CERIAS Tech Report 2000-28

BETTER LOGGING THROUGH FORMALITY
APPLYING FORMAL SPECIFICATION TECHNIQUES TO IMPROVE AUDIT LOGS AND LOG CONSUMERS

by Chapman Flack and Mikhail J. Atallah

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47909

Better Logging through Formality

Applying Formal Specification Techniques to Improve Audit Logs and Log Consumers

Chapman Flack[*] and Mikhail J. Atallah[**]

CERIAS, Purdue University
1315 Recitation Bldg., West Lafayette, IN 47907-1315 USA
{flack,mja}@cerias.purdue.edu

Abstract. We rely on programs that consume audit logs to do so successfully (a robustness issue) and form the correct interpretations of the input (a semantic issue). The vendor's documentation of the log format is an important part of the *specification* for any log consumer. As a specification, it is subject to improvement using formal specification techniques. This work presents a methodology for formalizing and refining the description of an audit log to improve robustness and semantic accuracy of programs that use the log. Ideally applied during design of a new format, the methodology is also profitably applied to existing log formats. Its application to Solaris BSM (an existing, commercial format) demonstrated utility by detecting ambiguities or errors of several types in the documentation or implementation of BSM logging, and identifying opportunities to improve the content of the logs. The products of this work are the methodology itself for use in refining other log formats and their consumers, and an annotated, machine-readable grammar for Solaris BSM that can be used by the community to quickly construct applications that consume BSM logs.

Keywords: log, formal, specification, documentation, reliability, interoperability, CIDF, BSM, grammar

1 Introduction

Audit logs maintained by computing systems can be used for a variety of purposes, such as to detect misuse, to document conformance to policy, and to understand and recover from software or hardware failures. Any such application presumes a log consumer, software that can read and analyze the log, and

[*] Supported in part by an Intel Foundation Graduate Fellowship, by contracts MDA904-96-1-0116 and MDA904-97-6-0176 from Maryland Procurement Office, and by sponsors of the Center for Education and Research in Information Assurance and Security.

[**] Supported in part by Grant EIA-9903545 from the National Science Foundation, and by sponsors of the Center for Education and Research in Information Assurance and Security.

draw conclusions of interest about the state history of the system that produced the log.

Two requirements that apply to any log consumer are easily stated: the consumer should be able to read any sequence of the possible log records without failure, and any results computed (or conclusions drawn) should be correct (or justifiable). These requirements may not have the same weight in all applications. Some unreliability in a tool devised ad hoc to count uses of a certain software package, for example, may be tolerated as a practical matter.

Also, it may suffice for an ad hoc tool to skim the log for a very small fraction of its information content, as the use to which the information will be put is known in advance. However, initiatives like the Common Intrusion Detection Framework (CIDF)[14] place a renewed emphasis on exchanging event information among multiple agents, where one agent should not discard information another may need. CIDF's Common Intrusion Specification Language (CISL) is necessarily expressive enough to convey the semantic nuances of event records; that very expressiveness increases the pressure on any system that would translate logs into CISL not to miss or mistranslate those nuances, lest later analysis be led into error. In a production intrusion detection system, failures caused by incorrect handling of the input, or unsound conclusions resulting from semantic misunderstandings, may be costly. In unlucky cases, they may represent new, exploitable vulnerabilities introduced by the security tool itself.

This paper describes a way to reduce the risk, observing simply but centrally that the usual documentation accompanying a system that produces logs is also a partial *specification* for software that must consume those logs. Software engineering techniques for formalizing specifications can therefore be applied to find and purge ambiguities and inconsistencies, and reshape the document into one from which reliable log consumers can more readily and consistently be built. Opportunities to improve the log itself may be revealed in the same process.

The amount of attention devoted here to the mere task of reading a stream of data may be surprising to a reader who has not been immersed for some time in extracting meaning from audit logs of general purpose systems. The case that an audit log is a peculiarly complex stream of data, presenting subtle issues of interpretation, will be built in Sect. 4 with quantitative support in Sect. 6.

2 Contributions

Contributions of this work include artifacts of immediate use to the community, and suggestions with demonstrated potential to improve design of future audit producers and consumers.

– A grammar and lexical analyzer package for Sun's Basic Security Module[8] (BSM) audit data through Solaris 2.6. The package, requiring Java[3] and the ANTLR parser generator[11], produces a parser for BSM audit that can be rapidly extended with processing specific to an application. The parser is conservative: it may signal a syntax error on an undocumented BSM record that was not covered in our test data, but will not silently accept invalid

input. Undocumented rules are readily added to the grammar as they are discovered. The package, available for educational and research purposes, has been used to speed development in one completed and several ongoing BSM-related projects.
- The BSM grammar in the package is extensively annotated, with hyperlinks from grammar rules to the corresponding pages of Sun documentation. While others working with BSM have undoubtedly noted some of the same ambiguous or misdocumented records that we have, and probably some not present in our test data, there has not always been a document available to the community and intended to detail such discoveries in one place.
- While grammars and parsing techniques arguably offer a natural approach to the reliable processing of structured information that auditing requires, they have been strangely often neglected in practice, as described in Sect. 4. Questions of practicality may have discouraged more widespread adoption. For example, while it is clear that audit records must be described by some grammar,[1] that observation does not alone guarantee a *tractably parsed* grammar. [4] Section 5.2 argues for more optimism, and this work demonstrates practicality and effectiveness on a widely-available, commercial audit format. At the same time, the complete grammar can be studied for useful insights such as which aspects of BSM logs are beyond the expressive power of, e.g. regular expressions.
- The requirement to draw justifiable conclusions, mentioned in Sect. 1, reveals that not only syntax but semantics must be captured. Semantic content is an, or possibly the, important thing to get right. Providing a grammar does not lay the issue to rest, but does help in two ways. First, Sect. 4 argues that the grammatical structure of the input cannot be ignored without sacrificing semantic information. Second, in a more cognitive than technical vein, without a concrete representation of the details to be resolved, promising discussions about audit content sometimes end up, to recycle Pauli, “not even wrong.”

3 Audience

This paper assumes a familiarity with parsing concepts and some parser generating tool, such as might be acquired in an undergraduate compilers course. Examples will be in the notation of the ANTLR parser generator, similar enough to other tools’ notations that the reader need not know ANTLR per se to follow the arguments, but may refer to [11] to pursue niceties that are tangential.

Examples of BSM event record formats will be presented and discussed. For the most part, these are records of UNIX system calls and the discussion may assume a familiarity with the operations and subtleties of the programming interface for UNIX or a similar operating system, and some of the ways those operations can be abused, such as might be expected in the intrusion detection

[1] They are produced by a Turing-equivalent machine.

community. Terms such as 'rules' and 'transition diagrams' familiar from rule-based (e.g. [10]) and state-based (e.g. [5]) intrusion detection efforts will be used freely. Sample BSM records, being binary rather than text, would add little to the perspicuity of the grammar examples they match, and such low level details play no role in the discussions. No specific familiarity with BSM will be required to follow the arguments, though the reader whose curiosity is piqued may refer to [8].

4 Other Approaches

It is not necessary to have a grammar to extract some useful information from an input stream. Various intrusion detection systems support BSM audit logs, and we obtained access to code or internals documentation to see how three of them do it.[2] ASAX[10], IDIOT[2], and USTAT[5] all skim information from BSM logs without concern for grammatical structure. This section will compare their approaches and examine some consequences. The critique is not intended to disparage these projects, which set out to shed light on other aspects of the intrusion detection problem and did so with acknowledged success. The consequences discussion will include some issues that apply to a grammar-based approach as well, and so should be considered in any audit project.

4.1 Canonical Form

All three systems have some notion of a canonical form into which the native BSM log is transformed as a preprocessing step. Their canonical forms are rather different in intent and reality from CISL. Where CISL sets out to allow heterogeneous applications across platforms to share event and analysis data and agree on their interpretation, the canonical forms of ASAX, IDIOT, and USTAT serve mostly to simplify porting of the tools themselves.

4.2 How the Log Is Processed

ASAX. In ASAX, most of the work of converting BSM to the "normalized audit data format" (NADF) is done in `conv_bsm.c`. Examination reveals a table, each row of which contains a BSM token ID, a base NADF field ID, and a pointer. There is a row for every BSM token type expected to occur; for tokens that can appear multiply in one event record (`arg`, for example), several rows are allocated. Before each BSM record is processed, the pointer is nulled in each row. Then, for each token in the record, if a row can be found that contains the token's ID and a null pointer, the first such row is updated to point to the token.

After the whole record has been read, the rows for token types that did not appear in the record still have null pointers, while rows for token types that

[2] We suspect the way these tools process the BSM input is typical, but there are certainly other tools that support BSM for which we did not obtain source code or sufficiently low-level documentation to make a determination.

appeared point to the corresponding tokens. Finally, in the order of appearance in the table, tokens are copied field-by-field to the NADF record. The final NADF record contains verbatim copies of the fields from the BSM record, but with padding to support aligned access, and reordered to match the order of token IDs in the table.

IDIOT. IDIOT's canonical form is slightly more abstracted from the underlying audit format than is NADF. IDIOT defines a set of attribute names and picks out field values of interest from BSM tokens as they are encountered, binding them to the corresponding IDIOT attributes. The mapping is done by a script that reads the line-oriented output of `praudit`, a Sun tool that renders the BSM data in readable text, one BSM token per line.

USTAT. USTAT also defines its own abstract event record, whose attributes (with a few exceptions) are bound to fields of BSM tokens as shown in Fig. 4.5 of [5], provided those tokens appear in the incoming event record.

4.3 Consequences

Invalid Input Detected Late or Not at All. A strategy of simply copying data from tokens as they appear, best seen in the ASAX source, will not detect ungrammatical input, such as an event header followed by an impossible sequence of tokens for that event. An invalid stream of native data can be silently transformed to an invalid stream in canonical form, leaving the problem to be detected in a later processing step, if at all.

Semantic Interpretation Left to Later Stages. The canonicalizers used by ASAX, IDIOT, and USTAT defer, to varying degrees, details of the native audit format to be handled by the later, ostensibly less platform-specific, stages of analysis. The situation is clearest in ASAX, whose canonical form is nothing more than the native form with fields reordered and padded for aligned access. The specific significance of information within the fields is left to be spelled out in RUSSEL, the language of ASAX rules. Dealing with native format issues in rules complicates porting those rules, even to other flavors of UNIX whose audit formats differ even though the system calls and their semantics are the same. Format idiosyncrasies of a given field or event must also be handled in each rule that involves the field or event, and a rule language may not have convenient facilities for functions or subroutines.[3]

IDIOT and USTAT take on more of the interpretation problem at the time of canonicalization, at least picking out certain fields of interest and mapping them into a set of attributes intended to be less platform specific. However, the

[3] It is possible in ASAX to isolate some format-specific processing in functions written in a traditional programming language, linked to ASAX, and invoked in RUSSEL rules.

combination of such selective inclusion of fields with disregard for grammatical structure can lead to loss of semantic content, as described next.

Lost Syntactic Cues to Meaning. Consider the BSM grammar rules in Fig. 1. To determine whether an `ioctl` system call was applied to a socket, an other non-file, a file descriptor with no cached name, or an ordinary good file descriptor requires careful consideration of what tokens appeared in the record. If a parser is used, subsequent analysis logic has only to look at the parse tree to determine which rule matched the input. Absent a parser, some analysis effort equivalent to deciding which rule would have matched is deferred to later processing stages. Again, in ASAX, the work can be done by RUSSEL rules that are tightly bound to the details of the native format, explicitly testing for the presence of certain fields or for distinctive field values.

```
IoctlGoodFileDescr
    : path (attr)? arg[2,"cmd"] arg[3,"arg"] ( arg[2,"strioctl:vnode"] )?
    ;
IoctlSocket
    : (socket)? arg[2,"cmd"] arg[3,"arg"]
    ;
IoctlNonFile
    : arg[1,"fd"] arg[2,"cmd"] arg[3,"arg"]
    ;
IoctlNoName
    : arg[1,"no path: fd"] ( attr )? arg[2,"cmd"] arg[3,"arg"]
      ( arg[2,"strioctl:vnode"] )* // have seen 0, 1, or 2 of these
    ;
```

Fig. 1. ANTLR grammar rules for a portion of a system call record. The parser's decision which rule to apply distills semantic content from a series of specific tests that would otherwise be left to later processing stages

In IDIOT or USTAT the situation is complicated by the mapping of native fields into canonical attributes. In Fig. 1, if the optional attribute and `vnode` arguments are absent, determining which of the last two rules to apply hinges on the text string of the first `arg` token. Because the text in an `arg` token is a constant string serving only as a syntactic cue, it is not copied to the canonical form used by IDIOT or USTAT. As a result, the semantic content conveyed in a parse of the native format cannot be recovered at all in IDIOT or USTAT, an example of the semantic sacrifice risked when grammatical structure is ignored.

Nonconservative Transformations. The problem of dangerous transformations must be considered in any audit project, grammar-based or not. However, the systems described in this section offer examples to illustrate this important issue.

IDIOT relies on Sun's `praudit` tool to preprocess the binary log format into a text representation. The transformations made by `praudit` go beyond representing the binary log. User and group IDs, for example, are presented as user and group names, and the transformation reflects the name-to-ID mapping in effect at the time `praudit` runs, not at the time of the audited event. That transformation can be disabled by a `praudit` option, but others cannot. IP addresses, for example, are displayed as domain names by `praudit`, a transformation that reflects the state of the Domain Name System[9] at the time the tool is run, not when the event was logged.[4]

The USTAT document in Sect. 4.1.2.6[5] describes another nonconservative transformation. BSM records often present path names in a non-canonical form such as `/etc/../usr/share`. USTAT's preprocessor, accordingly, includes a "filename correcting routine," the wisdom of which can be questioned on two grounds.[5] First, such a transformation cannot be said to conserve correctness without knowing the state of the affected file systems, including symbolic links and any cycles in the directory graph reflecting accidental or deliberate file system corruption at the time the event was logged. Second, the exact form of the path name appearing in the log reflects the kernel's construction of the path from a process root and working directory and any symbolic links encountered; it conveys part of how the event came to pass, and may have forensic value.

Audit records are often consulted in cases where the integrity of the system that produced those records is in doubt. It seems prudent, in transformations applied to those records after the fact, to avoid unnecessary assumptions about the state of the system that produced them. The same concern need not apply to transformations reliant on mappings that are widely known and independent of any single computer system. Those transformations, such as IP protocol numbers to the names of those protocols, may arguably be used wherever they would be useful.

4.4 Discussion

The intrusion detection systems just described all translate native BSM audit logs into some canonical form without looking at grammatical structure. They do so, however, by defining canonical forms that offer little semantic support to later analysis stages. The proposal by Bishop[1] is another example of such a canonical form. Where such a form is used, the authors of rules or transition diagrams must still account for what is meant *in the original native form* when, for example, a certain field is absent from a record. The rules, therefore, become platform specific.[6]

[4] A more fundamental, equally fatal, but less enlightening nonconservative transformation applied by praudit is the presentation of the log data in a delimited form with no escaped representation for occurrences of the delimiter in the data. This feature alone rules out any role for praudit in a reliable consumer of BSM logs.

[5] This transformation is also done by `praudit`.

[6] While this paper was in preparation, a portion of the EMERALD project[6], eXpert-BSM became available for review. Unfortunately, the distribution terms prohibit

Such lazy translation is not even an option if the target representation, like CISL, intends to convey the semantic nuances revealed by a careful parse of the original. A translator that overlooks or mistranslates those nuances will produce a false translation that may lead later analysis into error.

Finally, canonical or intermediate representations of audit data should be scrutinized for assumptions that would require nonconservative transformations during conversion. An example would be a canonical form that identifies machines by domain names, if IP numbers are used in the native form.

5 Appropriateness of a Grammar Approach

At least two objections may be considered to a grammar representation of a logging format.

5.1 Efficiency

Logs are voluminous and efficiency in their processing is important. Developers observing that constraint may lean toward ad hoc and handcrafted techniques and away from strict attention to grammatical structure. Section 4, however, showed that the price of parsing, if saved up front, must be paid later if the full information content is to be extracted from the input. In fact the price is paid with interest, as a single test and decision not made on the initial parse of the data may have to be duplicated in many rules that apply to the same records. It was not in the scope of this work to build otherwise-comparable intrusion detection systems and obtain a performance comparison, but these observations, coupled with the importance of reliability and maintainability, suggest that grammar techniques in audit processing should not be dismissed out of hand on efficiency grounds.

5.2 Applicability

The foregoing discussion breaks down unless it is reasonable to expect that audit logs can be described by grammars in the classes that enjoy efficient parsing algorithms.

A distinction must first be made, just as in the specification of programming languages. A grammar like that given in the Java specification[3] does not purport to describe the language "all semantically reasonable Java programs"; it describes the simpler language, "syntactically valid Java programs." The grammar, therefore, is a specification with a deliberately limited scope. Aspects of the language excluded from its scope fall into two broad categories:

reverse-engineering to discern just how the log is processed, but [7] presents some sample detection rules for this newer tool and here again, rules that describe attacks applicable to UNIX systems generally must be written to the specifics of the BSM format.

Unspecified Aspects. Some aspects of a language are not addressed by any part of the specification. For example, the Java grammar imposes no structure on a *methodBody* or other *block*, other than that it be some sequence of zero or more *blockStatement*s. The statements themselves, and their subproductions, are explicitly specified, but it is considered beyond the scope of a language specification to characterize how those statements might be placed in meaningful blocks by programmers.

Aspects Specified Extragrammatically. Some details of the language are explicitly specified elsewhere. For example, Java's official grammar is Chapter 19 of the Java specification. Other chapters, in prose or grammar-like notation, contain requirements not embodied in the grammar itself, such as those for casts and parenthesized expressions, or field and method modifiers. Therefore, the grammar describes a superset of conforming programs, which must be culled after parsing by enforcing the extragrammatical requirements.

For Java, two factors contributed to the exclusion of these details from the grammar itself: the choice to provide a grammar no more complex than can be parsed left to right without backtracking and with only one token of lookahead, and the choice to adopt a C-like syntax, which includes constructs that cannot be parsed that way.

Minimizing Extragrammatical Requirements. The need for the second kind of scope restriction can be reduced by relaxing restrictions on the grammar to be provided. For example, ANTLR supports LL(k) grammars for configurable k with predicates (a form of localized backtracking)[12], and comes with a $k = 2$ Java grammar that explicitly embodies requirements for casts, etc., that had to be left out of the official LALR(1) Java grammar.

If not constrained to perpetuate difficult features of an existing language, a designer can so craft a new language that a simple, efficiently parsed class of grammar is adequate to specify it, and few or none of its syntactic features need to be specified extragrammatically. The designer of a new audit logging format is in such a position.

Application to Audit Logs. The specification for an audit log, like that for a programming language, may be deliberately restricted in scope. While each individual event, and its subproductions, should be explicitly specified, the sequences in which events may appear in actual use of the system depend on user and program behavior, and their easy characterization *a priori* is unlikely. As with a block of "zero or more statements," the simple "zero or more events of any type" is a permissive superset of the expected event sequences and presents no difficulty in parsing.

The individual event records are produced by code that must execute when, and only when, the corresponding events take place. Necessary restrictions on the logging code (e.g. termination guarantees) limit its complexity and, with it, the complexity of the grammar required to describe the record, even if the format was not designed with a specific grammar class in mind. The commercial log format described in the next section was successfully described in ANTLR notation with 1-token lookahead for most choices. The predicates required at other choice points all amount to constant-depth additional local lookahead.

6 Formalizing the BSM Audit Format

This work began when a robust consumer for Sun's Basic Security Module (BSM) audit log format[8] was needed for another project. The existing documentation on the format was transcribed into a grammar notation. Before a parser could be generated to test the grammar against actual logs, it was necessary to modify the grammar to resolve all ambiguities detected by the parser generator. The grammar was then iteratively refined by generating a parser, running it on actual logs, and observing parse errors. A parse error could represent an error in the BSM documentation, or a fault in the Solaris log production code. It could be resolved for the next iteration by modifying either the grammar or the Solaris code. For this project, modifying the code was not an option, so all parse errors were resolved by modifying the grammar, leading ultimately to a grammar that describes closely the log that Solaris actually produces, even in instances that seem unintended.

The resulting grammar contains 327 named, nonterminal rules. Examination shows that the rules are associated in a straightforward manner with the 267 kernel and user event types and 41 token types found in the Solaris 2.6 system files, and follow the Sun documentation with only necessary departures. That is, the number of rules does not reflect an especially obfuscated grammar but, rather, an indication of the intrinsic complexity of the audit log alluded to in Sect. 1. By comparison, the example grammar supplied with ANTLR for the Java 1.1 programming language includes 64 such rules.

The remainder of this section will discuss selected examples of the flaws or ambiguities in BSM documentation or implementation that were detected by this methodology. The entire grammar, with annotations describing discrepancies, can be downloaded from `http://www.cerias.purdue.edu/software/` with the other files needed to compile and run a working BSM parser. To print the grammar as an appendix would be impractical because of its size, and would sacrifice the hyperlinks that connect the grammar rules to the corresponding sections of Sun's BSM documentation.

6.1 Difficulties Detected by Static Analysis in Parser Generator

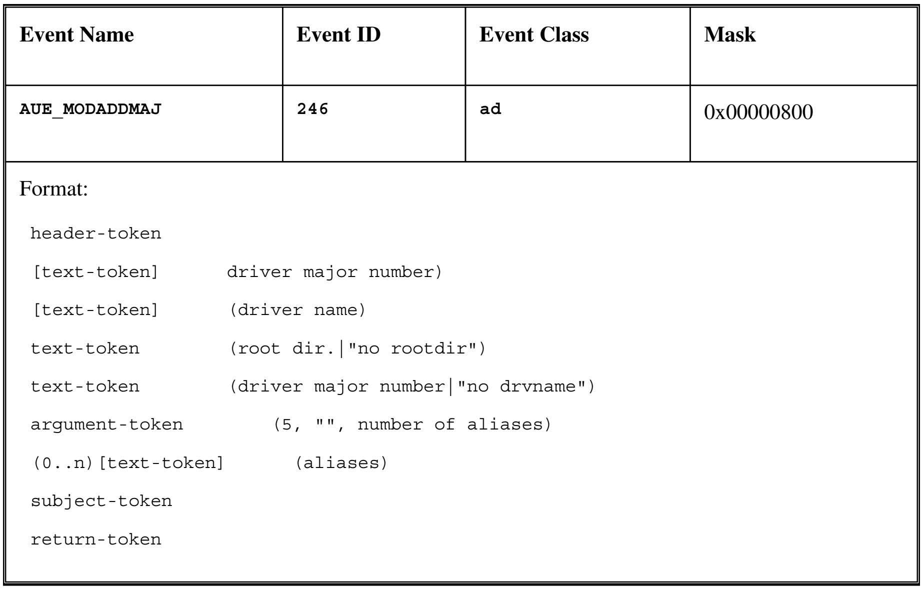
Non-LL(1) Constructs. Many of the ambiguities that were automatically detected simply reflected features of the BSM log format that cannot be recognized by an LL parser with one lookahead token; they were resolved by adding explicit lookahead at strategic places in the grammar. They do not reflect inherent ambiguity in the log format, but nevertheless are possible pitfalls for developers who attempt to develop a straightforward BSM consumer tool from the documentation without a parser generator's rigorous analysis.

Constructs Resolvable with Semantic Information. Compiling a naïve version of our BSM grammar will result in 16 warnings of ambiguity apparently inherent in the log syntax, any one of which would suffice to dash the hope of reliably processing BSM logs, whether by a conventional parser or by any other means. Although it is impossible in these cases to determine the correct grammar rule to apply from the sequence of BSM token types alone, they can be resolved by looking explicitly into the *values* carried by certain of those tokens, an operation known in ANTLR terms as a "semantic predicate." Specifically, a BSM 'arg' token contains a `data` field whose value is necessary and sufficient to resolve these 16 cases. The `data` values, which are constant character strings, are shown in the printed documentation, albeit without an explanation that they are essential at parse time to properly interpret the log. Both IDIOT and USTAT appear to discard these values in the conversion to canonical form, perhaps on the assumption that a token field whose value is constant does not convey essential information.

True Ambiguity. It may not be surprising, given the complexity of what BSM must log and the lack of formal analysis in its original design, that a few ambiguities remain. Instead of reflecting limitations of a particular parsing technique they are, if the BSM documentation is correct, inherent in the log format. Figure 2 is an example.

The description with two optional 'text' tokens followed by two mandatory ones leads to a formal ambiguity if an event record has exactly three text tokens following the header. It is clear in that case that one of the two optional text tokens is present, but the parser cannot determine whether it is the driver major number or the driver name. The ambiguity cannot be resolved, even with a semantic predicate, unless there is a way to tell decisively by looking at the text string whether it is a driver major number or a driver name. Perhaps the number is always a text string of only digits and the name must begin with a non-digit, but this should be stated in the BSM documentation if programs are expected to depend on it. Or, it may be that the documentation is mistaken in showing the number and name as being independently optional: perhaps it should be "[text text] text text" with the first two both there or both absent. If that is the case, the documentation should be corrected.

Without access to the intent of the BSM developers, the grammar was modified to embody the last interpretation, which is reasonable and conservative under the circumstances. It will work if the first two 'text' tokens are both present and if they are both absent. If an instance is encountered of the ambiguous case with one of the two present, a parse exception will be signaled, avoiding an undetected misinterpretation.



Event Name	Event ID	Event Class	Mask
AUE_MODADDMAJ	246	ad	0x00000800

Format:

```
header-token
[text-token]      driver major number)
[text-token]      (driver name)
text-token        (root dir.|"no rootdir")
text-token        (driver major number|"no drvname")
argument-token        (5, "", number of aliases)
(0..n)[text-token]      (aliases)
subject-token
return-token
```

Fig. 2. Description of a record from [8]

6.2 Difficulties Detected in Testing

After the statically-detectable problems were resolved, the grammar was repeatedly used to generate a parser. The parser was applied to a collection of 2.2 megabytes of BSM audit data obtained in-house and from other institutions, from SunOS and Solaris systems as recent as Solaris 2.6. Two general classes of discrepancy were detected between the BSM documents and the actual logs.

Undocumented Records. Some records were encountered in the sample logs that simply do not appear in the documentation. Corresponding rules were added to the grammar to allow the logs to be parsed. Fig. 3 is an example.

```
AUE_CONNECT
        : %AUE_CONNECT socket socket subj ret
        ;
```

Fig. 3. ANTLR grammar rule for an event record that appears in our sample logs but is not documented

Misdocumented or Misimplemented Records. Some records were encountered for event types that were documented, but parse errors were detected because the records did not conform to the published format. Fig. 4 is an example. So that the logs could be parsed, the affected grammar rules were changed from direct transcriptions of the documentation to reflect the records actually encountered.

Event Name	**Program**	**Event ID**	**Event Class**	**Mask**
`AUE_su`	`/usr/bin/su`	`6159`	`lo`	0x00001000

Format:

```
header-token
text-token       (error message)
subject-token
return-token
```

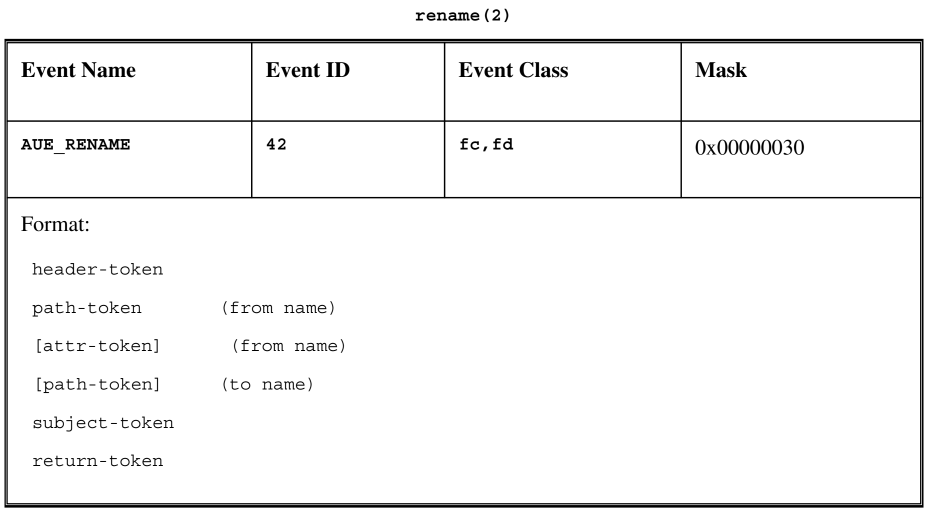
Fig. 4. Description of a record from [8]. In actual audit logs examined in this work, the text and subject tokens appear in the reverse order

6.3 Difficulties Not Automatically Detected

Figure 5 illustrates a point where the published BSM documentation is incomplete, and hence the interpretation of a log record is not completely determined, but the formal method described in this work could not detect the problem. The problem was recognized, however, during the process of transcribing the documentation into a grammar, and the discipline of that process may have contributed to that recognition.

Two tokens are shown as optional: the file attributes for the source file, and the rename destination path. The rule is quite readily parsable, but has two suspicious features. First, the optional tokens are shown as *independently* optional, implying four possible record variants for the rename event. In actual logs, only two—both tokens present, both absent—have been observed. The documentation may be incorrect, but the methodology will not detect the problem. The rule as stated presents no parsing difficulty that would be detected in static analysis, and, if incorrect, it matches a superset of the records that can be encountered, so no parse error will be produced. Nevertheless, it should spur any conscientious developer of a log consumer to wonder exactly what should be inferred about the state of the audited system when each of the—as written—four variant forms is encountered.

The second suspicious feature is that the destination path is shown as optional at all. It is absent in our samples only when the file attributes are absent also, which seems to happen only when the source file is not found. The feature



rename(2)

Event Name	Event ID	Event Class	Mask
AUE_RENAME	42	fc,fd	0x00000030

Format:

```
header-token
path-token        (from name)
[attr-token]       (from name)
[path-token]      (to name)
subject-token
return-token
```

Fig. 5. Description of a record from [8]. Why is the "to name" optional?

may be an artifact of some implementation detail within the rename system call. It might be worth changing, however. An intrusion detection system might recognize a certain intrusion attempt from a rename with a specific destination. Detection could be delayed if the intruder mistypes the source file name the first time, causing the recognizable destination path to be omitted from the record.

7 Methodological Recommendations

The ideal time to apply the ideas of this work would be during the design of the audit log format for a new system. A new log format can be designed to fall in a language class that is easily parsed with modest lookahead, and specification ambiguities detected by static analysis can be eliminated before implementation. Specification-driven tools can speed implementation and testing, and the annotated grammar can be provided as documentation.

The ideas can still be applied, however, when an existing log format is reviewed for possible improvement, and even in the simple development of tools to consume an existing format. In this less ideal setting, too late for the other formal-method benefits cited above, the technique has valuable potential for improved understanding of the log nuances and more thorough validation and verification of the software. It was applied in that way to BSM in this work, suggesting a methodology for similar projects. An existing audit format can be approached by iterating these four steps:

1. Prepare a grammar by transcription from whatever documents are available. As ambiguities are detected by the parser-generator's analysis, return to the documents, sample logs, experimentation, or system source code (if available)

to determine if any information present in the log can be used in explicit predicates to resolve the ambiguities. Also make note of constructs whose semantic significance is unclear to the human reviewer, even if not formally ambiguous.
2. When the grammar can be successfully compiled, apply the parser to a good sample of audit data and note any parsing diagnostics. Determine whether these represent flaws in the log documentation, the logging implementation, the grammar, or combinations of these. If this process is undertaken by a vendor, whatever needs to be corrected can be. Otherwise, options may be limited to suggesting fixes or documenting the issue and complicating the grammar.
3. Given a grammar that successfully describes the logs, scrutinize it for the semantic nuances of the rules. Choice points in the grammar always have semantic significance: because log records are produced by an automaton, the production of one of several forms of a record depends on and conveys information about the state of the system. An event record whose grammar rule shows three optional fields, for example, can make eight distinguishable statements about a particular event and the system state in which it occurred, beyond what is conveyed by field values. If the eight semantic nuances are not clear, return to documents, experiments, or source code until a satisfactory account of them can be made, or until the grammar rule can be tightened to imply fewer cases.
4. Update grammar, documentation, or code as necessary and possible, and repeat.

8 Future Work

- BSM and other auditing systems can have configuration options that control the inclusion or omission of certain optional fields in some records. Our grammar was tested on audit logs produced on systems with similar settings for those options. A single grammar could rapidly grow unwieldy if extended to accept the logs produced under all settings of the configuration options. *Environment grammars*[13] address the problem of parsing such classes of similar languages as efficiently as context-free languages, and could offer a cleaner solution.
  As it happens, the difference between an environment-grammar parser and an ANTLR parser resides entirely in the analysis algorithms used during parser generation. The structures and features required at run time by a parser specified by an environment grammar are exactly those of a parser generated by ANTLR.
- The modest cost of careful parsing might be further discounted in a self-contained application where the exact information needed from the log is known in advance. For example, a self-contained intrusion detection system might compile its rule base together with the full log grammar, producing a parser that skims lazily where possible.

References

[1] Matt Bishop. A standard audit trail format. In *Proceedings of the 1995 National Information Systems Security Conference*, pages 136–145, Baltimore, Maryland, October 1995. 7

[2] Mark Crosbie, Bryn Dole, Todd Ellis, Ivan Krsul, and Eugene Spafford. IDIOT users guide. Technical Report TR-96-050, Purdue University, September 1996. 4

[3] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996. 2, 8

[4] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979. 3

[5] Koral Ilgun. *USTAT: A Real-Time Intrusion Detection System for UNIX*. MS thesis, University of California, Santa Barbara, November 1992. 4, 5, 7

[6] SRI International. EMERALD website. http://www.sdl.sri.com/emerald/, April 2000. 7

[7] Ulf Lindqvist and Phillip A. Porras. Detecting computer and network misuse through the production-based expert system toolset (P-BEST). In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, California, October 1999. 8

[8] Sun Microsystems. *SunSHIELD Basic Security Module Guide*. Sun Microsystems, 901 San Antonio Road, Palo Alto, California, Solaris 2.6 edition, 1997. Part Number 802-5757-10. 2, 4, 10, 12, 13, 14

[9] P. Mockapetris. Domain names – concepts and facilities. STD 13, ISI, November 1987. 7

[10] Abdelaziz Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection*. D.Sc. thesis, Universitaires Notre-Dame de la Paix Namur (Belgium), September 1997. 4

[11] Terence Parr. ANTLR website. http://www.antlr.org/, February 2000. 2, 3

[12] Terence John Parr. *Obtaining Practical Variants of $LL(k)$ and $LR(k)$ for $k > 1$ by Splitting the Atomic k-tuple*. PhD thesis, Purdue University, August 1993. 9

[13] Manfred Ruschitzka. Two-level grammars for data conversions. *Future Generation Computer Systems*, pages 373–380, 1990. 15

[14] Brian Tung. Common intrusion detection framework. http://www.gidos.org/, November 1999. 2