2012 IEEE 36th International Conference on Computer Software and Applications

Model-based Static Source Code Analysis of Java Programs with Applications to Android Security

Zheng Lu
Department of Computer Science
Louisiana State University
zlu5@lsu.edu

Supratik Mukhopadhyay
Department of Computer Science
Louisiana State University
supratik@csc.lsu.edu

***Abstract*—We combine static analysis techniques with model-based deductive verification using SMT solvers to provide a framework that, given an analysis aspect of the source code, automatically generates an analyzer capable of inferring information about that aspect. The analyzer is generated by translating the collecting semantics of a program to a "marked" formula in first order logic over multiple underlying theories. The "marking" can be thought of as a set of holes or contexts corresponding to the "uninterpreted" APIs invoked in the program. Just as a program imports packages and uses methods from classes in those packages, we import the semantics of the API invocations as first order logic assertions. These assertions constitute the models used by the analyzer. Logical specification of the desired program behavior (rather its negation) is incorporated as a first order logic formula. An SMT-LIB formula solver treats the combined formula as a "constraint" and "solves" it. The "solved form" can be used to identify logical (security) errors in Java (Android) programs. Security properties of Android are represented as constraints and the analysis aims to show that these constraints are respected.**

***Keywords*-Static Code Analysis; Android Security; SMT**

I. Introduction

Software systems routinely manage mission-critical activities in organizations that rely on dependable, situation-aware, and timely delivery of classified or sensitive information. Information flows in such enterprises are processed by custom-built, open-source, and/or COTS software programs. These software programs can be uncertified and may contain malicious code or vulnerabilities that can be exploited by an insider or an outsider to leak confidential data, misclassify documents, perform pernicious activities, and destroy or modify valuable information. Hence, the correctness and reliability of software driving these systems have become issues of utmost importance.

Application-independent errors in software systems like buffer overflows and null dereferences can be exploited by malicious applications to create security holes through which confidential data can be leaked. Many of these bugs are not detected until much later when catastrophic effects are already visible [1] making difficult the task of runtime fault handling mechanisms for ensuring recovery. A more important concern is the lack of proper tool support for detecting logical application-dependent errors in programs. An examination of a list of well known incidents resulting from software glitches reveals that application-dependent logical errors were the causes of most [2] [3] [4] (e.g., the USS Yorktown breakdown and the failure of the Patriot missile). Many of these logical errors were deep (as opposed to simple typos) and are difficult to detect using state-of-the-art testing techniques alone [5].

We combine static analysis techniques with model-based deductive verification using SMT solvers to provide a framework that, given an analysis aspect of the source code, automatically generates an analyzer capable of inferring information about that aspect. A model-based technique is necessary since for many of the APIs invoked as well as the objects instantiated, the (source) code is not available. The only information that the analyzer can get about the properties of these artifacts is from their models. The analyzer is generated by translating the collecting semantics of a program to a "marked" formula in first order logic over multiple underlying theories. The "marking" can be thought of as a set of holes or contexts corresponding to the "uninterpreted" APIs invoked in the program. Just as a program imports packages and uses methods from classes in those packages, we import the semantics of the API invocations as first order logic assertions. These assertions constitute the models used by the analyzer. Logical specification of the desired program behavior (rather its negation) is incorporated as a first order logic formula. An SMT-LIB formula solver treats the combined formula as a "constraint" and "solves" it. The "solved form" can be used to identify logical (security) errors in Java (Android) programs.

II. Related Work

Techniques for software verification and validation fall into three main categories. The first category involves informal methods such as software testing and monitoring. Such techniques scale well; this is by far the most used technique in practice to validate software systems. Testing accounts for forty to sixty percent of the development effort [5] [6]. Traditional software testing methods [7], however, are too ad hoc and do not allow for formal specification

0730-3157/12 $26.00 © 2012 IEEE
DOI 10.1109/COMPSAC.2012.43

and verification of high-level logical properties that a system needs to satisfy. In the realm of safety critical software where exponential blow up in the number of possible situations to be dealt with is inevitable, traditional testing techniques can hardly be used to provide any amount of confidence. The second category of techniques for software verification and validation involves formal methods. Traditional formal methods such as model checking and theorem proving are usually too heavy and rarely can be used in practice without considerable manual effort.

Model checking is an automatic approach to verification, mainly successful when dealing with finite state systems. It not only suffers from the infamous state explosion problem but also requires construction of a model of the software.

The third category of techniques for software verification and validation are static analysis [8] and abstract interpretation [9]. Static analysis refers to the technique(s) for automatically inferring a program's behavior at compile time. While static analysis tools have met with tremendous practical success and have been routinely integrated with state of the art compilers, such tools can only detect shallow and simple errors due to their lack of deductive power. For example, traditional static analysis tools cannot detect the presence of deadlocks or the violation of mutual exclusion in concurrent programs. Abstract interpretation is a technique for collecting, comparing, and combining the semantics of programs. It has been successfully used to infer run time properties of a program that can be used for program optimization. The next few paragraphs review the most successful approaches to program analysis. In recent years, much work has been done on static analysis of software. Some static analysis tools, such as Uno [10], Splint [11], Polyspace [12], Codesurfer [13], PREfix and PREfast [14], ESP [15], and PAG [16] perform lightweight data flow analysis. Coverity [17] performs data flow analysis as directed by checkers written in MetaL, a language designed to encode checking automata. Astree is a static program analyzer that is aimed at proving absence of runtime errors in embedded programs. Astree can handle only a "safe" subset of C, rather than the full C language. Also, it applies only to particular runtime errors rather than general properties of programs. Halbwachs et al [18] use linear relation analysis for discovering invariant linear inequalities among the numerical variables of a program. Their techniques have been used to validate (e.g., analyze delays) in synchronous programs written in the language Lustre. Several abstractions have been considered to provide an approximate (conservative) answer to the validation problem such as widenings, convex approximations and Cartesian factoring [19]. These approximations are implemented using the polka [18] polyhedral library. Alur et al [20] have used predicate abstraction for analyzing hybrid systems. In this technique, a finite abstraction of a hybrid automaton is created a priori using the initial predicates provided by the user.

III. Overview of the Android Platform

Android is a software stack for mobile devices that includes an operating system, middleware and key applications. A central feature of the Android system is that an application can make use of elements of other applications. To accomplish this, the system needs to start a process when any part of an application is needed. Android doesn't have a single entry point (no main() function). They have essential components that the system can instantiate and run as needed. The four main types of components are:*Activity*, *Service*, *Broadcast receivers*,*Content providers*. Android uses intent to activate the first 3 components. For activities and services, the intent is the pair: <action name, data>, which indicates the predefined action that the receiver needs to take and the *data* to process. In our program analysis framework, this intent object is our *keyword*, we use this intent object to perform data flow analysis and function call analysis.

The Android architecture supports building applications with phone features and protecting users by minimizing the consequences of bugs and malicious software. In Android, an application can share its data and functionality with other applications. These accesses must be controlled carefully for security.

Android permissions are rights given to applications to allow them to perform functions like take pictures, use the GPS, or make phone calls. When applications are installed, they are given a unique UID, and each application always runs under that UID on that particular device. The UID of an application is used to protect its data sharing with other applications.

IV. Architecture of our model

Figure 1 shows the architecture of our model-based static analysis approach. The abstract collecting semantics of Java programs are represented as "marked" constraints. The "marking"s can be thought of as a set of holes or contexts corresponding to uninterpreted APIs, i.e., library APIs whose semantics are not known. Just as a program imports packages and uses methods from classes in those packages, we import the semantics of the API invocations as first order logic assertions or constraints. These assertions are the models that are used to "unmark" the abstract collecting semantics constraints, i.e., "filling in" the "holes" left by uninterpreted APIs. Analysis aspects are specified as constraints. Basic constraint solving is done using a combination of decision procedures provided by the Yices [21] constraint solver.

The key steps involved in our analysis framework are

1) Verify the permissions of the Android APIs invoked in the Java source code based on the `Manifest.xml`.

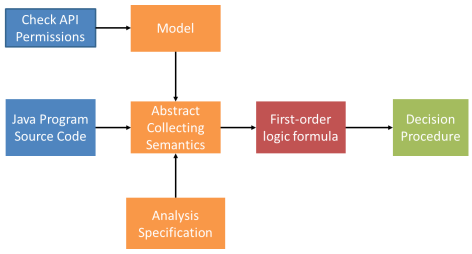

Figure 1. Architecture of our model

The results of the permission verification are used to modify the models of the APIs

2) Generate abstract collecting semantics constraints from the Java source code,
3) Import models of uninterpreted methods and objects as assertions into the already generated constraints; uninterpreted methods/objects need to be annotated by the programmer; annotation is needed since a particular method might be overridden by the developer and hence importing its "conventional" model from a model library may result in unsoundness of the analysis, and
4) Generate an analyzer by adding appropriate analysis "aspect" constraints,
5) Analyze by solving the constraints.

A. Permission verification

In this section, we discuss how we verify whether an application has the correct permission settings. Android permissions can be categorized into different protection levels [22] [23].

Permissions are defined in the `Mainifest.xml` file. Our framework will examine this file and retrieve the permission information. We build a required permission list by analyzing the APIs invoked in a program and compare it with the permission information $\mathcal{P}$ retrieved from the `Mainifest.xml` file. We map the Android API calls to the required permission list using Stowaway [24]. If every required permission is provided we consider that the application has no permission violation. Otherwise, any API calls which have no appropriate permissions provided will be asserted as returning -1 in the analysis that follows.

B. Constraints, SMT-LIB formula Solvers, and Satisfiability

Constraints are special formulas of first order logic [25]. A constraint system formally specifies the syntax and semantics of constraints.

A constraint solver implements an algorithm for checking *satisfiability/consistency* of a set of constraints using the constraint theory, i.e., determining if there exists an assignment of the variables that satisfies the constraints. A solver uses axioms of the constraint theory together with simplification rules as rewrite rules to transform the constraints to a "normal" form called the "solved form". The final constraint that results from such a computation is called the answer.

C. SMT-LIB formulas and Yices

Satisfiability Modulo Theories (SMT) libraries [26] provide a framework for checking the satisfiability of first-order formulas with some background logical theories. SMT-LIB is an SMT library that provides a standard description of the background theories used in SMT systems; it gives a common input and output languages for SMT formula solvers.

Yices [21] is an efficient SMT-LIB formula solver that decides the satisfiability of arbitrary formulas containing uninterpreted function symbols with equality, linear real and integer arithmetic, scalar types, recursive datatypes, tuples, records, extensional arrays, fixed-size bit-vectors, quantifiers, and lambda expressions.

V. Inferring Collecting Semantics of Java Programs

We need to perform both intraprocedural and interprocedural analysis for analyzing deep logical properties of Java programs. In intraprocedural analysis, from a data flow analysis of the source code, in a series of steps we build a constraint system that captures its collecting semantics. In case of interprocedural analysis, we need to build a call graph and define some external rules relating the different API invocations and detect if the analyzed code breaks these rules.

In our analysis framework, we follow the following sequence of steps to check if the a program satisfies a user-defined analysis aspect.

1) We first perform a dataflow analysis of the Java source code and generate its collecting semantics
2) Based on the dataflow analysis results, we generate the static single assignment [27] graph of the program
3) We convert the SSA graph to the SMT-LIB formulas (see below)
4) Finally, we import models of uninterpreted API invocations as first order logic assertions

We illustrate the above steps using the following example

```
1 class udhpcd
2 {
3   int getSocket(int listen_mode)
4   {
5     int fd = 0;
6     if (listen_mode == 2)
7     {
8       fd = listen_socket();
9     }
10    else
11    {
```

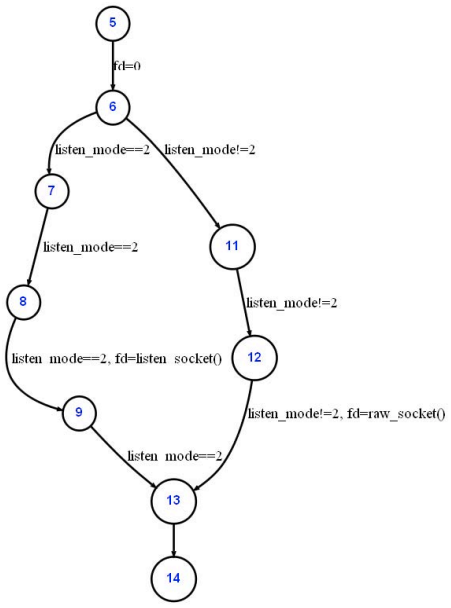

Figure 2. Data Flow

```
12          fd = raw_socket();
13        }
14      }
15    }
```

In the program above, `listen_mode` is the user input. `listen_socket()` is a method, which will return a positive integer; `raw_socket()` is a method provided by operating system, which will return a specific integer number greater than zero.

We perform a data flow analysis of the source code and represented in Figure 2; the integer number in the data flow graph indicates the line number of each statement in the program.

After line 12 the value of the variable `fd` can be the return value of `listen_socket()` or `raw_socket()`; since this program has two branches. At compile time, we cannot determine which path the control will follow; so we consider the value of `fd` is $\{listen\_mode = 2 \bigwedge fd = listen\_socket(); listen\_mode \neq 2 \bigwedge fd = raw\_socket()\}$ where the semicolon represents disjunction.

To construct SMT-LIB logic formulas capturing the collecting semantics of the program, we need to convert the program to a static single assignment graph as in Figure 3.

From the SSA graph in Figure 3, we create an assertion for each node. For example, the first node in the graph is fd1=0, we can create (assert (= fd1 0)). The node of "fd4 =Phi{fd3,fd2}" is a ϕ function, we build a disjunction (or (= fd4 fd3)(= fd4 fd2)). There are two labeled edges, so we need to create two implications: (=> (= listen_mode 2) (= fd2 listen_socket)) and (=> (distinct listen_mode

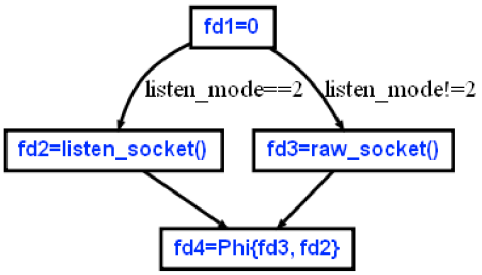

Figure 3. Static Single Assignment of the program

2) (= fd3 raw_socket)). The semantics of the APIs are incorporated as follows. If an API has no permission provided, we assert the API returns -1. Else we import an assertion characterizing the API from a model library. For example, for the `listen_socket` and `raw_socket`, we import the assertion (assert (and (> listen_socket 1) (= raw_socket 1))).

The post condition of the program considered above is `fd`> 1. Verifying whether this postcondition holds is considered the analysis aspect for this program. This analysis aspect is incorporated into the SMT-LIB formula characterizing the collecting semantics of the program by adding the conjunct ($<$ fd4 1). The combined SMT-LIB formula was found to be unsatisfiable by the Yices solver indicating the program satisfies the specification.

We now describe an algorithm for converting an SSA graph of a program to SMT-LIB formulas that capture its collecting semantics. Let $\mathbb{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ be the SSA graph of the program. In this graph, each node represents a statement in the program. We represent the if and loop conditions as edge labels in the graph. We can generate SMT-LIB formulas capturing collecting semantics of the program using Algorithm 1 that formalizes the intuition described above.

VI. Experiments and Limitations

In this section, we describe experiments that we conducted using our analysis framework.

We analyzed the source code from Android Bluetooth ChatServices application. This application builds a Bluetooth network platform to allow a device to exchange data with other Bluetooth devices. It has three main functionalities: 1. Discover the Bluetooth devices, 2. Paire and connect the devices, 3. Transfer data among these devices.

The application uses several API calls such as: `BluetoothAdapter.getDefaultAdapter()`, `BluetoothAdapter.getRemoteDevice(address)`, `BluetoothSocket` and `BluetoothChatService`.

We provide these API models based on the Android Development Documentations. For this application, we simply consider that the application can set up the Bluetooth

Algorithm 1 Converting SSA to SMT Algorithm

for $n \in \mathcal{N}$ **do**
  if n is a simple assignment statement $VAR = EXP$ **then**
    Create an assertion (assert (= VAR EXP)) in SMT;
  end if
  if n is a assignment statement with API call $VAR = API$ **then**
    Create an assertion (assert (= VAR API));
  end if
  if n is a ϕ function statement **then**
    Let v be the variable in this statement and the set W be the values of this ϕ function;
    Create a disjunction $v = w_i$, where $w_i \in W$;
  end if
  if n is a function call statement $FUN()$ **then**
    Create an assertion (assert (= FUN FUN_SUMMARY));
  end if
  for $e \in \mathcal{E}$ **do**
    if e is labeled **then**
      Let n be the node directed by this edge;
      Create a conjunction of implication formula $e \rightarrow n$;
    end if
  end for
  if The API permission is provided **then**
    Provide the API model as (assert (= API API_specification value));
  else
    Set the API model as -1 (assert (= API -1);
  end if
  Provide the function summary as the function return value after the function analyzed (assert (= FUN_SUMMARY FUN_return value)).
end for

service and can connect to any devices discovered. So we set `BluetoothChatService` to return a `non-null` object, and in the SMT specification, we model the API function value as (not -1). The source code analyzed satisfied the specification. We downloaded several free android application source code and ran our static analysis tool to the source code. In Table I, we report some possible program vulnerabilities we detected from the analysis. In the application `Android SMSPopup`, we detected a possible command injection error; the command statement is an array that comes from another function which may possibly provide a wrong statement. In `openGPStracker`, we found that it has more permissions than required; this may lead to the overprivileged permission problem. We also detected that the application `openGPStracker` has a hardcoded password.

Android SMSPopup	1) In class SmsReceiverService.java there is a false null checker for statement. The branch **if** (message.isSms()&&message. getMessageClass()==MessageClass. CLASS_0) will never be reached. 2) in class SmsPopupUtils.java the method getUnreadSmsCount(Context context) is never called. 3) in class SmsPopupUtils.java there is a system command call Runtime.getRuntime().exec (commandLine.toArray(**new** String [0])).getInputStream()), this statement may lead to a command injection error.
openGPStracker	1) In class Constants.java on line 98, there is a hardcoded password. 2) The function serializeWaypoints () in GpxCreator.java fails to perform a null checker for variable mediaUri on line 440. 3) The expression **if** (startImmidiatly &&mLoggingState==Constants .STOPPED) in GPSLoggerService.java line 563 is always evaluated as true, the else branch will never be reached.
OpenSudoku	1) The method update() in IMNumpad.java fails to perform a null checker for statement on line 208 and line 229. 2) The method saveToFile() in FileExportTask.java returns in a catch block on line 156, which may lead to a return value lost error.

Table I
EXPERIMENTAL RESULTS

A. Limitations

In the intraprocedural analysis, the constraint system includes all the possible values of variables; this may lead to false positives. More accurate abstract interpretation techniques are required to provide a precise analysis. In the interprocedural analysis, we model functions based on summaries; this abstraction loses accuracy and gives out false negatives. Another problem is that our tool needs developers to create external XML files to specify the correctness properties of the program; these files may not be easy to construct. In future, we need to develop XML patterns and good graphic user interfaces to help developers specify properties.

VII. CONCLUSIONS

Android applications can communicate with each other using system provided mechanisms like files, Activities, Services, BroadcastReceivers, and ContentProviders. If developers use one of these mechanisms they need to be sure

that they are communicating with the right entity. It is easy to violate the permissions inadvertently. Our program analysis framework can help developers to detect programming errors and permission violation statically. Developers need to understand which permissions need to be set up correctly and need some background knowledge in logic and constraint solving. Our future work will focus on designing a good user interface to help users easily set up the constraints and uncover logical errors in programs.

REFERENCES

[1] W. D. Yu, "A software fault prevention approach in coding and root cause analysis," *Bell Labs Technical Journal*, vol. 3, no. 2, pp. 3–21, 1998. [Online]. Available: http://dx.doi.org/10.1002/bltj.2101

[2] "Software horror stories," http://www.cs.tau.ac.il/~nachumd/verify/horror.html.

[3] "Nasa mariner 1," http://www5.informatik.tu-muenchen.de/~huckle/bugse.html#mariner.

[4] "Forum on risks to the public in computers and related systems," http://catless.ncl.ac.uk/Risks/19.88.html.

[5] B. Beizer, *Software testing techniques (2nd ed.)*. New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[6] K. I. Woldman, "A dual programming approach to software testing," Master's thesis, Santa Clara University, 1992.

[7] J.-F. Collard and I. Burnstein, *Practical Software Testing*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2002.

[8] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999.

[9] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '77. New York, NY, USA: ACM, 1977, pp. 238–252. [Online]. Available: http://doi.acm.org/10.1145/512950.512973

[10] G. J. Holzmann, "Software analysis and model checking," in *CAV*, 2002, pp. 1–16.

[11] D. Evans, J. Guttag, J. Horning, and Y. Tan, "Lclint: A tool for using specifications to check code," in *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 5. ACM, 1994, pp. 87–96.

[12] "Polyspace," http://www.polyspace.com.

[13] P. Anderson, T. W. Reps, T. Teitelbaum, and M. Zarins, "Tool support for fine-grained software inspection," *IEEE Software*, vol. 20, no. 4, pp. 42–50, 2003.

[14] D. Evans, J. Guttag, J. Horning, and Y. Tan, "Lclint: A tool for using specifications to check code," in *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 5. ACM, 1994, pp. 87–96.

[15] M. Das, S. Lerner, and M. Seigle, "Esp: Path-sensitive program verification in polynomial time," in *PLDI*, 2002, pp. 57–68.

[16] F. Martin, "PAG – an efficient program analyzer generator," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 1, pp. 46–67, 1998.

[17] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A system and language for building system-specific, static analyses," in *In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. ACM Press, 2002, pp. 69–82.

[18] N. Halbwachs, Y.-E. Proy, and P. Roumanoff, "Verification of real-time systems using linear relation analysis," in *FORMAL METHODS IN SYSTEM DESIGN*, 1997, pp. 157–185.

[19] N. Halbwachs, D. Merchat, and C. Parent-vigouroux, "Cartesian factoring of polyhedra in linear relation analysis," in *In Static Analysis Symposium, SAS03*. Springer Verlag, 2003, pp. 355–365.

[20] R. Alur, T. Dang, and F. Ivancic, "Counterexample-guided predicate abstraction of hybrid systems," *Theor. Comput. Sci.*, vol. 354, no. 2, pp. 250–271, 2006.

[21] B. Dutertre and L. D. Moura, "The yices smt solver," Tech. Rep., 2006.

[22] J. Burns, "Developing Secure Mobile Applications for Android: An Introduction to Making Secure Android Applications," iSec Partners, Tech. Rep., Oct. [Online]. Available: http://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf

[23] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka, "Towards formal analysis of the permission-based security model for android," in *Proceedings of the 2009 Fifth International Conference on Wireless and Mobile Communications*, ser. ICWMC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 87–92. [Online]. Available: http://dx.doi.org/10.1109/ICWMC.2009.21

[24] S. H. D. S. Adrienne Porter Felt, Erika Chin and D. Wagner, "Android permissions demystified," in *ACM Conference on Computer and Communication Security*, 2011.

[25] T. Frhwirth and S. Abdennadher, "Principles of constraint systems and constraint solvers," 2005.

[26] C. Barrett, A. Stump, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2010.

[27] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, vol. 13, pp. 451–490, 1991.