication (indicated with => in the middle of the rule) is meant to be interpreted as it would in boolean logic, so the test succeeds if both the left and right sides are true, or if the left side is false. A match is used to provide a regular expression which must recognize the string in question. The match could, strictly speaking, be replaced by one or more BNF rules, but a regular expression is more compact.

shared_type: Crypt_Password
  type: String
  desc: the cryptographic hash of a password, as produced by crypt(3)
  rules:
    ::= <Crypt_Password_ID>$<Crypt_Password_Salt>$<Crypt_Password_Hash>
    Crypt_Password_ID == "1" => Crypt_Password_Hash.length = 22
    Crypt_Password_ID == "5" => Crypt_Password_Hash.length = 43
    Crypt_Password_ID == "6" => Crypt_Password_Hash.length = 86

Crypt_Password_ID
  type: String
  desc: ID of hashing algorithm
  rules:
    ::= 1 | 2a | 5 | 6

Crypt_Password_Salt
  type: String
  desc: Salt used in hash generation
  rules:
    match: /[a-zA-Z0-9./]*/
    length: <= 16

Crypt_Password_Hash
  type: String
  desc: Actual cryptographic hash of password
  rules:
    match: /[a-zA-Z0-9./]*/
A.2 The Passwd/File Type

This type is specifically for validating the `passwd` file itself, and provides a good example of validation rules for a plain-text, record-based file. The `Text_Record` type is used to indicate to the system that the file will be composed of text-based records. We bind this file type to the file `/etc/passwd`, which is likely the only file that uses this format, and indicate that we wish to use the `Crypt_Password` type as part of our definition. The `record_type` parameter allows us to identify which subtype will be used to indicate a complete record, will be used for validating file sets and providing temporal constraints, both of which appear later.

The rules directly associated with a file type are meant to describe the overall structure of the file, and apply any restrictions that require consideration of the entire file, like uniques in a record field. Rules related to the composition of a single record or field would appear with the corresponding record or field definition.

It is worth noting that this description sets validation constraints according to what the login system actually requires. Two examples are presented later which extend this description to include additional, site-specific constraints.

```plaintext
file_type: Passwd_File
type: Text_Record
bind_to: /etc/passwd
use: Crypt_Password
record_type: Passwd_File_Record
rules:
  ::= <Passwd_File_Record><Newline> | <Passwd_File><Passwd_File_Record><Newline>
  Name is Unique

Passwd_File_Record
type: String
desc: A single record in the passwd file
rules:
  ::= <Name>:<Password>:<UID>:<GID>:<GECOS>:<Directory>:<Shell>
  /* the following applies to each record, but not across records */
  Name == "root" => UID == 0

Name
type: String
desc: User’s name
rules:
  match: /\[a-z_][a-z0-9_]*\/
  length: <= 32 /* length is available because this is a string */

Password
type: String
desc: Hash of user’s password, or a special character
rules:
  ::= * | x | <Crypt_Password>

UID
type: Integer
desc: Numeric ID for the user
rules:
  /* knowing UID is an integer allows us to do numeric comparisons */
  UID >= 0
  UID <= 32767
```
GID
    type: Integer
    desc: Numeric ID corresponding to the user's primary group
    rules:
      value: >= 0
      value: <= 32767

GECOS
    type: String
    desc: Optional, informational field
    /* if no rules are specified, the system assumes any value is valid */

Directory
    type: String
    desc: User's home directory

Shell
    type: String
    desc: Program run when user logs in. If nonexistent, cannot login
A.3 The Shadow_File Type

The shadow file is structurally quite similar to the passwd file, so this ruleset is quite similar to the Password_File type. In the case of the last field, we assume that the system provides a Reserved type which indicates that nothing should appear in a field. We could, equivalently, create an actual subtype in this definition and configure it to require that no data be entered in the field.

```
file_type: Shadow_File
  type: Text_Record
  bind_to: /etc/shadow
  use: Crypt_Password
  record_type: Shadow_File_Record
  rules:
    ::= <Shadow_File_Record><newline> | <Shadow_File><Shadow_File_Record><newline>

Shadow_File_Record
  type: String
  desc: A single record in the shadow file
  rules:
    ::= <Name>:<Password>:<Last_Password_Change>:<Minimum_Password_Age>:
        <Maximum_Password_Age>:<Password_Warning_Period>:<Password_Inactivity_Period>:
        <Account_Expiration_Date>:<Reserved>
    Name == "root" => UID == 0

Name
  type: String
  desc: User's name
  rules:
    match: /[a-z_][a-z0-9_]*/
    length: <= 32

Password
  type: String
  desc: Hash of user's password, or a special character
  rules:
    ::= * | ! | !<Password> | <Crypt_Password> | <Null>

Last_Password_Change
  type: Integer
  desc: Epoch time of last password change

Minimum_Password_Age
  type: Integer
  desc: Min number of days between password changes

Maximum_Password_Age
  type: Integer
  desc: Maximum number of days before a user must change their password

Password_Warning_Period
  type: Integer
  desc: Number of days before password expiration to warn user
```
Password_Inactivity_Period
  type: Integer
  desc: Days after expiration during which password should be accepted

Account_Expiration_Date
  type: Integer
  desc: Date of account expiration, in epoch time
A.4 The Group_File Type

Once again we have a text record file definition, this time for the group file. The User_List field is interesting in that it possesses its own subtype, the User_Name.

file_type: Group_File
type: Text_Record
bind_to: /etc/group
use: Crypt_Password
record_type: Group_File_Record
rules:
::= <Group_File_Record><Newline> | <Group_File><Group_File_Record><Newline>

Group_File_Record
type: String
desc: A single group record in the group file
rules:
::= <Group_Name>:<Password>:<GID>:<User_List>

Group_Name
type: String
desc: Name of the group
rules:
match: /[a-z_][a-z0-9_]*/
length: <= 32

Password
type: String
desc: Password to access group, or empty
rules:
::= <Crypt_Password> | <Null>

GID
type: Integer
desc: Numeric ID for the group
rules:
GID >= 0
GID <= 32767

User_List
type: String
desc: Comma separated list of group member names
rules:
::= <User_Name> | <User_List>,<User_Name>

User_Name
type: String
desc: Name of a single group member
rules:
match: / [a-z_][a-z0-9_]*/
length: <= 32
A.5 The Passwd_Set Type

Being correct of itself does not necessarily make a file valid in its larger context. Grouping types like this Text_Record_Set sets allow us to specify the way individual files need to interact with one another, in a way largely analogous to referential integrity or inter-table constraints on a relational database. Care must be taken, however, to ensure that the set rules allow for files to actually be modified.

In this case, we specify which file types are included in the set, and then provide rules that much apply between the files. In order to do that efficiently, we employ set-based reasoning based on records within the files, with precisely what constitutes a record defined by the record_type parameter of each file type. We use the familiar dot notation to access fields within a record, and try to provide plain-text-friendly versions of the standard set notation symbols. For example, the first rule below would traditionally be written as: $\forall p \in \text{Password File}, p.\text{Password} == 'x' \Rightarrow \exists s \in \text{Shadow File} | p.\text{Name} == s.\text{Name}$

```plaintext
group_type: Passwd_Set
type: Text_Record_Set
desc: File set for the passwd authentication file
includes: Password_File, Shadow_File, Group_File
rules:
   for_every record p in Password_File:
      p.Password == 'x' => exists record s in Shadow_File st p.Name == s.Name
      exists record g in Group_File st p.GID == g.GID

   for_every record s in Shadow_File:
      exists record p in Password_File st s.Name == p.Name

   for_every record g in Group_File:
      for_every uname in g.User_List:
         exists record p in Password_File st uname == p.Name
```

27
A.6 Restricting the Password_File Type

There will be cases where the default set of validations rules are judged to be too permissive. In these cases, we propose a mechanism for providing additional rule specifications which further restrict validity, rather than requiring the modification of the original specification. In this example, we add restrictions to the Password_File definition to require that usernames be less than eight characters long, only root have UID 0, and all users either have passwords in the shadow file or not be allowed to log in.

The restricts keyword is used to indicate that we wish to apply additional rules to an existing definition. We then specify the particular (sub-)types to which we will be adding rules. New rules are added as though they appeared in the original definition, and must be satisfied in addition to all existing rules. This could be implemented by checking all rules, though the runtime system could probably identify cases where one rule is a strict subset of another (as with the length requirement for Name) and only check the stricter rule.

restricts file_type Password_File

    Password_File_Record:
        rules:
            UID == 0 => Name == "root"

    Name:
        rules:
            length <= 8

    Password:
        rules:
            ::= * | x
A.7 Temporal Restrictions on the Password_File Type

The descriptions presented so far govern the state a file must be in to be considered valid, but says nothing about how the file got into that state. The passwd file and its associates are generally only writable by root, which prevents arbitrary users from altering the files. However, it would not be unreasonable to allow users to update their own records, which presents a problem: how do we allow users to change their own record, while prevent them from changing the records of other users?

Typically we’d solve this problem with special software, likely run with Set UID permissions. However, this introduces the potential for problems ranging from (accidentally) trashing the password file to compromising the root account. Temporal restrictions allow us to not only constrain the contents of a file at a moment in time, but also to govern how the file changes over time. By melding temporal restrictions with a knowledge of which user initiated a change, we can derive rules that will permit a normal user to change the passwd file, but only their own record and only in certain ways. Combined with a world-writable passwd file, this gives us the advantages of a set UID solution without the risks.

Once again, employ the system’s ability to deal with the passwd file on a per-record basis. In this case, we can classify any change to the passwd file as adding, removing, or changing a record, and then consider the user who initiated the change to determine whether it is allowed. The Unchanged() construct is used to indicate that a user may change most of their own file, but may not change their own name or UID.

```plaintext
restricts file_type Password_File
rules:
  UID is Unique

temporal:
  Record.Add => User.Name == 'root'
  Record.Remove => User.Name == 'root'
  Record.Update =>
    User.Name == Record.Name OR User.Name == 'root'
    User.Name != root => Unchanged(Name) AND Unchanged(UID)
```
A.8 SSH Config and Key/Value Files

Key-value based files are popular when storing configuration data for servers and the like. In addition to individual key-value pairs having particular validation rules, it is also important to be able to deal with groups of key-value pairs. The SSH configuration file (generally found at "/.ssh/config") provides an excellent example of both of these properties. To that end, we present a partial ruleset for the file.

Where Text_Record files have records, Key_Value files are understood to contain key/value pairs. We provide the system with a way to identify keys, values, and pairs, and can then craft rules in terms of those constructs. This is also helpful as it allows us to provide information about all of the keys or values, such as whether they are case-sensitive, at once, rather than requiring that information to appear for every possible pair.

Once we have set up the basic format of the file, we can construct Key_Groups which consist of sets of k-v pairs that appear together. Each group contains an indication of how to identify the start of a group, along with a number of other parameters. It then contains a listing of all keys which could belong to that group. We treat these keys as subtypes of the group, and define their rules to specify constraints on the value associated with the key.

```plaintext
file_type: SSH_Config
type: Key_Value
desc: per-user or system host configuration for SSH
bind_to: HOMEDIR/.ssh/config, /etc/ssh/ssh_config

kv_pair: SSH_Config_KV_Pair
desc: ssh config key/value pair
rules:
    ::= <SSH_Config_Key><SSH_Config_Sep><SSH_Config_Value>

SSH_Config_KV_Pair
type: String
desc: k/v pairs may be separated by whitespace or an =
rules:
    ::= <Whitespace> | <Whitespace>=<Whitespace>

SSH_Config_Key
type: String
desc: a single line in the ssh config file
rules:
    ::= <SSH_Config_Comment> | <SSH_Config_KV_Pair>

SSH_Config_Comment:
type: String
desc: a comment left in the config file. Always starts with a #
rules:
    match: /^#/

SSH_Config_Value:
type: String
desc: a comment left in the config file. Always starts with a #
```
SSH_Config_Key
  type: String
  desc: a key within the ssh config file
  key_groups: Host_Config
  rules:
    Case_Insensitive

SSH_Config_Value
  desc: values within the ssh config file
  rules:
    Case_Sensitive

Host_Config:
  type: Key_Group
  desc: properties associated with a particular host
  rules:
    begin_with: Host
    Allow_Multiple
    Optional: all

  key: Host
    type: String
    desc: (Set of) hosts associated with this host configuration
    rules:
    ::= <Hostname> | <Host>,<Hostname>

Hostname
  type: String
  desc: a network host name
  rules:
  ::= <Alphanumeric_String> | *

key: AddressFamily
  type: String
  desc: Address families to use when connecting
  rules:
  ::= any | inet | inet6

key: Ciphers
  type: String
  desc: Cipher allowed for protocol version 2, in order of preference
  rules:
  ::= <Cypher> | <Cyphers>,<Cypher>
  Cypher is Unique

Cypher:
  type: String
  desc: A single protocol V2 cypher
  rules:
  ::= 3des-cbc | aes128-cbc | aes192-cbc | aes256-cbc
key: CompressionLevel
type: Integer
desc: Compression level to use, if enabled. V1 only
rules:
  CompressionLevel >= 1
  CompressionLevel <= 9

key: HashKnownHosts
type: Boolean
desc: Hash host names / addresses added to ".ssh/known_hosts"
true_value: "yes"
false_value: "no"

key: HostName
type: String
desc: Real host name to log into
rule:
  /* we assume that rules for the following are provided elsewhere */
  ::= <Valid_Hostname> | <IP_Address_V4> | <IP_Address_V6>

key: LocalForward
type: String
desc: Local port and remote host/port for forwarded connection
rules:
  ::= [<Bind_Address>:]<Port><Whitespace><Host>:<Host_Port>

  Bind_Address
type: String
desc: optional local address to which to bind forward
rules:
  ::= <Valid_Hostname> | <IP_Address_V4> | <IP_Address_V6>

  Port
type: Integer
desc: local port for port forward
rules:
  Port >= 0
  Port <= 65535

  Host
type: String
desc: remote host to which to connect
rules:
  ::= <Valid_Hostname> | <IP_Address_V4> | <IP_Address_V6>

  Host_Port
type: Integer
desc: port on remote host for port forward
rules:
  Host_Port >= 0
  Host_Port <= 65535
References

5. Trithemius, J.: Steganographia. (1499)


