Verifying Resilient Software

P. E. Black*
Computer Science Department
Brigham Young University
Provo UTAH 84602-6576
p.black@ieee.org

P. J. Windley
Computer Science Department
Brigham Young University
Provo UTAH 84602-6576
windley@cs.byu.edu

Abstract

We explore the tension between adding functionality to create resilient software and minimizing functionality to make it more feasible to formally verify software. To illustrate the effects of this trade-off, we examine a tiny example in detail. We show how code written with a good style may be hard to verify, specifically that the test condition is troublesome. We also show that a test condition "improved" in an attempt to make the verification more straight-forward worsens the failure characteristics.

To demonstrate the effect in an actual situation, we examine a secure web server, thttpd, its design principles and security features. We discuss how the security features introduce redundancies making verification harder, but also present some of its formal verification to show that verification is feasible.

We conclude that software should be designed with necessary redundancies and that the temptation to oversimplify the design in order to formally verify it should be resisted.

1 Introduction

Safety critical and other kinds of high reliability software are often required to be tolerant of faults. This tolerance usually comes by anticipating faults and adding functionality to prevent faults from causing failures.

> One key approach used to tolerate failures is *redundancy*. In this approach, a system may employ a multiple number of processes, a multiple number of hardware components, multiple copies of data, etc. [11]

This includes, of course, the software to update copies, switch to backups, detect and recover from faults, etc. Redundancy in software usually appears as extra features to check results. In security applications this redundancy may be manifest as multiple guards to prevent undesired outcomes.

*This work was sponsored by the National Science Foundation under NSF grant MIP–9412581

Formal verification is a means to investigate the consistency and completeness of specifications, properties, models of implementations, assumptions, and inference rules [1]. Since by definition formal verification excludes informal arguments, one cannot gloss over properties which are "obviously true" but are hard to prove. Thus formal verification tends to be tedious, covering all facets. We tend to abstract away detail and eliminate any features which make verification difficult. There is also the temptation to believe that a successful formal verification means the code is completely error-free and that therefore fault handling is unnecessary. If taken to an extreme, we would remove or ignore the very redundant functionality which adds reliability or safety.

Section 2 is a very simple example which we examine in detail. We show that the good, resilient style is harder to verify, and that an arguably acceptable implementation is easier, but has worse failure modes. Section 3 presents an actual product, thttpd, a secure web server. We show that redundant security features do, indeed, translate into additional software and difficult-to-verify properties and realizations. Section 4 examines the verification of one facet of the security properties to demonstrate the difficulties encountered. Since we use notations and concepts which are quite familiar to some people, but totally new to others, we explain some syntax and meaning of axiomatic semantics, the inference system we use in this paper, in Appendix A.

2 Loop Redundancy

To illustrate the apparent tension between resilient software and software which is easy to verify, we begin with a simple example. We will show that a good, resilient style leads to code which is superficially harder to verify. We also show that there is a temptation to change the code to make the verification easier, but the change loses desirable properties of the good style.

The example problem is to write a fragment of C code to compute $z = x^y$ using repeated multiplication. We state that x and y are integers, that $y \geq 0$, and ignore the limitations of computer arithmetic to simplify the example. A good programmer might produce the following code fragment.

```
z = 1;
for (c = 0; c < y; c++) {
    z *= x;
}
```

To make this example clearer, we replace the semantically complex *for* loop with an equivalent and semantically simpler *while* loop.

```
z = 1;
c = 0;
while (c < y) {
    z *= x;
    c++;
}
```

Let's attempt to verify the code fragment. Informally our aim is to prove that $z = x^y$ at the end. More formally we must prove that the following:

- The loop terminates.
- The variable z has the value x^y if the loop terminates.

Note that we must be sure that x and y have the same values as at the beginning of the code fragment. Otherwise we would judge a code fragment that merely sets x, y, and z to 1 (thus trivially satisfy $z = x^y$!) to be correct. Using x' and y' to represent the initial values of x and y, a more complete goal is to prove

$$\{x = x' \wedge y = y'\} \texttt{ code } \{x = x' \wedge y = y' \wedge z = x^y\}$$

Preserving initial values is a vital and often-overlooked point, but since it adds nothing to the example, we will assume x and y do not change. We defer discussion of the proof of loop termination.

Ignoring the side issues, the obvious loop invariant is $z = x^c$. The inference rule for *while* loops ([6], pp. 24–25) is

$$\frac{\vdash \{\texttt{Invariant} \wedge \texttt{COND}\}\ \texttt{BODY}\ \{\texttt{Invariant}\}}{\vdash \{\texttt{Invariant}\}\ \texttt{while}\ (\texttt{COND})\ \texttt{BODY}\ \{\texttt{Invariant} \wedge {}^{\sim}\texttt{COND}\}}$$

If we assume the loop invariant holds, we can conclude from the rule that at termination the invariant is true and the loop condition is false. That is, $z = x^c \wedge {}^{\sim}(c < y)$ or more simply, $z = x^c \wedge c \geq y$. However, we cannot prove $z = x^y$ since we cannot prove $c = y$. But anyone can see that the code is correct; where is the problem?

First, let us convince ourselves that the failure to prove is proper, in other words that the problem is not in the inference rule. Note that the rule for *while* loops, doesn't explicitly include the body in supporting the postcondition. Suppose the increment in the loop is different:

```
while (c < y) {
    z = z * x * x;
    c += 2;
}
```

Clearly c might not be equal to y when the loop exits. Since this is a possibility, the inference rule was correct in *not* letting us conclude $z = x^y$.

2.1 Simplifying the Loop to Verify It

Obviously we can change the code so the termination condition is exactly what we need:

```
while (c != y) {
```

From this we can conclude $z = x^c \wedge {}^{\sim\sim}(c = y)$ and therefore $z = x^y$. Since we can prove our result with this condition, why does good programming style prefer `c < y`?

The test `c < y` leads to termination under a wider variety of conditions than `c != y`. If the test is `c != y`, we can draw very strong conclusions at termination precisely because the loop only terminates in a very particular condition. However since it is often easier to compensate for an incorrect result than an infinite loop, resilient coding practice prefers `c < y` even though we can infer less about the conditions which hold at termination.

It is interesting to note that Edsger W. Dijkstra ([5], pp. 56–57) advocates the opposite coding style: clear verification even at the expense of good failure characteristics. He argues that if a later program change introduces an error or a machine failure occurs, using `c < y` may terminate with an undesired (wrong) result, but no other notice. Using `c != y` *always* establishes the termination condition, or else it doesn't terminate! We disagree with this coding style since a single failure in the compiler, operating system, or hardware could lead to complete collapse of the program function or security.

2.2 Verifying the Original Loop

So that the reader does not incorrectly conclude one must choose between verification and resilience we show how we can keep the test `c < y` and still prove the termination condition. We noted above that the exit condition of a *while* loop doesn't refer to the body code. This suggests we carry more information in the loop invariant. The invariant must guarantee that c doesn't increase much in each loop. An adequate loop invariant is

$$z = x^c \wedge c \leq y$$

At the end of the loop we can conclude

$$z = x^c \wedge c \leq y \wedge {}^{\sim}(c < y)$$

that is

$$z = x^c \wedge c \leq y \wedge c \geq y$$

From the two inequalities we can conclude $c = y$, and therefore $z = x^y$ if the loop terminates.

Let us return to the issue of verifying termination. We can prove termination by showing that a value, such as $y - c$, is always nonnegative in the loop and decreases with each iteration. As was pointed out to us, proving this is straight forward with the resilient loop test `c < y`. Interestingly enough, Dijkstra's supposedly more direct test `c != y` needs the condition $c \leq y$ in the invariant in order to prove termination. In this case changing the code to ease the verification does not really help. The actual example in the next section shows there may be a trade off between resilient design and ease of verification.

3 A Secure Web Server

In June 1995, Management Analytics wrote a secure World Wide Web server called thttpd. (It is named for HTTP, the HyperText Transfer Protocol, which is the most common protocol for WWW on the Internet. HTTP is operationally similar to FTP, but is enhanced and optimized for Web interaction.) The code consists of about 100 lines of C. They point out [3, 4] 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.

Their

> ...solution to the security problem with servers is to design a secure server with security properties that can be explicitly demonstrated.

They then list the general properties of interest 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).

3.1 Design Features

To assure these properties, they incorporate the following design features.

- Small Size, so it is feasible to thoroughly examine for security properties,
- Confinement of Operating Privileges, using the Unix command `setuid` to run as a non-privileged user (named "www") so

  > ...the basic operating system protection features (e.g., access control, process separation, limited input and output capabilities) that are used by millions of people every day are used to protect the server ...

  and to

  > ...trace the activities of the daemon [which] makes automatic detection of anomalous behavior very easy ...

- Confinement of File System Access, using the Unix command `chroot` to avoid access to sensitive or special files, such as password or device files,
- Confinement of File Output, by only writing to a single, predefined log file,
- Confinement of Information Flow, by reading only one fixed size request from the user,
- Confinement of File Inputs, by only opening files which are owned by "www" and "world" readable.

All of these features lead to an intentional redundancy in protection methods. They

> take a multitude of precautions so that if and when one fails, the others will prevent harm, give warning, and allow response to counter the detected weakness before it becomes a full-fledged vulnerability.

For example since thttpd makes several different checks before returning a file, the failure of one library function, such as `chroot` or `geteuid`, is unlikely to compromise security. (Redundant precautions also means that users or operators on the server must take several particular steps to make files available, but also reduces the chance that a human error will allow a breach.)

4 Verifying Properties of Thttpd

This redundancy adds to the verification burden. About ten percent of the code could be completely eliminated if thttpd did not have redundancy. In addition to just having less code to examine, the code which could be eliminated includes system calls and concerns properties used no where else in the program. such as `chroot` and the process' user ID. Not only must the calls (`chroot`, `setuid`, etc.) be modeled to demonstrate that they cause no problems, but their areas of concern (process privilege, file system access, etc.) must be taken into account for all other commands and calls to assure that those don't interfere.

The temptation appears to remove some of the redundant features to ease the task of verification. After all, if the code is formally verified, there are no errors in it, right? Formal verification exposes hidden assumptions and classes of errors, but are not an absolute guarantee. For example, Management Analytics reports a human error of incorrectly putting files in the thttpd's file area. Since the permission were not correct, they were not exposed. Had file access been the only confinement, they would have been. An operating system flaw or compiler failure could have led to similar consequences. We maintain, then, that sound design is vital, even with complete formal verification. As the termination condition showed in the example in Section 2, verification of the resilient design may not be much harder than a "simpler" design. To illustrate, we show some details of verifying thttpd hereafter.

To begin, we note that the top-level goal or predicate must be a conjunction of all the highest level properties. That is, all the properties must hold in order to conclude that the software satisfies our conditions. The properties can be verified separately, then we can conclude their conjunction.

We start with wanting to prove $isSecure(thttpd)$. We define $isSecure(thttpd)$ as

$$hasIntegrity(thttpd) \wedge isAvailable(thttpd) \\ \wedge isConfidential(thttpd)$$

which are the three general properties of interest. Confidentiality is provided in this case by redundant features: confinement of operating privileges, confinement of file system access, and confinement of file inputs. Any one of these *should* be sufficient to establish confidentiality. But to offer defense in depth, thttpd has all three. We must prove all the properties one at a time to show that, indeed, we have redundancy. If instead one property depends on another, we don't have complete independence or redundancy. To conclude that thttpd is secure even if one or two of the properties fail, we must verify security given only the weaker disjunction of these redundant properties.

Notice than that the software may be proved reliable by proving the top-level properties in conjunction, that is, all properties must hold. However the properties should be proven independently (without relying on each other) if possible. Redundant properties which support a top-level property by lines of defense must be proven independently of each other. The top-level property, such as security, is then be proven with the disjunction of redundant properties. That is, if any of the redundant properties hold, the software is safe.

4.1 Verifying a Single Property

To use axiomatic semantics, we must turn predicates on programs into statements about initial and final states at some point. Concentrating on the first general property (and skipping some details), we can define *hasIntegrity*(*thttpd*) as

$$\vdash \{\texttt{FileState file}\}\ \texttt{thttpd} \quad \{\texttt{file} \neq \texttt{LogFile} \Longrightarrow \texttt{FileState file}\}$$

This states that thttpd will never change any file except possibly the log file. A more literal reading is, given that each file is initially in some state, executing thttpd leaves each file (except the log file) in the same state.

The actual definition is more complicated since the predicates are given in terms of files in a file system. For example, `FileState file` above is actually

$$\vdash \forall\, \texttt{fhandle}\, .\ \texttt{StateOfFile fhandle} \quad (\texttt{getFile FileSystem fhandle})$$

where `getFile` gives the contents of a file (handle) in a file system and `StateOfFile` is an under-defined predicate which "fixes" the contents of a file (handle) at an undefined time. However this complexity is not relevant to the discussion at hand.

What about changing file permissions, moving or renaming a file, or deleting or creating files? File permissions can be included in the file state. Moving or renaming a file can be thought of as deleting the original file and creating a new file. Since we have abstracted the file system, we can consider `FileState` a function over *all* files which could ever exist, not just the files currently in the file system. Deletion is marking a file as "completely inaccessible," and creation is marking a file as "accessible" and setting its contents. So all these changes can be seen as changing the state of files.

Each system call must have an axiom which states if and when the file system is changed. Here is an example of the axiom for `fprintf` calls (from [9]).

$$\begin{array}{l}
\vdash \{\texttt{FileState file}\}\ \texttt{fprintf(stream, format, ...)} \\
\{(\exists \texttt{FPRINTF_error}.\texttt{C_Result} = -\texttt{FPRINTF_error}) \\
\quad \wedge(\texttt{inSomeCasesOf C_Result} \\
\qquad (\exists \texttt{FPRINTF_errno}.\texttt{errno} = \texttt{FPRINTF_errno})) \\
\quad \wedge \texttt{FileState file} \vee \\
(\texttt{C_Result} = 0) \wedge \texttt{FileState file} \vee \\
\texttt{C_Result} > 0 \\
\quad \wedge(\texttt{file} = \texttt{fileOf(deref stream)}) \Rightarrow \\
\qquad \texttt{writeFile(deref stream)} \\
\qquad\quad (\texttt{fprintfSpec format C_args}) \mid \\
\quad\ \texttt{FileState file}\}
\end{array}$$

The post condition means that `fprintf` has three possible results: it may fail with a negative result (and in some cases set the global variable `errno`), it may fail with a 0 result, or it may write the information to the stream.

As a side note, while verifying thttpd we proved that the transaction logging function does not change any other files, but we could not prove that it *did* write to the log file to log transactions. We coded the `fprintf` axiom directly from the manual page, which allows that `fprintf` may fail to write anything. The formal verification exposed our assumption that `fprintf` calls always succeed. We can still prove that transactions are logged if we change the model so that `fprintf` never fails or add the assumption that it never fails.

From the beginning of software verification, researchers understood that finding serviceable verification conditions took skill. As Manna and Waldinger state [10]:

> *Finding Invariant Assertions:* Although the invariant assertions required to perform the verification are guaranteed to exist, to find them one must understand the program thoroughly. Furthermore, even if we can discover the program's principal invariants ... we are likely to omit some subsidiary invariants (e.g., $y \geq 0$ above) that are still necessary to complete the proof.

4.2 How Much Verification is Done?

The complete verification of thttpd is still some time in the future. We still need to incorporate arrays and handle time-out's and non-local jumps. We also want to justify our inference system from a denotational semantics.

It has been our experience that the proof proceeds fairly quickly once the infrastructure is in place and the properties and system calls have been formalized. It is building the infrastructure and this formalization which takes the time. We have worked on the verification over a year and hope to finish within a year. At that time we expect that verification of similar software, such as a secure gopher server, will only take a few weeks.

5 Conclusions

Highly reliable software will intentionally contain redundant functionality. This redundant functionality may increase the verification task simply because there is more code to verify, but also because there are more properties to verify and the code has tolerance ("sloppiness") built-in.

Even if the verification task is harder, it is still tractable. In some cases it is not much harder than thorough verification of apparently "simpler" code. In other cases simple approaches, such as proving properties independently then combining them, suffice to prove safety. So although there may be a temptation to simplify the code in order to simplify the proof, this may not be a gain, and will likely lose desirable properties.

Although we have considered only post-hoc formal verification of source code, we believe these conclusions apply to complimentary quality control approaches such as validation and testing, too. Although some changes may ease the task of verification, validation, or testing, sound designs practices should not be (and perhaps need not be) sacrificed.

Acknowledgments

We are grateful to Dr. Frederick B. Cohen of Management Analytics for letting us use thttpd as a test case for verification. We are also grateful to the anonymous reviewers whose detailed and thoughtful comments greatly improved this paper. We regret we cannot cite them properly.

References

[1] P. E. Black, K. M. Hall, M. D. Jones, T. N. Larson, and P. J. Windley, "A Brief Introduction to Formal Methods," *Proceedings of the Custom Integrated Circuits Conference*, San Diego, California, May 1996.

[2] P. E. Black and P. J. Windley, "Inference Rules for Programming Languages with Side Effects in Expressions," to appear in *Proceedings of The 1996 International Conference on Theorem Proving in Higher Order Logics*, Turku, Finland, August, 1996.

[3] F. B. Cohen, "Why is thttpd Secure?" http://all.net/ManAl/white/whitepaper.html or http://all.net/ → *Products* → *Secure http and gopher daemons.* (14 March 1996).

[4] F. B. Cohen, "A Secure World Wide Web Daemon," *Computers and Security*, submitted, 1995.

[5] E. W. Dijkstra, *A Discipline of Programming.* Prentice-Hall, Inc., New Jersey, 1976.

[6] M. J. C. Gordon, *Programming Language Theory and its Implementation.* Prentice-Hall, Inc., New Jersey, 1988.

[7] M. J. C. Gordon and T. F. Melham (Eds.), *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[8] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, October 1969, pp. 576–583.

[9] HP-UX on-line manual, HP-UX Release 9.0: August 1992, Hewlett-Packard Company, Palo Alto, California.

[10] Z. Manna and R. Waldinger, "The Logic of Computer Programming," *IEEE Transactions on Software Engineering*, vol. SE-4, no. 3, May 1978, pp. 199–229.

[11] M. Singhal and N. G. Shivaratri, *Advanced Concepts in Operating Systems.* McGraw-Hill, Inc., New York, 1994, page 330.

A Axiomatic Semantics

This appendix briefly explains some notations we use in the paper.

Our verification work is done in the HOL theorem prover [7, 2]. The syntax

$$\vdash A$$

means that A is a theorem. In HOL, all free variables are assumed to be universally quantified. Variables may range over functions as well as simple types, and function application is implicit. Thus $\vdash Px$ means "for all P and x, $P(x)$ is true."

Our notation is based on axiomatic semantics [8]. 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 meaning is: if `Code` is executed in a state which satisfies `Precondition` and it terminates, `Postcondition` will be true. In this case, "terminates" means that the code doesn't loop indefinitely or abort abnormally. For example,

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

A statement of total correctness is similar, but asserts that the computation always terminates. The syntax uses square brackets ([and]) instead of curly braces ({ and }). For instance, the above assignment always terminates, so we can prove

$$\vdash [y = 3]\ \mathtt{x = y;}\ [x = 3]$$

An inference rule has the form

$$\frac{\vdash A, \ldots, \vdash B}{\vdash C}$$

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