Formal Verification of Secure Programs in the Presence of Side Effects

Paul E. Black*
National Institute of Standards and Technology
Gaithersburg, Maryland 20899
paul.black@nist.gov

Phillip J. Windley
Computer Science Department
Brigham Young University
Provo, Utah 84602-6576
windley@cs.byu.edu

 Published in the Proceedings of the Hawai'i International Conference on Systems Sciences, January 6–9, 1998, Kona, Hawaii.

Abstract

Much software is written in industry standard programming languages, but these languages often have complex semantics making them hard to formalize. For example, the use of expressions with side effects is common in C programs. We present new inference rules for conditional (`if`) statements and looping constructs (`while`) with pre- and postevaluation side effects in their test expressions. These inference rules allow us to formally reason about the security properties of programs.

We maintain that formal verification of secure programs written in common languages is feasible and can be worthwhile. To support our claim, we give an example of how our verification of a secure web server uncovered some previously unknown problems.

Automated theorem proving assistants can help deal with complex inference rules, but many components must be brought together to make a broadly useful system. We propose elements of a formal verification system which could be widely useful.

1. Introduction

We have been working on proving security properties of an http daemon, thttpd, written in C. It is engineered to provide information to the World Wide Web and to be free of security flaws, even in the presence of a few operating system bugs or administrative errors. The code has a five page informal proof of correctness, and has been reviewed and critiqued by dozens of experts. It seems like an ideal candidate for a formal proof, and a proof would increase confidence in its security.

A formal verification is a means to investigate the consistency and completeness of specifications, properties, models of implementations, and assumptions [2]. To verify thttpd, we need a specification of the security properties of interest, a semantics of the language and the pertinent parts of the environment in which it runs, and a set of inference rules. In practice we also need a mechanized theorem proving assistant to handle details of the proof and to act as "a tireless mechanical skeptic." [14, page 53]

We chose Hoare's axiomatic semantics [12] as a powerful, but simple-to-understand model. We found that most work in axiomatic semantics has been on relatively simple languages in order to focus on particular concepts, such as distributed system [11] or a fully verified tool set [13]. Production languages, such as C or COBOL, are generally very rich with many overlapping features, instead of a minimal set, to express different kinds of algorithms and data structures succinctly. To verify thttpd we developed new inference rules which handle pre- and postevaluation side effects in simple assignment statements, conditional statements, function calls, and looping constructs.

We implemented these inference rules in HOL [8] beginning with code from Harrison [11]. We extended it with rules documented in Gordon [7] and Homeier's rules [13]. We then wrote goal-directed tactics to automate the proof. An ad-hoc parser adapted from Paulson [18, chapter 9] converts C code into equivalent abstract syntax trees.

Why verify programs written in C? Much software is written in industry standard programming languages because people are trained in its use, compilers and libraries are widely available and experience has shown them to be reliable, many tools exist to work with and analyze programs written in them, etc.

Sect. 2. presents inference rules for basic statements with preevaluation and postevaluation side effects. Sect. 3. shows rules which allow pre- and postevaluation side effects in the test expressions of conditionals and `while` loops. This section also explains how to create similar rules for other looping constructs and suggests how diversions in the control flow might be handled. We include an example from thttpd in Sect. 4.2 to demonstrate how formal verification can uncover faults. Finally Sect. 5. lists future work, including a proposal for the components of formal software verification system which could be widely used, and Sect. 6. has our conclusions.

*This work was sponsored by the National Science Foundation under NSF grant MIP–9412581 while Black was at BYU.

Since we use notations and concepts which are quite familiar to some people, but new to others, we explain some syntax and meaning of axiomatic semantics in Appendix A. Appendix B explains some relevant nuances of Unix permissions.

2. General Inference Rules for Side Effects

We use Hoare axiomatic semantics to express the correctness of program statements. The representation for partial correctness is

$$\vdash \{\mathtt{P}\}\ \mathtt{code}\ \{\mathtt{Q}\}$$

This means if predicate `P` is true of the current state, then when `code` finishes, the result is a state in which predicate `Q` is true. See Appendix A for more details.

An inference rule has the form

$$\frac{\begin{array}{c} H_1 \\ \vdots \\ H_n \end{array}}{C}$$

The meaning is if all of the hypotheses $H_1, \ldots, H_n$ are true, one can conclude C. Any number of hypotheses may qualify one conclusion.

Much of this section is taken from [3], but the rules presented here are more general.

The basic axiom for an assignment statement `v = expr;` is

$$\vdash \{\mathtt{Q}^{\mathtt{v}}_{\mathtt{expr}}\}\ \mathtt{v} = \mathtt{expr};\ \{\mathtt{Q}\}$$

as long as `expr` doesn't have any side effects [7, pp. 15–17]. That is if $\mathtt{Q}^{\mathtt{v}}_{\mathtt{expr}}$ is true and the assignment finishes, `Q` will be true. The notation $\mathtt{Q}^{\mathtt{v}}_{\mathtt{expr}}$ denotes `Q` with all free occurrences of `v` replaced by `expr`. For instance, $(\mathtt{g} * (\mathtt{h}+1))^{\mathtt{g}}_{(\mathtt{i}-1)}$ is `((i-1)*(h+1))`.

Since expressions in the C language may have side effects, this rule does not always apply. As a simple example, the semantics of `a = 2 * ++b;` is well defined [10]: it is equivalent to `++b; a = 2 * b;`. Applying the above rule we could conclude $\{b = 2\}\ \mathtt{a} = 2 * ++\mathtt{b};\ \{b = 2\}$ which is wrong.

To reason about statements with side effects, we introduce a general inference rule which derives the correctness of one statement from the correctness of a semantically equivalent statement.

$$\frac{\begin{array}{c}\mathtt{SEM\_EQ}\ \mathtt{stm1}\ \mathtt{stm2} \\ \vdash \{\mathtt{pre}\}\ \mathtt{stm1}\ \{\mathtt{post}\}\end{array}}{\vdash \{\mathtt{pre}\}\ \mathtt{stm2}\ \{\mathtt{post}\}} \qquad (1)$$

The predicate `SEM_EQ` is true if its two statement arguments are semantically equivalent. The inference rule means if

- two statements are semantically equivalent, and
- there is a partial correctness theorem for precondition, statement `stm1`, and postcondition

we can conclude an analogous partial correctness theorem for statement `stm2`.

Preevaluation side effects may be separated from statements with the following rule.

$$\frac{\mathtt{PreEval}\ \mathtt{expr}\ \mathtt{stm1}\ \mathtt{stm2}}{\mathtt{SEM\_EQ}\ (\mathtt{Seq}\ (\mathtt{Simp}\ \mathtt{expr})\ \mathtt{stm1})\ \mathtt{stm2}} \qquad (2)$$

`Seq` is the abstract syntax constructor which creates a statement from a sequence of two statements. `Simp` converts an expression into a simple statement, which is allowed in C. The `PreEval` is a predicate which is true if extracting the preevaluation side effects expression `expr` from statement `stm2` yields `stm1`.

Informally the rule means that if `stm2` can be separated into `expr` (which has all preevaluation side effects) and `stm1`, then `expr` (in a statement) followed by `stm1` is semantically equivalent to `stm2`.

For example, we can derive the correctness of `a = 2 * ++b;` from the correctness of the sequence of simpler statements `++b; a = 2 * b;` with

$$\frac{\begin{array}{c}\mathtt{SEM\_EQ}\ (++\,\mathtt{b};\ \mathtt{a} = 2 * \mathtt{b};)\ \mathtt{a} = 2 * ++\,\mathtt{b}; \\ \vdash \{\mathtt{P}\}\ ++\,\mathtt{b};\ \mathtt{a} = 2 * \mathtt{b};\ \{\mathtt{Q}\}\end{array}}{\vdash \{\mathtt{P}\}\ \mathtt{a} = 2 * ++\,\mathtt{b};\ \{\mathtt{Q}\}}$$

and

$$\frac{\mathtt{PreEval}\quad ++\mathtt{b}\quad \mathtt{a} = 2 * \mathtt{b};\quad \mathtt{a} = 2 * ++\,\mathtt{b};}{\mathtt{SEM\_EQ}\ \ (\mathtt{Seq}(\mathtt{Simp}\ \ ++\mathtt{b})\ \mathtt{a} = 2 * \mathtt{b};)\ \ \mathtt{a} = 2 * ++\,\mathtt{b};}$$

C allows postevaluation side effects in expressions in addition to preevaluation side effects. The statement `d = 2 * f++;` is well defined, as is equivalent to `d = 2 * f; f++;`. Postevaluation side effects may be separated with the following rule.

$$\frac{\mathtt{PostEval}\ \mathtt{stm1}\ \mathtt{expr}\ \mathtt{stm2}}{\mathtt{SEM\_EQ}\ (\mathtt{Seq}\ \mathtt{stm1}\ (\mathtt{Simp}\ \mathtt{expr}))\ \mathtt{stm2}}$$

Informally if extracting the postevaluation side effects expression `expr` from statement `stm2` yields `stm1`, then `stm1` followed by `expr` is semantically equivalent to `stm2`.

Why add semantic equivalence and more inference rules in order to handle side effects? Homeier's language, Sunrise [13], has an operator with a side effect, increment, which can occur in test expressions. He handles this by embedding the semantics of the operator in the inference rules. However user-written functions, which may have arbitrary side effects, can occur in loop and conditional test expressions in C. Even statements without function calls can have multiple side effects using, say, increment and assignment operators. We take this more general approach to be able to separate a side effect from the expression in which it occurs.

Having a semantic equivalence rule (1) also allows us to more uniformly express other semantically complexities. We can use semantic equivalence to clearly express the associativity and composition of sequences of statements, the relation between one-armed (`if...then`) and two-armed (`if...then...else`) conditionals, the semantics of the empty statement, etc., in addition to pre- and postevaluation side effects. Although this rule "factoring" would greatly complicate a manual proof, but we have written proof subroutines, called "tactics" in HOL, invoke and prove instances of the rules automatically.

3. Side Effects in Control Structures

The rules presented above are inadequate for control statements. For instance, suppose we were allowed to apply the postevaluation rules to the following code.

```
if (b++ > 0) {
    t = b;
} else {
    e = b;
}
```

It would be transformed into this (note the postincrement afterward) which is *not* the same. The increment would be delayed until after the entire conditional statement.

```
if (b > 0) {
    t = b;
} else {
    e = b;
}
b++;
```

In this section we present inference rules for some control structures and indicate how the general approach could cover many other structures.

3.1 Side Effects in Conditionals

Conditionals are the simplest form of control statements for our purposes. Without side effects the inference rule is straight forward:

$$\frac{\begin{array}{c}\texttt{IS\_VALUE}\ \texttt{expr}\ \texttt{test}\\ \vdash \{\texttt{pre}\ \wedge\ \texttt{test}\}\ \texttt{thenCode}\ \{\texttt{post}\}\\ \vdash \{\texttt{pre}\ \wedge\ \texttt{\~{}test}\}\ \texttt{elseCode}\ \{\texttt{post}\}\end{array}}{\vdash \{\texttt{pre}\}\ \texttt{If}\ (\texttt{expr})\ \texttt{thenCode}\ \texttt{elseCode}\ \{\texttt{post}\}}$$

`IS_VALUE` means that `test` is the assertion language equivalent of `expr`.

Any preevaluation side effects can be separated and handled with the semantic equivalence (1) and preevaluation (2) rules.

Figure 1 shows the flow in a conditional statement with side effects in the test expression. In summary the sequence of events is

1. Determine the test condition in the initial state (when the precondition is true),
2. Evaluate the postevaluation side effects, yielding new conditions, then
3. Evaluate the code in the body, yielding a postcondition.

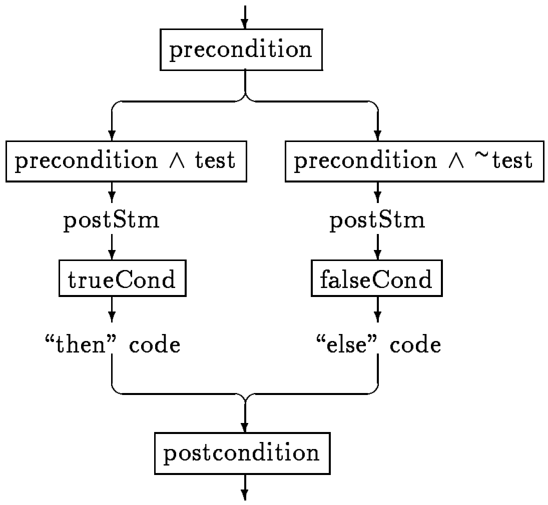

Figure 1: Control Flow in a Conditional

This is the corresponding inference rule.

$$\frac{\begin{array}{c}\texttt{SEM\_EQ}\ (\texttt{Seq}\ (\texttt{Simp expr})\ \texttt{postStm}))\ (\texttt{Simp ex})\\ (\texttt{postStm} = \texttt{EmptyStmt})\ \vee\\ (\texttt{postStm} = (\texttt{Simp}\ \texttt{postSeEx})\\ \wedge\ \texttt{NoPreSE postSeEx})\\ \texttt{IS\_VALUE expr test}\\ \vdash \{\texttt{pre}\ \wedge\ \texttt{test}\}\ \texttt{postStm}\ \{\texttt{trueCond}\}\\ \vdash \{\texttt{pre}\ \wedge\ \texttt{\~{}test}\}\ \texttt{postStm}\ \{\texttt{falseCond}\}\\ \vdash \{\texttt{trueCond}\}\ \texttt{thenCode}\ \{\texttt{post}\}\\ \vdash \{\texttt{falseCond}\}\ \texttt{elseCode}\ \{\texttt{post}\}\end{array}}{\vdash \{\texttt{pre}\}\ \texttt{IfElse}\ (\texttt{ex})\ \texttt{thenCode}\ \texttt{elseCode}\ \{\texttt{post}\}}$$

Informally the above means that if the following conditions are met, we can conclude the partial correctness of the conditional statement.

- The original test expression code `ex` is split into a side effect free test expression `expr` followed by a statement for any postevaluation side effects `postStm`. (Any preevaluation side effects can be removed by Rule 2.)
- Either the postevaluation side effect statement `postStm` is the empty statement (if there are no side effects), or it is a simple statement of an expression `postSeEx` having the postevaluation side effect conditions, but no preevaluation side effects.
- `expr` in the programming language corresponds to `test` in the assertion language.
- Executing `postStm` with `test` true or false establishes the "true" or "false" conditions respectively.
- Executing the "then" and the "else" code establishes the post condition.

Typically most of these theorems are proven automatically, thus minimizing the user's work.

An inference rule for one-armed conditionals can be derived from the above rule and the following rule. It states the semantic equivalence of one-armed conditionals and two-armed conditionals with an empty "else" case.

$$\frac{}{\texttt{SEM\_EQ (IfElse t s EmptyStmt) (If t s)}}$$

3.2 Side Effects in Loop Statements

In simple languages the inference rule for a `while` loop, or backward jump, is straight forward:

$$\frac{\begin{array}{c}\texttt{IS\_VALUE expr test}\\ \vdash \{\texttt{invariant} \wedge \texttt{test}\}\ \texttt{body}\ \{\texttt{invariant}\}\end{array}}{\begin{array}{l}\vdash \{\texttt{invariant}\}\ \texttt{while expr body}\\ \qquad\qquad \{\texttt{invariant} \wedge \texttt{\~{}test}\}\end{array}} \quad (3)$$

When test expressions can have side effects, the rule is more complex. We cannot use the preevaluation rule as we could with conditionals. If we could use the preevaluation rule, we could prove

```
while (preeval side-effects in expr)
    body
```

by proving

```
preeval side-effects;
while (expr)
    body
```

But in the second form, the side effect is not executed every loop! The flow of control in a `while` loop with pre- and postevaluation side effects is as follows.

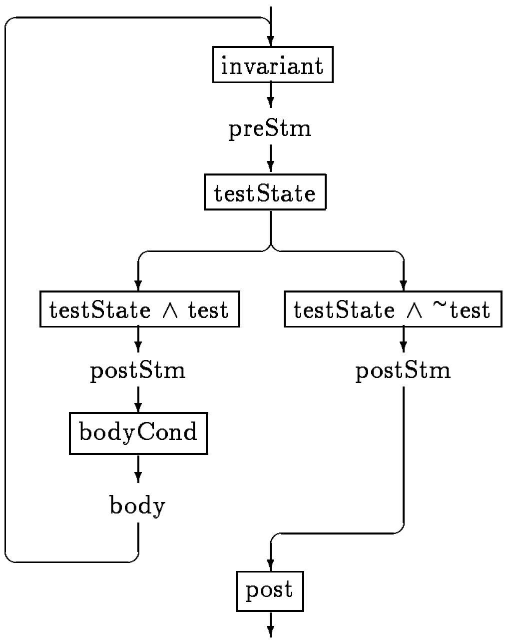

Figure 2: Control Flow in a `While` Loop

The corresponding inference rule for `while` statements is then

$$\frac{\begin{array}{l}(\texttt{preStm} = \texttt{EmptyStmt}) \vee\\ \quad (\texttt{preStm} = (\texttt{Simp preSeEx})\\ \qquad \wedge\ \texttt{NoPostSE preSeEx})\\ (\texttt{postStm} = \texttt{EmptyStmt}) \vee\\ \quad (\texttt{postStm} = (\texttt{Simp postSeEx})\\ \qquad \wedge\ \texttt{NoPreSE postSeEx})\\ \texttt{SEM\_EQ (Seq preStm(Seq(Simp testEx)postStm))}\\ \qquad\qquad \texttt{(Simp ex)}\\ \texttt{IS\_VALUE testEx test}\\ \vdash \{\texttt{invariant}\}\ \texttt{preStm}\ \{\texttt{testState}\}\\ \vdash \{\texttt{testState} \wedge \texttt{test}\}\ \texttt{postStm}\ \{\texttt{bodyCond}\}\\ \vdash \{\texttt{testState} \wedge \texttt{\~{}test}\}\ \texttt{postStm}\ \{\texttt{post}\}\\ \vdash \{\texttt{bodyCond}\}\ \texttt{body}\ \{\texttt{invariant}\}\end{array}}{\vdash \{\texttt{invariant}\}\ (\texttt{while ex body})\ \{\texttt{post}\}}$$

In other words if

- The preevaluation (`preStm`) and postevaluation (`postStm`) side effects statements are either empty statements or are expressions with just pre- or postevaluation the side effects respectively.
- Executing `preStm`, then the remaining test expression (`testEx`), then `postStm` is equivalent to the original test expression.
- `testEx` in the programming language corresponds to `test` in the assertion language.
- Executing `preStm` in the invariant condition establishes a test condition.
- Executing `postStm` with `test` true or false establishes the "body" or "post" conditions respectively.
- Executing the body code in the body condition reestablishes the loop invariant.

then {`invariant`} (`while ex body`) {`post`} is true.

We allow `preStm` and `postStm` to be the empty statement in case the original test expression has no side effects. To support our confidence in the rule, we note that when `expr` has no side effects, `preStm` and `postStm` are the empty statement. Therefore the test condition is the same as the invariant, the body condition is `invariant` ∧ `test`, and the post condition is `invariant` ∧ `~test`. This reduces to the basic `while` loop rule (3).

The HOL tactic to reduce a `while` loop optionally takes a test condition and a body condition. The user can skip either or both if there are no side effects. The tactic also proves most conditions automatically. Thus the complexity of the rules are only exposed when necessary, and the user's work is minimized.

3.3 Other Looping Constructs

Other looping constructs can be handled similarly. The `for` and `do...while` loops in C, `do...until` in Pascal, and `loop...begin...again` in Forth can be broken apart into side conditions and correctness conditions over pieces of code. Built-in tactics can keep track of where correctness conditions are needed and with regard to which expressions or pieces of code.

Directives which change the flow of control within loops, such as `break` and `continue` in C, can be handled with multiple post conditions as originally set forth in [1]. For example, a `break` statement would have a formalization something like this.

$$\vdash \{\texttt{pre}\}\ \texttt{break;}\ \{[\texttt{next} : \texttt{false}, \texttt{break} : \texttt{pre}]\}$$

In other words, the next sequential condition is "false" (control never arrives at the next statement), and the precondition of the `break` is the condition where the break control flow arrives.

4. Example from Verifying a Secure Web Server

In June 1995, Management Analytics wrote a secure World Wide Web server called thttpd. The code consists of about 100 lines of C. They point out [5] that

> The main risk to providers of [web] services is that someone might be able to fool their server software into doing something it is not supposed to do, thus allowing an attacker to break into their server and do some harm.

Thus they wrote a small server with intentionally redundant security features. They listed the security properties as information integrity (no information on the server can be corrupted by outside users), availability of service (outside users cannot deny services to other users), and confidentiality (the server only provides information which is explicitly authorized for outside access). A five page detailed, but informal, review argues that the code has these properties. Additionally it has been tested for typical programming errors, and it ran over a year without a single known security breach.

4.1 Definition of Confidentiality

In this section we discuss the definition of confidentiality we use. Our formal definition involves many details not relevant to verification in the presence of side effects.

Above we define confidentiality as a property of the information which may flow to a remote user. We assume that the only channel to a remote user is through standard out. (When invoked for the web, standard out *is* directed back to the remote user.) Since contents of files are the primary objects of interest, we model the server simply as a set of files, i.e., the file system. We follow the Unix convention in treating standard out as one of the files.

With this model, we say the code has confidentiality if each file has confidentiality after the code executes. Each file has confidentiality if the file is not standard out (assuming remote users cannot access local files) or all information in the file is nonconfidential. Information is nonconfidential if it is defined as such (e.g., fixed strings in the program) or if it is read from a file authorized for outside access.

4.2 Confidentiality of the Function, `cat`

To illustrate reasoning about expressions with side effects, we discuss the proof of confidentiality of one function in thttpd. Other properties can be proven separately [4]. This function, `cat`, returns the contents of the requested file to the user. Here is the code with applicable global declarations.

```
#define BUFSIZE 4096
#define MAXSIZE 2048
char bs2[BUFSIZE];

void cat(s)
char s[];
{int i,n;FILE *F;
 i=open(s,0);
 while ((n=read(i,bs2,MAXSIZE)) > 0)
      write(1,bs2,n);
 close(i);}
```

In more detail, `cat` is passed the pathname, `s`, for a file. The code opens the file for reading, copies its contents to standard out, then closes the file. In thttpd code preceding the call to `cat` checks that the file is authorized for outside access, in particular, it is "other" readable. See Appendix B for more detail on Unix permissions.

First we prove `nonConfFD i` after opening a file authorized for outside access, where `nonConfFD` is true if its argument refers to a nonconfidential file and `i` is the file descriptor. If open succeeds, this is indeed the case. However, if the file is not user-readable and the process and file user are the same, open fails and returns -1. Thus we can only prove the weaker postcondition $\texttt{i} \neq -1 \Rightarrow \texttt{nonConfFD i}$. The code was intended to have the stronger postcondition, so it does not perform as desired. However it does not cause a security breach.

The test in the loop has preevaluation side effects, so we use Rule 3.2. Referring to Figure 2, the test state satisfies the condition $(\texttt{i} \neq -1 \Rightarrow \texttt{nonConfFD i}) \wedge (\texttt{n} > 0 \Rightarrow \texttt{nonConfS bs2})$, where `nonConfS` means its argument is a nonconfidential string. The test is $\texttt{n} > 0$. Since there are no postevaluation side effects, the body condition is the test condition "and" $\texttt{n} > 0$, and the postcondition must be implied by the test condition "and" $\sim(\texttt{n} > 0)$.

We use the preevaluation rule (2) to separate the `read` call from the assignment. Then we use the axiom of calls to `read` to prove that executing `n=read(i,bs2,MAXSIZE)` establishes the test condition. From the test condition and the test, we can conclude `nonConfS bs2`, that is, the information in the buffer is not confidential. (If the open failed, the documentation and some experiments suggest that the `read` will fail.) We use the nonconfidentiality of the buffer and the axiom of calls to `write` to prove that the invariant is reestablished: the file system (still) has confidentiality.

While writing and using the `write` axiom, we found another problem. Since `write` might not write all (or any!) of the characters passed, thttpd might not return the complete contents of a file. This does not cause a security breach, but a more thorough implementation could retry `write` until all characters were written or some permanent failure occurred.

As a side note, the code which calls `cat` assumes `cat` succeeds (having checked "everything" before calling it) and unconditionally logs a "cat *filename*" message. However the `open` or `write` calls may fail. Although this is not a security breach, the log file may be misleading.

Proving that the `close` maintains confidentiality takes a single line. We give the HOL tactic here to give a idea of the proof although it is unclear without additional documentation.

```
e (CALL_TAC SYS_close THEN
                          STRIP_THEN_REWRITE_TAC);
```

This invokes the function call tactic with the axiom of calls to the system function `close`, then strips quantifiers and implications the rewrites the foal with assumptions. The proof of confidentiality of `cat` is about two pages of similar tactics.

The verification of thttpd security was finished in July 1997. The proof is about 2,500 lines of definitions and tactics. The description of the Unix environment and system calls is about 1,000 lines.

5. Future Work

This section describes possible future improvements and outlines the components required for a complete, broadly usable software verification system.

5.1 Sequence Points

The inference rules given in Sect. 3. handle pre- and postevaluation side effects. However, they are not valid in the presence of sequence points with side effects. Sequence points arise in C from logical OR's (`||`), logical AND's (`&&`), and the comma operator (`,`), among others. Consider the semantics complexity of the following code fragment. The variable `c` may or may not be incremented and three more intermediate states arise compared with Figure 1.

```
if (b++ || c++ > b) ...
```

Arbitrarily many sequence points may occur in an expression leading to arbitrarily branching control flows. Future work should find a more general scheme of inference rules which addresses side effects with sequence points.

5.2 More Formalization

Much of the current logic is shallowly embedded. For instance, the inference rules are embedded as axioms with only informal arguments of correctness, and the predicates `PreEval` and `PostEval` are only partially formalized. There are surely errors or unnecessary restrictions given how complex the semantics of C are. Since one of the values of post-hoc verification of source code is examining extreme cases, the language model must be highly reliable.

This reliability can come from proving the correctness of the logic from a lower level, definitional description such as operational semantics [13, 16] or abstract state machines [9]. Definitional descriptions are much easily to get correct, but may be harder to reason with.

5.3 A Complete Verification System

This paper concentrates on one aspect of practical formal verification: inference rules for complex semantics. But merely having a formal model of the language does not constitute a broadly useful software verification system. We believe the following elements would be necessary and sufficient.

1. A library of examples of design formalizations and examples of how to formalize common programming patterns. A formal specification is the first step in verification [2] and can, in itself, be of great benefit [6]. But finding a formalization and avoiding lapses can be hard. The specification of sorting in an early version of [7] could have been trivially satisfied by setting all the values to zero: it didn't specify that values at the end are a permutation of beginning values.

2. A high level model of the language along with rules of inference, such as axiomatic semantics. As explained above (5.2) the logic must be proven correct from a low level, definitional semantics.

3. Formal models of the environment. This begins, of course, with the programming language, but includes standard libraries, operating system routines, network services, etc. Larger programs often use other services rather than being standalone entities.

4. A powerful, highly automated theorem proving environment. This corresponds to PVS [17], very powerful tactics in HOL, verification condition generators [13, 15], etc. An environment which finds loop invariants and proves most lower level theorems automatically allows a lower entry training cost and less user time.

A system like this could be as widely used as compilers or project management tools are today.

6. Conclusions

Semantic complexity of a language need not prevent formal verification of programs written in that language. We presented new inference rules for `if` and `while` statements in C which may be used when test expressions have some types of side effects. The same general scheme can be applied to develop inference rules for control constructs in other languages. With these new rules, we can use axiomatic semantics to formally reason about a broader class of statements.

A widely-used formal verification system is practical. We outlined the components needed for a complete software verification system, and believe that such a system could be as widely accepted as compilers and configuration management tools are now.

Formal verification can be beneficial, even for well engineered and tested programs. We give an example of how formal verification uncovered hitherto unknown (or undocumented) errors. The discipline of formal verification forces us to think more clearly about our specification and goals.

Although we presented these ideas in the context of post-hoc verification of source code, they also apply to complementary quality control approaches such as validation and testing. For example, formal semantics of a programming languages and formal specifications can drive automatic generation of test cases.

Acknowledgments

We are grateful to Frederick B. Cohen for letting us use thttpd as a test case for verification. We thank for referees for their critiques and suggestions, especially the clearer formulation of inference rules for `if` and `while` statements.

References

[1] Michael A. Arbib and Suad Alagić. Proof rules for gotos. *Acta Informatica*, 11(2):139–148, 1979.

[2] Paul E. Black, Kelly M. Hall, Michael D. Jones, Trent N. Larson, and Phillip J. Windley. A brief introduction to formal methods. In *Proceedings of the IEEE 1996 Custom Integrated Circuits Conference (CICC '96)*, pages 377–380. IEEE, 1996.

[3] Paul E. Black and Phillip J. Windley. Inference rules for programming languages with side effects in expressions. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs '96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 51–60. Springer-Verlag, 1996.

[4] Paul E. Black and Phillip J. Windley. Verifying resilient software. In Ralph H. Sprague, Jr., editor, *Proceedings of the Thirtieth Hawai'i International Conference on on System Sciences (HICSS-30)*, volume V, pages 262–266. IEEE Computer Science Press, January 1997.

[5] Frederick B. Cohen. A secure world-wide-web daemon. *Computers & Security*, 15(8):707–724, 1996.

[6] Ben L. Di Vito and Larry W. Roberts. Using formal methods to assist in the requirements analysis of the space shuttle gps change report. Contractor Report 4752, NASA Langley Research Center, August 1996.

[7] Michael J. C. Gordon. *Programming Language Theory and its Implementation.* Prentice-Hall, Inc., 1988.

[8] Michael J. C. Gordon and Tom F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[9] Yuri Gurevich. Evolving algebras: An attempt to discover semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, 1993.

[10] Samuel P. Harbison and Guy L. Steele, Jr. *C, A Reference Manual.* Prentice-Hall, Inc., 1991.

[11] William L. Harrison. Mechanizing the axiomatic semantics for a programming language with asynchronous send and receive in HOL. Master's thesis, Dept. of Computer Science, University of California, Davis, September 1992.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[13] Peter Vincent Homeier. *Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures.* PhD thesis, University of California, Los Angeles, 1995.

[14] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. Contractor Report 4527, NASA Langley Research Center, July 1993.

[15] W. Douglas Maurer. A minimization theorem for verification conditions. In *Proc. 8th International Conference on Computing and Automation (ICCI '96)*, 1996.

[16] Michael Norrish. An abstract dynamic semantics for C. Technical Report 421, Computer Laboratory, University of Cambridge, May 1997.

[17] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: a prototype verification system. In Deepkar Kapur, editor, *11th Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artifical Intelligence*, pages 748–752. Springer Verlag, June 1992.

[18] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1993.

A Axiomatic Semantics

Our verification work is done in the HOL theorem prover [8]. This appendix briefly explains some HOL conventions and some notations we use in the paper. In HOL, all free variables are assumed to be universally quantified. Variables may range over functions as well as simple types. Function application is implicit. Thus the theorem Px means "for all P and x, $P(x)$ is true."

Our notation is based on axiomatic semantics [12]. There are two main types of statements: partial correctness and total correctness. An axiomatic statement of partial correctness is

$$\vdash \{\texttt{Precondition}\}\ \texttt{Code}\ \{\texttt{Postcondition}\}$$

where `Precondition` and `Postcondition` are predicates on the state of the computation and `Code` is a fragment of code. The above means if `Code` is executed in a state which satisfies `Precondition` and it terminates, `Postcondition` will be true of the resulting state. In this case, "terminates" means that the code doesn't loop indefinitely or abort abnormally. For example,

$$\vdash \{y = 3\}\ \texttt{x = y;}\ \{x = 3\}$$

Although not used in this paper, a statement of total correctness is similar, but asserts that the computation always terminates.

B Unix Permissions

Typically in Unix systems, processes have two identification numbers pertinent to our paper: user ID (UID), representing the person running the program, and group ID (GID), representing the person's group affiliation. Files also have a UID and a GID, and have three sets of permissions:

- those for the file owner ("user"),
- those for people in the same group as the file ("group"), and
- those for any other person in the world ("other").

If the process' UID matches the file UID, user permissions are checked to authorize the operation. If the UID's don't match, but the GID's match, group permissions are checked. If neither UID's nor GID's match, the "other" or world permissions are checked.

Although permissions are typically used hierarchically, it need not be so. Thus a file may be readable by everyone on the system except the owner, if it has read permission for others but no read permission for the owner.