Applying Formal Methods Informally

Matt Bishop
Dept. of Computer Science
University of California at Davis
bishop@cs.ucdavis.edu

Brian Hay
Dept. of Computer Science
University of Alaska Fairbanks
brian.hay@alaska.edu

Kara Nance
Dept. of Computer Science
University of Alaska Fairbanks
klnance@alaska.edu

Abstract

While many problems associated with software development and the associated vulnerabilities are well documented and discussed, there is a distinct and obvious lack of consensus on the means to overcome and remedy these identified issues. This paper introduces the idea of integrating the concepts of formal methods into the programming process from the beginning. This method of applying formal methods informally has the potential to change the programming paradigm to include formal methods; and, when formal methods cannot be applied, an ancillary application of the philosophy and underlying foundational concepts to move towards a culture of more secure programming.

1. Introduction

The poor quality of software is well known and has been for some time. Indeed, in the early 1970s, Gerald Weinberg coined Weinberg's Second Law: "if builders built buildings the way programmers wrote programs, then the first woodpecker to come along would destroy civilization" [4]. To remedy this state of affairs, formal methods were developed. Formal methods allow the mathematical validation of programs, and in some cases, implementation in such a way that programs can be proved to be correct.

The benefit of formal methods is multifold. The two most important ones are the validation of the correctness of the software, assuming the preconditions are met. The second is less formal. In order to verify the software, one must understand it thoroughly, especially how different components interact and how interactions with the system (such as input and output) are constrained. The very exercise of deriving the constraints, preconditions, and postconditions makes the analyst understand the software, and potential problems, in greater depth than otherwise.

The problem with formal methods is their associated expense. They require time, specialized tools, and trained people. Furthermore, they must be integrated into a software development life cycle at points beyond the design and development stages. For example, should the software have to be modified to meet new requirements, or to compensate for erroneous assumptions about how users interact with the program, it will have a significant impact when formal methods are applied. Any such changes must go through much of the design and development phases, and then the *entire* program must be re-evaluated. Thus, formal methods are used in two situations: first, when the system or program being built is small; and second, when the developers have the resources and the need for very high assurance that specific requirements have been met.

Even though formal methods are widely seen as impractical, there is much wisdom and rigor in them that can be applied to programs for a much lower cost. These ideas, of course, produce programs with much less assurance than the application of full formal methods. But given the choice between applying *ad hoc* rules and a framework derived from formal methods, the choice to try to be as complete as possible seems appropriate.

The goal of this paper is to demonstrate how to adapt and apply the ideas and concepts behind formal methods for use in everyday programming. A subtheme of this paper is that formal methods are useful even when not applied fully; they are much more than an academic exercise.

Throughout this paper, we use an example program to illustrate our methodology. This program, the *login* program from UNIX-like systems, is critical to the security of the system. It has four clear goals:

1. Authenticate the user as required;
2. Change the UID of the process (which is initially *root*) to that of the authenticating user;
3. Update log files to reflect the new login; and
4. Initiate a command interpreter (a *shell*) for the user.

Although it is a small program (1405 lines of C code spread over 5 files), *login* controls most ways to access

the system[1], and has many points of interaction with both the user and system resources. Therefore, it provides a good demonstration of how some moderate analysis can improve one's confidence in the correctness of the software.

2. Formal vs. Informal vs. *Ad Hoc* Methods

A *method* is an organized way to attain an objective; *formal* means mathematically or logically verifiable. Thus, for our purposes, *formal methods* are a way to construct programs that can be proved correct. If a program developed using formal methods fails, then the error(s) must lay in either the assumptions or axioms—in other words, the statement of preconditions—only. We assume that all verification steps are performed correctly (whether by theorem provers, other automated techniques, or by humans).

Informal in this context means not involving proof, but providing a strong, rigorously reasoned argument for correctness. Thus, informal methods do not prove correctness. Instead, they provide a strong argument for correctness by providing assurance evidence that others can evaluate. Unlike formal methods, errors can occur in the techniques used by informal methods, but points at which errors can (or are likely to) occur can be described.

Ad hoc describes the methods generally used now. Programmers think through the design, and may or may not document it. They then implement the code, and test it. Rarely is the test coverage complete; usually, it involves those paths of control that the tester believes are the most likely to have errors, or that are most critical to the correct functioning of the program. No proofs of correctness are given. The assurance evidence is, in essence, the result of the testing. The maxim "testing can provide proof of errors, but not proof of their absence" illustrates that this argument for correctness is quite weak. Errors may occur throughout this process.

3. Formal Specification

First, let us examine a specification that uses formal methods. We will then be able to apply the lessons to our example program.

3.1. Formal Methods

A specification states goals. Formal methods do so using a precise language. The associated language may be a purely mathematical formulation, or it may be stated in a specification language such as SPECIAL, HOL, or Z. Once the specifications are clearly defined, one then verifies that they are consistent (otherwise, they cannot be satisfied) and that they meet the stated requirements. With both a mathematical and a specification language formulation, this requires axioms be stated explicitly, and from those that theorems be proven. This may be done manually, although in practice the programs and systems are too large, and automated theorem provers and other tools are used.

As an example, consider the venerable Bell-LaPadula model [5]. One of the rules of this model, which gives a subject access to an object, is the *give_access* rule. The specification for that rule in SPECIAL is [7]:

```
MODULE Bell_LaPadula_Model Give_access

TYPES

Subject_ID:        DESIGNATOR;
Object_ID:         DESIGNATOR;
Access_Mode:       {OBSERVE_ONLY, ALTER_ONLY,
                              OBSERVE_AND_ALTER};
Access:  STRUCT_OF(     Subject_id        subject;
                        Object_id         object;
                        Access_Mode       mode);

FUNCTIONS
VFUN active (Object_ID object) -> BOOLEAN active;
HIDDEN;
INITIALLY
   TRUE;

VFUN access_matrix () -> Access accesses;
HIDDEN:
INITIALLY
   FORALL Access a: a INSET accesses => active (a.object);

OFUN give_access(Subject_ID giver; Access  access);
ASSERTIONS
   active(access.object) = TRUE;
EFFECTS
  'access_matrix() = access_matrix () UNION (access);

END_MODULE
```

The other rules are defined similarly. There is no ambiguity in the Bell-LaPadula SPECIAL specifications as written in SPECIAL. They are precise. Their consistency (or rather, lack of inconsistency) can be verified. The lessons that this demonstrates include the following:

[1] Under some conditions, physical access to the computer enables someone to gain access to the system.

Lesson 1. State goals clearly and unambiguously. This falls out of the use of formal specification languages.

Lesson 2. Be complete; adding a new goal later may create inconsistencies with earlier ones. With formal methods, adding a new rule or goal requires re-verifying the system is consistent.

Lesson 3. Understand the environment in which the program runs. This springs from formal methods' requiring axioms and propositions.

Lesson 4. Know your assumptions. In the realm of formal methods, this is the same as stating axioms.

3.2. Applying the Lessons

We now apply these to the example *login* program. First, as lessons 1 and 2 suggest, we list the goals:

1. Only allow users with accounts on the system to use the system;
2. Restrict that user's privileges to those allowed to the user; and
3. Log sufficient information to reconstruct any unauthorized login.

Lessons 3 and 4 take us to the environment and assumptions. The first assumption is that the *login* program accesses the correct authentication data. For example, is this data stored in /etc/master.passwd, or obtained from a remote server (such as a Kerberos authentication server) or somewhere else? The second assumption is that the login data is up to date.

Take the first assumption. The question is how the *login* program determines what authentication data to access. If the location of that data is determined from the environment, we next ask how it reads the information from the environment. Consider the following code:

```
if ((p = getenv("HOST")) < 0)
        … handle the error …
if (strcmp(p, "host1") == 0)
        authenticate(SKEY);
else if (strcmp(p, "host2") == 0)
        authenticate(KERBEROS);
else
        authenticate(PASSWORD_FILE);
```

This code first determines the name of the host on which the program is running. It then uses this information to determine where the correct authentication data is. So, the authentication data depends on the host named, which is, in turn, determined from the environment variable HOST. But the user can set that variable; therefore, the user can control which authentication data the program uses.

This means that, when writing a program, one must control the environment. This ensures that the program can be constrained to access only trusted, or trustworthy, data and resources. If the environment cannot be controlled, neither the program nor any subprograms should trust it. In the broadest sense, this is impossible because the program must trust some foundational elements such as the CPU. So, we must determine where the program explicitly uses information from the environment (as in the first line of the example). If the user can control that, there is a potential problem.

A similar, much more subtle problem occurred in a vendor's *login* program. The code was subsequently adapted for open source software—and the problem was not completely eliminated for 10 years:

```
authenticate = YES;
while ((o = getopt(argv, argc,  "fph:n")) != EOF){
        switch(o){
        case 'n':  authenticate = NO;  break;
        …
}
```

The reason for the –n option was to allow *login* to be called whenever a new window opened; were it not present, the user would need to enter her password whenever she opened a window. The reason this is an environmental problem, under the control of the user, is clear.

The dependency on environment may not always be obvious. For example:

```
if ((fp = popen("mail staff", "w")) != NULL){
        fprintf(fp, "Send help soon!\n");
        fclose(fp);
}
```

The function *popen* executes the first argument as a command using the standard command interpreter. It, in turn, uses multiple command environment variables to locate the desired "mail" program. The correctness of the program with the above code fragment depends entirely on those environment variables that control *which* mail program is accessed. In particular, the environment variable **PATH** controls the directories searched for the mail program. More fundamentally, the environment variable **IFS** controls which characters separate words in a command. For example, if **IFS** is set to "l \t\n"[2], the command will be interpreted as invoking the program "mai" (because the

[2] That is, the letter "l", space, tab, and newline.

"l" is a word separator, not a part of a word) with the argument "staff". By defining the **PATH** variable appropriately, or by adding an executable program with name "mai", one can force the program to execute arbitrary actions, contrary to its intended function.

Understanding assumptions often requires knowing the fine points of how a system works, something frequently overlooked by programmers. As an example, consider the following code, which resets the program's search path (contained in the **PATH** environment variable, as noted above). Then the program executes a subcommand, which the system locates by searching the directories in the search path in the order given:

```
for(k = 0; environ[k] != NULL; k++)
        if (strncmp(environ[k], "PATH=", 5) == 0)
                break;
if (environ[k] != NULL)
        environ[k] = "PATH=/bin:/usr/bin:/usr/etc";
…
system("echo hithere | mail bishop");
```

This code looks for the first occurrence of **PATH** in the environment, and sets it to something known to be safe. The problem is that there may be multiple definitions of **PATH** in the environment, and this code only changes the first occurrence. And the system may use the second occurrence to determine the directories to search.

This leads to a short list of basic questions:

Question 1. What will the users, and the remote servers, be supplying to the program? How can it be checked for validity? How should the program act if it is invalid?

Question 2. What assumptions are made about each library function's actions? What are its side effects? What assumptions does the library function make?

4. Formal Design

Designing a system requires a structured approach. We first examine one formal methodology for system design, and then extract and apply its lessons to our example program.

4.1. Formal Methods

One approach in formally designing a system is to use layers. In this case, the system functions are broken into layers, and each layer is defined in terms of the layer beneath it. Thus, the layers are in a linear hierarchy and each layer is represented by an abstract machine.

An example of such a methodology is the Hierarchical Decomposition Methodology (HDM). It was used to produce a formally verified design the Provably Secure Operating System (PSOS) [1]. (Unfortunately, the sponsor never funded an implementation, although the importance of the underlying concepts was revisited in 2003 [2].) The HDM proceeds in 5 stages.

The first stage is the *interface definition*. In this stage, the system interface is designed and decomposed into a set of modules. System security requirements are formulated. Two examples of such requirements in PSOS are:

Detection principle: one cannot get information without authorization.

Alteration principle: one cannot alter information without authorization.

The second stage is the actual hierarchical decomposition. In it, modules are arranged into a linear hierarchy, and each is represented by an abstract machine. Using these machines, consistency of structure and function names are verified.

As an example, the PSOS hierarchy had 17 layers, the lowest being capabilities. These layers are shown in Table 1 [1].

Table 1. The PSOS Hierarchy

16	Command interpreter
15	User environments and name space
14	User input/output
13	Procedure records
12	User processes and visible input/output
11	Creation, deletion of user objects
10	Directories
9	Abstract data types
8	Virtual memory (segmentation)
7	Paging
6	System processes and system input/output
5	Primitive input/output
4	Basic arithmetic, logical operations
3	Clocks
2	Interrupts
1	Real memory (registers, core)
0	Capabilities

The third stage is to develop formal specifications for each module, and verify it is consistent and meets the system specifications. In PSOS, the detection and alteration principles were expressed in terms of

capabilities, and then the modules were verified to be consistent with the two principles.

The next stage refined the layering. It defined the functions of each module in terms of the interfaces at the next lower layer. These functions, and the decomposition as a whole, were verified to be consistent with the specification.

The final stage (not accomplished by the PSOS project) was to implement the system. Working from the bottom up, each layer was implemented in terms of the layer beneath it, so each layer could be verified before the layer above it was implemented. Each layer's implementation was proven to meet the specifications.

Two lessons from this decomposition will prove useful.

Lesson 5: *Break the problem into parts, and so forth, refining each step*. Try to keep modules to one task each.

Lesson 6: *Layer the design*. Then each layer can be implemented in terms of the next lower layer. This simplifies checking and debugging.

4.2. Applying the Lessons

In order to apply these important lessons to our example, the *login* program can be broken into four modules.

The first module authenticates the user. A further refinement of this module might be:

1. Get user's authenticator (hashed password);
2. Get user's authentication data (password); and
3. Compare the two; on match, success; else failure

The second module restricts the user's rights to those that are authorized. In this context, this means the program must determine the user and group identifications, and reduce its privileges to them:

1. Get the user's UID and primary and secondary GIDs; and
2. Change the program's UID and GIDs to them.

The third module updates the appropriate log file:

1. Call the function to open the log file;
2. Add a record to the log file to show the user has logged in; and
3. Close the log file.

The last module spawns the appropriate shell, so the user can use the system:

1. Obtain the file name of the (executable) shell associated with that user;
2. Verify that the name of the shell is a valid shell; and
3. Overlay the login program with the shell

Each of these steps can be broken down further. This is the process of *stepwise refinement* that every introductory programming student learns. In essence, lesson 5 reinforces the usefulness of stepwise refinement.

Each of the modules above relies on lower-level modules—in this context, most probably library functions—so lesson 6 states that we need to know the assumptions of the modules (library functions) that are called. Unfortunately, the UNIX-like systems do not have manuals documenting the assumptions; the existing descriptions list some, but not all, of the associated assumptions. The omitted information is almost always "fence" or "boundary" cases.

As examples, the string copy function *strcpy* copies the string identified by the second parameter into the space identified by the first. Thus, there is an implicit assumption that the string is no longer than the space. If this assumption is not acceptable, the function *strncpy* stops copying after a given number of characters in the string have been copied; but if that happens, no terminating NULL byte is added to the (new) first string.

More subtle conditions occur when bogus arguments are given. What happens if you try to allocate 0 or –2048 bytes? Or if the program tells *strncpy* to copy –5 bytes of the string?

Some of these questions are defined by the standards for the C library functions; others are machine- or implementation-dependent. In either case, not knowing how the library functions will react means that the programmer must check that the parameters are such that the result is well-defined, and is consistent with what is desired.

This leads to the following questions:

Question 3: Is the program structured so that security relevant elements are separate from non-security relevant elements? (This simplifies checking the correctness of the security relevant parts.)

Question 4: Are security relevant elements modularized so that each module performs exactly one security related function? (Again, this simplifies checking.)

Question 5: Are the interfaces compatible and simple to use? (In general, all parameters and return values should be checked to ensure their values are consistent with what is expected, except where the

value under consideration is never used. The software should be designed so that validation is as straightforward as possible. For example, passing pointers is an invitation to problems. Other techniques, such as using tokens or tickets that encapsulate the pointers make checking for illegal or undesired values much simpler.)

5. Implementation

The final step, implementation, requires that the actual code be proved correct. The methods and techniques for this vary greatly, but all have the same basic structure. First, state the precondition and postconditions to the functions or modules; then prove that, if the preconditions hold, so will the postconditions. Underlying these proofs are the assumption that the compiler correctly implements, and the processor and other hardware or firmware components execute, the program's operations correctly. We make that assumption in what follows.

Consider the following formal proof that a routine properly computes the remainder of a division:

```
{ x > 0, y > 0 }
int rem(int x, int y){
        while (x >= y)
                { x ≥ y, x > 0, y > 0 }
                x –= y;
                { x – y ≥ 0, x > 0, y > 0 }
        { x < y, x > 0, y > 0 }
        return(x);
}
{ x > 0, y > 0, rem ≥ 0, rem < y, rem ≡ x mod y }
```

The preconditions, postconditions, and proof steps are in **boldface**. The precondition here is that x and y are both positive. On entry into the *while* loop, the preconditions hold, as does the property $x \geq y$ (else the loop would not have been entered). After subtracting y from x, the difference is non-negative (because before, $x \geq y$), so the preconditions hold. When the *while* loop exits (or if it is bypassed), $x < y$. The value of x becomes the value of the function; thus, the postcondition that the function return a non-negative value ($rem \geq 0$) less than y ($rem < y$) that is congruent to x mod y ($rem \equiv x \pmod{y}$) holds.

Thus, when this function is called with two positive arguments, it returns the remainder of the first divided by the second. Notice the *if*; this proof says nothing about the value if either x or y is non-positive. Were we formally verifying the program, we could check that the code never made such a call (or reconstruct the proof).

In our more informal view of verification, though, we simply take the preconditions and turn them into tests. We ensure that, each time *rem*() is called, our program checks that the preconditions hold—in this case, that both arguments to *rem*() are positive. Thus, the following would be considered poor coding:

```
int rem(int x, int y)
{
        while(x >= y)
                x –= y;
        return(x);
}
```

This is considered poor coding because we do not know what will happen when it is called. For example, suppose $x > 0$ and $y < 0$. Also, in most programming languages, there is an ambiguity to the definition of "remainder" when $x < 0$. Specifically, when –5 is divided by 2, is the remainder –1 or 1?

A better coding style is to note, or handle, the cases *not* covered by the preconditions, explicitly. So, we rewrite *rem*() to set an error code *remerror* indicating whether the arguments meet the preconditions (in conventional C style, 1 means they do and 0 means they do not) and then have *rem*() return the remainder:

```
int remerror = 0;          /* no error */
int rem(int x, int y)
{
        if (x <= 0 || y <= 0){
                remerror = 1;
                return(0);
        }
        while(x >= y)
                x –= y;
        return(x);
}
```

This allows the program to check for errors when the routine is called.

The lessons from this are:

Lesson 7: *Know the assumptions made by each function*. Check them whenever possible, and if not possible (for example, when a program must use a pointer as an argument) check the results.

Lesson 8: Check the results of system calls and library functions to ensure they worked as expected. Often, programming manuals either omit side effects or describe them ambiguously. Document such instances in your program, so a reviewer will know why any seemingly unnecessary checking is present.

5.2. Applying the Lessons

We turn to the routine in *login.c* that makes environment variables and their values available to the user. This is called *exporting* environment variables. As noted above, these variables are critical to the correct functioning of the command interpreters and many programs. The environment variables are stored in an array of strings, and each has the form "*name-value*". The preconditions are as follows:

1. Do not export any environment variable with a string length of over 1024 characters;
2. Do not export any environment variable without a value (note the value may be empty, so this means that there is no "=" sign); and
3. Do not export any environment variable named in the *noexport* list.

There is one postcondition:

1. The environment variable is placed in the list of environment variables to be made available to the user.

The following simple function is called for each environment variable to be exported, and it places them in the environment:

```
static int exportv1(const char *s)
{
        p = strchr(s, '=');
        *p = '\0';
        (void) setenv(s, p + 1, 1);
        *p = '=';
        return(1);
}
```

But there is no checking of preconditions. We add the following code to do that:

```
static int export(const char *s)
{
   char *p;
   const char **pp;
   size_t n;
   /* check precondition 1 */
   if (strlen(s) > 1024) return(0);
   /* handle precondition 2 */
   if (strchr(s, '=') == NULL)
      return(0);
   /* handle precondition 3 */
   for (pp =- noexport; *pp != NULL; pp++){
      n = strlen(*pp);
      if (s[n] == '=' && strncmp(s, *pp, n) == 0)
         return(0);
   /* now the original code */
   p = strchr(s, '=');
   *p = '\0';
   rv = setenv(s, p + 1, 1);
   *p = '=';
    /* check the postcondition */
   if (rv == -1) return(0);
   return(1);
}
```

In fact, this is almost exactly part of the *login.c* code used in FreeBSD 8.0. Aside from differences that are cosmetic (for example, *p* is assigned at the first *strchr* call), the precondition checking code is the same. Interestingly, the postcondition check is omitted; in case of error, no recovery is attempted and the error is ignored. A better approach would be to force the use of a preset environment, or at least log the failure for future investigation.[3]

This leads to the following questions:

Question 6: Does the function have a well-defined goal? In the above function, the goal was to make the environment variable available to the user. The function *setenv* does that.

Question 7: Are assumptions checked? In this case, the assumptions occur in two places. The first is in the entry to the routine, because there are three preconditions the argument to the function must meet. The second is that the postcondition is met. The above code checks it; as noted, the original FreeBSD code does not.

Question 8: Is all security-relevant code checked? The checking does indeed add overhead, but if properly modularized the security relevant parts of the program are typically called infrequently (because most of the computations will be done *after* the checking), so the overhead is typically negligible. Also, while the checking may seem unnecessary once a program is thoroughly debugged, it will become invaluable should the program ever be altered or moved to a different system or environment.

6. Conclusion

While formal methods have been used with spectacular success in restrictive environments and on small, limited, or specialized programs and systems [3,6], they are not yet practical for the average programmer nor do they fit within the budgets of most software development environments. Verifying

[3] Whether this is inconsequential, a feature, or a bug is left to the reader to determine.

implementations is seen as too complex and time-consuming, and indeed if high assurance is not a priority, formal specification, design, and implementation validation is seen as consuming too much time and resources. The specialized training and software required is another drawback.

The assumption inherent in this approach is that the correct use of formal methods improves assurance; therefore, the use of this method will also improve assurance (although, as noted above, not so much as formal method would). Of course, just as formal methods can be applied poorly, and thus not provide assurance (or, worse, mislead the analyst into believing that the software is of higher assurance than is appropriate), this method also can be misleading or inadequate if poorly applied. In other words, assurance methods that are applied poorly or incorrectly do not work as well as those applied completely and correctly.

The thesis of this paper is that the philosophy and concepts of formal methods can be integrated into the vast majority of programming that goes on, for all levels of programmers. The evidence of correctness is by no means as strong as when formal methods, or other high assurance methods, are applied—but that the evidence of correctness is much stronger than when the usual *ad hoc* methods are used.

Indeed, analyzing how formal methods work leads to approaches that are useful for any kind of programming. They increase awareness in programmers by suggesting what to look for, where in the code to check, and how to design and implement programs. Thus, even when formal methods cannot be used in their full glory, the *philosophy* and underlying concepts and ideas should be.

Changing the programming culture to one that produces more secure code is essential if we want to be able to protect our digital assets. Historically, much of the content discussed in this paper is taught once in beginning programming classes, and then glossed over in later classes. We can begin to change the culture by increasing the emphasis on these methods the academic arena throughout the curriculum, as well as in industry during all phases of the software development process. This increased emphasis will help to improve the safety and security of the resulting software products.

7. References

[1] P. G. Neumann, L. Robinson, K. N. Levitt, R. S. Boyer, and A. R. Saxena, "A Provably Secure Operating Systems: The System, Its Applications, and Proofs," Technical Report CSL-116, Computer Science Laboratory, SRI International, Menlo Park, CA (May 1980).

[2] P. G. Neumann and R. J. Feiertag, "PSOS Revisited," Proceedings of the 19th Annual Computer Security Applications Conference pp. 208–216 (2003).

[3] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cockl, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal Verification of an Operating-System Kernel," *Communications of the ACM* **53**(6) pp. 107–115 (June 2010).

[4] G. Weinberg, *The Psychology of Computer Programming; Silver Anniversary Edition*, Dorset House, New York, NY (1998).

[5] D. E. Bell and L. LaPadula, "Secure Computer System: Unified Exposition and Multics Interpretation," Technical Report ESD-TR-75-306, ESD/AFSC, Hanscom AFB, Bedford, MA (1975).

[6] J. Pan and K. N. Levitt, "A Formal Specification of the IEEE Floating-Point Standard with Application to the Verification of Floating-Point Coprocessors," *Proceedings of the 24th Asilomar Conference on Signals, Systems and Computers* pp. 505–510 (Nov. 1990).

[7] M. Bishop, *Computer Security: Art and Science*, Addison-Wesley Professional, Boston, MA (2003).