Mechanized Information Flow Analysis through Inductive Assertions

Warren A. Hunt, Jr., Robert Bellarmine Krug, Sandip Ray, and William D. Young
Department of Computer Sciences
University of Texas at Austin
{hunt,rkrug,sandip,byoung} @cs.utexas.edu

***Abstract*— We present a method for verifying information flow properties of software programs using inductive assertions and theorem proving. Given a program annotated with information flow assertions at cutpoints, the method uses a theorem prover and operational semantics to generate and discharge verification conditions. This obviates the need to develop a verification condition generator (VCG) or a customized logic for information flow properties. The method is compositional: a subroutine needs to be analyzed once, rather than at each call site. The method is being mechanized in the ACL2 theorem prover, and we discuss initial results demonstrating its applicability.**

I. Introduction

Security of many critical computing systems depends on *information flow policies* that prohibit access to sensitive information without proper authorization. With the increasing application of software systems to secure applications, it is vital to ensure that a software implementation properly enforces information flow policies. The goal of this paper is to develop techniques for mechanized information flow analysis.

In its simplest form, modeling an information flow policy involves labeling certain program variables as classified (or high security), with the requirement that the value of an unclassified variable is not influenced by the initial values of any classified variable. Such a policy can be formalized by *noninterference* [1]. A deterministic program satisfies the policy if, from a pair of initial states differing only in classified variables, any pair of computations leads to final states with identical values for unclassified variables. Noninterference naturally generalizes to a lattice of security levels.

This paper proposes a method for verifying information flow properties of software programs through general-purpose theorem proving. Programs are formalized through an *operational semantics* of the underlying language defined by an interpreter that specifies the effect of executing instructions on the system state. Our approach uses *inductive assertions*. Given a program annotated with assertions at cutpoints, we derive *verification conditions* that ensure requisite information flow control, to be discharged with a theorem prover.

A key feature of our approach is that it obviates the need for implementing a custom verification condition generator (VCG) for information flow properties of the underlying language constructs. Instead, we show how to configure an off-the-shelf theorem prover to *mimic* a VCG through symbolic simulation of the operational model. The method is inspired by, and an extension of, our previous work [2] which showed how to prove functional correctness via symbolic simulation. The method is compositional; properties of subroutines can be verified individually rather than at each call site. We demonstrate the method by analyzing a small but illustrative program with the ACL2 theorem prover.

II. Basic Framework

We use operational semantics to model a program by its effects on the machine states. A state is a tuple of values of all machine variables—the program counter (pc), registers, memory, etc. The semantics is then given by a transition function $\mathit{next} : S \rightarrow S$ where S is the set of states: for a state s, $\mathit{next}(s)$ returns the state after executing one instruction from s. Executions are modeled by the function $\mathit{run} : S \times \mathbb{N} \rightarrow S$ which returns the state after n transitions from s.

$$\mathit{run}(s,n) \triangleq \begin{cases} s & \text{if } n = 0 \\ \mathit{run}(\mathit{next}(s), n-1) & \text{otherwise} \end{cases}$$

To illustrate how to formally specify information flow properties of programs using operational semantics, we consider the simple version of noninterference mentioned in Section I. Assume a partition of the variables into sets H (high) and L (low), corresponding to classified and unclassified data. Furthermore, assume that we have two predicates poise and exit on set S. For any state s, $\mathit{poise}(s)$ stipulates that s is poised to initiate execution of the program of interest: it specifies that the program is in the current call frame and the pc points to its first instruction. The predicate exit characterizes the termination states. To formalize the noninterference statement, we make use of the function esteps below, which returns, for any state s, the number of transitions to the first exit state reachable from s (if such an exit state exists).

$$\mathit{estpt}(s,i) \triangleq \begin{cases} i & \text{if } \mathit{exit}(s) \\ \mathit{estpt}(\mathit{next}(s), i+1) & \text{otherwise} \end{cases}$$
$$\mathit{esteps}(s) \triangleq \mathit{estpt}(s,0)$$

The definition of estpt is *partial*: its return value is unspecified if no exit state is reachable from s. Defining a recursive function generally requires a termination proof. However, since the definition is tail-recursive, it is admissible in theorem provers whose logics support Hilbert's choice operator [3].

A formalization of noninterference is shown in Fig. 1, and can be paraphrased as follows. "Let s and s' be any two states

$$\mathit{pre}(s,s') \triangleq \mathit{poise}(s) \wedge \mathit{poise}(s') \wedge (\bigwedge_{l \in L} l(s) = l(s'))$$
$$\mathit{post}(s,s') \triangleq (\bigwedge_{l \in L} l(s) = l(s'))$$
$$\mathit{nexte}(s) \triangleq \mathit{run}(s, \mathit{esteps}(s))$$

Noninterference Condition:

$$\mathit{pre}(s,s') \wedge \mathit{exit}(\mathit{run}(s,n)) \Rightarrow \mathit{exit}(\mathit{nexte}(s')) \wedge \mathit{post}(\mathit{nexte}(s), \mathit{nexte}(s'))$$

Fig. 1. Formal Definition of Noninterference. Here $l(s)$ is assumed to be the value of variable l in state s.

poised to execute the program, such that the variables in L have the same valuation in both s and s'. Suppose that there is an *exit* state reachable from s. Then the following conditions hold. (1) There is an *exit* state reachable from s'. (2) Let s_0 and s_0' be the first *exit* states reachable from s and s' respectively; then s_0 and s_0' have the same valuation for all variables in L."

Note that the statement of a practical information flow property might differ from the above. For instance, one might have a lattice of security levels. Or one might be interested only in a subset $L' \subseteq L$ at *exit*; then the conjunction in the definition of *post* would range over L'. Nevertheless, such concerns affect only the concrete definitions of *pre* and *post*; once they have been defined for the desired information flow requirements, the noninterference statement can be used as is.

Our approach is based on *inductive assertions* which involve annotating a program with assertions at cutpoints that include loop tests, program entry, and exit. For our purpose, the set of cutpoints is characterized by a predicate *cut* on S, commonly depending only on the pc. One then proves that whenever the program control reaches a cutpoint, the corresponding assertion holds. For functional correctness, this is achieved by a VCG as follows. A VCG crawls over an annotated program, generating verification conditions to be discharged by theorem proving. The guarantee from the VCG process is informally stated as follows. "Let p be any non-exit cutpoint satisfying the assertions. Let q be the next subsequent cutpoint. Then the assertions hold at q." It follows that the corresponding assertion holds whenever the program control reaches a cutpoint.

How do we extend the above for information flow properties? Since information flow is characterized by *pairs of states*, the assertions involved are formalized by a predicate *assert* over $S \times S$. The associated VCG guarantee is as follows. "Let p and p' be any two corresponding non-exit cutpoints of the program such that $\mathit{assert}(p,p')$ holds. (See below for an explanation of the role of "corresponding cutpoints.") Let q and q' be the next cutpoints from p and p' respectively. Then $\mathit{assert}(q,q')$ holds." Then, if additionally, (1) *assert* holds for the initial pair of states, and (2) for the pair of *exit* states *assert* implies *post*, it follows that the pair of *exit* states reachable from any pair of initial states satisfies *post*.

The formal verification conditions for information flow are shown in Fig. 2. Condition 4 formalizes the VCG guarantee. We now discuss the predicate *C*, which formalizes the notion of "corresponding cutpoints." Recall that the VCG guarantee specifies that when *assert* holds between a pair of cutpoints

$$\mathit{cstpt}(s,i) \triangleq \begin{cases} i & \text{if } \mathit{cut}(s) \\ \mathit{nextc}(\mathit{next}(s, i+1)) & \text{otherwise} \end{cases}$$
$$\mathit{cstps}(s) \triangleq \begin{cases} \mathit{cstpt}(s,0) & \text{if } \mathit{cstpst}(s,0) \in \mathbb{N} \\ \omega & \text{otherwise} \end{cases}$$
$$\mathit{cut}(\mathrm{D}) \Leftrightarrow (\forall s : \mathit{cut}(s))$$
$$\mathit{nextc}(s) \triangleq \begin{cases} \mathit{run}(s, \mathit{cstps}(s)) & \text{if } \mathit{cstps}(s) \in \mathbb{N} \\ \mathrm{D} & \text{otherwise} \end{cases}$$

Verification Conditions

1. $\mathit{C}(s,s') \Rightarrow \mathit{cut}(s) \wedge \mathit{cut}(s') \wedge (\mathit{exit}(s) \Leftrightarrow \mathit{exit}(s'))$
2. $\mathit{pre}(s,s') \Rightarrow \mathit{C}(s,s') \wedge \mathit{assert}(s,s')$
3. $\mathit{exit}(s) \Rightarrow \mathit{cut}(s)$
4. $\mathit{C}(s,s') \wedge \mathit{assert}(s,s') \wedge \neg \mathit{exit}(s) \wedge \mathit{exit}(\mathit{run}(s,n)) \Rightarrow \mathit{assert}(\mathit{nextc}(\mathit{next}(s)), \mathit{nextc}(\mathit{next}(s')))$
5. $\mathit{C}(s,s') \wedge \mathit{assert}(s,s') \wedge \neg \mathit{exit}(s) \wedge \mathit{exit}(\mathit{run}(s,n)) \Rightarrow \mathit{C}(\mathit{nextc}(\mathit{next}(s)), \mathit{nextc}(\mathit{next}(s')))$
6. $\mathit{assert}(s,s') \wedge \mathit{exit}(s) \wedge \mathit{C}(s,s') \Rightarrow \mathit{post}(s,s')$

Fig. 2. Verification conditions for information flow. Each verification condition is implicitly quantified over all free variables. Here ω is the first infinite ordinal. The function $\mathit{cstps}(s)$ returns the number of transitions to the closest cutpoint reachable from s if one exists, and ω otherwise. The reasons for returning ω when no cutpoint is reachable from s are technical, and not germane to this paper.

then it also holds for the subsequent pair. However, the next subsequent cutpoints might be out of sync. For instance, computation might exit from p and not from p'; thus the information flow theorem cannot be derived from the VCG guarantee. The predicate *C* eliminates this possibility by requiring the following: (1) if $\mathit{pre}(s,s')$ holds then s and s' must be corresponding; (2) for any two corresponding cutpoints, the subsequent cutpoint pair must be corresponding; and (3) if s satisfies *exit* and s' is a corresponding cutpoint, then s' must also satisfy *exit*. We assume that there is a binary predicate *C* on $S \times S$ characterizing the corresponding cutpoints; in practice, the definition of $\mathit{C}(s,s')$ will usually reduce to the condition that the pc values for s and s' are equal.

Conditions 4 and 5 involve multiple steps of program execution. Contrary to common practice, we discharge them *without a VCG* as follows. We prove the following two theorems, which are easy consequences of the definition of *nextc*:

SSR1: $\neg \mathit{cut}(s) \Rightarrow \mathit{nextc}(s) = \mathit{nextc}(\mathit{next}(s))$
SSR2: $\mathit{cut}(s) \Rightarrow \mathit{nextc}(s) = s$

We treat SSR1 and SSR2 as oriented conditional equations or *rewrite rules*. For any symbolic state s, the rules rewrite the term $\mathit{nextc}(s)$ to either s or $\mathit{nextc}(\mathit{next}(s))$; in the latter case the definition of *next* is symbolically expanded, simplified, and the rules applied again. The proof attempt causes the theorem prover to symbolically simulate the program from a cutpoint until the next cutpoint is reached; the process mimics a forward VCG. For symbolic simulation to terminate, each program loop must contain a cutpoint, as with traditional VCGs.

We briefly remark on how to automate the method above in ACL2. The derivation of noninterference from Conditions 1–6 is independent of the definitions of *pre*, *post*, etc. This allows the development of a *proof template* for generating the

```
tricky1 (int high, low, n) {
   int temp = low;
   for i = 0 to n do {
     if even(i) {
       out = out + temp;
       temp = high;
     } else {
       temp = low;
     }
   }
   out = out + 7;
   return out;
 }
```

Fig. 3. Pseudo-code for the Tricky Program. Variable `out` is incremented by `temp` only when `i` is even, and `temp` is equal to `low` in that case. Thus, the final value of `out` is independent of `high`.

verification conditions as follows. First, we introduce functions *pre*, *post*, *cut*, *assert*, *C*, and *next* constrained to satisfy Conditions 1–6, and prove the noninterference theorem for these constrained functions.[1] We can then implement an ACL2 macro that automates the information flow proofs as follows:

- Mechanically generate the concrete version of *nextc*.
- Establish conditions 1–6 for the concrete versions of *pre*, *post*, etc., using symbolic simulation.
- Derive the information flow theorem by functionally instantiating the generic version.

We have developed a corresponding proof template for (partial and total) functional correctness [2]. We are working on extending the template for information flow properties.

III. A "TRICKY" EXAMPLE

Our approach, although extremely simple, nevertheless provides a scalable framework for information flow analysis. One key strength is the ability to use expressive predicates for proving information flow theorems: if an information flow property depends on functional invariants of the system state, then assertions can easily account for such invariants. This is in stark contrast to traditional security type systems for information flow verification, which depend on the syntactic analysis of the program to deduce information flow [4].

As an illustration, consider the program shown in Fig. 3. The information flow specification for the program is that the final value of `out` depends only on the initial values of `low` and `n`. The program is adapted from one in a recent paper by Amtoft and Banerjee [5][2] which, although simple, was motivated by an actual program used in operational verification of hardware amplifiers provided by Rockwell Collins. The information flow property depends on a key observation: whenever the value of `out` is incremented by `temp`, the value of `i` is even and the value of `temp` is equal to `low`; thus the final value of `out` is dependent only on `low` (and the loop count `n`). The property cannot be inferred by type reasoning which would infer dependence of `temp` on `high` and `out` on `temp`.

We formalized the program through an operational semantics of a simple machine model, and proved the information flow specification using inductive assertions. The precondition stipulates that s and s' are poised to execute the program and the values of `low`, `n`, and `out` are the same in both states; the postcondition specifies that the value of `out` is the same after exiting the program. The only "creative" assertion is in the loop invariant. In addition to the boiler-plate assertion that the values of `low`, `n`, `out`, and `i` are the same in s and s', we need the condition that if `i` is even s and s' have the same value of `temp`. With this assertion, the verification conditions shown in Fig. 2 are easily verified through symbolic simulation.

It is instructive to compare our approach with that of Amtoft and Banerjee [5]. Their approach is built around the axiomatic semantics for a special logical construct $\bowtie$ stipulating *agreement assertions*: for a variable x, two states p and q satisfy $x \bowtie$ if and only if $x(p) = x(q)$. They develop axiomatic semantics for specifying loop flow and object flow using $\bowtie$, and a VCG for the semantics. In contrast, we generate and discharge the verification conditions directly through symbolic simulation of the operational semantics. Nevertheless, our approach requires no more creative insight than theirs, namely manually constructing the loop invariant above. On the other hand, our approach can harness the power of a general-purpose theorem prover for symbolic simulation and requires no axiomatic semantics for information flow.

IV. COMPOSITIONALITY

The above treatment did not consider compositionality. Consider verifying a program P that invokes a subroutine Q. Symbolic simulation from a cutpoint of P might encounter an invocation of Q, resulting in simulation of Q. We prefer to separately verify Q, and use the result for verifying P.

To achieve composition, we must handle the *frame conditions* to justify that P can continue execution after Q returns. Note that an information flow property of Q is not sufficient for this; for instance, we must show that the execution of Q does not corrupt the call stack. For functional correctness, the frame problem is addressed by phrasing the postcondition as an equality to characterize how each state component is modified by Q [2]. However, a full characterization of Q is often irrelevant to the information flow of P. The challenge is to effectively augment the postcondition of Q with frame conditions to facilitate symbolic simulation of P.

How do we address the challenge? Let V be the set of state components governing the control flow on return from Q; typically V includes the call stack. Then we define $\mathit{modify}_Q(s)$ to update each component of s as follows. For each component $v \in V$, $\mathit{modify}_Q(s)$ updates v by precisely characterizing its modification on exit from Q. The update to the call stack is characterized by popping the current call frame. For $c \notin V$, the update is simply $c(\mathit{nexte}_Q(s))$, which

[1] This proof has been completed with ACL2 at the time of this writing.

[2] The difference between Amtoft and Banerjee's program and that shown in Fig. 3 is that the former involved 7 iterations of `i` in the loop while we use `n` iterations. Note that if the number of loop iterations is a constant then the loop can be unrolled by symbolic simulation, obviating loop invariants.

```
int out;
main (int high, low, n, flag) {
  out = 0;
  if flag < 0 {
    tricky1(high, low, n);
  } else {
    tricky3(high, low, n);
  }
  out = out - 1;
}
```

Fig. 4. A Program to demonstrate compositionality

may or may not need to be characterized depending on the caller. We then prove the following two conditions.

1) $\mathit{poise}_Q(s) \wedge \mathit{exit}_Q(\mathit{run}(s,n)) \Rightarrow \mathit{nexte}_Q(s) = \mathit{modify}_Q(s)$
2) $\mathit{pre}_Q(s,s') \wedge \mathit{exit}_Q(\mathit{run}(s,n)) \Rightarrow \mathit{post}_Q(\mathit{modify}_Q(s), \mathit{modify}_Q(s'))$

Here, $\mathit{poise}_Q(s)$ states that s is poised to invoke Q. Its definition is derived from $\mathit{pre}_Q(s,s')$ by collecting the conjuncts that only mention s. Condition 2 follows from 1 and the information flow property of Q. We prove 1 through inductive assertions, viewing modify_Q as a functional characterization of Q [2]. The necessary assertions can be culled from the information flow proof of Q. Recall that the predicate $\mathit{assert}_Q(s,s')$ is strong enough to characterize the flow of control from each cutpoint of s (and s') to the next. Thus, the conjuncts in $\mathit{assert}_Q(s,s')$ that only involve s can be used for symbolic simulation from each cutpoint in s to prove 1. The theorems facilitate compositional reasoning. If states s and s' encountered during symbolic simulation of P are poised to execute Q, then 1 permits simulation to "skip past" Q, and 2 enables us to assume *post* on the generated pair $(\mathit{modify}_Q(s), \mathit{modify}_Q(s'))$ during subsequent simulation. Note that this can be automated using macros as hinted at in Section II for the basic framework.

We used the approach above to compositionally verify the program shown in Fig. 4. The program is artificial but illustrative. It invokes one of two separate versions of the `tricky` procedure depending on `flag`: `tricky1` is as shown in Fig. 3; `tricky3` iterates `3n` times instead of `n`. Our information flow specification is that the final value of `out` on *exit* from `main` is independent of `high`. Note that the information flow analysis of `tricky3` is exactly analogous to `tricky1`, but its return value is different. Thus, a complete functional characterization of the two routines would involve separate analysis. However, the actual return value of the subroutines is immaterial to the information flow of `main`. With our approach, `main` can be verified using only the noninterference of each subroutine and the frame conditions.

V. Related Work and Conclusion

Information flow analysis was formulated by Denning and Denning [6]. Sabelfeld and Meyers [4] contains a comprehensive survey of the area. Traditional approaches to information flow analysis involves *security type systems* [7], [8]: program variables and expressions are annotated with security levels, and flow of information is controlled by typing rules. There has also been significant recent work on axiomatic semantics for information flow. Clark *et al.* [9] develop a semantics for Idealized Algol. Joshi and Leino [10] develop a weakest precondition calculus for information flow. Darvas *et al.* [11] use dynamic logic to express information flow for Javacard.

Our work provides the first framework for information flow analysis through inductive assertions directly on operational semantics. No separate VCG or axiomatic semantics for information flow is necessary. Instead, the generation and discharge of verification conditions are handled by the theorem prover through symbolic simulation. Furthermore, we can compose information flow properties of subroutines without requiring full characterization of their functional specification. The framework is in an early stage of development. As mentioned in Section II, we are developing proof templates to facilitate the automation of information flow verification. Some planned future enhancements include (1) automated static analysis of data structure shapes, (2) extension to multithreaded programs, and (3) analysis of dynamic and declassification policies.

Acknowledgements: This material is based upon work supported by DARPA and the National Science Foundation under Grant No. CNS-0429591. Matt Kaufmann provided numerous comments and suggestions in course of this work.

References

[1] J. Goguen and J. Meseguer, "Security policies and security models," in *Proc. 1982 IEEE Symposium on Security and Privacy*, April 1982, pp. 11–20.

[2] J. Matthews, J. S. Moore, S. Ray, and D. Vroon, "Verification Condition Generation Via Theorem Proving," in *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2006)*, ser. LNCS, M. Hermann and A. Voronkov, Eds., vol. 4246, Nov. 2006, pp. 362–376.

[3] P. Manolios and J. S. Moore, "Partial Functions in ACL2," *Journal of Automated Reasoning*, vol. 31, no. 2, pp. 107–127, 2003.

[4] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security," *IEEE Journal on Selected Areas of Communication*, vol. 21, no. 3, Jan. 2003.

[5] T. Amtoft and A. Banerjee, "Verification Condition Generation for Conditional Information Flow," in *Proceedings of the 2007 ACM workshop on Formal methods in security engineering (FMSE 2007)*, P. Ning, V. D. G. V. Atluri, and H. Mantel, Eds. ACM Press, Nov. 2007, pp. 2–11.

[6] D. Denning and P. Denning, "Certification of Programs for Secure Information Flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[7] P. Orbaek and J. Palsberg, "Trust in the λ-calculus," *Journal of Functional Programming*, vol. 7, no. 6, pp. 557–591, 1997.

[8] D. Volpano and G. Smith, "A Type-Based Approach to Program Security," in *Proceedings of Theory and Practice of Software Development (TAPSOFT 1997)*, ser. LNCS. Springer-Verlag, 1997, pp. 607–621.

[9] D. Clark, C. Hankin, and S. Hunt, "Information Flow Analysis for Algol-like Languages," *Computer Languages*, vol. 28, no. 1, pp. 3–28, 2002.

[10] R. Joshi and K. R. M. Leino, "A Semantic Approach to Information Flow," *Science of Computer Programming*, vol. 37, pp. 113–138, 2000.

[11] A. Darvas, R. Hähnle, and D. Sands, "A Theorem Proving Approach to Analysis of Secure Information Flow," in *Proceedings of the 2nd International Conference Security in Pervasive Computing (SPC 2005)*, ser. LNCS. Springer, 2005, pp. 193–209.