UiO : Department of Informatics
University of Oslo

Inline Monitoring Attribute Grammar for Security Policies

Toktam Ramezanifarkhani

Research report 490, July 23, 2019

ISBN 978-82-7368-455-4
ISSN 0806-3036

Inline Monitoring Attribute Grammar for Security Policies

Toktam Ramezanifarkhani, *Member, IEEE*

Abstract

Attribute Grammars have been a powerful and the most widely applied formalism in language processing in a vast variety of application domains. With the ever-increasing need to prevent security attacks of software systems, Inlined Reference Monitors (IRMs), as one of the policy enforcement mechanisms, are practical in many modern and security critical systems. The burden of doing critical tasks in the process of security policy enforcement is with the end users or the programmers, who are not usually security experts. Moreover, the existing approaches are syntax-based, non-integrated with program generation, and not completely automated and thus error-prone. Our main goal in this paper is providing a framework with the ability of specifying programming languages with innate power of security policy enforcement at runtime for written programs in these languages. Therefore, we introduce the Inline Monitoring Attribute Grammar (IMAG) formalism as a framework that can efficiently provide such ability. IMAG shifts the burden of securing programs to the expert designers of secure languages in a semantic-directed, integrated and automated approach, and especially applicable to enforce attribute-based policies. Moreover, we present a simple method for IMAGs evaluation, and describe how to derive an evaluation order on attribute occurrences in IMAG based on Ordered Attribute Grammars. We describe IMAG power in security policy enforcement and elaborate an IMAG example for runtime satisfaction of a critical policy. To the best of our knowledge, at the time of increasing interest in language-based security, it is the first time that the intrinsic power of AGs is re-emphasized in the domain of security.

Index Terms

Language-based security, Attribute Grammar, Semantic Specification, Inlined Monitoring, Security Policy Enforcement, Data Flow Integrity

I. Introduction

In recent years, addressing the security of software systems, programs and applications has been one of the major concerns facing the overall computer field [1]. Main approaches are based on static or dynamic checking of desired security properties. Approaches based on static checking attempt to find security problems before program executions e.g. [2],[3]. These approaches usually appear as characteristics of programming languages. For example, every program written in a type-safe language preserve type-safety [3]. Because of not having the real execution of programs, these static approaches are rather pessimistic or optimistic which usually result in low precision (i.e. high level of false negative or false positive) [4] and, of course, less overhead. Instead, approaches that dynamically check program executions are more realistic such that usually result in higher precision. In dynamic approaches, program monitoring is a well-known and common mechanism to ensure the secure behavior of programs at runtime which dynamically checks satisfaction of desired security policies and takes remedial actions in case of a policy violation detection [5]. Among program monitoring techniques, Inlined Reference Monitors (IRMs) [6] merge the code of the enforcement mechanism, i.e. the security code, that checks the policy satisfaction into the target code with a trusted tool called rewriter to make the code secure. By residing within a target, an IRM can observe security-relevant activities more precisely than other reference monitoring mechanisms [6]. Triggering the security code and allowing it to modify the program state with the capability of maintaining desired properties of program executions give IRMs unprecedented flexibility in enforcing security policies [6] such that if a policy is enforceable at runtime then it is enforceable by an IRM [7].

We categorise the dynamic security policy specification and enforcement in two categories; *program-based* approaches in the sence that the policy specification and enforcement techniques are applied to each program, and *language-based* approches where these are characteristic of the programming language itself. Figure 1 (a) and (b) explaines these approches, respectively. Various examples of dynamic approaches based on IRM, which can check if the target adheres to a policy, e.g. [8], [9], [10] usually fall in the first category. In these approaches there are two independent phases; generating a program, and securing it by inlining the security code to the program to satisfy a desired policy at runtime. In program-based techniques, which is summarized in Figure 1 (a), in the first phase a program is written and compiled completely regardless of the policy satisfaction, and then the second phase of securing the program is usually accomplished in one of two different ways depicted in Figure 1 (a1) and (a2). In many approaches such as [11], [12] with respect to a specific security policy, a program becomes secure by specifying the security policy, its translation to the program language, compiling the program and the policy code by the programming language compiler, and finally merging these codes. Sometimes merging these codes proceeds their compilation such as in [10]. In some other works such as Polymer [9], Naccio [13] and PoET/Pslang [14], shown in Figure 1 (a2), instead of specification of a specific policy, a policy specification language is provided. Therefore, another tool as the policy compiler

T. Ramezanifarkhani is with the Department of Informatics and Department of Technology Systems, University of Oslo, Norway, (e-mail: toktamr@ifi.uio.no).

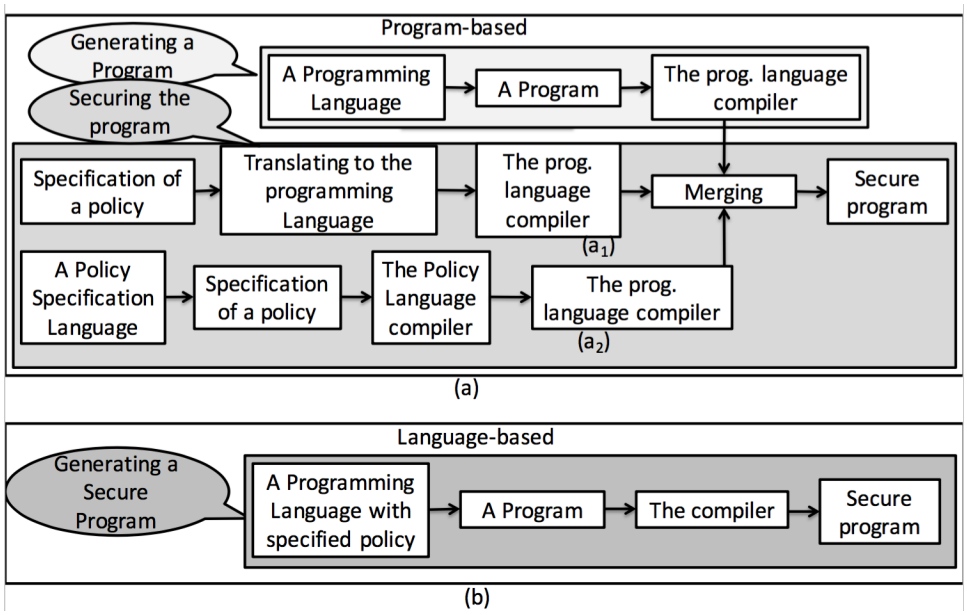

Fig. 1. Program-based and language-based security policy enforcement techniques

is needed to translate the policy language code to the program language code while the rest is like the previous. Techniques in these program-based approaches are applicable for programs in a specific programming language such as Java [15] and/or for some specific policies such as [12], [16], [10] and thus are not applicable for others.

Program-based approaches are useful in special cases and are usually non-uniform, non-integrated, non-automated, and syntax-based. These approaches are applicable when customers, i.e. the code recipients as the end users, need to run untrusted accessible applications, and/or when the customers desired security policies are discretionary, i.e. the policy satisfaction is not always necessary for the application. Besides, using these techniques that come with or without tools, is usually a responsibility of customers or programmers and is not automatic. They should provide the security code, analyze the code and make decisions about locations in the target application where the security code should be inserted, and the reaction in case of a violation detection. Such responsibilities require expert knowledge for example about the desired security policy, underlying language and existing vulnerabilities. Reliance on programmers and especially customers skills to do these critical tasks make the processes error-prone. Moreover, these approaches are non-uniform because usually there are more than one technique or in some cases there might be no technique to specify a desired policy and to convert a target program to a secure one (in the policy point of view). Therefore, there might be several or no form of a secure code for a special program and policy. Generating and securing programs in these approaches are completely different phases coming with a variety of tools, and thus are non-integrated. Moreover, static or dynamic security checking approaches usually look for special syntax patterns in the program code [17], [7]. For example, in [9] a list of all program method names in the target, that their content might have an impact on security policy, are determined by programmers and then such method calls will be instrumented. For example, method calls with the same name would be instrumented the same while they might need different treatment for a security policy enforcement which can be distinguished according to semantic analysis of the program. For instance, consider an overloaded function named $Print$ with two different implementation that differ because of data types of the function parameters such as $Print(string - objo)$ and $Print(integer - objo)$ when the type of parameters should be inferred by a semantic analysis. Also, assume that in the enforcement of a security policy just calls to functions by a string parameter needs monitoring and thus instrumentation. In such a case, according to the attribute types of the parameters the security code can be inlined more effectively.

In this paper, we aim to introduce a language-based approach on the basis of Attribute Grammars (AGs) to specify languages with the capability of security policy enforcement. As shown in Figure 1 (b), everything needed for dynamic policy enforcement by inline monitoring is a built-in feature of the language, and thus in the programmers and the customers point of view, writing and compiling a program is the whole process that is required to generate a secure program as well.

Such language-based approaches are useful when trusted programs that are generated to satisfy security policies, especially for mandatory policies are desired for both of customers and producers. In contrast with untrusted applications by customers, there are remarkable applications especially with critical security services such as intrusion-detection systems, anti-viruses, anti-spyware programs and many others that are always intended to guarantee satisfaction of a security policy. Hence, it is reasonable to generate these applications secure in the first place to satisfy mandatory security policies, e.g. data integrity, which in contrast with discretionary policies, are almost always necessary to be satisfied. Therefore, in addition to customers, the producer of such security critical applications intend to produce unexploitable applications that increase their trust. The key

idea underlying the use of IMAGs to enforce security policies is to shift the burden of critical tasks of securing programs from the customers or the code producers to the designers of secure languages, and thus the framework is prevented from being error-prone. Moreover, this language-based approach is an effective, uniform, integrated and automatic way to enforce security policies. Its effectiveness is because we present a semantic-directed approach to inline the enforcement code conditionally. In contrast with syntax-directed instrumentation, considering instructions semantics while inlining security code or checking a policy, increases the precision of security checks by decreasing false alarm. In addition, decreasing the number of unnecessary checks because of analysis in semantic-directed approach results in less runtime overhead [18]. By considering a security policy, every program has one secure and automatically generated form. In such a language-based approach, the task of specifying a language with a built-in security policy satisfaction feature, which is of increasing interest, is in the language design.

AGs are first devised by [19] for formal language syntax and especially semantic specification, and have proven to be suitable not only for the design but also for the implementation of both domain-specific and general purpose languages [19]. AGs are extended in several ways, e.g. in [20], [21], [22], in order to facilitate the specification of programming languages and analysis. In addition, extensive studies on AGs show their power in wide areas including but not limited to semantic analysis, language processing, and environmental services [23]. We believe that the wide variety of applications of AGs in addition to their simple description and straightforward formal evaluation can generate great enthusiasm for extending AGs to deal with other challenging and interesting areas such as built-in security features in programming languages. Hence, we introduce Inline Monitoring Attribute Grammar (IMAG) based on AGs as a language specification formalism and a framework to specify languages that programs written in these languages have the capability of self-monitoring. Therefore, satisfaction of desired policies are guaranteed by execution of those programs. In other words, the capability of inline monitoring for a security policy appears as a characteristic of the programming languages.

A. Contributions and Goals within the Paper Structure

Let us describe the paper contributions, goals and characteristics in more details as follows.

1) Based on the standard definition of policies and properties, e.g. in [5], we present some basics including *Control Flow Semantics about Status Manipulation (CFSS)* to specify policies on control flow paths such that their specification and enforcement based on AGs becomes straightforward. For simplicity, let us call security policies when they are specified with these basics, *CFSS* policies.
2) We aim to present IMAGs that integrate the grammar-oriented compilation process in AGs with inlining the security code, especially in semantic analysis and intermediate code generation phases.
    a) Attributes are evaluated in the semantic analysis and keep semantic information of program constructions in AGs. Therefore, we utilize the power of attributes in IMAGs to inline the enforcement code in each program. Hence, the instrumentation in our approach is semantic-based.
    b) In IMAGs, we use the compiler to generate the security code and merge it in a suitable places in the rest of the program and thus to generate the secure intermediate code of the program in the first place. It has the following advantages: 1- *Avoiding the replication of some semantic analysis*, e.g. type analysis, that might be required for both of intermediate code generation in compiled languages, and in the policy enforcement [18]. 2- *No need for extra policy language, tools with different verification approaches, and thus TCB will be smaller*: Almost every piece of software in compiled languages used on any computer is produced by a compiler [24]. Therefore, by using the programming language and its compiler to specify, translate and merge the main program and the security code, without replicated modules, the need for an extra language such as policy specification language, and tools such as a separate rewriter is obviated. Since the Trusted Computing Base (TCB) for IRMs implementations usually includes these tools, in the case of trust to one compiler the size of TCB would be decreased. Moreover, in the case of having a verification approach for each tool, reliance on one tool and its verification provides more compatible and integrated approach than having separate tools and applying several and different verification techniques. 3-*Intermediate representations have a finer granularity* than source code and are not as low level as the binary representation. Therefore, they are more suitable to inline the security code [6], and it can make the policy enforcement independent of a wide variety of source code and machine representations.
3) We extract all characteristics that are required to provide the capability of effective and efficient inline monitoring in AGs as a list of requirements. Some of these requirements are already satisfied in the existing AGs, and we utilize them, such as extendable structured attributes and their partial definitions, and conditional semantic rules satisfied in JastAdd [25] and [26]. In addition, there are some other requirements that might be achieveble by providing certain conditions and algorithms on other existing extensions of AGs, such as efficiently writable attributes by more than one production, semantic-based instrumentation at attribute evaluation time or shortly at attribution, and protected and reusable runtime attributes. However, as we will explain later, supported capabilities by existing extensions of AGs with totally different goals do not totally meet the above requirements. Moreover, with/without extra constraints and algorithms, their capabilities cause unnecessary complexities in the specification of the language. Moreover, they can cause circularity, infinity or undecidability of these features in attribute evaluation. However, supporting the least required capabilities as

simple as possible and avoid every unnecessary feature to prevents such complexities. Providing the more clarity and simplicity for the purpose of security results in the more trustworthy, and applicability of the approach. Therefore, to prevent any ambiguity we provide our simplified and clear notations to satisfy our requirements exactly by defining IMAG. Moreover, some compiler tools such as JastAdd [27], and Kiama [28] can be used as basic tools to support IMAGs requirements that we are working on.

4) We present an IMAG pattern for enforcement of security policies that are specified as CFSS. Moreover, we describe how IMAGs that are defined according to this specified pattern are non-circular and guaranteed to terminate. Also, we describe a relatively simple approach to evaluate IMAGs and derive an evaluation order of their attribute occurrences.
5) We demonstrate IMAGs strong power to enforce security policies by defining IMAG automata and describing their operational semantics. The semantics of IMAG automata show the model of IMAG program monitoring. To satisfy policies, IMAG automata can permit, suppress or insert instructions and halt such as edit automata. Moreover, we describe how to evaluate the effectiveness of a specific IMAG in satisfaction of a given security policy.
6) We specify *DFI-IMAG* as an application of IMAGs to automatically enforce *Data Integrity* (DI) and *Data Flow Integrity* (DFI) as two CFSS security policies. These policies are necessary security policies in software security and useful to satisfy some other security policies such as Control Flow Integrity (CFI) [16], [11]. Written programs in DFI-IMAG are proven to guarantee satisfaction of these policies at runtime with the capability of data recovery as the remedial action. In addition, the policy satisfaction in DFI-IMAG comes with no false positive and false negative. Finally, the space and time overhead of DFI-IMAG is computed.

To the best of our knowledge, it is the first work that re-emphasizes the intrinsic power of AGs in software security whereby policy enforcement is integrated into the semantic specification, analysis and code generation of programming languages. Besides, it gives an evidence that AGs have a great potential beyond their traditional scope to affect challenges such as enabling further built-in security features in other areas of using AGs in language processing.

The rest of the paper is organized as follows. A brief background of AGs and inlined reference monitoring is given in Section III. Section IV describes CFSS security policy specification and enforcement with some examples. In Section V, IMAGs and IMAGs pattern are introduced after more discussion about their requirements and existing AGs. Section VII provides the IMAGs evaluation and describes how to check IMAGs effectiveness to satisfy a policy. In Section VI, DFI-IMAG as an application of IMAGs is presented and discussed. Finally, Section IX concludes the paper and briefly suggests future works.

II. Overview

Before formal descriptions, let us go through an application of IMAG to see how it is going to improve security policy specification and enforcement, intuitively. For example, we show how IMAG can support specification and enforcement of attribute-based policies such as Attribute-Based Access ControlÂă(ABAC). ABAC is anÂăaccess controlÂămethod whereby access rights are granted or denied based on assigned attributes to the subjects who request an access resources or objects, environment, and etc. using a combination ofÂăthese attributes and relation and comparisons between them [29]. Since ABAC provides dynamic and context-aware access control that include attributes from many different point of veiws to be specified and applied for authorization, it is considered as the "next generation" authorization model although the general concept is not new, and makes it applicable in authorization which with the traditional access control models is impossible. It is because historically used access control such as mandatory access control (MAC), discretionary access control (DAC), or role-based access control are user-centric and do not take into account additional parameters such as information about resources and their relationsh with the subjects, and dynamic information e.g. time of the day.

IMAG can support ABAC in several aspects such as its dynamic and context-based policy enforcement. Moreover, as expected in ABAC, attributes in IMAGs can be assigned to complicated objects with courser granularity such as sessions and to the finer granularity such as data, to deliver the greatest level of flexibility. As a characteristic of ABAC, IMAG can support set-valued attributes, e.g. the list of user roles or atomic-valued such as data confidentiality level. Attributes can be compared to staticÂăvaluesÂăor to one another, and thus relation-based access control becomes possible.

IMAG could be also useful to address some challanges in ABAC as well. For example, a critical component for the ABAC is the access control policy language which is erroneously and created specifically for a single model [29] and thus have the same challenges of the program-based policy enforcements. One standard access control policy language that implements ABAC with a request and response scheme isÂăXACML (the eXtensible Access Control Markup Language) that has some dynamic permissions and actions are only defined and stored in the framework. At runtime, Visual Guard will dynamically load and apply them. As a result, the application code is completely separated from the security code [29]. Although XACML supports attributes, lacks any kind of formal model [29] while IMAG gives the required formalism and evaluation. In addition, ABAC has the problem of attribute management [29] which due to central control of attributes in IMAG is mitigated. Moreover, although there are several published ABAC models, they are domain specific and not general that can be applied in other domains [29] while IMAG is not limited to a specific domain.

For an example, Fig. 2 shows a pruducyion rule in an attribute grammar for the addition and the assignment. In AGs, each grammar symbol can have an attribute, and each production rule can have a set of semantic rules. When the syntax of

a program is going to be checked according to the language grammar, the Abstract Systax Tree is generated. In addition to syntax, semantics is also checked like the typing rules. In AG, semantic check is done by different semantic analysis schemas and algorithms at this time by the semantic rules which define and use semantic attributes.Then the program code can be translated into the target or an intermediate code as well. Therfore, based on the attributes of elements involved in the program AST, and the attribute valuses extracted by the semantic analysis, lots of useful information about the whole program and each part of the program in the context of the other program parts become available, statically. For example, let us assume that each expression (Exp) has an attribute called "definers" which is a list of all other elements in the progrm that can change the expression. This attribute is extracted during the semantic analysis of the whole program in the AG, statically. Here, for the sake of the simplicity, we ignore the explanations related to such analysis which is based on use-def chain or reaching definition analysis [30]. However, for the assignment, the list of all program elements that can change the amount of Exp_2 can also change the amount of Exp_1 as well. This semantic information is specified by the first semantic rule that updates $Exp_1.definers$ by using $Exp_1.definers$. The second semantic rule states that both of Exp_1 and Exp_2 inherit the enviromental attribute of Exp. Then, the code for the assignment can be generated by the concatenation of the amount of the attribute $code$ of the two expresion, Exp_1 and Exp_1, which would be available at this time in the process of the semantic analysis. These elements can have some other attributes such as $EnvType$, $EnvRole$, $Time$ for Exp in which $EnvType$ attribute is the type of the context such as the class type in OOP which the exp belongs to. $EnvRole$ is the context role of the assignment, and $Time$ is the time tag of the last execution of the expression.

For example, let us assume that in ABAC, we need to check if Exp_1 belongs to a course material, and the context role of the assignment is the student rule, and all elements that may effect the value which is going to be assigned to the Exp_1 is in a specific domain, then the assignemnt is allowed. So, we need to define a security code that can be combined with the exp code as is shown in Figure 2. Instead of the material and student we can also use other attributes that their values could be determined during the program execution. Moreover, the time attribute will be set when the assignment is alowed and thus the required code is added as well.

To do so, what is needed and supported by IMAG is to define suitable attributes that can be extracted statically based on the whole program analysis, and can be saved for further use at runtime such as $Exp.definers$. Some other attributes can be define statically and be used to in the generated code by a default or even without it, and be saved and assigned a value at the late steps in the compile time such as attributes related to the memory address, or runtime such as $Exp.EnvRole$. And it gives a pwerful capabilities for inlining code.

Moreover, it is needed to have the possibility to defined the security code such as the one in this grammar which entails this access policy, assign a protected memory to define the attributes and keep them for further use at runtime. And as one of the main objectives, include the security code in the process of semantic and syntax analysis of the program while the effects of this instrumention can be observe during the these analysis automatically while it is supported by the compiler process.

Now, considering the program sketch,

III. Backgrounds

A. Attribute Grammar

Programming Languages are specified using formal syntax and formal semantics [31], [32]. Syntax concerns the form of language constructs while semantic concerns the meaning of the language and allows us to enforce semantic rules to provide the information we need to generate an equivalent output program [31]. Context-free grammars play a central role in the description and design of programming languages and compilers, and almost all modern languages are context-free.

There is a grammar-oriented compiling technique known as syntax-directed definition and translation that fundamentally associates with each grammar symbol a set of attributes, and with each production a set of semantic rules for computing values of attributes associated with symbols appearing in the production [30]. These grammars are called *Attribute Grammars (AGs)*.

The attractive formalism of AGs introduced by [19] allows context-sensitive properties of individual constructs in a language to be described and automatically computed for any program in the language [21]. Defining context-sensitive syntax, and code generation for languages are important applications of AGs [21]. Properties in AGs are called attributes and are used to establish meaning relationships among symbols and carry semantic information of them [30]. Both semantic analysis and intermediate code generation can be described in terms of attributes in attribute grammar [32]. The simplicity of the Attribute Grammar formalism appears as an important strength. In fact, it allows many static analysis techniques to come up with very precise results [33]. Therefore, AGs are used in the domain of static language specification and processing [33]. In this paper, we extend them for policy enforcement that can guarantee policy satisfaction dynamically at runtime.

Definition 1 (Attribute Grammar)**.** *According to [34] and [35] an attribute grammar is a tuple $AG = (G, SD, AD, R)$ where:*
• *$G = (N, T, Z, P)$ is a context-free grammar in which N is a finite set of non-terminal symbols; T is a finite set of terminal symbols, $N \cap T = 0$ and $N \cup T = V$ which denotes all grammar symbols; $Z \in N$ is the root non-terminal (start symbol); P is a finite set of productions in the form of $p : X_{p0} \rightarrow X_{p1}...X_{pn}$ where $n \geq 0$, $X_{p0} \in N$ and for $1 \leq i \leq n$, $X_i \in (T \cup N)$. Also, $LHS(p) = X_{p0}$ and $RHS(p) = \{X_{p1}...X_{pn}\}$ which are the set of grammar symbols respectively on left and right-hand side of production p.*

$Exp \rightarrow Exp_1 := Exp_2(Assign)$
$\quad Exp_1.code = Exp_1.code||Exp_2.code$
...

$Start \rightarrow His\ LHis\ Stms$
$\quad (His, Stms, LHis).env = Start.env$
$\quad Stms.rhis = ref\ \ to\ \ His.his$

$Exp \rightarrow Exp_1 := Exp_2\ SemanticInstSym\ \ (Assign)$
$\quad Update(Exp_1.definer, Exp_2.definer)$
$\quad (Exp_1, Exp_2, SemanticInstSym).env = Exp.env$

$SemanticInstSym \rightarrow \epsilon$ semantic production
$\quad DetRecI.sec - code =$ "
$\quad if \quad (Exp_1.EnvType = SpecificTyp(material)$
$\quad and\ Exp.EnvRole = SpecificRole(students)$
$\quad and\ Maxtime(exp_1.definer) \in \{...\})$"
$\quad ||Exp_1.code||Exp_2.code||$
$\quad "Exp_1.Time := timetag,$
$\quad Exp.Time := Exp_1.DefTim$"

$\quad Instbool = Instrument\{$
... $\quad SemanticInstSym$ by $DetRecI.sec - code$
$\quad$ through $SemanticInstSym \rightarrow Stms\}$
$\quad if\ !Instbool\ then$
$\quad Error\}$

$SemanticInstSym \rightarrow Stms$ instrumentation semantic
...

Fig. 2. Attribute-based policy enforcement using AGs

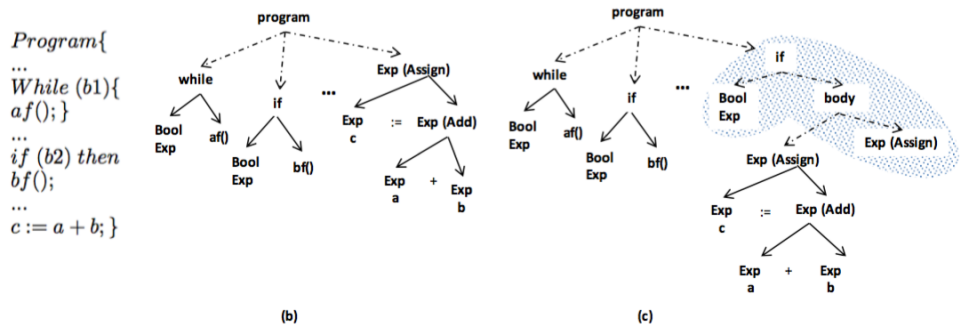

Fig. 3. Syntax tree of the program, (a) before the instrumentation, (b) after the instrumentation.

• *$SD = \{TYPE - SET, FUNC - SET\}$ is a semantic domain. $TYPE - SET$ is a finite set of types. $FUNC - SET$ is a finite set of total functions of type $type_1 \times ... \times type_n \rightarrow type_0$, where $n \geq 0$ and for $0 \leq i \leq n, type_i \in TYPE - SET$.*
• *$AD = (A, I, S, L, TYPE)$ is a description of attributes. Each symbol $X \in V$ has a set $A(X)$ of attributes which can be partitioned into two disjoint subsets $I(X)$ and $S(X)$ of inherited and synthesized attributes. The set of all attributes will be denoted by A which means $A = \bigcup_{X \in V} A(X)$. Furthermore, for each production p, an attribute grammar may define local attributes. Therefore, L^p is the set of local attributes in p.*

An attribute is called synthesized if its value at a syntax tree node N is determined from attribute values at the children of N and at N itself. Moreover, inherited attributes have their value at a syntax tree node determined from attribute values at the node itself, its parent, and its siblings in the syntax tree [30].

An attribute a of symbol X will be denoted by a of X (or $X.a$) when $a \in A$ and $TYPE(a) \in TYPE - SET$ is the type of possible values of a.
• *$R(p)$ is a finite set of semantic rules or attribute evaluation rules associated with production $p \in P$. It is said that production*

p has the attribute occurrence (a, p, k) if $a \in A(X_{pk})$ where $0 \leq k \leq n$ and thus $X_{pk} \in \{X_{p0}...X_{pn}\}$. The set of all attribute occurrences of production p will be denoted by $AO(P)$. A part of $AO(P)$ is called defined occurrences as the following: $DO(p) = L^p \cup \{(s, p, 0)|s \in S(X_{p0})\} \cup \{(i, p, k)|i \in I(X_{pk})$ and $1 \leq k \leq n\}$.

The rules of $R(p)$ specify how to compute the values of attributes occurrences in $DO(p)$ as a function of certain other attribute occurrences in $AO(p)$. The semantic rule defining attribute occurrence (a, p, k) has the form $(a, p, k) := f((a_1, p, k_1), ..., (a_m, p, k_m))$ where $(a, p, k) \in DO(p)$, $f : TYPE(a_1) \times ... \times TYPE(a_m) \rightarrow TYPE(a)$, $f \in FUNC - SET$ and $(a_i, p, k_i) \in AO(p)$ for $1 \leq i \leq m$.

An unambiguous context-free grammar assigns a single derivation tree to each of its sentences. The nodes of derivation trees are labeled with grammar symbols. It is said that production $p : X_{p0} \rightarrow Xp1...X_{Pn}$ is applied or instantiated when an interior node of a derivation tree is labeled with X_{p0} and its children are labeled with $X_{p0}...X_{Pn}$, respectively. In such a case, each grammar symbol X is instantiated and will have instances of all attributes in $A(X)$. The semantic rules of a production apply to all instances of productions too. Therefore, for the applied production p in a derivation tree with an associated semantic rule such as $(a_0, p, k) := f((a_1, p, k_1), ..., (a_m, p, k_m))$ all $a_0, ..., a_m$ denote attribute instances. In both cases of attributes and attribute instances, a_0 is said to be dependent on $a_1, ..., a_m$. A dependency graph $D(T)$ is a directed graph whose vertices are attribute instances in a derivation tree T, and there is an edge from attribute instance a to the attribute instance b if and only if b depends on a and indicates that the value of the attribute instance a must be defined before b. If $D(T)$ is acyclic, its edges specify a partial ordering of attribute instances. *Attribute evaluation* is the process of computing values of all attribute instances attached to a derivation tree according to semantic rules associated with attribute instances. The process that determines the order of attribute instances in an evaluation is the attribute scheduling process that uses $D(T)$.

B. Inlined Reference Monitor

Reference monitors that comprise the most current runtime security enforcement mechanisms, for a given security policy, observe software execution and take remedial actions if an operation violates the policy [6]. There are several kinds of reference monitors. Among them, Inlined Reference Monitors (IRMs) merges the code of the enforcement mechanism into the target software or application code whose activity is to be restricted. IRMs are strictly more flexible than traditional reference monitors for enforcing policies. IRM enforcement code consists of the security state and security code that update the security state. Security codes are inlined program fragments that: 1) maintain a summary of the target execution history in the added security state, as relevant to the policy, and 2) check satisfaction of the security policy and take the remedial action in case of a violation detection. The resulting code is called secured because it is guaranteed not to violate the security policy at runtime.

To insert an IRM as a security enforcement mechanism into a target code, i.e. *instrumentation*, it is necessary to identify places in the application where security-policy-relevant operations may occur at runtime; each such place forms an *insertion point* for the IRM enforcement code. Given a particular security policy, only a subset of operations of a target code may be security-relevant and thus at runtime should comply with the security policy. Effective enforcers adhere the principles below:

- Soundness:
  The IRM must fully mediate all security-policy-relevant operations in the target code. Therefore, observable executions of the secured application by IRM must satisfy the desired property.
- Transparency:
  In the absence of security policy violations, observable behavior of secured application must not be modified.
- Integrity:
  The integrity of the IRM must be protected, either by the IRM itself or by some external means, i.e. the secured application must not be able to circumvent or subvert IRM enforcement code.

IV. Policy Specification and Enforcement Compatible with AGs

To specify and dynamically enforce policies we aim to utiliz the power of attributes and semantic rules in AGs together with supported capabilities in IMAGs. To do so we can also model program state and state transitions, and analyze the execution of program instructions with theie effects on the program state.

Definition 2 (Status)**.** *Each information unit extractable from the whole system state and related to the program execution that i) are affected by some instruction executions, and ii) by its analysis can monitor satisfaction of desired security policy, are called a status.*

Statuses can directly depict information about program memory, e.g. values of program variables, or indicate access information to program variables. They can be extracted by complicated semantic analysis of the programs as well. We model each status by an n-tuple with labeled components in which the number of components, n, depends on the status characteristics while one of them represents the status identity. Moreover, we say a status is *determined* if its tuple with the basic information is prepared and we say a status is *defined* or changed if its other components are assigned values. Some

status components that we call *basic information of statuses* can be defined once, i.e. status identities. The other components of a status can be changed by instruction execution.

Definition 3 (State)**.** *The set of all statuses constitutes the program state.*

At each point of a program execution, contents of statuses show the program state in that point denoted by a unique index, e.g. s_i. To enforce specified policies in programs based on IMAGs, status changes, and thus program state transitions could be monitored. Therefore, extraction of statuses to constitute the program state is one of the main parts, plays an important role in security policy specification and thus in their enforcement by IMAGs because they will be mapped to attributes.

Previously program executions were modeled as sequences of program actions, e.g. in [5], or as sequences of states, e.g. in [36]. In this paper, we model program execution as a sequence of states and instructions to emphasize instruction effects on program states. Therefore, we define Control Flow State Machine as follows.

Definition 4 (Control Flow State Machine)**.** *A control flow state machine Φ for each program is a 5-tuple $(S, start, stop, V, T)$ where S is the set of all possible states, i.e. s_i s, $start, stop \in S$ are the initial and final states, V is the set of all program instructions, and $T \subseteq S \times V \times S$ is the set of all possible one-length one-step state transitions in the program.*

The semantics of instructions determines how they affect program states at execution [37]. For each instruction the associated state with the program point before the instruction, and the associated state with the program point after the instruction are respectively called input and output state of the instruction [30]. Therefore, according to semantics of each instruction, its execution transforms its input state into its output state which can be shown by a one-length state transition such as $s_i \xrightarrow{I} s_{i+1}$ where I, s_i , and s_{i+1} are respectively a particular instruction, the input, and the output states of the instruction.

Definition 5 (Extended State Transition)**.** *Extended state transitions are given by $\Sigma \subseteq S \times V^* \times S$ with the form of $\sigma^n = (s_1, (v_1, ..., v_{n-1}), s_n)$ for $2 \leq n$ if and only if for $1 \leq j \leq n-1$ there exists a sequence of one-length state transitions $(s_j, v_j, s_{j+1}) \in T$. The length of an extended state transition with the above form is $n-1$ and it is also called an (n-1)-length state transition.*

For simplicity, we call every n-length state transition with an arbitrary n, a state transition.

A. Security Policy Specification

To specify security policies on runtime control flow, some relationships between states and state transitions are defined and used here. Let su and sus denote a status identity and a subset of status identities, respectively, where we identify all statuses for each program by their identities in a set which we call $SuSet$. Also, let $st.u.f$ denote the component with label f of the status that is identified by u in the state st.

Definition 6 (State Equivalence)**.** *Let $\sim$ be an equivalence relation on states. We say two states s_i and s_j are $su-equivalent$ with notation $s_i \sim s_j$ if and only if for every $su \in SuSet$, two statuses $sr_m \in s_i$ and $sr_{m'} \in s_j$ with status identities equal to su are equal, i.e. all components of these two statuses are equal that is shown by $sr_m = sr_{m'}$. Also, $s_i \overset{su}{\sim} s_j$ denotes that the above equivalence holds for a specific $su \in SuSet$.*

Definition 7 (Sub-State Transition)**.** *State transition $\sigma'^m = (s_{\sigma'1}, (v_{\sigma'1}, ..., v_{\sigma'm-1}), s_{\sigma'm}) \in \Sigma$ is a sub-state transition of another state transition $\sigma^n = (s_{\sigma1}, (v_{\sigma1}, ..., v_{\sigma n-1}), s_{\sigma n}) \in \Sigma$ if $n \geq m$, and $\exists\ k$, $1 \leq k \leq n-m+1$, such that for each i, $1 \leq i \leq m-1$ and j, $1 \leq j \leq n-m+1$, the next conditions hold:*
i) $(s_{\sigma'i} \sim s_{\sigma k})$, and ii) $(v_{\sigma'i} = v_{\sigma k})$.
Such a sub-state transition is denoted by $\sigma'^m \subseteq \sigma^n$.

Definition 8 (State Transition Equivalence)**.** *Let $\approx$ be an equivalence relation on state transitions of a system. We say two state transitions σ and σ' are $su-equivalent$, denoted by $\sigma \approx \sigma'$, if and only if $\forall s_i, s_j \in (\sigma \cup \sigma')$, $s_i \sim s_j$. Also, $\sigma \overset{su}{\approx} \sigma'$ denotes that the above equivalence holds for a specific $su \in SuSet$.*

Definition 9 (Control Flow Semantics about Status Manipulation (CFSS))**.** *Assuming the set of all statuses, control flow state machine ϕ and the set of all possible state transitions Σ, control flow semantics about status manipulation determine a property on statuses, states and sub-state transitions caused by instructions in every program execution such as $\sigma = (s_0, ..., s_i, v_i, s_{i+1}, ..., s_n) \in \Sigma$ which is shown by $p(\sigma)$.*

By specifying control flow semantics about status manipulation, legal effects of instructions on every status in program states are determined. In addition, by utilizing relationships between state transitions of instructions we can specify security policies.

To describe some examples of CFSS policies, we assume that legal status manipulations by every instruction may be changing or using statuses, and we define *status-change* and *status-use* instructions as follows.

Definition 10 (Status-change Instruction)**.** *An instruction v is a $status-change$ instruction if and only if according to its semantics it makes a change to a non-empty subset of statuses that the set of their identities is denoted by $SuDef(v) \subset SuSet$ in program state. In other words, if $s_i \xrightarrow{v} s_{i+1}$ is the state transition made by v, then $s_i \overset{\forall su \in SuDef(v)}{\sim} s_{i+1}$.*

Definition 11 (Status-use Instruction)**.** *An instruction v is a $status-use$ instruction if and only if according to its semantics it uses information of a subset of statuses in program state such that the set of their identities is denoted by $SuUse(v) \subset SuSet$. In this case, if $s_i \xrightarrow{v} s_{i+1}$ is the state transition made by v, then $s_i \overset{\forall su \in SuUse(v)}{\sim} s_{i+1}$.*

We define Status Integrity, and Status Flow Integrity as two general policies which deal with integrity of statuses along state transitions corresponding every program execution. Moreover, these policies aim is integrity and thus we assume that there is no illegal uses for a status in programs.

Definition 12 (Status Integrity)**.** *The status integrity policy dictates that there should not be any unauthorized status changes in any program execution at runtime. Formally speaking, for all state transition of every program execution $\sigma_{RT} \in \Sigma$ and all $(s_i, v_i, s_{i+1}) \subset \sigma_{RT}, 1 \leq i < n$:*

$$\forall\ su \in (SuSet - SuDef(v_i)),\ \text{then}\ s_i \overset{su}{\sim} s_{i+1}.$$

Definition 13 (Status Flow Integrity)**.** *We define the status flow integrity policy in the form of Time of Legal Change to Time of Legal Use (TLCTLU) dictating that if an instruction in any state transition of a program execution, e.g. $\sigma_{RT} \in \Sigma$, legally changes a status, and another instruction legally uses the same status while it is not legally changed by any other instructions in-between, then this status in program states after the former and before the latter instructions should be the same. In other words:*

If for each specific $su \in SuSet$,
a) there is a $staus - change$ instruction v_i such that $su \in SuDef(v_i)$, and $(s_j, v_j, s_{j+1}) \subset \sigma_{RT}$, and
b) there is v_j such that $(s_j, v_j, s_{j+1}) \subset \sigma_{RT}$, $i+1 \leq j \leq n$ and $su \in SuUse(v_j)$, and
c) there is not any $v_k \in (s_{i+1}, ..., s_j) \subset \sigma_{RT}$ such that $su \in SuDef(v_k)$, then TLCTLU dictates that: $s_{i+1} \overset{su}{\sim} sj$.

By considering different kinds of informative entities as statuses in program states, these policies can describe many other CFSS security policies. For example, if every status reveals information about the program data, then Data Integrity and Data Flow Integrity are two policies that can be defined based on these policies (see Section VI for more details).

B. CFSS Policy Enforcement

A brief description and overview of policy enforcement which is followed in IMAGs to enforce CFSS policies with some examples are presented in this subsection.

To insert an IRM as a security enforcement mechanism into a target code, it is necessary to identify all security-relevant operations, insertion points, and the IRM enforcement code. Given a particular security policy, only a subset of operations in a target code may be security-relevant that should be determined explicitly. In general, in the enforcement of CFSS policies in IMAGs, operations that manipulate statuses in program states are security-relevant. Moreover, it is needed to maintain a summary of the target execution history relevant to the policy in the security state by the security code. The security code and its corresponding insertion points in IMAGs are divided into three major parts that we call *State Update, State Check,* and *State Remedy*.

Although CFSS policies are specified as properties over a sequence of states and instructions, at every point of a program execution at runtime, just the last current actual system state is available. At every such point, the current program state, composed of all statuses, corresponding to the last current actual system state related to the program execution must be extracted. Therefore, any change or use of the current program state implies corresponding manipulations in the current actual system state and vise versa. Hence, we just explain changes or uses of the current program state which is denoted by CS.

A part of the security code in IMAGs aims to save the effects of all legal status changes in a protected security state at runtime. Therefore, programs written in IMAGs should keep track of legal state transitions to dynamically comply with the specified security policy. Hence, the protected security state at every program point, demonstrates the expectation of CS, along the history of all executed instructions at runtime until that point. Let us call such protected state the *expected history state* or for short EHS. With this aim, at compile time IMAGs should statically construct the basic structure of the EHS composed of statuses, determine legal changes to statuses in program states by every instruction, and then, insert the security code to update the expected state by extracting required information for statuses at runtime. This part of the security code in IMAGs is the state update.

In addition to state update, the enforcement code includes specific validity checks, and remedial actions if checks fail which respectively constitute the state check and the state remedy, that are inserted to suitable insertion points in programs. State check uses the current state and the expected history state to check the desired property along the runtime control flow path. If these checks fail, it means that a violation of the CFSS policy is detected, and therefore, an alarm will be raised. Finally, according to the specified policy, the state remedy takes the remedial action. In IMAGs, remedial actions include but are not limited to raising an alarm, recovering the current program state such that the next operations can be executed as if no violation happens, continuing the execution with the user permission, and terminating the secured application.

1) Examples of CFSS Policy Enforcement: As examples of security policy enforcement in IMAGs, let us consider the enforcement of Status Integrity and Status Flow Integrity in Definition 12 and 13. Moreover, assume that statuses in EHS are determined, and $Ins.SuDef$ and $Ins.SuUse$ for each instruction Ins denote the sets of all sus that can be legally manipulated by that instruction corresponding to $SuDef(Ins)$ and $SuUse(Ins)$, respectively. As we will see later, these sets are considered as attributes in IMAGs and are defined in the format of semantic specification and analysis. There might be different insertion points and security code with different assumptions that can be used to enforce these integrity-concerned policies. We present one of these possiblities as follows:

1) An alternative for the insertion point of the state update is after instructions Ins where $Ins.SuDef \neq \emptyset$, in which the state update can be inserted to the target code to update statuses in the expected history state according to the actual and legal changes in the current state that have been made by Ins for all statuses sus in $Ins.SuDef$. Therefore, state update in the IMAGs enforcement code is shown as EHS $\overset{\forall su \in Ins.SuDef}{:=}$ CS.
2) for the state check and its insertion points an alternative is
   After instructions such as Ins where $Ins.SuDef \neq \emptyset$, the correspondence between the current state and the expected history state can be checked for all statuses that according to Ins should not be changed. In other words, the IMAGs enforcement code checks EHS $\overset{\forall su \notin Ins.SuDef}{\sim}$ CS.
3) An alternative for the state remedy and its insertion point is after the state check for an instruction Ins, the state remedy security code can recover the current state according to the expected history state in case of detection of an inconsistency between the same statuses in these two states. In other words, the IMAGs enforcement code will recover those sus in the CS according to the same sus in the EHS at runtime which is shown as CS $\overset{su}{:=}$ EHS.

The combinations of these alternatives as a security code can detect Status Integrity violations and recover the current state afterwards. In addition, it can prevent Status Flow Integrity violations when instructions that are legally changing some sus have a vulnerability of changing the others as well. It is because, the state update insertion point is placed after each instruction, e.g. Ins where $Ins.SuDef \neq \emptyset$. State update in such points of every target program updates the expected history state according to the legal changes in the current state that have legally been made by those instructions. Moreover, the state check insertion point is placed after such instructions and checks the correspondence between the current state and the expected history state for all statuses that according to Ins should not be changed. And finally the insertion point for the state remedy is after the state check. The state remedy security code can recover the current state according to the expected history state in case of detection of an inconsistency between the same statuses in these two states.

In Section VII-C it is described how to check the effectiveness of an IMAG for dynamic enforcement of a CFSS policy. Moreover, to show the application of IMAGs for security policy enforcement, in Section VI, this explained security code is used and its effectiveness is proved.

It is noticeable that the overhead of each of these approaches depends on the number of instructions that will be inserted and the time and space consumption of the added security code. Therefore, choosing the optimum approach for an underlying IMAG depends on characteristics of written programs, assumptions and the specified security policy. In general, although approaches based on runtime checks increase overhead and runtime execution, it is realistic to use them especially in security-critical systems [38] as we can see in almost all works in this area.

V. Inline Monitoring Attribute Grammar (IMAG)

Inline Monitoring Attribute Grammars (IMAGs) establish a foundation in which semantic specification and analysis use attributes and semantic rules statically to a) extract the basic structure and information of statuses in the program state, b) determine instructions legal effects on the program state, and c) insert the IRM to the target code at insertion points by utilizing attributes and semantic rules to dynamically enforce the specified policy at attribute evaluation time. The inserted IRM monitors status manipulations and checks satisfaction of a property p corresponding to the CFSS policy to enforce it dynamically at runtime. IMAGs are introduced after describing its requirements while comparing to existing extensions on AGs.

A. IMAGs Requirements and Existing AGs

Preparing a general foundation to specify and automatically enforce CFSS policies at runtime, IMAGs should be suitable to define the corresponding security state and security code. Therefore, IMAGs should support the following features which we describe in more details here, and in an example in Section VI:

1) extendable structured attributes and their partial definitions,
2) conditional semantic rules,
3) efficiently writable attributes by a specific set of productions,
4) semantic-based instrumentation at attribute evaluation time, and
5) protected and reusable runtime attributes.

Some of these requirements are available, or might be achievable by providing certain conditions and algorithms to some existing extensions of AGs. In this section, we describe how supported capabilities by those extensions does not totally meet the

above requirements, and they may unnecessarily cause complexity in the specification, and circularity, infinity or undecidability of these features in the evaluation. Although we utilize capabilities of some existing AGs, to make the above requirements quite clear and prevent any ambiguity, we define IMAG.

1) Extendable Structured Attributes: Let us call productions of every security-relevant operation, *security-relevant productions*. Each security-relevant production should have attributes and semantic rules to determine its set of legally manipulated statuses, e.g. $SuDef$ and $SuUse$. These attributes that are sets of tuples can be modeled as sets of records. Therefore, some attributes that we call *Structured Attributes* need to have such composite types and thus should be supported in IMAGs. Therefore, EHS, in which basic information about statuses is gathered, is modeled as a structured attribute. This information should be extracted at attribute evaluation time through semantic rules of all security-relevant productions which are instantiated in the program syntax tree. Each status, as an element of these structured attributes, may be defined by a different semantic rule and thus IMAG need to support extendable structured attributes with the ability of their partial definition. This feature is supported in many extensions of AGs, and we assume the same support in IMAG.

2) Conditional Semantic Rules: Another requirement in IMAGs is to have conditional semantic rules. For example, it is needed to have some boolean attributes to determine if the instrumentation is required for production rules corresponding to each program part. For an instance, the enforcement code is usually guaranteed not to have any vulnerability against the specified security policy. Therefore, for production rules related to the enforcement code, these attributes must be set to false. Hence, such attributes may be switched to on or off in subtrees related to each part of the code during semantic analysis to activate and deactivate the policy enforcement. If such attributes for a specific grammar symbol entails the instrumentation, then according to the semantic rules, the corresponding node of that symbol in the tree can be expanded to instrument the code by a derivable sentence from that symbol. Hence, IMAGs support conditional semantic rules, and thus conditional syntax tree expansion at attribute evaluation.

There are some works that somehow support such concepts such as [35], [26], [39]. In [35], a conditional structure for semantic rules and productions called semantic rules block is introduced with two forms. The first form of semantic rules block is a production with a set of unconditional rules, and its second form is a triple consisting of a boolean condition and two other semantic rules blocks. The latter form shows conditional semantic rules. We simplified this definition in IMAGs by not including the production itself in the semantic rule block, and just including a set of semantic rules under a condition which may be a boolean function on some attributes. We call this block of semantic rules which are evaluated under a condition a *Conditional Semantic Rule Block*. In [39] semantic conditions are used which are boolean expressions, and enforce the semantic requirements of a programming language such as a variable must be declared before being used in the entire syntax tree. These semantic conditions must be evaluated to true in every legal syntax tree. Therefore, these kinds of conditions are different to our required conditional semantic rules. Moreover, Boyland [26] introduces conditional attribute grammars in which semantic rules may be guarded, and a rule may be evaluated if its guard is satisfied. IMAGs uses a boolean attribute to do the same as the Boylands to support conditional semantic rules.

3) Efficiently Writable Attributes by a Specific Set of Productions: In IMAGs, it is required to have attributes that are efficiently accessible to be written by semantic rules of instances of the same or different specific productions that can be far from each other in the syntax tree. For example, to gather information of statuses, the attribute for the expected history state should be accessible for more than one production. It is because each status can be related to different instances of the same or different productions in the syntax tree. In IMAGs attributes that collect basic information of statuses are not used at attribution and just are defined to be used at runtime. Since in the security point of view read access and write access are totally different features, to control the effect of attribute accesses for CFSS policies based on IRM, IMAGs just need to support writable attributes at attribution.

To have such attributes, one approach in standard AGs is to define an attribute for the start symbol of the grammar, and then collect all required information as different statuses. To do so, it is needed to propagate this attribute many times which is not suitable and efficient especially in specification and enforcement of security policies because this propagation involves attributes of all intermediate syntax tree nodes even those that are not security-relevant and thus does not provide any information about statuses. Moreover, it can cause error-prone situations especially in the case of extending an IMAG with new productions that intermediate others, or in the management of such attributes by adding or omitting statuses, e.g. in case of any changes in the security policy, or in case of enforcement of more than one security policy. Therefore, this solution will increase complexity and make trustworthy and evaluation of the approach debatable.

Another suitable approach to have such attributes is supporting non-local dependencies in the underlying AG. Non-local dependencies between attributes occur when an attribute of one symbol in the syntax tree is dependent on an attribute of another symbol whose node is far away in the syntax tree. In standard AGs, with respect to syntax tree nodes, all attribute dependencies are local which is not suitable to maintain and update accessible attributes for many parts of a tree. However, non-local dependencies in AGs are supported in various ways that usually have capabilities which not only are unnecessary in IMAGs but also may cause challenges with IMAGs security purposes or impose unnecessary complexities in its evaluation. For example, consider the following most related works to this feature:

- Reference Attribute Grammars [21] allow attributes to be references to other nodes arbitrarily far away in the syntax tree. As a result, all attributes of those nodes can be accessed via such reference attributes. Instead, in IMAGs a limited

access, which is write to certain attributes and not others, is needed between certain attributes belonging to some nodes far away in the syntax tree. This limited capabilities help to prevent unnecessary accesses and simplify dependency graph and thus evaluation of IMAGs. Moreover, since the dependency graph for an RAG can not be completely determined before evaluation, it has to be determined during the evaluation which restrict RAGs evaluation to demand-driven evaluation algorithms. However, in IMAGs dependencies can be determined before attribute evaluation, as described in Section VII-A, which simplifies attribute evaluation.

- Based on the object-oriented view of attribute grammars, some extensions to RAGs are provided such as object-oriented reference attributed grammars (ORAGs) [21]. IMAGs do not require capabilities in ORAGs, e.g. inheritance, which may also cause complexities in accessibility of attributes in IMAGs.
- There are collection attributes defined in [40] that can be accessible by all productions and have a starting value and a combination function which is used to form a final value from all their definitions that might be useful in IMAGs. There is an extra limitation at evaluation of these collection attributes such that any use of them will not be scheduled until all definitions have been combined into a final value. However, instead of a global attribute and total access to it, in IMAGs just security-relevant productions required to have the write access to a special attribute.
- The LIGA attribute grammar system [41] supports some features such as semantic rules that can refer to an ancestor node. In IMAGs, an attribute of an unrelated node must be accessible to a specific attribute instead of all attributes of its ancestor. Moreover, providing this capability in LIGA needs some additional control attributes which their sufficiency can not be always determined, and increase dependencies between attributes which makes the evaluation more complicated.
- Remote Attribute Grammars [42] generally allow attributes that refer to other attributes in nodes arbitrarily far away in the syntax tree. However, it is possible in these AGs to read and write attributes through references. Both of these capabilities make dependency graph more complicated and cause circularity in these AGs undecidable.

IMAGs efficiently support writable attributes by a set of productions through their references by introducing *Reference-to-Write Semantic Rules*, and *(Local & Non-local) Reference-write Dependencies*. Moreover, to support this feature, IMAGs do not impose any extra cycle, any limitation in evaluation algorithm, and any unnecessary attribute accessibility.

4) Semantic-Based Instrumentation at Attribution: Attributes keep semantic information of grammar symbols and are defined at semantic analysis during attribution. In standard AGs, the priority is totally given to syntax and syntax tree construction than semantics and semantic analysis through attributes, and attributes are mostly used to diagnose or reject existing syntax after attribution.

To instrument the code in IMAGs, first the security code does not exist in the program main code to trigger derivation and syntax tree expansion before evaluation. Second, we want to instrument the code based on semantic information extracted into attributes. In other words, in IMAGs attributes which are defined in the semantic analysis of the target code are also used to determine the enforcement code or some parts of it, and to embed this code in the target code. For an instance, to enforce the specified integrity-concerned policies in Section IV-A, the security code in most of described alternatives in Section IV-B1 is determined using attributes such as $SuDef$ or $SuUse$. Moreover, syntax tree nodes of the security code that extend the current tree are themselves grammar symbols which may have attributes that must take part in the semantic analysis. As a result, for instrumentation every target program, it is needed to expand its syntax tree at attribution. In other words, some syntax tree nodes need to be computed and grafted to necessary places of the generated syntax tree. Moreover, the security code should correspond to the underlying context-free grammar, i.e. it should be drivable from the grammar. Therefore, for instrumentation purpose in IMAGs, the priority is given to semantics than syntax, and attributes are able to guide the syntax during and after attribution instead of just rejection or diagnosing the existing code. Hence, for the purpose of instrumentation, the definition and translation of IMAGs is semantic-directed.

In the sense of changing syntax tree, IMAGs may seem similar to some works such as [43], [44], [20], [45] which are based on abstract syntax tree rewriting. In these works, rules are used to specify how to transform a part of an abstract syntax tree to another form. The main difference between these works and IMAGs lies between typical purposes and applications of these systems that are code optimization and re-engineering of the program source code which results in equally rewriting abstract syntax tree from derivation point of view. For example, in syntax tree rewriting in [20], the first and the second syntax trees are equal in all features and correspond the same code except that the latter simplifies compilation. In IMAGs, because of the embedded security code that enforces a specified policy at runtime, syntax trees before the attribute evaluation corresponding to the main target code are not equal to the trees corresponding to the secure code after the attribute evaluation.

IMAGs have a common feature with some other works such as HAGs [22] and their extensions, e.g. [46], [47]. This common feature is the tree expansion at attribute evaluation time. HAGs want to use attributes to guide the syntax before attribution and to do so, some attributes are defined as non-terminals as well, and are called non-terminal attributes (NA).

However, HAGs attempt to remove the difference between syntax and semantics such that the boundary between syntax tree nodes, which are grammar symbols, and attributes is totally disappeared. Moreover, HAG implies that production rules of a non-terminal attribute are also the type specification of that attribute, i.e. $p(NA) \subset TYPE - SET$. Further, in HAGs production rules and thus attribute types can be recursive. Therefore, the existence of a non-terminal attribute which has an attribute of its type may create a case of infinity. Moreover, due to these characteristics, HAGs are not cycle free and finite tree expansions is undecidable. Furthermore, although NAs can be useful for instrumentation, effects of the above complicated

characteristics on security is debatable while IMAGs do not benefit from them all. Therefore, for the sake of security which its reasoning is complicated IMAGs that do not need to support HAGs capabilities which are useful in other sophisticated areas. Instead IMAGs allow semantic-based instrumentation at attribute evaluation through new semantic rules called *Instrumentation Semantic Rules* while are non-circular and guarantee to terminate.

5) Protected and Reusable Runtime Attributes: Another requirement in IMAGs is to have some attributes, e.g. EHS, that can be defined at attribution and maintained in a protected part to be manipulated by security code through a secure communication at runtime. These attributes constitute the program security state. Basic information about these attributes is saved in a protected part at attribution for latter use at runtime. At runtime, security code retrieves information in these attributes to trace its legally changeable parts by program operations to dynamically guarantee satisfaction of the specified security policy.

In general, retaining some information respectively at compile time and runtime are not new. For example, symbol table as a data structure is generated and used by compilers to keep track of some information about programs and might be retained after compiling and made available at runtime, e.g. in [39], [32]. Moreover, a lot of works such as [16], [48], [12] with the aim of runtime policy check are based on using a protected part to save some information about programs to use at runtime, especially in inline monitoring [6]. In practice, the machine model established in [49], [50], [51] can provide such a protected part. In this machine model, there is a protected memory such that other parts of the program resided out of the protected part have an efficient stream cipher designed to communicate with this protected part while any other manipulations including malicious ones to such part is prevented. IMAGs defines *protected live attributes* to support such reusable runtime attributes.

B. Inline Monitoring Attribute Grammar (IMAG)

Defining some preliminaries in this section, we introduce the IMAG formalism. Let us call a sequence of semantic rules a *Semantic Rule Block*. Moreover, if all rules in a semantic rule block are performed under satisfaction of the same condition, then let us call it *Conditional Semantic Rule Block (CSRB)*. In other words, a sequence of semantic rules of a production p denoted by $CSRB_{Cond}$ is a Conditional Semantic Rule Block if all its rules are performed under satisfaction of a condition named $Cond$. The condition itself is a semantic rule of the form $f((a_1, p, k_1), ..., (a_n, p, k_n))$ and type $type_1 \times ... \times type_n \rightarrow \{true, false\}$, where $n \geq 1$, $(a_i, p, k_i) \in AO(p)$ and $TYPE(a_i) = type_i$ $(0 \leq i \leq n)$, $true$ and $false \in TYPE - SET$ of the underlying AG. A conditional semantic rule block with an empty condition is just a semantic rule block. Also, all defined attributes in semantic rules in a $ICSRB_{Cond}$ are dependent on attribute occurrences of the semantic rule that defines the condition.

In addition to productions in context-free grammars, IMAGs support other productions with the same style which we call *Instrumentation Semantic Production*. Instead of being directed by syntax, these new productions are directed by semantics at attribution to instrument the code. Moreover, in addition to semantic rules in AGs, IMAGs support other semantic rules which trigger the instantiation of instrumentation semantic productions resulting syntax tree expansion at attribution. We call these semantic rules "Instrumentation Semantic Rules".

Definition 14 (Instrumentation Semantic Production)**.** *A production rule is an Instrumentation Semantic Production p_{IS} if and only if its derivation can only occur at attribution during execution of a semantic rule.*

Definition 15 (Instrumentation Semantic Rule)**.** *An Instrumentation Semantic Rule, r_{IS} for a production p is a function shown as f_{ISR} defined by the following general form:*

$$\begin{aligned} \alpha = & \textit{ Instrument } \{ \\ & X \textit{ by } X.InstruStr \\ & \quad \textit{through } p_{IS} \} \end{aligned}$$

where $X = LHS(p)$, $InstruStr \in S(X)$, $TYPE(InstruStr) = string$ while $InstruStr$ is an attribute storing security code, p_{IS} is an ISP such that $LHS(p_{IS}) = X$, and $\alpha \in L^p$ where $TYPE(\alpha) = boolean$. Therefore, the type of the instrumentation semantic rule is $string \rightarrow boolean$ with attribute occurrence $(\alpha, p, X) = (f_{ISR})(InstruStr, p, X)$ that will extend the derivation tree and the syntax tree at attribution if according to p_{IS} the code string in $InstruStr$ is derivable from X. Therefore, by evaluation of a r_{IS} , the inlined security code will be considered in the rest of syntax and semantic analysis of the program main code.

According to the dependency graph definition, it is clear that α depends on $X.InstruStr$. The evaluation of α leads to execution of the instrumentation semantic rule. If according to p_{IS} , $X.InstruStr$ is derivable from X, then α will be evaluated to true, and in this case, all attributes of the new subtree, related to symbols in p_{IS} , will then be entered into the evaluation process. Therefore, all these new attributes depend on α.

Definition 16 (Semantic Production)**.** *A Semantic Production p_S is a production in the form of $p_S : x \rightarrow \epsilon$ such that $R(p_S)$ contains instrumentation semantic rules.*

Although non-terminals in AGs are motivated from the context-free syntax, IMAGs also support different non-terminals that we call *Instrumentation Semantic Non-terminals* which their application is determined based on semantics. These non-terminals have one semantic production and a number of instrumentation semantic productions.

Definition 17 (Instrumentation Semantic Non-terminal)**.** *A non-terminal is an Instrumentation Semantic Non-terminal shown by X_{ISN} if the following conditions are satisfied:*

1) *there is just one production rule with the form of $p_S : X_{ISN} \rightarrow \epsilon$ in the grammar,*
2) *p_S has at least one CSRB containing an instrumentation semantic rule such as f_{ISR} with the form of:*

$$\begin{gathered}\alpha\text{= Instrument } \{\\ X \text{ by } X_{ISN}.InstruStr\\ \text{through } p_{IS}\}\end{gathered}$$

According to the above definitions, p_{IS} is an instrumentation semantic production such that $LHS(p_{IS}) = X_{ISN}$. Moreover, derivation from an instrumentation non-terminal before attribution is just possible through a semantic production. Using evaluated attributes at attribution time, the security code is constructed in an attribute. Then, through an instrumentation semantic rule in the semantic production, the target code is instrumented by the security code corresponding to a derivation of an instrumentation semantic production. In other words, each instrumentation semantic non-terminal x_{ISN} uses conditional semantic rule blocks to instrument the code according to attributes of production instances in the program context. Therefore, derivation from an instrumentation non-terminal in an instrumentation semantic production is not triggered by any sentence of the main target code.

As explained in IMAGs requirements, just writing access for some attributes is sufficient at attribution. Moreover, protected attributes that could be reusable at runtime is needed in IMAGs. Therefore, we define *Write-access* and *Protected Live Attributes* as follows.

Definition 18 (Write-access Attribute)**.** *An attribute $\overleftarrow{w}$ is a write-access attribute if it can be just defined with no use at attribution.*

Definition 19 (Protected Live Attribute)**.** *An attribute $\|a\|$ is protected live if it is stored in a protected part at attribution, and it is retrieved and manipulated at runtime.*

Based on the above definition, protected live attributes have two liveness phases: attribution and runtime. Example of these attributes is the expected history state or EHS such that IMAGs statically construct the basic structure of the EHS composed of statuses, determine legal changes to EHS statuses by every instruction at attribution, and then, the inserted security code at runtime, i.e. state update, updates EHS by extracting required information for statuses. Moreover, this attribute is a write-access attribute at attribution because it is just defined during attribution. Therefore, we denote this attribute with the form of $\|\overleftarrow{EHS}\|$.

In addition to basic types for attributes, IMAGs may have composite types such as sets and records to support structured attributes. Each structured attribute may consist of some other structured attribute with the same or different types. For example, an attribute that is a set can store attributes with the same type, without any particular order, and no repeated values. In addition, each attribute has an address identity which could be referred to. Moreover, there is a reference type such that an attribute with this type can store a reference to another attribute. We call an attribute with the reference type a *Reference Attribute*. A reference attribute can access the stored value in the referred attribute. Moreover, we precede the identifier of a reference attribute with an asterisk (*) which acts as dereference operator and is a known notation in C++.

Definition 20 (Reference-to-Write Semantic Rule)**.** *A semantic rule of the form $v = ref\ to\ w$ is called a reference-to-write semantic rules where w and v are attributes such that v stores a reference to the attribute w to totally or partially define it, and the type of such a reference rule is $TYPE(w) \rightarrow reference\ to\ TYPE(w)$ such that $TYPE(w)$ and $reference\ to\ TYPE(w)$ are in TYPE-SET of the underlying AG while w is a write-access attribute.*

In a reference rule with the above form, it is said that v directly refers to w. In such a case, any change to the value of a referred attribute w can be made by the same change to $*v$ and vice versa.

According to the preliminary definitions above, IMAGs are introduced as follows.

Definition 21 (Inline Monitoring Attribute Grammar (IMAG))**.** *An inline monitoring attribute grammar is a tuple IMAG=(IMG(N', T, Z, P'), IMSD(Γ, F), IMAD(A, S, I, W, P_L, R, L, τ), IMR(R, ISR)) where:*

- *IMG=(N', T, Z, P') is a context-free grammar in which:*
  - *N' is a set of non-terminals and instrumentation semantic non-terminals $ISN = \{x_{ISN} | x_{ISN}$ is an instrumentation semantic non-terminal, $|ISN| \geq 1\}$. Therefore, $ISN \subset N'$.*
  - *T is a finite set of terminal symbols, $N' \cap T = 0$ and $N' \cup T = V$ which denotes all grammar symbols.*
  - *$Z \in (N' - ISN)$ is the root non-terminal (start symbol).*
  - *P' is a set of productions in the form of $p : X_{p0} \rightarrow X_{p1}...X_{pn}$ where $n \geq 0$, $X_{p0} \in N'$ and for $1 \leq i \leq n$, $X_{pi} \in (T \cup N')$. Also, P' includes semantic productions and instrumentation semantic productions. Therefore, there is a non-empty set $SP \subset P'$ such that $SP = \{p_S | p_S$ is a semantic production and $|SP| \geq 1\}$ and a non-empty set $ISP \subset P'$ such that $ISP = \{p_{IS} | p_{IS}$ is an instrumentation semantic production and $|ISP| \geq 1\}$.*
- *IMSD=($FUNC - SET(F)$, $TYPE - SET(\Gamma)$) such that $\{\Gamma_{st}$, string, boolean (false, true), $\Gamma_{ref}\} \subseteq \Gamma$ where:*

- ∘ Γ_{st} *and* Γ_{ref} *determine the type domain of structured and reference attributes.*
- ∘ *false, true, and string respectively are domains for conditions and instrumentation code.*
- ∘ *Similar to standard AGs,* F *is a finite set of total functions including reference rules and instrumentation semantic rules.*

• *IMAD=*$(A, S, I, W, P_L, R, L, \tau)$ *is a description of attributes. Each symbol* X *in the set of grammar symbols* V *has a set* $A(X)$ *of attributes which can be partitioned into two disjoint subsets* $I(X)$ *and* $S(X)$ *of inherited and synthesized attributes that have the same definitions as in standard AGs. Moreover,* $W(X) \subset (I_W(X) \cup S_W(X))$ *is the set of write-access attributes for a symbol* X *which is a subset of the set of inherited or synthesized attributes that are just definable during attribution. Similarly,* $P_l(X) \subset (\|S(X)\| \cup \|I(X)\|)$*, is the set of protected live attributes and are maintained in a protected part and retrieved at runtime. In addition,* $R(X) \subset (I(X) \cup S(X))$ *is the set of reference attributes which are attributes with reference type that refers to another attribute of a grammar symbol. Moreover,* L *is the set of local attributes in all productions. Again, the set of all attributes is denoted by* $A = \bigcup_{X \in V} A(X)$*. For* $a \in A$*,* $\tau(a) \in \Gamma$ *is the type of possible values of* a*.*

The set of all attribute occurrences for production $p : X_{p0} \rightarrow X_{p1}...X_{pn}$ *that are directly defined is denoted by* $DO(p)$*. It consist of synthesized attributes of* X_{p0} *, inherited attributes of* X_{pk} *, local attributes of* p *in addition to reference attributes of* X_{pk} *, i.e. while* $1 \leq k \leq n$*,* $DO(p)$ *is according to the follows:*

$DO(p) = L^p \cup \{(s, p, 0) | s \in S(X_{p0})\} \cup \{(i, p, k) | i \in I(X_{pk})\} \cup \{(r, p, k) | r \in R(X_{pk})\}$.

Also, the set of attribute occurrences for production p *that are indirectly defined is the content of attributes to which the reference attributes of* X_{p0}*. This set for each production* p *is denoted by* $DOR(p)$ *which is shown as* $DOR(p) = \{(*r, p, 0) | r \in R(X_{p0})\}$*. Moreover, the set of all attribute occurrences for a production* p *that may be used is the set of inherited attributes of* $LHS(p)$ *and synthesized attributes of the rest of symbols in* p *except for the contents of write-access attributes, i.e.:*

$UO(p) = \{(i, p, 0) | i \in (I(X_{p0}) - *I_W(X_{p0}))\} \cup \{(s, p, k) | s \in (S(X_{pk}) - *S_W(X_{p0}))$ *and* $1 \leq k \leq n\}$

• *IMR(*R, ISR*) is a description on semantic rules.* $R(p)$ *is a finite set of semantic rules and conditional semantic rules associated with productions* $p \in P'$*. Moreover,* $ISR(p) \subset R(p)$ *such that* $ISR(p)$*=*$\{f_{ISR} |$ f_{ISR} *is an instrumentation semantic rule in* $R(p_S)\}$ *, and ISR is the set of* $ISR(p)$ *for all* $p \in P'$*.*

Definition 22 ((Local & Non-local) Reference-write Dependency)**.** *A dependency between two attributes* v *and* w *in IMAGs is called a reference-write dependency when there is a reference-to-write rule such as* $v = ref\ to\ w$ *in semantic rules of a production* p *where* v *is defined and address of* w *is used. Since both of* v *and* w *belong to attribute occurrences of the same production, we say* v *is local reference-write-dependent to* w*. Moreover, in any copy rule* $v' = v$ *in a production* p'*, if* v *has a reference type, then* v' *should have the same type. In such a case,* v' *is locally dependent on* v *and also there should be a chain of copy rules, e.g.* $v' = v$*, to copy the address of an attribute started by a reference rule, e.g.* $v = ref\ to\ w$*. In such cases, we say* v' *indirectly refers to* w *and non-local reference-write-dependent to* w*.*

Since a reference rule $v = ref\ to\ w$ is a function on w that defines v, and a copy rule such as $v' = v$ is a function on v that defines v', the dependency between these attributes can be demonstrated in the same way as standard AGs. Therefore, construction of dependency graphs in IMAGs is the same as in standard AGs.

To ensure about the access write to attributes in W a simple algorithm can check that there is no semantic rule which accesses an attribute $w \in W$ or the content of a direct or indirect reference attribute to w on its right-hand side. Similarly, it can be checked by a simple algorithm that just for semantic rules of the certain set of productions such as security-relevant productions w and $*v$ can be defined in the left-hand side of semantic rules.

C. IMAGs Pattern

According to the security policy enforcement in *IMAGs* described in Section IV-B, we present a common pattern to specify languages that their written programs can guarantee satisfaction of a specified CFSS policy at runtime. Moreover, according to this pattern we describe an example in Section VI that exemplified this pattern with more details. Assuming an IMAG=(IMG(N', T, Z, P'), IMSD(Γ,F), IMAD=($A, S, I, W, P_L, R, L, \tau$), IMR($R$, ISR)) with a production $p : Z \rightarrow X_{p1}...X_{pn} \in P'$ for the root non-terminal, the IMAG pattern is as follows.

1) According to the CFSS security policy all security-relevant productions, insertion points and the enforcement code should be determined, e.g. see the sample policies described in Section IV-B1.
2) An approach that supports the protected part and its secure interface should be determined.
3) The grammar should have an input flag which we call it the *instrumentation flag* to turn the program monitoring on and off. Therefore, according to this flag, production rules in IMAGs, especially security-relevant productions, may have different sets of semantic rules which will be appeared as different CSRBs. Such a flag can be considered as an inherited attribute of the start symbol supplied as an initial value before attribute evaluation begins.
4) There should be a non-terminal $X_{pi} \in RHS(p)$ with an attribute which is used to collect basic information of statuses through semantic rules at attribute evaluation time that we call it the *history attribute*. This attribute should be a write-access and protected live attribute. Therefore, this attribute should store a set of statuses. After semantic analysis and finalizing this information at compile time, it will be saved in the protected memory by compiler. Later, through the

enforcement code, this information will be used and manipulated to provide the expected history state of the program at runtime.

5) $R(p)$ should include some semantic rules to start a flow of the instrumentation flag and a flow of a reference to the history attribute down the tree. Moreover, semantic rules of other productions may continue the flow of these attributes down the tree by copy rules.
6) There should be a non-terminal $X_{pj} \in RHS(p)$ that instruments the program to securely connect to the basic information of the history attribute at runtime if the instrumentation flag is on.
7) According to the specified policy, for every security-relevant productions such as p':
   a) $R(p')$ should include some semantic rules to insert statuses into the history attribute directly or through a reference to it.
   b) $R(p')$ should include semantic rules to extract the sets of statuses which might legally be manipulated through instructions corresponding these productions. These sets are defined as attributes which are used for enforcement of the desired policy.
   c) According to insertion points and the enforcement code, $RHS(p')$ should include instrumentation non-terminals $x_{ISN} \in ISN$ to embed the security code in the program.
8) For every instrumentation semantic non-terminal, $x_{ISN} \in ISN$, there should be a semantic production $p_S : x_{ISN} \rightarrow \epsilon$ that contains semantic rules to perform required instrumentation. In other words, p_S should have a CSRB that defines the security code and an instrumentation semantic rule that embed the security code in the target code.
9) IMAGs should use the safe enforcement code that have no vulnerability against the specified security policy or at least the effects of the enforcement code on the security policy should be handled by an external mechanism. In both of these cases, semantic rules in $R(p_{IS})$ for each $p_{IS} \in ISP$ from which such safe code is derivable, should start a new flow of the turned off instrumentation flag in subtrees related to the safe code.
10) Generating instrumented intermediate code, IMAGs are code generator AGs too. Therefore, IMR should include semantic rules for code generation.

VI. Data Flow Integrity IMAG

In Section 3, we defined Status Integrity and Status Flow Integrity, in Definition 12 and 13 as examples of CFSS policies that by considering different information in the program states present definition of different integrity-concerned policies. For example, considering statuses as records with some fields including variable name as status identity, and value, e.g. $< var, [var], ... >$, the program state demonstrates information of program data, $SuSet$ includes a set of sus which are variable names. Moreover, assume that $\sigma_{RT} = (s_0, ..., s_i, v_i, s_{i+1}, ..., s_n)$ is the state transition of a program execution, and according to the semantics of every instruction v_i , $SuDef(v_i)$ denotes a specific set of variables that v_i defines, i.e. v_i assigns them a value. Moreover, $SuUse(v_i)$ is the set of variables that are used by v_i , then according to definitions of Status Flow Integrity and Status Integrity, they respectively present Data Flow Integrity (DFI) and Data Integrity (DI). Therefore, the TLCTLU case of DFI and DI are defined as follows while other cases of DFI can be defined similarly:

Definition 23 (Data Flow Integrity: Data TLCTLU)**.** *If for each variable $var \in SuSet$ the next conditions are satisfied:*

a) there is an instruction v_i such that $(s_i, v_i, s_{i+1}) \subset \sigma_{RT}$, v_i defines var, i.e. $var \in SuDef(v_i)$,
b) there is an instruction v_j such that $var \in SuUse(v_j)$, $(i < j)$ $(s_j, v_j, s_{j+1}) \subset \sigma_{RT}$, and
c) there is not any $v_k \in (s_{i+1}, ..., s_j) \subset \sigma_{RT}$ such that $var \in SuDef(v_k)$,

then the TLCTLU case of Data Flow Integrity, which we call Data TLCTLU dictates that: $s_{i+1} \overset{var}{\sim} sj$.

Definition 24 (Data Integrity (DI))**.** *The data integrity security policy dictates that there should not be any unauthorized variable changes in any program execution at runtime. This formally means that for all state transition of every program execution $\sigma_{RT} \in \Sigma$ and all $(s_i, v_i, s_{i+1}) \subset \sigma_{RT}, 1 \leq i < n$:*

$$\forall\, var' \in \{SuSet - SuDef(v_i)\} \text{ , then } s_i \overset{var'}{\sim} s_{i+1}$$

DI and TLCTLU case in DFI are security policies that their satisfaction is necessary almost always unless in isolated environment that no security attack may occur. Violations of these policies usually happen due to exploiting software vulnerabilities such as buffer overflow or format strings that overwrite an arbitrary memory location which make some unauthorized changes in value of some program variables. Therefore, vulnerability-based mitigation techniques such as elimination of buffer overflows [52], program protection from format string attacks [53], and many other techniques are useful to prevent these attacks. Nevertheless, all vulnerabilities that can cause such violations are not mitigated [11], [54], [12], [55]. Moreover, every vulnerability-based technique is not applicable for other existing and newly detected vulnerabilities, while the policy enforcement by IMAGs is independent on any specific vulnerability.

Attacks that violate DFI can cause subversion of the program execution to an arbitrary program path [16], [54] which is a Control Flow Integrity (CFI)[11] violation. For example, to arbitrarily change the program control flow, attackers can exploit some vulnerabilities to overwrite critical memory locations such as return addresses or program variables that are, in fact,

violations from DFI. A few live exploits that violate DFI and may also cause CFI violations by arbitrary changing a program data memory are included but not limited to the widely used `SSH` by overwriting an authenticated variable [16], the most widely used `FTP` server `WU-FTPD` by exploiting user identification data [56], and `Sendmail` [48] and `Telnetd` [48] by exploiting a configuration data. Therefore, DFI satisfaction helps to reduce violations of these policies too. Programs are often subject to such attacks and combined effects of these attacks make them one of the most important challenges in computer security [11], [57].

DI and DFI enforcement in languages with more sophisticated features need more complicated semantic specifications and further analysis, such as reaching definition or pointer analysis, which can be considered as future works. In this section, we present some parts of a small and simple language that we call DFI-IMAG to show an application of IMAGs that enforces these policies and also prevents CFI violations if they are results of DFI violations.

A. DFI-IMAG

To illustrate important concepts of IMAGs, a part of the specification of DFI-IMAG is described in Definition 25 which explores issues that arise during the specification, translation and instrumentation of simple expressions and statements. Moreover, we ignore typing rules to make the example easier to follow and thus we assume that all variables and expressions are integers. This definition is according to IMAGs pattern.

In this section, we use the enforcement code explained in Section IV-B to enforce DFI and DI with the same assumptions and results. Therefore, we assume that instructions having some authorized manipulations on variables might have a vulnerability causing unauthorized changes in other variables. In other words, instructions that may have a non-empty set of $SuDef$ are assumed security-relevant. Hence, in DFI-IMAG attribute $SuDef$ is computed for instructions and after each instruction, e.g. Ins where $Ins.SuDef \neq \emptyset$, state update is inserted to the target code such that it updates the set of vars in the expected history state, according to the actual and legal changes in the current program state that has been made by Ins to variables in $Ins.SuDef$. Then, for all vars, the correspondence between the current state and the expected history state is checked by state check. If the instruction Ins has made an unauthorized change to the value of any other variables, i.e. $var \notin Ins.SuDef$, then an inconsistency is detected. Moreover, state remedy security code can recover the current state according to the expected history state in case of detection of inconsistency for any var.

Definition 25 (DFI-IMAG)**.** *A tuple (IMG, IMSD(F, Γ), IMAD, IMR) is DFI-IMAG where:*

- *IMG=(N', T, Z, P') is a context-free grammar in which $N' = \{Start, Stms, Stm, Exp, Trm, UPDI, DetRecI, LHis, His, id\}$ while $LHis$, $UPDI$ and $DetRecI$ are instrumentation non-terminals that are used respectively to instrument the program to connect to the expected history state at protected part and embed security code for state update, check and remedy. Moreover, non-terminal His is used to provide an attribute that according to IMAGs pattern is called the history attribute. $T = \{\text{“}:=\text{”}, \text{“}*\text{”}, \text{“}+\text{”}, \text{“};\text{”}, \text{“}\neq\text{”}, \text{“(”}, \text{“)”}\}$, start symbol $Z = Start$ and P' is a set of all productions, shown in Figure 4, includes semantic productions, labeled p_S , and instrumentation semantic productions, labeled p_{IS}.*
- *IMSD=(FUNC-SET(F),TYPE-SET(Γ)) $\Gamma_{st} \in \Gamma$ defines the domain of sets of statuses modeled as records such as $SuRecord$=< $NamAdr$: string, val : int, $realaddr$: int > that we call status record whose instances correspond to program variables, where $NamAdr$ is a unique name or a compiler-generated temporary, val and $realaddr$ respectively keeps the value and real address of each variable. Moreover, $\Gamma_{ref} \in \Gamma$ includes reference to sets of statuses with the type of $SuRecord$. Moreover, F corresponds to rules in Figure 4.*
- *IMAD(A,S, I, W, P_L,R,L, τ) such that:*
  - *$\tau(env) = boolean$, $env \in I(X)$ where $X \in \{Start, LHis, Stms, Stm, His, Exp, UPDI, DetRecI\}$; $Start.env$ indicates the instrumentation flag. As noted before, this attribute determines whether instrumentation is needed.*
  - *$\tau(\|\overleftarrow{his}\|) \in \Gamma_{st}$ where $\|\overleftarrow{his}\| \in S_W(His)$, and $\|\overleftarrow{his}\| \in (W \cap P_L(His))$, $\tau(rhis) = \tau(Ihis) \in \Gamma_{ref}$ where $rhis \in (R(Stms) \cap S(Stms))$, $Ihis \in (R(X) \cap I(X))$ where $X \in \{Stms, Stm, Exp\}$.*
  - *$\tau(val) = \tau(realaddr) = int$, val and $realaddr \in \{L^{(Assign)}, L^{(Add)}\}$.*
  - *$\tau(surec) = SuRecord$, $surec \in L^{(Assign)}$, $L^{(Add)}$, and $L^{(UPDI)}$.*
  - *$\tau(Instbool) = \tau(do) = \tau(ok) = boolean$, $Instbool \in L^{(UPDI)}$, $L^{(DetRecI)}$ and $L^{(LHis)}$, $do \in I(DetRecI)$ and $ok \in S(UPDI)$.*
  - *$\tau(NamAdr) = string$, $NamAdr \in S(X)$ where $X \in \{Exp, Trm\}$, and $\tau(lexeme) = string$, $lexeme \in S(X)$ where $X \in \{id\}$.*
  - *$\tau(SuDef) \in \Gamma_{st}$, $SuDef \in S(X)$ where $X \in \{Exp, UPDI\}$.*
  - *$\tau(InsLStr) = \tau(UpdStr) = \tau(DetRecStr) = string$, $InsLStr \in S(LHis)$, $UpdStr \in S(UPDI)$ and $DetRecStr \in S(DetRecI)$.*
- *IMR is shown in Figure 4.*

Referring to Figure 4, semantic rules in production $Start \rightarrow His\ LHis\ Stms$, determine the env attribute according to the inherited attribute of the start symbol, i.e. $Start.env$, as an instrumentation flag and start a flow of this attribute down the tree. Having an inherited attribute for the start symbol in IMAGs is not the only option to determine the instrumentation flag.

$Start \rightarrow His\ LHis\ Stms$
$(His, Stms, LHis).env = Start.env$
$Stms.rhis = ref\ \ to\ \ His.his$

$Stms \rightarrow Stms_1; Stm$
$(Stms_1, Stm).env = Stms.env$
$(Stms_1, Stm).Ihis = Stms.rhis$

$Stms \rightarrow \epsilon$
$Stm \rightarrow Exp$
$Exp.env = Stm.env$
$Exp.Ihis = Stm.Ihis$

$LHis \rightarrow \epsilon\ \ (p_{sp_0})$
$if(LHis.env = 1)\{$
$LHis.InsLStr =$
"$if!(ConnectToProtectedPart())$
$\ \ then\ print($"$Error$"$);$"
$Instbool = Instrument\{$
$LHis$ by $LHis.InsLStr$
through $LHis \rightarrow Stms\}$
$\}$
$if\ !Instbool\ then$
$Error$
$His \rightarrow \epsilon$
$if(His.env = 1)$
$His.\|his\| = InitSetOfSus$

$Exp \rightarrow Trm$
$Exp.NamAdr = Trm.NamAdr$

$Trm \rightarrow id\ (VarAccess)$
$Trm.NamAdr = id.lexem$

$Exp \rightarrow Exp_1 := Exp_2\ UPDI\ DetRecI\ (Assign)$
$(Exp_1, Exp_2, UPDI, DetRecI).env = Exp.env$
$gen(Exp_1.NamAdr\,' =' \ Exp_2.NamAdr)$
$if(Exp.env = 1)\{$
$surec = (Exp_1.NamAdr, 0(var), 0(realaddr))$
$AddSu(surec, Exp.SuDef)$
$AddSuHis(surec, Exp.Ihis)$
$UPDI.SuDef = Exp.SuDef\}$
$(Exp_1, Exp_2).Ihis = Exp.his$
$DetRecI.do = UPDI.ok$

$Exp \rightarrow Exp_1\ +\ Trm\ UPDI\ DetRecI\ (Add)$
$Exp.NamAdr = newTemp()$
$(Exp_1, UPDI, DetRecI).env = Exp.env$
$gen(\ Exp.NamAdr\,' \quad =' \quad Exp_1.NamAdr\,' \quad +'$
$Trm.NamAdr\ )$
$if(Exp.env = 1)\{$
$surec = (Exp.NamAdr, 0(var), 0(realaddr))$
$AddSu(surec, Exp.SuDef)$
$AddSuHis(surec, Exp.Ihis)$
$UPDI.SuDef = Exp.SuDef\}$
$Exp_1.Ihis = Exp.Ihis$
$DetRecI.do = UPDI.ok$

$UPDI \rightarrow \epsilon\ \ (p_{sp_1})$
$if(UPDI.env\ and\ UPDI.SuDef)\{$
$surec = UPDI.SuDef$
$UPDI.UpdStr =$ "$writeProtected($" $||\ surec.NamAdr\ ||$ "$);$"
$Instbool = Instrument\{$
$UPDI$ by $UPDI.UpdStr$
through $UPDI \rightarrow Stms\}$
$if\ !Instbool\ then$
$Error$
$else$
$UPDI.ok = true$
$DetRecI \rightarrow \epsilon\ \ (p_{sp_2})$
$if(DetRecI.env\ and\ DetRecI.do)\{$
$DetRecI.DetRecStr =$
"$surecRT := readProtected(1);$
$While(surecRT \neq EndRec)$
$\{if(surecRT.val) \neq *(surecRT.realaddr)$
$then\quad \{print($"$Violation$"$);$
$\ \ *(surecRT.realaddr) := surecRT.val;\}$
$surecRT := readProtected(0);\}$"

$Instbool = Instrument\{$
$DetRecI$ by $DetRecI.DtcRecStr$
through $DetRecI \rightarrow Stms\}$
$if\ !Instbool\ then$
$Error\}$
$UPDI \rightarrow Stms\ \ (p_{IS_0})$
$Stms.env = 0$
$...$
$LHis \rightarrow Stms\ \ (p_{IS_1})$
$Stms.env = 0$
$...$
$DetRecI \rightarrow Stms\ \ (p_{IS_2})$
$Stms.env = 0$
$...$
$AddSu(SuRecord, t \in \Gamma_{st})\{...\}$
$AddSuHis(SuRecord, t \in \Gamma_{ref})\{...\}$
$writeProtected(string)\{...\}$
$readProtected(int)\{...\}$

Fig. 4. DFI-IMAG

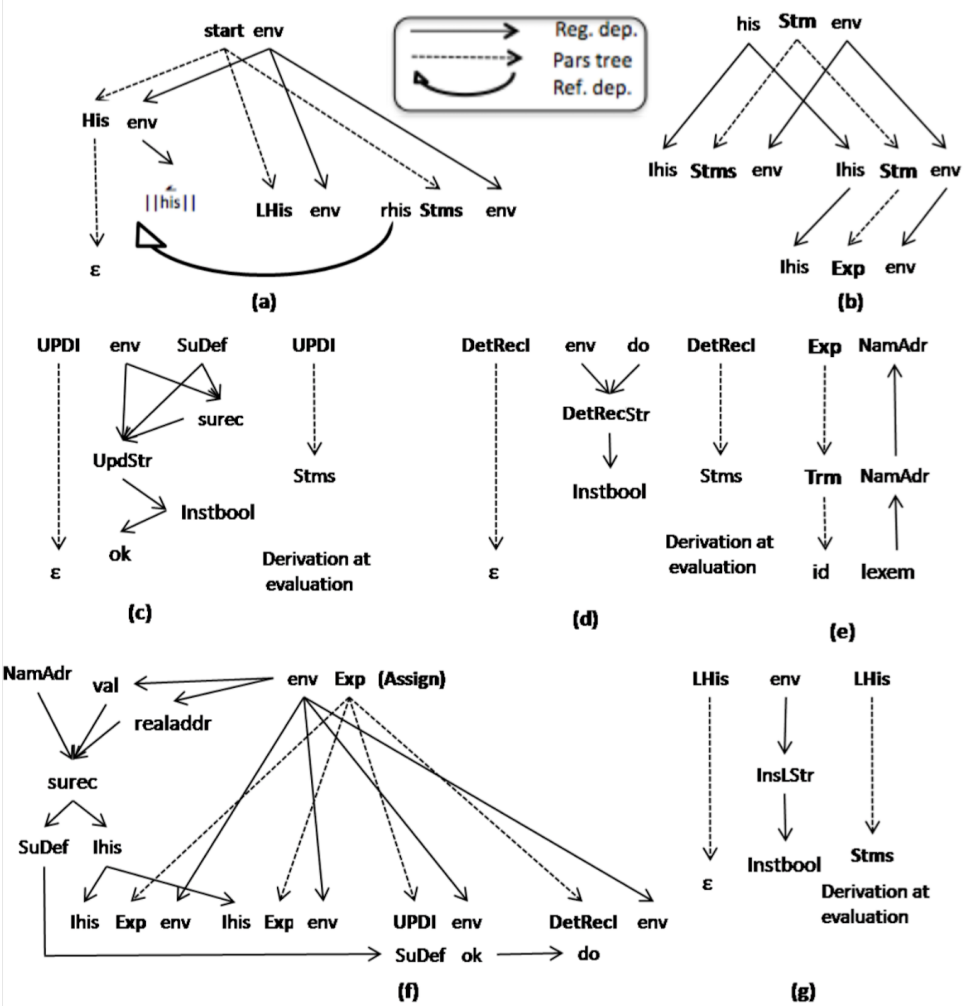

Fig. 5. Local dependencies of DFI-IMAG

Also, it is possible to have another non-terminal as the start symbol, e.g. S with a production such as $S \rightarrow flg\ Start$ such that its right-hand side includes a $boolean$ flag indicating the instrumentation flag. Semantic rules of such a production may include $Start.env = flg.value$ that defines the attribute env. Considering this approach, the start symbol does not have an inherited attribute. In IMAGs, both of these cases can be used.

Semantic rules of many productions form CSRBs with conditions on the instrumentation flag. For example, CSRB in production $His \rightarrow \epsilon$ initially defined the history attribute, $\|\overleftarrow{his}\|$, as a structured attribute that is an extendable set of status records to collect basic information of statuses later by other semantic rules. Moreover, a reference to this attribute is defined by a semantic rule of the start production, i.e. $Stms.rhis = ref\ \ to\ \ His.\|\overleftarrow{his}\|$. By considering attribute $rhis$ in IMAG, there are semantic rules of instantiated production rules all over every syntax tree that propagate this reference and then insert some instances of the status record into $\|\overleftarrow{his}\|$ through the inherited attributes $Ihis$ that also refer to $His.\|\overleftarrow{his}\|$ by a non-local reference-write dependency. Since $\|\overleftarrow{his}\|$ is a write-access attribute, its contents will not be used at attribution through a direct or indirect accesses. Moreover, since $\|his\|$ is a protected live attribute, it will be saved in the protected memory by compiler after semantic analysis. This information will be used and manipulated by the enforcement code to provide the expected history state of the program at runtime.

$LHis$ which is an instrumentation semantic non-terminal and belongs to the RHS of the root production has a CSRB that in case of turned on instrumentation flag, instruments the code such that the inserted code can connect to the protected part at runtime. We provide a security code in $LHis.InsStr$ which is an if-statement that is derivable from $Stms$ while semantic rules of p_{IS_1} will lead to generation of intermediate code corresponding to the provided security code in attribute $InsStr$. Therefore, according to instrumentation semantic rule, the derivation and syntax tree will be expanded if the syntax expansion for the inserted code does not have any syntax error, i.e. based on p_{IS_1} , $LHis.InsStr$ is derivable from $LHis$. In such a case, the instrumentation rule returns true to the local attribute $Instbool$. Otherwise, an error is raised. Moreover, execution of

this inserted code at runtime checks the possibility of connecting to the protected part. If this connection is possible, then other security code that will be added through other semantic rules can update this basic information and check desired properties related to the policy at runtime. Since in this example inserted code with the aim of inline monitoring assumed safe, semantic rules in every $R(p_{IS})$ start a flow of turned off instrumentation flag down the subtrees related to the instrumentation. It implies that no more instrumentation in these subtrees is needed.

In DFI-IMAG, two productions named Add and $Assign$ may have a non-empty set of defined sus, and therefore are security-relevant productions. In Add, the generated code is an instruction that adds values of E_1 and Trm and puts the result of the addition into a new temporary name for E, denoted by $E.NamAdr$, and in the $Assign$ production, the generated code is an instruction that performs the assignment. First, semantic rules in both of these productions pass the attribute env down the tree and according to other attributes, generate the related code for these productions. To generate intermediate code in AGs and instrumented intermediate code in IMAGs, it is possible to define and use a string attribute, which is usually named $code$, and define it by construction of corresponding strings. Moreover, there is another incremental approach with the same application that makes a smaller number of semantic rules. This approach uses a function usually called gen which not only constructs a string of three-address codes but also appends them to the sequence of codes generated so far [30]. In this example, we use the latter approach to generate instrumented intermediate code. Then, both of productions Add and $Assign$ have a CSRB that in case of turned off env contain a semantic rule to build a status for a variable that its value is defined by execution of the generated instructions of these productions. Therefore, these statuses will be added to the $SuDef$ set of the Exp, denoted by $Exp.SuDef$ and also to the status set that $Exp.Ihis$ refers. Functions $AddSu$ and $AddSuHis$ add a status to a set of them if it does not cause any status replication. The second argument in $AddSuHis$ is a reference to $\|\overleftarrow{his}\|$ and in $AddSu$ is a non-reference synthesized attribute. Since we assumed that all Exps and thus program variables are integers, default values of statuses are zero. Also, the default value of $realaddr$ for every status is zero, and after determining the real addresses of variables in the compilation process, this field for each status can be completed to finalize basic information of the expected history, and then the history attribute can be saved in the protected part for every program.

Based on the enforcement code, there are insertion points for enforcement code after the corresponding code of productions Add and $Assign$. Therefore, they have instrumentation non-terminals, $UPDI$ and $DetRecI$, in their right-hand side. CSRB in $UPDI$ for instances of these productions that actually define a non-empty set of variables, i.e. for those that define the inherited attribute $SuDef$ of $UPDI$ as a non-empty set, instruments the program. The security code embedded by $UPDI$ includes a function call $writeProtected$ to securely communicate with the protected part. At runtime, this security code updates the value of legally defined statuses in $SuDef$ in the expected history state according to the current state. To do so, function $writeProtected$ for a variable or temporary name x, finds the corresponding status in the protected part and then copies the current value of x, which is accessible through its name at runtime, into the field val of the status in the expected history through a secure communication. Moreover, CSRB in $DetRecI$ instruments the program such that the inserted security code loads each status to $surecRT$ by using another function named $readProtected$ that securely communicates with the protected part. This function has a parameter such that, if is 1, the first status in the protected part is read, if is 0, the next status is read. Therefore, the inserted security code by $DetRecI$ for every loaded status dynamically checks if the expected value for the retrieved status, i.e. $surecRT.val$, corresponds to its current value in the program memory, i.e. $*(surecRT.realaddr)$ which depicts the content of the real address of the status at runtime. If they do not correspond to each other, a violation is detected and the current value is recovered according to the expected value.

There are some attributes such as $DetRecI.do$ and $UPDI.ok$ whose occurrences in semantic rules of $Assign$ and Add and dependencies between them make sure that state check and the state recovery security code generated by $DetRecI$ follow the state update security code generated by $UPDI$.

All local dependencies including regular dependencies and local reference-write dependencies of productions in DFI-IMAG are shown in Figure 5 respectively by reg. dep. and ref. dep. According to the evaluation of IMAGs and dependency graph of DFI-IMAG, the evaluation of DFI-IMAG is simple because there is no circular dependency, DFI-IMAG is well-defined, and a unique solution can be found by evaluating attributes in the topological order of the dependency graph for each syntax tree. Moreover, recalling the definition of L-attributed grammar as follows (from [30]), we see DFI-IMAG is an L-attributed grammar and thus has a straightforward evaluation.

Definition 26 (L-attributed Grammar)**.** *An attribute grammar is said to be L-attributed if and only if each inherited attribute of X_{pi} in production $p : X_{p0} \rightarrow X_{p1}, ..., X_{pn}$ depends only on the set $\bigcup_{1 \leq j \leq i} A(X_{pj}) \cup Inh(X_{p0})$ for $i = 1, ..., n$ and each synthesized attribute of X_{p0} depends only on the set $\bigcup_{1 \leq j \leq n} A(X_{pj}) \cup Inh(X_{p0})$.*

L-attribute grammars are the most general class for which evaluation can be implemented on-the-fly during LL parse [32]. Therefore, semantic analysis, intermediate code generation, and inlining the enforcement code with parsing through semantic rules will be done along with parsing. Moreover, the effectiveness of DFI-IMAG is discussed in the next theorem.

VII. IMAGs Evaluation and Effectiveness

In this section, attribute evaluation and characteristics of well $ICSRB_{Cond}$ IMAGs are explained, followed by describing enforcement power of IMAGs and determining how to check the effectiveness of a particular IMAG to enforce a specified

security policy.

A. IMAGs Evaluation

The generation of a syntax tree T of an AG is started by instantiating the grammar start symbol which repeatedly continued by replacing instances of non-terminals by the body of productions for those non-terminals. In addition, all attributes and semantic rules for each of these productions are instantiated.

According to the extensive bibliography on AGs in [58], evaluation of an attribute instance $x.a$ is done by applying the related semantic rule to its arguments and assigning the resulted value to $x.a$. This evaluation is possible when values of all attributes that are arguments of the semantic rule are available. In this case, it is said that $x.a$ is ready for evaluation. Initially, all attribute instances attached to a syntax tree are unavailable, with two exceptions: first, the inherited attribute instances attached to the root which may contain information concerning the environment of the program, second, synthesized attribute instances attached to the leaves which may be determined by the parser.

Assume that instances of semantic rules are equations. These equations can make an equation system for every syntax tree T of an AG, where all attribute instances constitute variables in the equation system. An attribute instance $x.a$ is consistent if all required arguments for its evaluation are available and its defining equation holds, i.e. the value of $x.a$ equals the result of its applied semantic rule to its arguments. Therefore, every tree T is consistent if all its attribute instances are consistent and, in this case, its attribute evaluation provides a solution to its equation system.

In IMAGs, attribute evaluation is the process of evaluating attribute instances in the current syntax tree and expanding it with nodes and symbols that may have attributes. These expansions include syntax tree expansion at attribute evaluation to embed the security code. Therefore, for each target program, a syntax tree T is derived from the start symbol of the grammar. Then, according to some attributes at attribute evaluation time, new subtrees and their attributes will be added to T and to the attribute scheduling process, respectively. Therefore, supporting syntax tree expansion at attribute evaluation, IMAGs need dynamic evaluation that extends the dependency graph of every syntax tree at evaluation time. Attribute instances of the tree that are nodes of the dependency graph are then topologically sorted and evaluated. Moreover, all local dependencies including local reference-write-dependencies can be extracted before evaluation. Also, according to Definition 22, for each non-local reference-write dependency between two symbols far away in the syntax tree, there is a local dependency that can be determined before attribute evaluation.

Let us assume that for a syntax tree T, a list of all attribute instances that are ready for evaluation is prepared. Utilizing a general evaluation approach in [19] and [22], we present an attribute evaluation algorithm for IMAGs shown by Algorithm 1. In this algorithm, it is assumed that the set of all attribute instances and those that are ready for evaluation are A and R, respectively. While there is a ready attribute instance for evaluation, this algorithm selects and removes an attribute such as $x_i.\alpha$ from R that is defined by f which may be a local attribute too. If f is not an instrumentation semantic rule, then α will be evaluated and all its dependent attributes that become ready for evaluation will be added to R. Otherwise, f is an instrumentation semantic rule for a $R(p_{IS})$ when $p_{IS} \in ISP$. If according to p_{IS} , the security code stored in the attribute $InstruStr$ is derivable from x_i , then α will be evaluated to true. Moreover, the syntax tree and thus dependency graph will be extended. Afterwards, all attributes of the new subtree will be added to A. Again, all dependent attributes to α that become ready for evaluation will be added to R. If security code is not derivable from p_{IS} , the CSRB that contains the instrumentation semantic rule will raise an error.

In IMAGs, evaluation terminates if syntax tree expansion at attribute evaluation does not expand the tree indefinitely. These expansions occur when for a $p_S \in SP$, the instrumentation flag is on that leads to instrumentation of the main code through a derivation of a $p_{IS} \in ISP$ by $f_{ISR} \in ISR$ in $R(p_S)$. If the instrumentation flag in $R(p_{IS})$ is on, it means that the enforcement code may need to be controlled again by inline monitoring. In such cases, more tree expansions may occur in the underlying subtree. Syntax tree expansion at attribute evaluation in IMAGs specified according to IMAGs pattern does not cause non-termination in evaluation. It is because in IRMs, such a nested instrumentation should be terminated by finally using a safe enforcement code. These IMAGs should satisfy the termination condition as follows:

Definition 27 (Termination Condition)**.** *In every possible syntax tree T of an IMAG, each branch in every subtree of T rooted in instance of an instrumentation semantic non-terminal in an instantiated semantic production $p_S \in SP$ should end up with a derivation of a production with turned off instrumentation flag in its semantic rules.*

It is easy to check the termination condition for an IMAG by checking whether there is a syntax tree T that does not satisfy termination condition. To do so, starting from each p_S with turned on instrumentation flag and considering all possible derivation of p_{IS} in $R(p_S)$, we can construct possible subtrees and check if there is a branch that again leads to the same p_S or another semantic production, e.g. p'_S , with turned on instrumentation flag that may cause non-termination.

B. Ordered IMAGs

An attribute grammar is well-defined if the equation system of every possible syntax tree has at least one solution [59]. Moreover, a dependency graph of attributes of a syntax tree has a simple circularity if an attribute a is defined directly or

Algorithm 1: Attribute Evaluation for IMAGs

```
evaluate (T: an unevaluated syntax tree)
begin
    D:= DT(T) the dependency relation over the tree T
    R:= the set of attribute instances that are ready for evaluation
    A:= the set of all attribute instances
    α, β, θ= attribute instances
    While R ≠ ∅ do
    begin
        Select and remove an attribute instance α(X_i.α := f or α ∈ L^p) from R
        If f ∈ ISR then
        begin
            α:=expand T at X_i (or X_p0 ) according to p_IS and InstruStr in f
            If α then
            begin
                D := D ⋃ DT(X_i)
                A := A ⋃ all attribute instances β in DT(X_i)
            end
        end
        else
            Evaluate α := f
        For all θ ∈ successor(α) in D do
        begin
            If θ is ready for evaluation then
                insert θ in R
        end
    end
end
```

indirectly using an attribute b and vice versa. Therefore, an attribute grammar is non-circular if $D(T)$ of every possible syntax tree T is acyclic, otherwise it is circular. While circular AGs may be well-defined under certain circumstances, all non-circular grammars are not only well-defined but also uniquely-defined, i.e. there exists exactly one solution for equation system of each possible syntax tree. Deciding whether an AG is non-circular is an exponential problem [60].

According to Kastens definition [61], a subclass of non-circular grammars are Ordered Attribute Grammar (OAG) and thus are uniquely-defined. In an OAG, for each symbol, a partial order over associated attributes can be given such that in any context of the symbol, attributes are evaluable in an order which includes that partial order [61]. Therefore, for each symbol, a partial order of attributes is used to construct a total order. If this total order has not any circle, then the AG is an OAG. This property for AGs and IMAGs can be checked similarly by a polynomial time algorithm on the grammar size.

Non-local dependencies usually cause problems such as circularity and extra limitations on the order of attribute evaluation [21], [42]. Although IMAGs support these dependencies that are results of reference-write dependencies, evaluation of IMAGs which follow the IMAG pattern does not face such problems. It is because such dependencies in these IMAGs are limited in two cases. First, it is not necessary to support a situation in which every attribute could be a reference to any other attribute in a syntax tree. Instead, IMAGs aim to provide reference attributes to the history attribute for security-relevant productions to statically extract and maintain basic information of statuses. Second, the enforcement code in IMAGs uses this information to monitor program behavior at runtime. Therefore, the manipulation of a referred attribute at attribution would be limited to definition. Hence, there is no use for reference attributes at evaluation time to take part in a circular dependency. Moreover, semantic rules at evaluation time insert statuses into the referenced attributes, and thus, partially define referenced attributes. Although the collection of this basic information of statuses is necessary, the order of them is not. Therefore, the union of statuses is commutative and associative which does not impose any limitation on the order of status definitions in related attributes. Therefore, specific characteristics related to IMAGs do not make any extra cycle or limitation order in the evaluation.

Now, we describe how to derive visit sequences in ordered IMAGs. Kastens evaluation of ordered AGs [61] is based on a very simple principle of visit sequences. At evaluator construction time, a visit sequence is computed for each production according to attribute dependencies of the grammar. A visit sequence for a production $p : X_{p0} \rightarrow X_{p1}...X_{pn}$ is a sequence of instructions of the following types:

- Eval($x_{pj}.a$): Evaluates the attribute instance $x_{pj}.a$ according to its defining semantic rule in $R(p)$.

- Visit(i, k): $\begin{cases} i = 0, & \text{Visit parent of p for } k^{th} \text{ time.} \\ i > 0, & \text{Visit child } x_{pi} \text{ for the} k^{th} \text{ time,} \end{cases}$

At evaluation time, a simple evaluator walks the tree and visits nodes according to the instructions in the visit sequence of instantiated productions. Evaluation starts with the first instruction for the root production of T. When an Eval instruction is encountered, the specified attribute is evaluated. Then, the evaluator moves on to the next instruction. The corresponding visit sequence of each production will be denoted as $VS(p)$ such that:

$$V(p) = \{Visit(i,k) | 0 \leq i \leq |p|, 1 \leq k \leq nov_x, x = x_i\}$$
$$AF(p) = \{Eval(x_{pj}.a) | 0 \leq j \leq |p|\}$$
$$AV(p) = AF(p) \cup V(p)$$
$$VS(p) \subseteq AV(p) \times AV(p)$$

$|p|$ denotes the number of non-terminals in production p and nov_x denotes the number of visits that will be made to a non-terminal x.

In IMAGs, the associated local tree walk for each production p, steps may be a: move up to the parent (Visit parent), move down to a certain child (Visit child), evaluate an attribute (Eval), and extend the tree and the equation system. Therefore, considering the last action in IMAGs, we define visit sequence for each production $IMVS(p)$ in terms of $VS(p)$. Each visit sequence associated to a production p in an IMAG, $IMVS(p)$, is a linearly ordered relation over evaluation of defined attribute occurrences in p, visits $V(p)$, and expansions $Exp(p)$ as follows:

$$Exp(p) = \{e_{ISN} \text{ as the expansion of } x_{ISN} | 0 \leq i \leq |p| \text{ and } x_{ISN} \in ISN\}$$
$$IMAV(P) = AV(p) \cup Exp(p)$$
$$IMVS(p) \subseteq IMAV(p) \times IMAV(p)$$

Since all described steps including the computation of visit sequences depend polynomially in time on the IMAGs size, it can be determined in polynomial time if an IMAG is well-defined. Based on these visit sequences and evaluation algorithms for OAGs [61] an efficient evaluation algorithms can be applicable to evaluate IMAGs.

C. IMAGs Effectiveness

Inline monitoring is effective to enforce every enforceable policy at runtime [7]. We demonstrate IMAGs strong power to enforce security policies by defining IMAGs automata which show the model of IMAGs program monitoring. Moreover, we describe how to evaluate the effectiveness of a specific IMAG in satisfaction of a given security policy.

1) IMAGs Program Monitoring Model: Some program monitors observe executions of target applications and dynamically recognize invalid behaviors and thus are called recognizers [62]. Some other program monitors are viewed as execution transformers rather than execution recognizers [5] that transform program execution into policy satisfying execution. Since by inlining security code of the enforcement mechanism in IMAGs program monitoring, programs and thus their behaviors are transformed to the ones that satisfy a policy, they are execution transformer. IMAGs statically transform programs and dynamically transforms their execution. We model IMAGs program monitoring formally by defining IMAGs automata as deterministic and infinite state machines to reason about their effectiveness.

Definition 28 (IMAG Automaton)**.** *An IMAG automaton denoted by IA is a triple (Q, q_0, δ) in which Q is the possibly countably infinite set of states, q_0 is the initial state and δ is the deterministic and total transition function with the form of: $Q \times A \rightarrow Q \times (A \cup .)$ while A is the finite set of actions that are derivable instructions from IMAG and the symbol "." denotes the empty sequence of actions. The transition function takes the automaton current state and input action, and returns a new state and an action to output. Automaton states is formed as (EHS, CS) such that EHS and CS indicate values of attributes in the history state and in current system state, respectively. Input actions are the next instructions the target wants to execute and output actions are those that the IMAG make observable and ready to execute. A transition of the automaton is triggered by the presence of an input action.*

We specify the execution of IA by using a labeled operational semantics in Figure VII-C1. The basic single-step judgment has the form $(q, \sigma) \xrightarrow{v}_{IA} (q', \sigma')$ where q denotes the current state of the automaton, σ denotes the sequence of instructions that the target program wants to execute, q' and σ' denote the state and instructions sequence after the automaton takes a single step, and v denotes the sequence of output instructions made by the automaton in this step. The input sequence of instructions is not observable from the generated code whereas the output is observable. Moreover, let "..." denote a processed sub-path in runtime paths.

The first rule, (Insert), determines that IMAG automaton may insert a new instruction in a runtime path if it reaches an instruction v in the processing runtime path, and according to the state and input action, transition function returns the new state and an action to insert into the output stream. All parts of the security code, i.e. security update, check and remedy code can cause instruction insertion. Security update changes EHS in q to depict the last legitimate changes by the output actions, i.e. if $q = (EHS, CS)$ then q' is (EHS', CS) while in insertion by security check $q' = q$, and in case of recovery through

$$\frac{\sigma = (..., v, \sigma_1) \quad \delta(q, v) = (q', v')}{(q, \sigma) \xrightarrow{v'}_{IA} (q', \sigma)} \tag{Insert}$$

$$\frac{\sigma = (..., v, \sigma_1) \quad \delta(q, v) = (q', .)}{(q, \sigma) \rightarrow_{IA} (q', (..., \sigma_1))} \tag{Suppress}$$

$$\frac{\sigma = (..., v, \sigma_1) \quad \delta(q, v) = (q', v)}{(q, \sigma) \xrightarrow{v}_{IA} (q', (..., \sigma_1))} \tag{Output}$$

$$\frac{\sigma = (..., v, \sigma_1) \quad \delta(q, v) = halt}{(q, \sigma) \dot{\rightarrow}_{IA} (q', (..., .))} \tag{Halt}$$

Fig. 6. Operational Semantic of IMAGs automatoa

security remedy code CS would be recovered according to expected legitimate changes, i.e. if $q = (EHS, CS)$ then q' is (EHS, CS'). Moreover, according to the insertion points, v' can be inserted before or after v by instrumentation rules for different purposes such as checking the policy, detecting a violation, taking a remedial action or updating the program state.

According to (Suppress), an input action such as v can be omitted and an empty sequence can be inserted to indicate that the input action should not be observable. In addition, rule (Output) shows that an input instruction may be accepted and returned as the output with no change when it is analyzed as usual and has no effect on the policy satisfaction. Moreover, in case of detection of a critical and non-recoverable violation, an IMAG automaton can halt the target by suppressing all next instructions which is shown in rule (Halt).

Theorem 1 (IMAGs Policy Enforcement)**.** *IMAGs can enforce every CFSS security policy.*

Proof. The operational semantics of edit automata in [5] and IMAG automata shows that IMAGs can enforce security policies defined in the form of CFSS policies with the same power as edit automata in the enforcement of security policies because they have the same capabilities. □

2) Policy-Preserving IMAG: We describe characteristics of a specific IMAG that guarantees satisfaction of a specific CFSS policy at runtime. In such a case, we call the IMAG with these characteristics a *Policy-Preserving IMAG* that is defined as follows.

Definition 29 (Policy-Preserving IMAG)**.** *For every CFSS security policy a well-defined IMAG is policy-preserving if the next conditions are satisfied:*

1) *IMAG is defined according to the IMAGs pattern.*
2) *Assume that:*
   - Σ *is the set of all possible state transitions of the program execution with turned off instrumentation flag while no violation of the specified policy happens,*
   - *for each* $\sigma \in \Sigma$*, we assume the set of all possible runtime state transitions such as* σ'' *which are the same as* σ *except that they have a violation of the security policy while the instrumentation flag is still off. Let us show this set for each* $\sigma \in \Sigma$ *by* Σ''*,*
   - Σ' *is the set of all possible runtime state transitions of the program with turned on instrumentation flag,*

   then the following requirements should be satisfied:
   a) *Integrity: There should be a guarantee that sub-state transitions related to the security code in every* $\sigma' \in \Sigma'$ *do not have any vulnerability against the security policy and it is not possible to be bypassed. We call them secure sub-state transitions.*
   b) *Transparency: For each* $\sigma \in \Sigma$ *there is one* $\sigma' \in \Sigma'$ *such that in case of no violation includes sequential sub-state transitions equivalent to* σ *with the same order in addition to some secure sub-state transitions generated by the enforcement code that are located in-between those equal sub-state transitions.*
   c) *Soundness: For each* $\sigma'' \in \Sigma''$ *that is generated to show a violation in a* $\sigma \in \Sigma$*, there is one* σ' *including sequential sub-state transitions equivalent to* σ *with the same order, and instead of sub-state transitions corresponding to the violation in* σ''*, consists of secure sub-state transitions related to the inlined security code that leads to continue sub-state transitions equivalent to* σ *as if no violation has happened. Moreover, the inlined sub-state transitions might be empty and thus the execution can be terminated after the secure sub-state transition.*

If the integrity and transparency are satisfied the state transition of a policy satisfying execution should still satisfy the policy after instrumentation, i.e. there should not be any false positive. Moreover, if soundness is satisfied then the state transition of a violating execution must be converted to a policy satisfying one which means the violation should be detected, and thus the approach should not have false negative. Therefore, by satisfying these requirements every semantic-preserving code written in

a policy-preserving IMAG satisfies the specified policy at runtime that is denoted by: $Policy-IMAG(\Sigma') \rightarrow \forall \sigma' \in \Sigma' : p(\sigma')$ *which implies for every state transition of the program execution* p *holds.*

The next section describes and discuses an application of the security policy enforcement by IMAGs.

VIII. DFI-IMAG Evaluation and Effectiveness

The effectiveness of DFI-IMAG is discussed in the next theorem.

Theorem 2 (Policy-Preserving DFI-IMAG)**.** *For DI and DFI security policies as specified CFSS policies (Definition 24 and 23), the well-defined DFI-IMAG is policy-preserving.*

Proof. A violation of these security policies is a result of an unauthorized su change in an arbitrary instruction v_i that may be caused by an attack. DFI-IMAG that is written according to the IMAGs pattern considers these violations.

Assume the runtime state transition of a written program in DFI-IMAG according to the policy preserving IMAGs as the following:

- $\sigma = (\overbrace{s_0, ..., s_i, v_i, s_{i+1}}^{\sigma_1}, \overbrace{v_{i+1}, ..., s_j, v_j, s_{j+1}, ..., s_n}^{\sigma_2})$ such that $\sigma \in \Sigma$ is a state transition without any violation of the security policy with turned off instrumentation flag,
- $\sigma'' = (s_0, ..., s_i, v_i, {s_{i+1}}^a, {v_{i+1}}^a, ..., {s_j}^a, {v_j}^a, {s_{j+1}}^a, ..., {s_m}^a) \in \Sigma''$corresponding to σ while it has a violation caused by v_i and the instrumentation flag is turned off, and
- There exists a $\sigma' \in \Sigma'$ that is resulted from turning on the instrumentation flag in the secured program such that:

  Integrity is satisfied because state transitions caused by $UPDI$ and $DetRecI$ in every σ' does not have any vulnerability against the security policy, and functions `writeProtected` and `readProtected` are safe because they securely write and read statuses from the protected part and does not make any arbitrary memory changes. Therefore, if the state transition for each of these functions is $s_i \longrightarrow s_{i+1}$, then $s_i \sim s_{i+1}$.

  Transparency is satisfied because in case of no runtime data integrity violation, $\sigma' = (\overbrace{s_0, ..., s_i, v_i, s_{i+1}}^{\sigma'_1}, \overbrace{UPDI, s_{UPDI}, DetI, s_{DetI}}, \overbrace{v_{i+1}$) while $\sigma'_1 \sim \sigma_1$. Moreover, secure state transitions produced by $UPDI$ affects the protected part. Also, since no violation occurs just instructions related to the state check in the detection instrumentation of $DetRecI$ are executed, which is shown by $DetI$, and no false alarm is raised. Since none of them makes any change in actual program states, then $s_{i+1} \sim s_{DetI}$. Therefore, sub-state transition after execution of v_{i+1} in σ' results in $\sigma'_2 \sim \sigma_2$.

  In case of a runtime violation, soundness is satisfied because $\sigma' = (\overbrace{s_0, ..., s_i}^{\sigma'_1}, \overbrace{{v_i}^a, {s_{i+1}}^a, UPDI, s_{UPDI}, DetRecI, s_{DetRecI}}, \overbrace{v_{i+1}, ..., s_j, v_j, s_{j+1}, ..., s_n}^{\sigma'_2})$ and $\sigma'_1 \sim (s_0, ..., s_i) \subset \sigma_1$. Moreover, state transitions corresponding to the violation in σ'' are changed in σ' such that the inserted security code by $UPDI$ and $DetRecI$ have no effects but raising an alarm and recovering the program state. Such recovery makes $s_{DetRecI}$ in σ' equivalent to s_{i+1} in σ without any unnecessary instruction execution and state changes, i.e. $s_{DetRecI} \sim s_{i+1}$ and then $\sigma'_2 \sim \sigma_2$.

DFI-IMAG detect violations from DI and because of recovering the current state after each violation, it prevents DFI violations. Therefore, for these policies, DFI-IMAG is policy-preserving. □

Presenting an optimum IMAG with minimum memory usage and runtime execution overhead, and/or maximum precision depends on the specified CFSS policy and some parameters such as supported capabilities and features in the underlying language, need for violation detection or prevention, exact machine model for the protected part and even characteristics of written programs. It is the same for DFI-IMAG such that presenting an optimum DFI-IMAG according to such parameters considered as a future work.

A. DFI-IMAG Overhead

The space and time overhead of DFI-IMAG for each syntax tree T is computed in Table 7 and Table 8, respectively. Attributes related to IMAG impose space overhead on attribution which is shown in Table 7. In space cost column, m_y denotes the size of attribute y and $|T_x|$ denotes the frequency of x appearance in a syntax tree T while x can be an instance of a non-terminal or a production rule named x. Therefore, the third column shows the total space overhead in bytes which is imposed by attributes related to inline monitoring in every T of DFI-IMAG. In general, some non-terminals such as Trm and id do not impose any cost because they do not have any effect in the policy enforcement. Some non-terminals such as $Start, His$, that just appear one time in each T, or non-terminals in ISN may have special effects in space and time overhead. Other non-terminals are denoted as the $rest$ non-terminals in computations. For example, for every instantiated non-terminal in every syntax tree except for Trm and id, denoted as $|T_{N^v}|$, there is an attribute env with the size of boolean, m_{bool} that is one byte. Also, the space overhead imposed by this attribute is 2 bytes for $Start$ and His. Moreover, the imposed space overhead by attribute $SuDef$ is the multiplication of the size of the status record, i.e. m_{surec} , and the frequency of security-related productions in T, denoted by $|T_{Add,Assign}|$. Space cost for for other attributes are computed similarly in Table 7.

Attribute	Attribute type	Space cost
env	bool	$\|T_{N"}\| = \|T_{ISN+His+Start+rest}\| = \|T_{ISN}\| + \|T_{rest}\| + 2$
Instbool	bool	$\|T_{ISN}\|$
Ihis & rhis	ref	$m_{ref}\|T_{N"-His-Start-ISN}\| = m_{ref}\|T_{rest}\|$
his	set of su rec.	$1m_{surec}\|T_{id,Add}\|$
Inst. str.	string	$m_{str}\|T_{ISN}\|$
surec	su rec.	$m_{surec}\|T_{Add,Assign,UPDI}\| = 3m_{surec}\|T_{Add,Assign}\|$
SuDef	su rec.	$m_{surec}\|T_{Add,Assign}\|$
ok	bool	$\|T_{Add,Assign}\|$
do	bool	$\|T_{Add,Assign}\|$

Fig. 7. DFI Space Overhead.

Attribute	Semantic Rule	Time cost
env	Copy rule	$\|T_{N"}\|$
Ihis	Copy rule	$\|T_{N"-ISN-N\in p_{IS}}\|$
rhis	Reference rule	1
his	Init, AddSuHis	$\|T_{Add,Assign}\|$
SuDef	Copy rule, AddSu	$\|T_{Add,Assign,UPDI}\| = 3\|T_{Add,Assign}\|$
ok	Copy rule	$\|T_{Add,Assign}\|$
do	Copy rule	$\|T_{Add,Assign}\|$
Instbool	Inst. sem. rule	$2\|T_{Add,Assign}\| + 1$
Inst. str.	Copy rule	$2\|T_{Add,Assign}\| + 1$

Fig. 8. DFI Time Overhead.

Time overhead in DFI-IMAG is imposed by extra semantic rules in IMAGs that compute some attributes related to inline monitoring. Table 8 shows the type and frequency of occurrence of defining semantic rules for each attribute. For example, there are $|T_{N"}|$ copy rules for definition of env attribute in every T. Moreover, to defined $Instbool$ there is an instrumentation semantic rule for every instantiated semantic production, $|T_{SP}|$, that should be executed. Similarly, semantic rules that impose time overhead are shown in Table 8.

According to semantic-based instrumentation in IMAGs, the inserted code by instrumentation would be inserted more precisely such that even values of every derivable code of the same production rule might be different, and thus are not necessarily instrumented in the same way. For example, all method calls to the same method, i.e. recognized with the same name, that according to knowledge of programmer might have an impact on system security, will not be instrumented the same. Instead, if according to the evaluated attributes of each call to the method, that are results of semantic analysis, the policy satisfaction might be affected, then the method call will be instrumented. Therefore, Policy enforcement by IMAGs does not cause extra runtime overhead on programs. In other words, considering the same security code as these works, the resulted code by IMAGs impose at most the same overhead.

B. DFI Related Works

There are some useful proposed techniques to satisfy DFI or DI which are most close to our desired policies such as [16], [10], and we compare them with DFI-IMAG in Table 9 using parameters including effectiveness in security policy enforcement, precision of the enforcement, categorization and user efficiency, necessity of the policy enforcement and its desirability. This comparison shows that DFI-IMAG is more effective, language-based, user efficient and precise in satisfaction of these necessary security policies while the approach is not error-prone.

Table III. Comparison of DFI to the most related works.

Parameters	Approaches		
	DFI-IMAG	**DFI** [Castro et al. 2006]	**DFI** [Ramezan-ifarkhani and Razzazi 2015]
Effectiveness in security policy enforcement			
Detects DI viol.	Yes	may be	Yes
Recovers in case of DI viol. det.	Yes	No	No
DFI viol. Prevention	Yes	No	No
DFI viol. Detection	No	may be	Yes
DFI case	Yes for all cases	no path cases-others may be	Yes for all cases
Precision and the possiblity of errors			
False positive	No	No	No
False negative	No	Yes	No
Error-prone	No	Yes	Yes
Categorization and user efficiency			
Language-based	Yes	No	No
Program-based	No	Yes	Yes
Automatic	Yes	No	No
Integrated	Yes	No	No
Responsibility	Language designer	Customer	Customer
Necessity and desirability			
Satisfaction of sec. pol.	Mandatory	Discretionary	Discretionary
Desired by	Producer and all customers	Customers	Customers
* yes - no			

Fig. 9. Comparison of DFI to the most related works.

As summarized in Table 9 and explained in Theorem 2, in DFI-IMAG, all unauthorized variable changes that may cause DI violations at runtime are detected, program data is recovered, DFI violations are prevented, and program execution continues as if no violation has been occurred. It is because by checking the amount of variables in the expected history state and the current program state after each possible definition instruction, any variable change that is not supposed to occur are detected. Therefore, even if an instruction between two legal definitions illegally changes the amount of a variable, an inconsistency would occur between the expected history state and the current program state, and thus would be detected show in an example. Moreover, after such a detection, the variable would be recovered. In addition, further violations such as control flow subversion that for example might occur because of checking a variable with an illegal amount can be prevented.

In [16], a definition of DFI is presented which actually is a specific case of DFI, and the program is instrumented to check if the last instruction that has changed a variable correspond to the static analysis, otherwise a violation is detected. If an instruction that for some variables is a data-change instruction illegally changes another variable, this approach will detect. Instead of checking the amount of data as denoted in the DI and DFI definition (IV-A), the changing instruction is checked to determine whether it is allowed to change or not. Therefore, this approach does not check DI and DFI satisfaction as defined in this paper. However, in some cases, when an alarm is raised a violation from DI or DFI might have occurred too, and we denoted this by “may be" in Table 9. In addition, again because of not keeping track of data amounts recovery is not possible in this approach.

In [10], a general definition for DFI is presented and then programs are instrumented such that every request for definition and use by instructions is checked to determine if it is legal. Therefore, no instruction can illegally change a data. Moreover, in [10] there are some assumptions that dictates no other applications can change the program data. Hence, data integrity and DFI are satisfied. Still, since this approach is not sensitive to illegal changes of the amount of data, data recovery is not possible in a violation detection.

In contrast with DFI-IMAG in which security policy enforcement is language-based, both approaches presented in [16] and [10] are program-based. Therefore, while DI and DFI are mandatory security policies, in both of these works the enforcement approach works like a patch and are not integrated with program generation, and thus would not be automatically applied. Moreover, in both of the related approaches transforming programs to preserve policy at runtime becomes a responsibility of the customers and relies on their knowledge and skills which especially in the concept of mandatory security policy satisfaction is not recommended. It is because such characteristics make the approach error-prone, non-user efficient and should be repeated for every individual program. To the best of our knowledge, there is no technique that after violation detection can recover the program state and continue the program execution while is not affected by the violation.

IX. Conclusion

As the number of attacks and their varieties against software security increases, developing secure software becomes more urgent and essential. Towards developing secure software, we provide an integrated and automated language-based foundation to specify languages such that programs written in these languages have the intrinsic capability to preserve desired security policies at runtime.

AGs provide a powerful formalism in language processing. Attributes and semantic rules in AGs are used by compilers to enforce static semantics, and also dynamic semantics through code generation [31]. We introduce Inline Monitoring Attribute Grammars (IMAGs) based on standard AGs in which attributes, semantic rules, and code generation are utilized to support inline monitoring as a policy enforcement mechanism. In other words, static semantic analysis techniques developed for AGs, which are of high interest in their own sake [33], are utilized in IMAGs to perform semantic-directed instrumentation at intermediate code generation. Hence, it leads to generate self-monitoring programs that check satisfaction of desired security policies dynamically at runtime even against powerful attackers.

There are some ways to continue this research such as extending IMAGs to support sophisticated language features, more effective specification and enforcement of critical and complex security policies, and to extend tools to support IMAGs features.

Appendix

ABAC

Appendix

References

[1] M. Hall, D. Padua, and K. Pingali, "Compiler research: the next 50 years," *Communications of the ACM*, vol. 52, pp. 60–67, 2009.
[2] M. Bishop, M. Dilger *et al.*, "Checking for race conditions in file accesses," *Computing systems*, vol. 2, no. 2, pp. 131–152, 1996.
[3] J. S. Foster, T. Terauchi, and A. Aiken, *Flow-sensitive type qualifiers*. ACM, 2002, vol. 37, no. 5.
[4] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security and Privacy*, vol. 2, no. 6, pp. 76–79, 2004.
[5] J. Ligatti, L. Bauer, and D. Walker, "Run-time enforcement of nonsafety policies," *ACM Transaction on Information System and Security*, vol. 12, no. 3, pp. 1–41, Jan. 2009.
[6] U. Erlingsson, "The inline reference monitors approach to security policy enforcement," Ph.D. dissertation, Cornell University, 2004.
[7] G. Gheorghe and B. Crispo, "A survey of runtime policy enforcement techniques and implementations," University of Trento, Tech. Rep., 2011.
[8] U. Erlingsson and F. B. Schneider, "Sasi enforcement of security policies: A retrospective," in *Proceedings of the 1999 workshop on New security paradigms*. ACM, 1999, pp. 87–95.
[9] L. Bauer, J. Ligatti, and D. Walker, "Composing security policies with polymer," in *ACM SIGPLAN Notices*. ACM, 2005.
[10] T. Ramezanifarkhani and M. Razzazi, "Principles of data flow integrity: Specification and enforcement," *Journal of Information Science and Engineering*, vol. 31, pp. 529–546, 2015.
[11] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security*, vol. 13, no. 1, pp. 1–40, Nov. 2009.
[12] P. Akritidis, C. Cadar, M. Costa, and M. Castro, "Preventing Memory Error Exploits with WIT," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP '08)*, 2008, pp. 263–277. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4531158
[13] D. Evans and A. Twyman, "Flexible policy-directed code safety," in *IEEE Security and Privacy*, Oakland, CA, 1999.
[14] U. Erlingsson and F. B. Schneider, "Irm enforcement of java stack inspection," in *Proceedings of the 2000 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2000.
[15] S. Nair, P. Simpson, B. Crispo, and A. Tanenbaum, "Trishul: A policy enforcement architecture for java virtual machines," Technical Report IR-CS-045, Department of Computer Science, Vrije Universiteit Amsterdam, Tech. Rep., 2008.
[16] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 147–160.
[17] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav, "A survey of static analysis methods for identifying security vulnerabilities in software systems," *IBM System Journal*, vol. 46, no. 2, pp. 265–288, 2007.
[18] F. B. Schneider, G. Morrisett, and R. Harper, "A language-based approach to security," in *Informatics*. Springer, 2001, pp. 86–101.
[19] D. E. Knuth, "Semantics of context-free languages," in *Mathematical Systems Theory*, vol. 2, no. 2, 1968, pp. 127–145.
[20] T. Ekman and G. Hedin, "Rewritable reference attributed grammars," in *ECOOP 2004*, 2003, pp. 144–169.
[21] G. Hedin, "Reference attributed grammars," in *Informatica 24*, vol. 3, 2000, pp. 301–317.
[22] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper, "Higher order attribute grammars," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 1989, pp. 131–145.
[23] A. M. Sloane, L. C. Kats, and E. Visser, "A pure embedding of attribute grammars," *Science of Computer Programming*, vol. 253, pp. 205–219, 2011.
[24] W. A. Barrett, *Compiler Design*. AS Print Shop, San Jose State University, 2005.
[25] "Welcome to jastadd.org," http://jastadd.org/.
[26] J. T. Boyland, "Conditional attribute grammars," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 1, pp. 73–108, 1996.

[27] G. Hedin and E. Magnusson, "Jastadd-an aspect-oriented compiler construction system," in *Science of Computer Programming*, 2003, pp. 37–58.
[28] T. Sloane, "Experiences with domain-specific language embedding in scala," in *Domain-Specific Program Development*, 2008, p. 7.
[29] D. Servos and S. L. Osborn, "Current research and open problems in attribute-based access control," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, p. 65, 2017.
[30] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 2006.
[31] R. W. Sebesta, *Concepts of Programming Languages*. Addison-Wesley, 2008. [Online]. Available: http://gigapedia.com/items:links?id=4597
[32] M. Scott, *Programming Language Pragmatics*, 2009, morgan Kaufmann.
[33] D. Parigot, G. Roussel, É. Duris, and M. Jourdan, "Attribute Grammars: a Declarative Functional Language," INRIA, Tech. Rep. RR-2662, 1995, projet CHARME. [Online]. Available: http://hal.inria.fr/inria-00074027
[34] H. Alblas and B. Melichar, Eds., *Attribute Grammars, Applications and Systems*, ser. Lecture Notes in Computer Science, vol. 545. Springer, 1991.
[35] D. Parigot, G. Roussel, M. Jourdan, and E. Duris, *Dynamic Attribute Grammars*. Springer-Verlag, 1996, vol. 45, no. 2, pp. 59–104. [Online]. Available: ftp://ftp.inria.fr/INRIA/publication/RR/RR-2881.ps.gz
[36] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, pp. 181–185, 1985.
[37] D. Jackson, *Software Abstractions*. MIT Press, 2006.
[38] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, *PAriCheck: an efficient pointer arithmetic checker for C programs*. ACM, 2010, no. June, pp. 145–156. [Online]. Available: http://portal.acm.org/citation.cfm?id=1755707
[39] J.-T. Chan and W. Yang, "An attribute-grammar framework for specifying the accessibility in java programs," *Comput. Lang. Syst. Struct.*, vol. 28, no. 2, pp. 203–235, 2002.
[40] J. T. Boyland, "Descriptional composition of compiler components," Ph.D. dissertation, University of California, 1996.
[41] U. Kastens, "Attribute grammars in a compiler construction environment," *Attribute Grammars, Applications and Systems Lecture Notes in Computer Science*, vol. 545, pp. 380–400, 2011.
[42] J. T. Boyland, "Remote attribute grammars," *Journal of the ACM*, vol. 52, no. 4, pp. 627–687, Jul. 2005.
[43] M. van den Brand and et al., "The asf+sdf meta-environment: a component-based language development environment," in *Proceedings of Compiler Construction Conference*. Springer-Verlag, 2001, volume 2027 of LNCS.
[44] E. Visser, "Stratego: A language for program transformation based on rewriting strategies," in *Proceedings of Rewriting Techniques and Applications (RTA'01)*. Springer-Verlag, 2001, volume 2051 of LNCS.
[45] J. R. Cordy, "Txl: A language for programming language tools and applications," in *Proceedings of the 4th Workshop on Language Descriptions, Tools, and Applications (LDTA'04) at ETAPS*, 2004.
[46] J. Alexandre and B. V. Saraiva, "Purely functional implementation of attribute grammars," Ph.D. dissertation, Department of Computer Science, Utrecht University, The Netherlands, 1999. [Online]. Available: ftp://ftp.cs.uu.nl/pub/RUU/CS/phdtheses/Saraiva/
[47] E. Van Wyk, O. De Moor, K. Backhouse, and P. Kwiatkowski, "Forwarding in attribute grammars for modular language design," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science, vol. 2304. Springer-Verlag, 2002, pp. 128–142.
[48] T.-P. Zhang, "Radar: compiler and architecture supported intrusion prevention, detection, analysis and recovery," Ph.D. dissertation, Georgia Institute of Technology, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1292991
[49] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 168–177, Nov. 2000. [Online]. Available: http://portal.acm.org/citation.cfm?doid=356989.357005
[50] C. Michael and A. Ghosh, "Using finite automata to mine execution data for intrusion detection: A preliminary report," *Lecture Notes in Computer Science*, vol. 1907, pp. 66–79, 2000. [Online]. Available: citeseer.nj.nec.com/michael00using.html
[51] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003. [Online]. Available: http://portal.acm.org/citation.cfm?id=822806
[52] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, vol. 38, no. 0086160. Citeseer, 2004, pp. 159–169. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.5.6065\&rep=rep1\&type=pdf
[53] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier, "Formatguard: Automatic protection from printf format string vulnerabilities," in *Proceedings of the 10th conference on USENIX Security Symposium*. USENIX Association, 2001. [Online]. Available: http://portal.acm.org/citation.cfm?id=1267627
[54] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of the 14th USENIX Security Symposium*, 2005.
[55] J. K. S.M. Christey, R.P. Glenn and et al., "Common weakness enumeration (cwe version 2.5)," 2013, aCM SIGAda Ada Letters.
[56] TF8, "Wu-Ftpd Remote Format String Stack Overwrite Vulnerability," 2000, http://www.securityfocus.com/bid/1387.
[57] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure execution via program shepherding," in *Proceedings of the 11th USENIX Security Symposium*, 2002, pp. 191–206.
[58] B. L. Pierre Deransart, Martin Jourdan, *Attribute Grammars: Definitions, Systems, and Bibliography*. Springer-Verlog, 1988, 323 of Lecture Notes in Computer Science.
[59] G. Hedin, "Incremental Semantic Analysis," Ph.D. dissertation, Lund University, 1992.
[60] M. Jazayeri, W. F. Ogden, and W. C. Rounds, "The intrinsically exponential complexity of the circularity problem for attribute grammars," *Commun. ACM*, vol. 18, no. 12, pp. 697–706, Dec. 1975.
[61] U. Kastens, "Ordered attribute grammar," *Acta Informatica*, vol. 13, no. 3, pp. 196–255, 1980.
[62] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, 2000.
[63] G. Hedin, "An introductory tutorial on jastadd attribute grammars," in *Generative and Transformational Techniques in Software Engineering III*. Springer, 2011, pp. 166–200.
[64] "Example of imag in jastadd," 2016, https://github.uio.no/toktamr/IMAGJA.
[65] T. Ramezanifarkhani and M. Razzazi, "Control flow integrity via data flow integrity specification and dynamic enforcement," *Technical report, Department of Informatics, University of Oslo, Norway*, vol. 31, p. Nr 481.

APPENDIX A
AN APPLIED EXAMPLE IN JASTADD

We provide an applied example in JastAdd which is an open-source system and extensible Java compiler [27]. Implementation details is available in the JastAdd documentation such as manual [25] and the tutorial [63]. We built our example (see [64] for more details) on top of the running state machine example of JastAdd [63] which supports several analysis and properties.

JastAdd also allows attributes to be programmed declarative. A parser that builds the AST from a text can be generated using an ordinary parser generator, building the AST in the semantic actions. JastAdd generates a Java API from the abstract grammar

$switch(auth)\{$
$case\ x : printf(\text{“}permitted\text{”});$
$case\ y : printf(\text{“}denied\text{”});$
$\}$
$EmptyVariavles();$
(a)
$state\ S0;$
$state\ S1;$
$state\ S2;$
$trans\ x = auth : S0 -> S1;$
$trans\ y = auth : S0 -> S2;$
$state\ S3;$
$state\ S4;$
$trans\ permitted : S1 -> S3;$
$trans\ denied : S2 -> S3;$
$trans\ emp : S3 -> S4;$
(b)

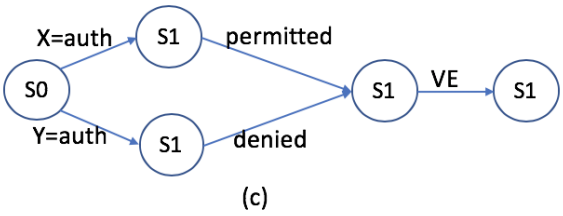

Fig. 10. (a) Example of a code, (b) its state machine in the example and, (c) its graphical representation.

including classes with AST constructors and AST traversal methods. Moreover, accessing the attributes are accomplished by methods.

The state machine that we are going to extend includes states with directional labeled transitions between them that are used to model programs with the aim of analysis purposes. To specify a simple policy, and its enforcement let us assume that multiple branches like those caused by switch statements are used for security purposes such as authentication. Moreover, a common policy in such cases is to empty all the used temporal variables afterward. It means in the state machine of programs satisfying this policy, the next instruction after the switch statements should be an instruction to empty the temporal variables, let us call it variable-emptiness or VE. Therefore, the state after a switch statement should have a transition labeled VE to the next state. To check the emptiness, we consider a policy p_{emp} dictating that the label for those transitions that their source state has several incoming edges should correspond the specific label VE. Such a policy might be selected for security critical applications and IRM implementation, and applicable to the programs labeled state transition machine model.

An example of such a code with its state machine is shown in Figure 10. The first part of this example, Figure 10 (a) is a switch statement for the authentication purpose and (b) is its state machine which will be the input text for the IMAG and, (c) shows a graphical view of the state machine. To provide an automatic approach to specify and enforce the p_{emp} policy in the state machine attribute grammar, we explain p_{emp}-IMAG in JastAdd with more details available at [64]. As the result of syntax and semantic analysis of the input, the policy specification and its automatic satisfaction will be a part of the state machine attribute grammar and involved in the process of parsing, analysis, compilation and code generation.

A. p_{emp}-IMAG

The state machine example in the form of inline monitoring attribute grammer, p_{emp}-IMAG, is explained here to automatically enforce the policy p_{emp}. p_{emp}-IMAG is shown in Figure 11 which later would be explained in notation of JastAdd abstract grammar, APIs and aspects.

In this grammar, $History$ is a NT with a *history attribute*, His, which is used to collect basic information of statuses through semantic rules at attribute evaluation time. This attribute should be a write-access and protected live attribute. After semantic analysis and finalizing this information at compile time, it will be saved in the protected memory by compiler. Later, through the enforcement code, this information will be used and manipulated to provide the expected history state of the program at runtime. We provide the protected part and its secure interface to store the protected-live attributes similar to [65]. $Connect$ is a ISN defined in IMAG to instrument the program to securely connect to the basic information of the history attribute at runtime if the instrumentation flag is on. StateMachine input attribute env is an *instrumentation flag* that can be considered

```
// Inline Monitoring Attribute Grammar

StateMachine ::= Declaration*; History Connect
(Declaration, History, Connect).env=StateMachine.env
Declaration.rhis =ref to History.His

Connect ::= \epsilon // (p_{sp_{0}})
    if(Connect.env=1){
    Connect.InstrumStr=
    ``if !(ConnectToProtectedPart())
     then print(``Error");"
  Instbool=Instrument {
    Connect by Connect.InstrumStr
    through Connect ::= IfStmt}
    }

    if !Instbool then  Error

    History ::= epsilon
    if(History.env=1)
    History.His=InitSetOfSus

    Connect ::= IfStmt //(p_{IS_{1}})
     Stmts.env=0

abstract Declaration;

State : Declaration ::= <Label:String>;
surec=(Label, 0)
AddSuHis(surec, Declaration.rhis)


Transition : Declaration ::=
<Label:String> <SourceLabel:String> <TargetLabel:String>; Inst
AddReachTrgSuHis(TargetLabel, Declaration.rhis)
Inst.Ihis =Declaration.rhis
Inst.SourceLabel=SourceLabel

Inst ::= <epsilon>
  Inst.InstrumStr:= "if ("
  sizeof(Inst.SourceLabel,Inst.Ihis)|
  %SourceLabel.reachablefrom())|
  ">1)  then  if ("|
  Inst.SourceLabel|
  "!=VE"|
  ") Printerror()"

  Instbool=Instrument{
   Inst by Inst.InstrumStr
      through Inst ::= IfStmt}

    if !Instbool then Error

Inst ::= IfStmt
```

Fig. 11. p_{emp}-IMAG: The state machine inline attriute grammar to automatically satisfy p_{emp} policy

```
// Abstract syntax
// See section 2.1 in the JastAdd tutorial paper
// Reference manual: http://jastadd.org/

StateMachine ::= Declaration*; History /Connect/

History ::= epsilon

Connect  ::= epsilon // (p_{sp_{1}})
Connect  ::= Stmts    //(p_{IS_{1}})

abstract Declaration;

State : Declaration ::= <Label:String>;

Transition : Declaration ::=
<Label:String> <SourceLabel:String> <TargetLabel:String>; //Inst//

Inst ::= epsilon          // (p_{sp_{2}})
Inst ::= ifStmt            //(p_{IS_{2}})
```

Fig. 12. JastAdd abstract grammar for p_{emp}-IMAG

as an inherited attribute of the start symbol supplied as an initial value before attribute evaluation begins, to turn the program monitoring on and off.

Status records here are in the form of $surec = (Label, 0)$ in which for each program state, $surec.Label$ contains the state label and $surec.reachablefrom$ stores the number of incoming edges to that state. In the state production, function $AddSuHis$ adds the status to the history attribute. $Inst$ as an ISN, will inherit the source label of the transition to inline the security code when it is necessary. Moreover, in the transition production, function $AddReachTrgSuHis(TargetLabel, Declaration.rhis)$ increases the $surec.reachablefrom$ by one when $surec.label$ is the target of the transition. In the production rules of $Inst$, $sizeof(Inst.SourceLabel, Inst.Ihis)$ returns $surec.reachablefrom$ when the $surec.label$ is $Inst.SourceLabel$.

ABS Grammar

The abstract grammar of p_{emp}-IMAG in JastAdd is shown in Figure 12 in which the nonterminals and productions are classes and alternative productions are subclasses. The terminals are corresponding the fields of classes. To generate AST from a text a parser generator can be used to generate the parser. In JastAdd nonterminal attributes (NTAs) are shown like /C/. For simplicity, we use the same notation instrumentation semantic nonterminal ISN like D In the abstract grammar. So, A ::= B /C/ entails that A has two children: B, and C while C is an ISN that will be created by the parser, but must be extended later at attribution.

API and Attributes in Aspects

JastAdd uses the abstract grammar to generate a Java API with constructors that build AST nodes, and methods for traversing the AST. To access the attributes, the API is furthermore augmented with required methods.

Figure 13 shows some parts of the generated API for the state machine language, including the attributes that will be defined in the coming sections.

Due to intertype declarations in JastAdd, the attributes and semantic rules of p_{emp}-IMAG have to be define in aspects such as instrumentation.jrag in [64]. Therefore, the attribute declarations that appears in the aspect file belong to the related AST classes and JastAdd reads the aspect files and inserts these intertype declarations into the AST classes. For example, defined methods in the aspects will be added to the corresponding classes.

In Figure 15 we show a program that makes the state machine shown in Figure 10 with directly used API construction that manually construct the AST. The process completes without any error which entails satisfaction of the required policy in the state machine.

```
class StateMachine {
  StateMachine();                              // AST construction
  void addDeclaration(Declaration node);       // AST construction
  void addHistory(History node);               // AST construction
  History getHistory();                        // AST traversal
  void addConnect(Connect node);               // AST construction
  Connect getConnect();                        // AST traversal
  List<Declaration> getDeclarations();         // AST traversal
  Declaration getDeclaration(int i);           // AST traversal
  boolean env();                               // Attribute access
  }
 abstract class Declaration extends ASTNode {
  Set<status> rhis ();                    // Attribute access
  }
  class History extends ASTNode {
  }
  class Stmt extends ASTNode {...
  }
 abstract class ISN extends ASTNode {
 abstract String getInstrumStr();              // AST traversal
 abstract void setInstrumStr(string instrumStr); // AST traversal
 abstract void addInstrum(ASTNode node);       // AST construction
 abstract ASTNode getInstrum();                // AST traversal
  }
  class Connect extends ISN {
  Connect();                                   // AST construction
    String getInstrumStr();                    // AST traversal
    viod setInstrumStr(string instrumStr);     // AST traversal
    void addInstrum(ASTNode node);             // AST construction
    ASTNode getInstrum();                      // AST construction
    void addIfStmt(IfStmt node);               // AST traversal
    List<Stmt> getStmts();                     // AST traversal
    Stmt getStmt(int i);                       // AST traversal
  }
  class State extends Declaration {
    State(String theLabel);                   // AST construction
    String getLabel();                        // AST traversal
    Set<Transition> transitions();
    // Attribute access, state has an attribute called transitions of
    //type of the set of transitions
    Set<State> reachable(); // Attribute access, state has an attribute
    //called reachable of type of the set of State
    }
  class Transition extends Declaration {
  Transition(String theLabel, theSourceLabel, theTargetLabel);// AST construction
  String getLabel();                     // AST traversal
  String getSourceLabel();               // AST traversal
  String getTargetLabel();               // AST traversal
  State target();                        // Attribute access
  State source();                        // Attribute access
  void addInst(Inst node);               // AST construction
  Inst getInst();                        // AST traversal
  }
    class Inst extends ISN {
      Inst();                            // AST construction
      String getInstrumStr();            // AST traversal
      viod setInstrumStr(string instrumStr); // AST traversal
      void addInstrum(ASTNode node);     // AST construction
      ASTNode getInstrum();              // AST traversal
      void addIfStmt(IfStmt node);       // AST construction
      List<Stmt> getStmts();             // AST traversal
      Stmt getStmt(int i);               // AST traversal
    }
    Class status{
      string Label;
      int ReachableFrom;
      public status();
      // Constructor, returns a new pair (null,0);
      public void SetSu(string label);
      // Adds the element su to this object.
      public void IncSuRF(string label);
```

```
aspect Instrumentation {
inh boolean StateMachine.env()=getFlagInput(boolean InstFlag);
//in-line equation

inh boolean StateMachine.getDeclaration(int i).env()=env();
//the right hand side of the equation is within the scope of StateMachine and
//thus it means StateMachine.env()

inh boolean StateMachine.getHistory.env()=env();

Syn statusSet History.His()=new statusSet(); //a synthesized %write-access
protected-live attribute which is a set of statuses.

inh StateMachine.getDeclaration(int i).rhis() = History.His()
//a reference to the his attribute

inh boolean StateMachine.getConnect.env()=env();


Syn string Connect.setInstrumStr(
"if !(ConnectToProtectedPart())
 then printError();"); // sets the Connect.InstrumStr by the input string

 Connect.Instrument ()
 {
 rewrite Connect  { // Conditional rewrite rule
   when (Connect.env())
     to IfStmt {
       ...    // extending IfStmt  based on Connect.setInstrumStr
       return Instbool;
     }
 If !Instbool then println("Instrumentation Error");
 }

 Syn boolean Stmts.env()=0;

 Syn status state.surec()=(SetSu(getLabel()),0);
 Declaration.rhis.AddSu(state.surec);

 Declaration.rhis.IncSuRF(Transition.lookup(getTargetLabel()));
 inh boolean Inst.env()=Transition.env();

 Inh statusSet Inst.Ihis=Declaration.rhis();
 Inh string Inst.SourceLabel=Transition.lookup(getTargetLabel());
 Inh string Inst.Label=Transition.lookup(getLabel());

 Syn string Inst.setInstrumStr(
 "if ("+Inst.Ihis. ReachableFrom(SourceLabel)+">1)  then  if ("+
   Inst.Label+ "!=VEC"+ " then printError();");

Inst.Instrument ()
{
rewrite Inst{ // Conditional rewrite rule
  when (Inst.env())
    to IfStmt {
      ...    // extending IfStmt  based on Inst.setInstrumStr
      return Instbool;
    }
If !Instbool then println("Instrumentation Error");
}
inh boolean ifStm.env()=Inst.env()
}
```

Fig. 14. A short version of the aspect that instruments the code.

```java
package exampleprogs;
import AST.*;

public class MainProgram {

    public static void main(String[] args) {
        // Construct the AST
        StateMachine m = new StateMachine(1);
        m.addDeclaration(new State("S0"));
        m.addDeclaration(new State("S1"));
        m.addDeclaration(new State("S2"));
        m.addDeclaration(new Transition("auth=x", "S0", "S1"));
        m.addDeclaration(new Transition("auth=y", "S0", "S2"));
        m.addDeclaration(new State("S3"));
        m.addDeclaration(new State("S4"));
        m.addDeclaration(new Transition("permitted", "S1", "S3"));
        m.addDeclaration(new Transition("denied", "S2", "S3"));
        m.addDeclaration(new Transition("emp", "S3", "S4"));

        // instrument the program and check the policy.
                m.Instrumentation();

    }

}

```

Fig. 15. A program with a state machine that satisfies the p_{emp} policy in p_{emp}-IMAG