Proceedings of the Twenty-Seventh Conference on Innovative Applications of Artificial Intelligence

HACKAR: Helpful Advice for Code Knowledge and Attack Resilience

Ugur Kuter, Mark Burstein, J. Benton, Daniel Bryce, Jordan Thayer and **Steve McCoy**
Smart Information-Flow Technologies
319 1st Ave N., Suite 400, Minneapolis, MN 55401-1689
114 Waltham Street, Suite 24, Lexington, Massachusetts 02421
{ukuter,mburstein,jbenton,dbryce,jthayer,smccoy}@sift.net

Abstract

This paper describes a novel combination of Java program analysis and automated learning and planning architecture to the domain of Java vulnerability analysis. The key feature of our "HACKAR: Helpful Advice for Code Knowledge and Attack Resilience" system is its ability to analyze Java programs at *development-time*, identifying vulnerabilities and ways to avoid them. HACKAR uses an improved version of NASA's Java PathFinder (JPF) to execute Java programs and identify vulnerabilities. The system features new Hierarchical Task Network (HTN) learning algorithms that (1) advance state-of-the- art HTN learners with reasoning about numeric constraints, failures, and more general cases of recursion, and (2) contribute to problem-solving by learning a hierarchical dataflow representation of the program from the inputs of the program. Empirical evaluation demonstrates that HACKAR was able to suggest fixes for all of our test program suites. It also shows that HACKAR can analyze programs with string inputs that original JPF implementation cannot.

Introduction

To write robust and secure code, programmers must know the input and output specifications of each imported and remote function the program uses. This knowledge often goes beyond documented information and presents some unique challenges. For example, the common C++ library function sscanf can be used to parse arguments to a UNIX procedure; however, unless a sufficiently large buffer is provided (e.g., larger than any possible input string), the transmission of external inputs through that call can cause an exploitable buffer overflow. User-provided inputs must be truncated in the calling code written by the programmer, in a way consistent with the size of the buffer they provide. The problems only get worse if the function being called is complex or uses such insecure functions internally. Developers desperately need tools that will help them identify and address such vulnerabilities.

This paper describes a novel program analysis, automated planning, and machine learning architecture for discovering security vulnerabilities, errors, and exceptions in Java programs. Program analysis for discovering vulnerabilities presents several challenges to state-of-the-art AI formalisms, algorithms, and systems. First, traditional AI planning and learning techniques are mostly symbolic, while program analysis require numeric and unbounded modeling and reasoning. Second, state-of-the-art AI techniques typically assume finite domains; however, programs are intrinsically infinite. Third, programs can create new objects during their execution, while most AI systems assume a closed world in their models.

We have developed a system called *Helpful Advice and Coding Knowledge for Attack Resistance (*HACKAR*)* that addresses these challenges. Throughout the development cycle, HACKAR provides suggestions to programmers on how to make their programs more secure and can work with opaque external library calls. HACKAR integrates into Eclipse® as a plugin for Java program analysis. HACKAR includes (1) our improved version of Java PathFinder (JPF) (Havelund and Pressburger 2000; Visser et al. 2003; Visser, Pasareanu, and Khurshid 2004) for simulating execution of Java programs and generating program path constraints, (2) an incremental learning algorithm for Hierarchical Task Networks (HTNs), (3) a variant of our HTN planner SHOP2 (Nau et al. 2003) for generating code advice to fix any vulnerabilities discovered during executions and learning.

We used HACKAR to analyze Java programs with mathematical computation, file operations, array vulnerabilities, buffer overflow and underflows, program logic errors, and string computations. Our experiments demonstrated that HACKAR is able to identify vulnerabilities caused by the inputs to the program, which JPF's concrete and state program analysis techniques cannot analyze. HACKAR successfully produced suggestions on how to avoid those vulnerabilities, in scenarios that are beyond the scope of JPF or other static or dynamic analysis tools.

HACKAR Architecture

Program analysis is the process of automatically analyzing the behavior of programs. HACKAR is concerned program analysis as the programmer is writing a Java program, to identify vulnerabilities that could be caused by the inputs to the program, and to suggest advice to the programmer on how to eliminate those vulnerabilities. Almost half of the 2010 Common Weakness Enumeration (CWE)'s Top 25 software weaknesses arise from library misuse, unexpected side effects of functions, and insufficient input sanitization

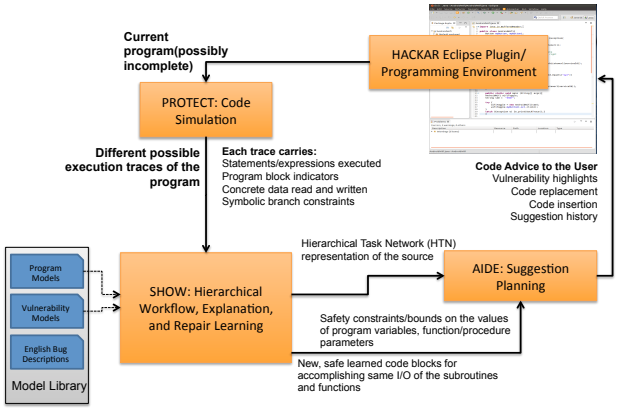

Figure 1: HACKAR high-level functional overview.

(http://cwe.mitre.org/top25/).

As an example, consider what happens when the user creates a string consisting of an SQL statement to an external database and feeds that statement with an array of values that are not specified properly.

Example 1. *Suppose that* HACKAR *is given a function with the following code fragment:*[1]

```
...
  int[] results = 0,1,2,3;
  results = external.computeResults(results);
  for (int i = 0; i < results.length; i++) {
      int dbCount = helper.queryForInt(
        "SELECT COUNT(FORENAME)
            FROM CUSTMR
            WHERE FORENAME='" + results[i] + "'",
        (Object[]) null);
      assertTrue("found in db", dbCount == 1);
  }
...
```

The above code illustrates an instance of a situation known as *SQL Injection*: if the programmer does not ensure the contents of the results array are set properly, arbitrary SQL commands can be executed against the database.

HACKAR dynamically tests programmer's code during development at the function level to identify inputs leading to vulnerabilities. By examining inputs at the function level, HACKAR captures a workflow representation of the program automatically, and utilizes that workflow in order to propagate the outcomes of the vulnerability analyses from the low-level functions to the high-level user procedures.

Figure 1 shows HACKAR's functional architecture and its components: dynamic test and execution generation, hierarchical workflow learning, and automated multi-suggestion planning. Each component shares a common program representation, and their own private models.

The program to be analyzed is first processed by the PROTECT component, which performs dynamic test-and-execution generation. For example, above, computeResults is an external function that does a computation over an integer array and returns the resulting integers as its output with one exception: computeResults returns an array of end-of-line characters, if the input contains non-positive integers. PROTECT generates test inputs for this program. Some of these test inputs will enable the program to succeed and others will fail it, enabling PROTECT to discover the conditions for the exception.

PROTECT also produces a AI planning domain description from the execution of the Java bytecode of a program. A planning domain description consists of action models (i.e., action names, parameters, preconditions and effects), as described in the subsequent section. The execution traces and action models are passed to the next component, HACKAR's hierarchical workflow learning system, SHOW. Learning generalizes the structure in the traces, i.e., the control flow of the program, into hierarchical task and data flows. Learning can create new hierarchical models of the program that differ from the original source code.

The output of the hierarchical workflow learning is a set of procedural descriptions of how to achieve tasks and functionality in a program. HACKAR uses the well-known *Hierarchical Task Network (HTN)* formalism to represent this output. We will describe HTNs in later sections. The learned HTNs are passed to HACKAR's multi-suggestion generation component, AIDE. This component is an automated HTN planner that generates plans, i.e., program traces, with the learned HTN knowledge. The hierarchical plans constitute an explanation of the vulnerability. An explanation carries the following information to the programmer:

In function SQL_Example:
failure point: helper.queryForInt
vulnerability point: computeResults
explanation: computeResults generates a non-numeric ASCII character with a negative integer as input. queryForInt uses the character instead of its integer encoding to create the SQL query.

As the programmer writes more code, this explanation would be revised and dropped eventually when HACKAR cannot justify the particular vulnerability as above.

AI Planning Formalisms

HACKAR uses PDDL (Planning Domain Description Language), the standard language for writing AI planning problems (McDermott 1998; Fox and Long 2003; 2001), to model Java program statements as *planning actions*, enabling our AI learning and planning algorithms reason about programs naturally.

PDDL is an action-centered language; at its core it provides a simple standard syntax for expressing the actions, using pre- and post-conditions to describe the applicability and effects of actions. The language separates the descriptions of parameterized actions that characterize domain behaviors from the description of specific objects, initial conditions and goals that characterize a problem instance. This is particularly useful for HACKAR, since program statements are usually parameterized. The pre- and post-conditions of actions are expressed as logical propositions constructed from

[1] Adapted from http://www.ukcert.org.uk/javasecurity.pdf.

predicates and argument terms (objects from a problem instance) and logical connectives.

These formal capabilities are particularly important for HACKAR: each action model corresponds to single Java program statement in the source program, describing the program variables and possible values used by that statement, preconditions which specify the conditions under which the statement was executed in those output traces, and the values produced of the execution of the statement.

PDDL can be used to represent primitive program statements, however it's not very suitable to represent compound program blocks, control structures such as branches and loops, and program methods. To model such constructs, HACKAR uses the *Hierarchical Task Network (HTN)* formalism (Erol, Hendler, and Nau 1994; Nau et al. 2003) on top of PDDL. HTN formalisms vary in terms of how they represent procedural descriptions, but all of them consist of the following abstract formalisms: the initial state (a symbolic representation of the state of the world at the time that the plan executor will begin executing its plan) and the goal task network (a set of tasks to be performed, along with some constraints over those tasks that must be satisfied). HTNs specify two main kinds of knowledge artifacts: *methods* and *operators*, described below.

Operators in HACKAR planning are represented in PDDL. The names of these operators are designated as *primitive tasks* (i.e., tasks that we know how to perform directly). Any task that does not correspond to an operator name is a *nonprimitive* task. Each method is a prescription for how to accomplish a non-primitive task by decomposing it into subtasks which may be either primitive or non-primitive tasks. A method consists of three elements: (1) the task that the method can be used to accomplish, (2) the set of preconditions which must be satisfied for the method to be applicable, and (3) the subtasks to accomplish.

Code Trace and Test Generation

HACKAR learns HTN methods and operators from program execution traces, which are generated by its PROTECT component. PROTECT uses Java Pathfinder (JPF) (Havelund and Pressburger 2000; Visser et al. 2003; Visser, Pasareanu, and Khurshid 2004) to execute programs, and a customized JPF listener to capture alternative program paths, resulting in different execution traces. The Symbolic PathFinder (SPF) extension of JPF enables symbolic execution of programs that can compute program inputs that guarantee execution of each program path. SPF executes Java methods by computing the constraints that must be satisfied by each program branch. SPF computes assignments to the method input parameters that satisfy the constraints by encoding and solving a Satisfiability Modulo Theory (SMT) problem. By negating the constraints imposed by a particular branch, SPF can explore each branch of a program.

PROTECT uses a custom JPF listener that reports the start and end of each of line of code and the entry and exit of each method. The listener records the data consumed and produced by each line of code, and the data passed to and returned by each method. From this hierarchical summary of program execution, the SHOW component in HACKAR learns operators and methods that describe the program. The data consumed and produced by lines of code map to operator preconditions and postconditions. The data passed to and returned by methods map to HTN methods.

Most SMT solvers used for reasoning over program constraints can natively handle symbolic data involving integers and floating point numbers. However, strings are often a fundamental limitation for SMT solvers. Though JPF's SMT solver can handle some rudimentary string operations, it cannot handle strings in general. To mitigate this issue, we use the well-known test case generation abilities offered by symbolic execution and combine it with a novel approach to test generation for strings that can take user-specified grammars on string inputs. The grammar-based theories help JPF reason with possible successful and failure test boundaries; i.e., they specify "interesting" program input values that might cause the program to fail or make the program complete its execution successfully.

Hierarchical Workflow Learning

The objective of hierarchical workflow learning is to learn the dataflow in the original program, as well as the causes of potential vulnerabilities highlighted by program execution, and generalize those dataflows in order to infer ways to fix the program. HACKAR uses the mathematical and structural foundations in HTN formalisms as a formal, high-level Java specification language for the program structures such as Java methods, program blocks, and statements.

Given the program execution traces generated by JPF, and the PDDL models for the primitive Java statements as planning actions, HACKAR's SHOW component learns HTN methods that represent compound Java program blocks. More specifically, the learning system represents a program block with its "parameters" as a planning *HTN method*, as described above. A Java method can be naturally translated into an HTN method definition, in which the Java method's name and parameters is the task for the HTN method and the statements and calls to the other functions in the Java method constitute the subtasks for the HTN method. For each other program blocks, such as a loop or a branch, that does not have a name or parameters explicitly in the source program, SHOW creates a special task symbol. The parameters of this task are generated by collecting all the program variables that are not defined locally in the block. Using this translation, SHOW learns an HTN method for any program block in a Java program.

SHOW learns both the task semantics and HTN methods from input traces for specific tasks. Unlike most of the existing HTN learners, SHOW learns the semantics of tasks; i.e., the applicability conditions and effects of Java methods or other program blocks. This enables HACKAR to reason about possible vulnerabilities and their fixes semantically and correctly. Given program execution traces generated by PROTECT, SHOW produces a knowledge base of HTN methods in a planning domain incrementally; that is, it successively updates its knowledge base when presented with new program execution traces for the same program

without requiring a large set of training data to be given a priori as in most traditional machine-learning algorithms.

The fundamental computation in SHOW is based on the well-known *goal regression* technique (Reiter 1991). Goal regression works toward finding some set of atoms g' such that, if we reach a state s in which they hold, we know a procedure to transform s' into a state s in which our goals g hold. In classical goal regression, the procedure would be a plan, but in our case, it is a task network that represents a Java program block. Given a set of goals g and a task network w, we can find the set $g' = R(g, w)$, extending the regression operator defined in (Reiter 1991):

- If w is the empty task network, then $R(g, w) = g$.
- If w contains a single primitive task t (i.e., a program statement), which corresponds to the action $a = (t, \mathrm{pre}(a), \mathrm{eff}(a))$, then $R(g, w) = (g \setminus a+) \cup \mathrm{pre}(a)$, where $a+$ denotes the positive effects in $\mathrm{eff}(a)$ (i.e., effects that do not make a fact false in the state of the world).
- If w contains a single nonprimitive task $(t\ \mathrm{pre}(t)\ \mathrm{eff}(t))$ for which we have an indexed method instantiation $m = (t\ \mathrm{pre}(m)\,\mathrm{subtasks}(m))$, then $R(g, w) = (g\ t+) \cup \mathrm{pre}(m)$, where $t+$ is the positive effects in $\mathrm{eff}(t)$.
- If w contains two or more tasks $\langle t_0, t_1, \ldots, t_n rangle$, then $R(g, w) = R(R(g, t_n), \langle t_0, t_1, ..., t_{n-1}\rangle)$.

The value of $R(g, w)$ is the minimal set of atoms that must be true in a state s' to guarantee that w will be decomposable resulting in a plan that produces a state s where g holds.

SHOW uses several mechanisms from our previous work in HTN learning (Hogg, Muñoz-Avila, and Kuter 2008; Hogg, Kuter, and Muñoz-Avila 2009; 2010). It uses a generalization of goal regression, called *hierarchical goal regression*, that can regress goals both horizontally (through the primitive actions) and vertically (up the task hierarchy). In hierarchical goal regression, a logical formula can be regressed over either a primitive action or an indexed method instance for a nonprimitive task. In the case of the former, the regression is performed using the preconditions and effects of the action in the same manner as traditional goal regression. In the case of the latter, the regression is performed over the postconditions of the annotated task and the preconditions of the method learned for that task.

Code Advice Planning and Ranking

In order to produce a new planning problem to produce a new Java program, AIDE uses the new HTN methods learned by SHOW, which specify improved versions of the vulnerability-prone methods in the program. The new HTN methods typically describe to program blocks with safety guards on values before their execution, such as checks that an index is in-bounds, or that a read length isn't larger than the size of the buffer.

We have extended SHOP2 (Nau et al. 2003) with a technique, which we call *blacklisting*. We provide SHOP2 with a list of methods that should only be used if no other means to solve the problem exists. These blacklisted HTN methods correspond to program fragments which SHOW has discovered contribute to vulnerable behaviors in the underlying Java program. The hope is that by avoiding methods we know to be unsafe in favor of learned alternatives, HACKAR, and AIDE specifically, can produce safer versions of programs by avoiding behavior automatically determined to be unsafe.

AIDE performs two planning epochs on the input HTN methods representing the program. In the first epoch, it uses the HTN methods that correspond to the original program, including the blacklisted methods. This generates the control-and-data-flow of the program that leads to vulnerabilities. In the second epoch, AIDE plans for the same program using the learned, vulnerability-free methods produced by SHOW as well as a blacklist of the unsafe methods that lead to the detected vulnerabilities. Thus, the first planning epoch produces a hierarchical execution tree representing the original flawed program, while the second produces an execution tree representing a program built with methods designed to avoid the original vulnerability.

AIDE is able to produce a diff, or a patch, by considering these two execution trees by walking the both from their roots. This procedure generates the suggestions to fix the program as follows: during the tree walk, AIDE compares each subtree in the two execution trees, starting from the roots and following the control flow of the program. When AIDE detects two different subtrees, it compiles the applicability conditions on both trees generated by SHOP2 and extracts those conditions stemming from the new, vulnerability-free HTN method that do not exist in the original one. These applicability constitute the safe-guard SHOW has learned for that particular program block to protect it against the discovered vulnerability. AIDE returns to the user the safe applicability conditions as a suggestion to modify the program, along with the program block that needs to be protected.

Evaluation and Validation

HACKAR is implemented in Common Lisp and integrates Java PathFinder (JPF), which is written in Java. The test cases (i.e., Java programs) for our experiments were developed based on MITRE's CWE (Common Weakness Enumeration) and CVE (Common Vulnerability Enumeration) databases that outline potential vulnerabilities in systems and programs written in Java, C, C++ and other programming languages.[2] We have identified several vulnerability/weakness types based on the latest CWE/CVE: division by zero errors, initialization exceptions, stack-overflows with intensive mathematical computing such as Euclidean GCD computations, file I/O errors, overflowing system file handlers, array indexing errors, buffer overflow and underflow scenarios, bad conditional/switch statements, casting errors, and multiple vulnerabilities that are dependent on each other or may be unrelated from each other.

We generated our test programs synthetically (i.e., generated by a script) as well as manually (i.e., written by a Java programmer). The manually constructed programs fall into two categories: those generated by HACKAR developers

[2]MITRE CWE/CVE: http://cwe.mitre.org.

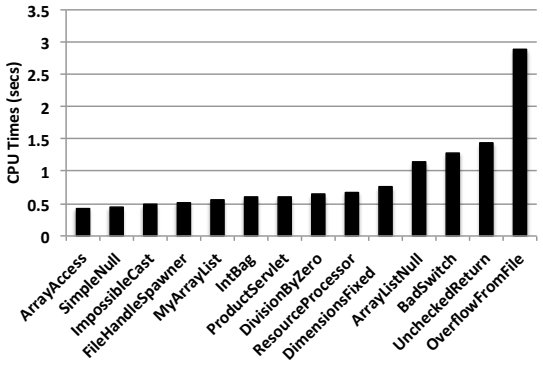

Figure 2: HACKAR's scability performance.

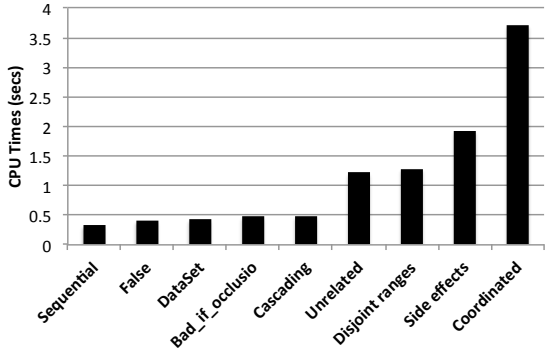

Figure 3: HACKAR's scability performance over programs with multiple vulnerabilities..

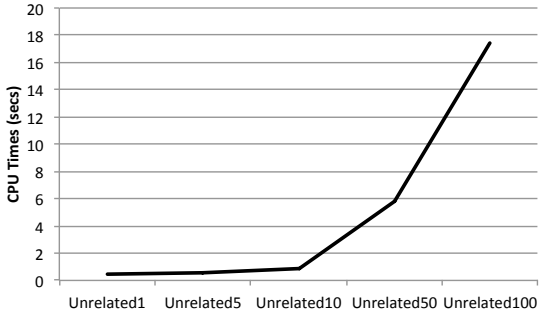

Figure 4: HACKAR's scability performance over synthetic programs with multiple unrelated (i.e., independent) vulnerabilities, as a function of the number of exceptions in the programs.

to illustrate (or test) specific portions of the HACKAR process, and those written by Java developers with no knowledge of HACKAR or JPF, including variants of open-source Java programs available on the Web (e.g. portions of the Amazon Web Services code library).

The core criteria in our experiments was to remove vulnerabilities from Java programs. The quality of the evaluation was measured in terms of the following factors:

- whether HACKAR can find a sound correction to a vulnerability 90% of the time;
- whether HACKAR can find the *earliest* correction in 90% of the time; and
- average running (CPU) times in seconds to find a correction is below 5 seconds.

Experiments with HACKAR

Comparative evaluation of the HACKAR system is difficult because we have not found a similar system that can dynamically test inputs of the programs and perform vulnerability analysis over execution traces of them. Hence we performed our evaluation of HACKAR against well-defined, easy-to-measure performance metrics.

Figure 2 shows the average CPU times of the end-to-end HACKAR system to generate a suggestion/correction to a vulnerability. HACKAR generated correct suggestions for all of our experimental programs. On average, the system identified the vulnerabilities and returned suggestions in 1.5 seconds, excluding three of our experimental programs where HACKAR's run-times were above 20 seconds, and in one case, it was 209.27 seconds. We analyzed HACKAR's behavior and concluded that generalization is not strong enough on programs with loops and HACKAR's learner's state models become very large. Since our planning algorithms use the same models, they produce suggestions very inefficiently. We're currently working on a new HTN learning algorithm to address this generalization issue.

We have also evaluated HACKAR on programs that include multiple vulnerabilities. Our experimental programs for this case included scenarios with unrelated errors, sequential exceptions, and cascading vulnerabilities with dependencies among them. Figure 3 shows the results of the experiment. HACKAR generated correct suggestions in under 4 secs on our programmer-written scenarios. Compared to the previous experimental suite, which include programs with single vulnerability, the system performed slower, due to more complex nature of workflows in the multiple-vulnerability programs.

We have also generated synthetic program suites for evaluating HACKAR on multiple-vulnerability analysis. When errors were sequential, HACKAR was able to generate correct suggestions under a second. Figure 4 show the results of these experiments on programs when the errors were unrelated (i.e., independent). The rate of performance of the system increases with the number of the independent errors in the programs. The fundamental reason for this performance degradation is that there are larger number of possible causal explanations to learn when programs have independent errors. This is a similar research issue as in Diagnosis problems (De Kleer and Williams 1987), and we are currently working on ideas to borrow from that literature.

Standalone evaluation of JPF within HACKAR

We have also performed an evaluation of the generalizations we made to the vanilla JPF. Most notably, we performed an ablation study in order to evaluate the impact of

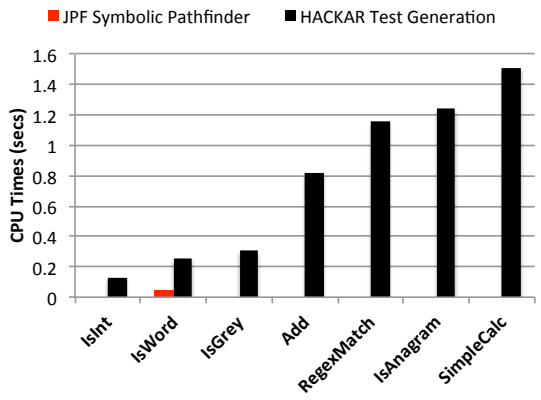

Figure 5: Comparison of the JPF's performance on programs with string inputs with our JPF-variant that uses grammar-based string generation. The original JPF was only able to generate test cases for the program "IsWord," which contains only primitive string comparison operation.

our test-generation algorithms we integrated in JPF. Figure 5 shows the results of the experiments, where we compared HACKAR with JPF using our grammar-based test generation over strings and with JPF without those new techniques.

JPF's test input generation algorithms (Visser, Pasareanu, and Khurshid 2004) allow simple string operations in symbolic and concrete execution, but cannot handle string inputs and operations in general. Our experiments showed that the new grammar-based algorithms we have incorporated into JPF alleviate this shortcoming; HACKAR with the new algorithms were able to generate execution traces for all of our programs with string inputs, whereas JPF was able to handle only one of the programs.

Additional experiments with JPF demonstrated that JPF can require considerable time to execute programs with many paths. In such cases, we impose a limit on the number of paths executed. We are currently researching alternative search strategies to quickly identify a subset of the most important program execution paths. By important paths, we seek those that are qualitatively different and exhibit either error-free or erroneous program behavior.

Conclusions and Future Work

We have described HACKAR, our novel architecture that combines program analysis and AI techniques in a unique way in order to discover vulnerabilities in Java programs and learn how to fix them. The learned information is returned to the programmer during development time as suggestions on how to protect his/her program. Experiments show that HACKAR can provide correct suggestions for fixing program vulnerabilities in 100% of our experimental suite, and it does so within a few seconds in most of the scenarios.

We are currently working to generalize HACKAR's scope of programs to include analyzing complex Java objects and external library calls/execution within Java programs. The latter is particularly useful for mobile apps written in Java or Java-like languages such as Android programs and in Web service analysis.

HACKAR can also be used for autonomic analysis and reasoning problems, where a program can monitor itself against potential vulnerabilities. As a near-future work, we're investigating this prospective research direction based on the technologies in HACKAR.

Acknowledgments. This research is funded by Contract N00014-12-C-0239 with Office of Naval Research (ONR). The views expressed are those of the authors and do not reflect the official policy or position of the U.S. Government.

References

De Kleer, J., and Williams, B. C. 1987. Diagnosing multiple faults. *Artificial intelligence* 32(1):97–130.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. Semantics for hierarchical task-network planning. Technical Report CS TR-3239, UMIACS TR-94-31, ISR-TR-95-9, University of Maryland.

Fox, M., and Long, D. 2001. PDDL+ level5 : An extension to PDDL2.1 for modeling domains with continuous time-dependent effects. technical note, University of Durham.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)*.

Havelund, K., and Pressburger, T. 2000. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer* 2(4):366–381.

Hogg, C.; Kuter, U.; and Muñoz-Avila, H. 2009. Learning hierarchical task networks for nondeterministic planning domains. In *IJCAI-09*.

Hogg, C.; Kuter, U.; and Muñoz-Avila, H. 2010. Learning methods to generate good plans: Integrating htn learning and reinforcement learning. In *AAAI-10*.

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required. In *AAAI-08*.

McDermott, D. 1998. PDDL, the planning domain definition language. Technical report, Yale Center for Computational Vision and Control.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research (JAIR)* 20:379–404.

Reiter, R. 1991. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Lifschitz, V., ed., *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, (Ed.)*. Academic Press.

Visser, W.; Havelund, K.; Brat, G.; Park, S.; and Lerda, F. 2003. Model checking programs. *Automated Software Eng.* 10(2):203–232.

Visser, W.; Pasareanu, C. S.; and Khurshid, S. 2004. Test input generation with java pathfinder. *ACM SIGSOFT Software Engineering Notes* 29(4):97–107.