ActionScript Bytecode Verification With Co-Logic Programming [*]

Brian W. DeVries
The University of Texas at Dallas
brian.devries@student.utdallas.edu

Gopal Gupta
The University of Texas at Dallas
gupta@utdallas.edu

Kevin W. Hamlen
The University of Texas at Dallas
hamlen@utdallas.edu

Scott Moore
The University of Texas at Dallas
scott.moore@student.utdallas.edu

Meera Sridhar
The University of Texas at Dallas
meera.sridhar@student.utdallas.edu

Abstract

A prototype security policy verification system for ActionScript binaries is presented, whose implementation leverages recent advances in co-logic programming. Our experience with co-logic programming indicates that it is an extremely useful paradigm for elegantly expressing algorithms that lie at the heart of model-checking technologies. This results in an unusually small trusted computing base, making the verification system well-suited to frameworks like certifying in-lined reference monitoring systems, which require small, light-weight verifiers. Preliminary experiments and progress are discussed.

Categories and Subject Descriptors D.2.4 [*Programming Languages*]: Software/Program Verification; D.4.6 [*Operating Systems*]: Security and Protection—access controls; D.3.2 [*Programming Languages*]: Language Classifications—constraint and logic languages

General Terms Languages, Security

Keywords ActionScript, Verification, Coinductive Logic Programming, Model Checking, In-lined Reference Monitoring

1. Introduction

ActionScript is a powerful, emerging mobile code language similar to Java and .NET bytecode. It is rapidly becoming a mainstay of modern web browsing technologies. Recently, it has also gained popularity as the base language for the Adobe Integrated Runtime (AIR), a cross-platform environment that allows third-party code-producers to construct ActionScript applications for the desktop.

With increased ubiquity comes increased exposure to malicious ActionScript code and therefore a need for protection systems that defend against it. Already ActionScript has served as a medium for numerous malware attacks. The first ActionScript virus (SWF/LFM-926) was identified in 2002, followed by numerous others including the Troj/SWFexp, Sus/SWFScene, Troj/SwfDL, and Troj/SWFdldr families of viruses, to name a few.

Existing security mechanisms for ActionScript and related technologies (including the ActionScript Virtual Machine, Flash, Flex, and AIR) mainly fall into two categories: code-signing and sandboxing. While these suffice for certain classes of attacks, code-signing places the code-producer in the Trusted Computing Base (TCB), and sandboxing enforces only a small class of coarse-grained access control policies built into the ActionScript virtual machine and runtime libraries. System- and application-specific policies, such as those that prohibit write access to files with certain names, or finer-grained policies that constrain arguments to individual ActionScript instructions (such as those that facilitate buffer overrun attacks) are not supported. Additionally, to our knowledge there has been little formal work that studies the language and its security implications, and therefore few tools for analyzing security vulnerabilities and threats in ActionScript bytecode binaries. It is therefore desirable to develop formal analysis techniques that fill this void.

In this paper we report on preliminary work toward developing an ActionScript verifier, with an LTL model-checker at its core written using co-logic programming. *Co-logic programming* (Co-LP) [30, 13, 4] is a cutting-edge logic programming technology that combines tabled and coinductive logic programming and allows extremely elegant and succinct formalization of properties defined in terms of least

[*] This research was supported by AFOSR YIP award number FA9550-08-1-0044, NSF award DGE 0742477 and an IASP scholarship from the Department of Defense.

PLAS '09 June 15, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-645-8/09/06...$10.00

and greatest fixed-points. Such properties lie at the heart of many model-checking analyses, such as those that regard potentially non-terminating loops. Our work applies this technology to build an unusually small but powerful ActionScript verification system, further minimizing the TCB of our framework.

Our verifier is particularly suited as a certification system for an ActionScript *In-lined Reference Monitoring (IRM)* system, currently under development at the University of Texas at Dallas. IRM's [27] enforce safety policies by injecting runtime security guards directly into untrusted binaries. The runtime guards test whether an impending operation constitutes a policy violation. If so, some corrective action is taken to prevent the violation, such as halting the program prematurely. The result is *self-monitoring* code that can be safely executed without external monitoring [14].

IRM's are powerful tools that can enforce a large class of policies including most safety properties [17], but the rewriters that in-line guards into the untrusted code can be large and complex, and therefore potentially difficult to trust. Certifying IRM systems [16] verify that the output of the rewriter constitutes a policy-adherent binary. This shifts the (typically larger) rewriter out of the TCB in favor of a (typically smaller) verifier.

Our verifier expresses security policies using *Linear Temporal Logic (LTL)* [23], which extends propositional logic with temporal operators. LTL underpins many modern software model checking technologies, and allows us to conveniently draw upon existing techniques from the field. With LTL, policy writers can specify security policies in a purely declarative and compositional manner; furthermore, formal analysis techniques, such as satisfiability checking [25], assist the policy writer in creating semantically valid policies.

The remainder of the paper is structured as follows. Section 2 discusses related work. Section 3 gives a detailed overview of our project, including its high-level design and main features. Section 4 presents the details of our implementation. Section 5 discusses the results of a few preliminary experiments. Finally, Section 6 suggests future work.

2. Related Work

ActionScript was developed by Adobe Systems as a scripting language for writing Flash Applications. ActionScript 1.0 was released in 2000 and includes Java-like language features such as an object model, function calls, dynamic types, and class inheritance. The most recent version (3.0) was released in 2006 and introduced a new VM, compile-time and run-time type checking, packages, namespaces, regular expressions, and direct access to security-relevant system resources such as the Flash runtime display list. The fact that the ActionScript language has become increasingly sophisticated and is used as the core for an increasing number of platforms demonstrates that it is an important language for language-based security researchers to study.

Prolog is an attractive language for implementing program analysis tools, as program semantics and decision procedures can be efficiently written in a concise and declarative manner. At least two major extensions to the standard resolution semantics of Prolog have further enhanced its usefulness in this area in recent years: tabled logic programming and coinductive logic programming—collectively, co-logic programming.

Tabled logic programming [7, 31] extends Prolog's standard inductive semantics with memoization and termination of cyclically infinite search paths. As a tabled recursive predicate executes, calls and intermediate solutions are automatically stored in a table. If a variant (duplicate) call is encountered that would normally cause the computation to cyclically diverge, the call is replaced by a previously discovered solution, if it exists; otherwise, the call is suspended while other alternatives are tried. Any solutions found by these alternatives are stored in the table and can be substituted for the suspended calls as well. This substitution process can be repeated until no new solutions can be found, indicating that the tabled solutions correspond to the least fixed-point of the original call. Any suspended calls will fail at this point, as no new solutions can be found inductively.

Coinductive logic programming [30, 13, 4] allows Prolog to reason about cyclically infinite data structures and proof trees. When a recursive call to a coinductive predicate is made, the call stack is searched for a unifiable ancestor call; if found, the call is unified with its ancestor and succeeds. The solution obtained from the unification corresponds to the greatest fixed-point interpretation of the original call, meaning a coinductive predicate can generate a proof without encountering—or even specifying—a base case.

Because tabled logic programming and coinductive logic programming correspond to the least and greatest fixed-point semantics of proof derivations, respectively, they are useful for implementing program analyses. In particular, program loops are by nature cyclic structures for which both inductive and coinductive properties are important for establishing policy-adherence. While many interesting coinductive properties can be proved by the absence of an inductive counterexample, the use of coinductive logic programming allows constructive proofs of these properties to be computed. Co-LP, which subsumes both of these strategies, is therefore particularly well-suited because it allows least and greatest fixed-point analyses to be combined in a well-defined and semantically meaningful manner.

Related research on general model-checking is vast, but there has been less work on applying model-checking to verifying virtual machine bytecode binaries. Instead, the majority of model-checking research has focused on detecting deadlock and assertion violation properties of source code. For example, Java PathFinder (JPF) [20] and Moonwalker [26] verify properties of Java and .NET source programs, respectively. Both provide built-in support for deadlock detection and unhandled exceptions, but not LTL model-checking. Other model-checking research [24, 5] has targeted abstract languages, such as the π-calculus [21] or Calculus of Communicating Systems (CCS). While important, these systems do not address certain significant practical issues, such as state space explosion, that typically arise when model-checking real software systems.

In-lined Reference Monitoring and program rewriting for security enforcement was first formalized by Erlingsson and Schneider [11, 27]. Subsequently, a wide variety of IRM implementations have been developed. ConSpec [2] restricts IRM-injected code to effect-free operations, which allows a static analysis to verify that a rewritten program does not violate the intended policy. The Java-MOP system [6] allows policy-writers to use a variety of formal specification languages, including LTL. Mobile [16] is an In-lined Reference Monitoring system for the Microsoft .NET framework. It rewrites .NET CLI programs to satisfy a declarative policy specification by transforming the program into a well-typed Mobile program. Finally, SPoX [15] rewrites Java VM bytecode programs to satisfy declarative, Aspect-Oriented security policies.

To our knowledge, ConSpec and Mobile are the only IRM systems to yet implement automatic certification. The ConSpec verifier performs a simple static analysis to verify that pre-specified guard code appears at each security-relevant code point; the guard code itself is trusted. Mobile implements a more general certification algorithm by type-checking the resulting Mobile code. While type-checking has the advantage of being light-weight, it comes at the expense of limited computational power. For instance, Mobile cannot enforce security policies based on data-flow; instead, it is limited to control-flow based policies. While the security policies described by these systems are declarative and therefore amenable to a more general verifier, both use a verifier tailored to a specific rewriting strategy.

3. Overview

ActionScript source code is typically compiled to ActionScript Virtual Machine (AVM) bytecode [1] and subsequently interpreted by the AVM. Our verifier models the AVM in Prolog and extends the semantics of each AVM instruction to include security-relevant state. It takes as input a program in ActionScript ByteCode (ABC) format and a security policy (expressed in LTL) and conservatively decides whether every program execution satisfies the policy.

We derive a particularly elegant implementation for our verifier by observing that an abstract interpreter with coinductive semantics facilitates the realization of a model checker. That is, where an interpreter loops on non-terminating programs, a model-checker based on co-LP succeeds (and terminates) when it revisits an abstract state that constitutes a valid loop invariant. This reduces much of the machinery normally required for model-checking to the relatively simple framework required for a standard abstract interpreter.

```
event(callpropvoid(open,[File,_Mode]),open) :-
    in_directory(File,'~/secret_files/').
event(callpropvoid(readByte,_Args),read).
event(callpropvoid(readUTF,_Args),read).
event(callpropvoid(writeByte,_Args),send).

ltl_policy(g(impl(open,g(impl(read,not(f(send))))))).
```

Figure 1. A policy prohibiting network-sends after file-reads from a restricted directory.

Both tabled and coinductive aspects of co-LP are useful in our approach. Using tabled LP, we implement model-checking as a search for a counter-example. Tabled LP semantics yield such a counterexample when this search succeeds. In the case of model-checking, the counter-example is a policy-violating sequence of AVM instructions that can be useful to a developer wishing to produce policy-adherent code. Coinductive LP yields a constructive proof of correctness when the search for a counter-example fails. Such a proof could be attached to the verified bytecode for use with a Proof-Carrying Code (PCC) system [22]. While our current prototype does not yet produce a complete, machine-readable proof automatically, most of the machinery necessary for implementing this feature has already been completed as a natural result of our use of co-LP as the foundation for the verifier. We therefore expect to add this feature relatively painlessly in future work.

Policies are expressed as LTL formulas encoding the set of all permissible sequences of security-relevant events. To reflect the reality that a real process may terminate at any point (e.g., due to practical issues like hardware failures or power outages), we model every program state as potentially terminating. This model limits our system to verifying safety policies; non-safety LTL properties are conservatively rejected. While LTL is therefore a more expressive language than necessary for our purposes, it is nonetheless convenient due to its familiarity in the model-checking community and the existence of many well-developed tools for writing and reasoning about LTL-specified policies.

As an example, Figure 1 specifies a policy that prohibits network-send operations after a file in a specific directory has been opened. This policy often appears in the IRM literature as a canonical example of a history-based policy that enforces data confidentiality. The `event` predicate defines three security-relevant events: `open`, `read`, and `send`. Method calls to `writeByte` (regardless of arguments) constitute `send` events. Method calls to `readByte` or `readUTF` constitute `read` events, demonstrating that events can abstract across multiple actions. The `open` event is defined as a call to the `open` method, where the first argument (the name of the file to open) satisfies the predicate `in_directory`. Here, the predicate `in_directory` checks whether the file name given resides within the prohibited directory.

The policy on line 7 encodes the LTL formula $\mathcal{G}(\texttt{open} \rightarrow \mathcal{G}(\texttt{read} \rightarrow \neg\mathcal{F}\texttt{send}))$. Informally, it stipulates that no execution may contain the sequence $(\texttt{open}\ldots\texttt{read}\ldots\texttt{send})$. This specification has been simplified for expository purposes, but a discussion of how to elaborate the specification for a real application is available in [18].

To keep the verifier implementation tractable, we make several important assumptions about untrusted programs that simplify the analysis. First, we assume that the compiled ABC code is syntactically valid. This can be verified separately by the AVM bytecode verifier. We also assume that AVM programs do not perform runtime code generation, since our analysis is limited to the statically observable code. Since runtime code generation is only available to ABC programs via a limited set of system calls, this assumption can be enforced in policy specifications by prohibiting calls to those methods.

Our dataflow analysis is conservative in the sense that it abstracts away certain details of the heap and program variables. This can cause some policy-adherent code to be conservatively rejected (though it never results in policy-violating code being accepted). However, we argue that this limitation can be overcome by an IRM system that inserts additional security guards that ease the verification process. The security guards effectively reduce the state space by explicitly ruling out unrealizable control flows and their associated data flows.

Our early prototype does not yet implement an interprocedural analysis. This limits our current experiments to simple programs whose security-relevant behavior does not involve method calls. We plan to extend our model-checker with a suitable analysis in future work.

4. Implementation

There are three major components to our verifier: a parser for ActionScript ByteCode, an encoding of AVM semantics, and an LTL model-checker. Each component's implementation is described throughout the remainder of the section.

4.1 ABC File Parser

Since typical code consumers do not have access to the original ActionScript source code, we verify compiled ActionScript Bytecode (ABC) files directly. Our ABC parser is a Definite Clause Grammar (DCG) that transforms a file in ABC binary format [1] to an annotated abstract syntax tree (AST). For more information about DCGs and their Prolog implementations, the reader is invited to consult [29].

4.2 AVM 2 Semantics

The AVM2 semantics module is a declarative implementation of an ActionScript VM derived from the AVM2 small-step semantics [1]. Using these semantics, we translate an ABC program into a transition system for use with the model-checker. The *transition system* [3] consists of a set of *states* that model the VM state at each point in program execution, and a *transition relation* that relates each state to the states reachable from the current instruction. We encode the transition relation as a Prolog predicate that is invoked by the model-checker as it explores the state space.

To avoid state space explosion, we use an abstract interpretation rather than a concrete interpretation. A concrete data model is inadequate because it requires every possible AVM memory configuration to be represented by a unique state in the transition system, and searching the resulting state space becomes intractable. In contrast, an abstract interpretation allows a collection of possible variable values to be represented by a single abstract state. The precision of this abstraction determines the power of our analysis. For instance, modeling all program values as unknown results in a simple control-flow analysis. While this would ensure that every policy-violating program is rejected, it would conservatively reject many policy-adherent programs, severely limiting the usefulness of the verifier. In particular, since IRMs enforce security policies by tracking security state in injected program variables, a control-flow analysis cannot verify that IRM-inserted guards that consult these variables suffice to prevent policy violations.

To avoid this limitation, we use concrete values where they can be statically determined and use the abstract state $\top$ otherwise. Our AVM2 semantics are therefore extended with transitions for $\top$. For instance, the `ifeq` instruction semantics in Figure 2 model the possibility of executing either branch when the outcome of the conditional expression cannot be determined statically; otherwise only the statically inferred branch is interpreted. Note that since model-checking explores each execution trace independently, the set of statically inferable values is much larger than would be available in a more traditional static analysis. The more powerful data-flow analysis provided by this abstraction captures relationships between data values and event sequences that the program might exhibit at runtime. As a result, the verifier can track the values of IRM security state variables.

4.3 Model Checker

The LTL model checker module takes as input a security policy specified as an LTL formula and the transition system supplied by the AVM2 semantics. Recall that LTL extends propositional logic with temporal operators; the logical propositions correspond to the individual security-relevant events, and the temporal operators specify the valid event sequences. Given two LTL formulae a and b, the formula $\mathcal{X}a$ mandates that a holds in the *next* state, $\mathcal{F}a$ mandates a holds in some *future* state, $\mathcal{G}a$ mandates that a holds *globally* (i.e., in the current and all subsequent states), $a\mathcal{U}b$ mandates that b holds in some future state and a holds in every state *until* that point, and $a\mathcal{R}b$ mandates that b holds in the current and all subsequent states until this requirement is *released* by a state where a and b both hold.

```
% trans/3 encodes the transition relation
% EE is the current execution environment: the scope
% stack, the operand stack, the register file, and
% the current instruction.
% NewEE is the resulting execution environment
trans(ifeq,state(H,[EE|EEs]),state(H,[NewEE|EEs])) :-
    EE =.. [env, SS, [V1,V2|Os], RF, Instr],
    equal(V1, V2, TruthValue),
    next_instr(TruthValue,Instr,NewInstr),
    NewEE =.. [env, SS, Os, RF, NewInstr].

% equal/3 returns true or false if equality between
% the arguments can be determined, otherwise top (tt)
equal(int(V1), int(V1), bool(true)).
equal(int(V1), int(V2), bool(false)) :- V1 =\= V2.
equal(V1, V2, tt) :- V1 \= int(_X); V2 \= int(_Y).

% next_instr/3 gives the next instruction to be
% executed, based on the result of the comparison.
% If the result is top, both branches are searched.
next_instr(bool(true), Instr, NewInstr) :-
    get_property(Instr, jumplabel, NewInstr).
next_instr(bool(false), Instr, NewInstr) :-
    get_property(Instr, next, NewInstr).
next_instr(tt, Instr, NewInstr) :-
    get_property(Instr, jumplabel, NewInstr).
next_instr(tt, Instr, NewInstr) :-
    get_property(Instr, next, NewInstr).
```

Figure 2. The `ifeq` transition relation clause

The expansion rules for the LTL operators [3] provide a declarative semantics for the interpretation of LTL formulae by constraining the current state and immediate successor states. These rules can be encoded directly in Prolog to obtain an interpreter for LTL formulae, as seen in Figure 3. To check whether a path through the transition system satisfies a given LTL formula, the model checker recursively invokes the expansion rules, first checking the requirements placed on the current state, then making a transition to the next state on the path and repeating the process.

Using these expansion rules, we can cleanly separate the temporal operators into two categories: $\mathcal{X}$, $\mathcal{F}$, and $\mathcal{U}$ are inductive, as their expansion rules provide base cases for deciding whether a path satisfies the operator, while $\mathcal{G}$ and $\mathcal{R}$ are coinductive, as they are satisfied by a cyclically infinite execution path. Based on this distinction, our implementation relies on tabling and coinduction to reason about loops encountered when traversing the transition system—without these extensions, loops would cause the model checker to diverge in an attempt to construct the proof for a formula. Tabling terminates the proof search with a failure if an inductive formula does not hold for a loop, while coinduction terminates the search with the infinite-length proof when a coinductive formula holds.

5. Experiments

As a preliminary test of our prototype, we verified three small test programs against the policy specified in Figure 1, which disallows network-send operations after a file has been read from a certain restricted directory. The three test programs—`Unsafe.as`, `Safe.as`, and `Loop.as`—were each compiled from ActionScript source code. The first,

```
% verify/2 takes a state and an existentially
% quantified LTL formula and checks
% whether the formula holds for that state.
%
% Atomic Propositions are labeled by 'ap'.
%
% holds/2 is true when the atomic proposition holds
% in the current state
%
% ftype/2 is a mapping from top-level temporal
% operators to their interpretation semantics
%
% The clause for 'a and b' should ensure that 'a' and
% 'b' hold on the same execution path. For simplicity
% of presentation, we omit this check here.

verify(State, F) :- ftype(F, inductive),
        verify_inductive(State, F).
verify(State, F) :- ftype(F, coinductive),
        verify_coinductive(State, F).

:- tabled verify_inductive/2.
verify_inductive(S, ap(AP)) :- holds(S,AP). % p
% Logical operators
verify_inductive(S, not(ap(AP))) :-      % not(p)
        \+ holds(S, AP).
verify_inductive(S, or(A,B)) :-          % a or b
        verify(S, A) ; verify(S, B).
verify_inductive(S, and(A,B)) :-         % a and b
        verify(S, A), verify(S, B).
% Inductive temporal operators
verify_inductive(S, x(A)) :-             % X(a)
        trans(S, S1), verify(S1, A).
verify_inductive(S, f(A)) :-             % F(a)
        verify(S, A); verify(S, x(f(A))).
verify_inductive(S, u(A,B)) :-           % a U b
        verify(S, B);
        verify_inductive(S, and(A, x(u(A,B)))).

:- coinductive verify_coinductive/2.
% Coinductive temporal operators
verify_coinductive(S, g(A)) :-           % G(a)
        verify(S, and(A, x(g(A)))).
verify_coinductive(S, r(A,B)) :-         % a R b
        verify(S, and(A,B)).
        % {a and b both occur, releasing b}
verify_coinductive(S, r(A,B)) :-
        verify(S, and(B, x(r(A,B)))).
        % {a does not hold, so b is not released}
```

Figure 3. A simple Co-LP LTL model checker

`Unsafe.as`, exhibits policy-violating behavior when executed. The second, `Safe.as`, was obtained by instrumenting `Unsafe.as` with runtime security guards similar to those that an IRM system would typically insert as part of the binary-rewriting process. This involved adding an additional program variable that tracks the current security state at runtime, along with instructions that test and update the variable as security-relevant events occur. The third program, `Loop.as`, enclosed the security-relevant operations of the second program in an infinite loop, allowing us to test our loop analysis. The source code of all three programs can be seen in Figure 4. Code inserted to generate `Safe.as` is marked in the listing with `*`, and the additional code in `Loop.as` is marked with `#`.

In this program the value of `flag` cannot be inferred statically, so our analysis conservatively assumes that it can take on any integer value. Thus, the outcome of the test `(flag > 0)` in line 12 is not statically known, leading to

```
 1 public function Test(flag:int) {
 2   var socket:Socket = new Socket();
 3   var file:File = new File("secret.txt");
 4   var fileStream:FileStream = new FileStream();
 5 * var security:int = 0;
 6
 7   fileStream.open(file, FileMode.READ);
 8
 9   socket.connect("example.com", 1234);
10
11 # while (true) {
12     if (flag > 0) {
13       fileStream.readByte();
14 *     security = 1;
15     }
16
17 *   if (security != 1) {
18       socket.writeByte(0);
19 *   }
20 # }
21
22   socket.close();
23   fileStream.close();
24 }
```

Figure 4. Source code of the test programs

a possible policy violation at the network-send operation in line 18 if no runtime security guards were present. Lines 14 and 17, however, update and test (respectively) a new program variable `security` that tracks the current security state. In this case the state is 1 if a file has previously been read and 0 otherwise. Thus, testing `(security != 1)` in line 17 before each network-send event suffices to prevent a policy violation.

We ran each of these tests 10 times on an Intel Pentium Core 2 Duo machine with 4GB of RAM running Ubuntu Intrepid and Yap Prolog v5.1.4 [32]. The median runtimes are reported below. `Unsafe.as` was correctly identified as policy-violating, while `Safe.as` and `Loop.as` were correctly identified as policy-adherent.

`Unsafe.as`	0.093s
`Safe.as`	0.110s
`Loop.as`	0.101s

6. Conclusion and Future Work

We have described preliminary work toward developing a security policy verifier for Adobe ActionScript bytecode programs. Our verifier consists of an interpreter for ActionScript bytecode and an LTL model-checker, both written in Prolog extended with tabling and coinduction. Experiments demonstrated that our prototype can efficiently verify simple but interesting history-based policies for small ActionScript programs.

Verifiers are typically part of a secure system's trusted computing base. It is therefore important that the verifier itself be amenable to formal verification. The declarative nature of co-LP Prolog yields several significant advantages in this regard. First, our verifier code base is very concise—the parser is 2 kSLOC while the AVM2 semantics and the model-checker are each 1 kSLOC. Second, our experiences indicate that it is straightforward to encode semantic rules, such as those from the AVM2 Specification [1], as a relation between program states (see Figure 2). Finally, implementing the expansion rules for LTL in co-LP avoids a great deal of tedious and error-prone implementation work by relying upon the well-defined termination semantics of tabling and coinduction in Prolog (see Figure 3).

Like all static analysis techniques, our verifier conservatively rejects some policy-adherent programs. Our ongoing work is presently focusing on extending our analysis to reduce this conservative rejection rate. This involves introducing richer abstractions to model data dependencies and data flows. We intend to use constraint logic programming to elegantly implement these abstractions while minimizing the state space that must be explored to verify useful security properties.

Additionally, we will provide a means by which a code producer or enforcement mechanism can supply hints to the verifier. These hints will greatly increase the efficiency of the verifier by pre-computing the set of possible variable values at particular points in the program. These precomputed values need not be trusted since the verifier can ignore hints that are inconsistent with its analysis. This capability will be implemented in conjunction with our IRM system currently under development.

We also plan to generate explicit policy-adherence proofs. This involves enhancing current implementations of coinductive Prolog [13] to support enumeration of all coinductive proofs of a goal. After completing these efforts, generating these proofs should require minimal changes to our system.

Future work also should investigate applying our technique to other bytecode languages such as Java and .NET, and the IRM systems that have been implemented for them. We expect that the modular design of our framework will simplify the task of extending our implementation to cover these domains.

There is a large body of existing work on optimizing LTL formulae used in model checking (e.g., [8, 12, 28]). We also intend to explore the use of other temporal logics, especially Computation Tree Logic (CTL) and the μ-calculus [19], for the specification of security policies. The method mentioned in [10] suggests a means of modularizing the temporal logic engine out of the model-checker, allowing the same model-checking system to be used for multiple temporal logics by changing which temporal logic engine is used. Examining how to use several of these engines at once seems a promising direction for our tool as well. These techniques are explored further in [9].

Finally, we plan to study some of the many existing examples of malicious ActionScript code toward deriving practically useful policies and analysis strategies that enforce them. This should lead to more robust certification of mobile ActionScript code in practical settings.

References

[1] Actionscript virtual machine 2 overview, 2007. `http://www.adobe.com/devnet/actionscript/articles/avm2overview.pdf`.

[2] I. Aktug and K. Naliuka. ConSpec - A Formal Language for Policy Specification. *Science of Computer Prog.*, 74:2–12, 2008.

[3] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[4] A. Bansal. *Next Generation Logic Programming Systems*. PhD thesis, The University of Texas at Dallas, Dallas, Texas, 2007.

[5] S. Basu and S. A. Smolka. Model checking the Java metalocking algorithm. *ACM Trans. Softw. Eng. Methodol.*, 16(3):12, 2007.

[6] F. Chen. Java-MOP: A monitoring oriented programming environment for Java. In *In Proc. of the Eleventh International Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 546–550. Springer, 2005.

[7] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43:43–1, 1996.

[8] N. Daniele, F. Guinchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. In *Computer Aided Verification, Proc. 11th International Conf.*, volume 1633 of *LNCS*, pages 249–260. Springer-Verlag, 1999.

[9] B. W. DeVries. Developing an optimized LTL model checker in coinductive prolog, *forthcoming*. Master's thesis, University of Texas at Dallas, June 2009.

[10] L. K. Dillon and Y. S. Ramakrishna. Generating oracles from your favorite temporal logic specifications. *SIGSOFT Softw. Eng. Notes*, 21(6):106–117, 1996.

[11] U. Erlingsson and F. B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *Proc. of the New Security Paradigms Workshop*, 1999.

[12] K. Etessami and G. J. Holzmann. Optimizing büchi automata. In *CONCUR '00: Proc. of the 11th International Conf. on Concurrency Theory*, pages 153–167. Springer, 2000.

[13] G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive Logic Programming and Its Applications. In *Proc. of the International Conf. on Logic Prog.*, 2007.

[14] K. W. Hamlen. *Security Policy Enforcement by Automated Program-rewriting*. PhD thesis, Cornell University, Ithaca, New York, 2006.

[15] K. W. Hamlen and M. Jones. Aspect-Oriented In-lined Reference Monitors. In *Proc. of the ACM SIGPLAN Workshop on Prog. Languages and Analysis for Security (PLAS)*, 2008.

[16] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified In-Lined Reference Monitoring on .NET. In *Proc. of the ACM SIGPLAN Workshop on Prog. Languages and Analysis for Security (PLAS)*, 2006.

[17] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability Classes for Enforcement Mechanisms. In *ACM Trans. on Prog. Languages and Systems*, 2006.

[18] M. Jones and K. Hamlen. Enforcing IRM security policies: Two case studies. In *Proc. of IEEE Intelligence and Security Informatics (ISI) Conference (to appear)*, June 2009.

[19] E. M. C. Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[20] W. Kisser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2), April 2003.

[21] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[22] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In X. Useni, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243. USENIX, 1996.

[23] A. Pnueli. The temporal logic of programs. In *Proc. of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE Comp. Soc. Press, Oct.-Nov. 1977.

[24] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient Model Checking Using Tabled Resolution. In *Computer Aided Verification (CAV '97)*. Springer-Verlag, 1997.

[25] K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In *In 14th International SPIN Workshop, volume 4595 of LNCS*, pages 149–167. Springer, 2007.

[26] T. C. Ruys and N. H. M. A. de Brugh. MMC: the Mono Model Checker. *Electron. Notes Theor. Comput. Sci.*, 190(1):149–160, 2007.

[27] F. B. Schneider. Enforceable Security Policies. *ACM Trans. on Information and System Security*, 3:30–50, 2000.

[28] R. Sebastiani, R. Sebastiani, S. Tonetta, and S. Tonetta. More deterministic vs. smaller bchi automata for efficient LTL model checking. In *In CHARME03, volume 2860 of LNCS*, pages 126–140. Springer, 2003.

[29] L. . Shapiro and E. Y. Sterling. *The Art of PROLOG: Advanced Programming Techniques*. The MIT Press, 1994.

[30] L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive Logic Programming. In *Proc. of the International Conf. on Logic Prog.*, 2006.

[31] H. Tamaki and T. Sato. OLD resolution with tabulation. In E. Y. Shapiro, editor, *ICLP*, volume 225 of *LNCS*, pages 84–98. Springer, 1986.

[32] Yap prolog, 2009. `http://www.dcc.fc.up.pt/~vsc/Yap/`.