# Chapter 4
# Formal methods for software verification

**Abstract** The growing importance of software in every aspect of our life has fostered the development of techniques aimed at certifying that a given software product has a particular property. This is especially important in critical application areas such as health care and telecommunications, where software security certification can improve a software product's appeal and reduce users and adopters concern over the risks created by software faults. In this chapter, we shall deal with a wide range of formal and semi-formal techniques used for verifying software systems' reliability, safety and security properties. A central notion is the one of a *certificate* i.e. a metadata item containing all information necessary for an independent assessment of all properties claimed for a software artifact. Here we focus on the notion of *model-based certification*, that is, on providing formal proofs that an abstract model (e.g., a set of logic formulas, or a formal computational model, such as a finite state automaton), representing a software system, has a particular property. We start by laying out some of the work that has been done in the context of formal method verification, including in particular the areas of model checking, static analysis, and security-by-contract. Then, we go on discuss the formal methods that have been used for analyzing/certifying large-scale, C-based open source software.

## 4.1 Introduction

Software systems are trending towards increased size and increasingly complex architectures. This is making it more and more difficult to achieve full assurance of a software product's non-functional properties by means of test-based techniques alone (see Chapter 3). An important alternative option is to use *formal methods*, that is, techniques based on logics, set theory and algebra for the specification of software systems models and the verification of the models' properties. The use of formal methods has become widespread,

especially during the early phases of the development process. Indeed, an abstract model of a software system can be used to understand if the software under development satisfies a given set of functional requirements and guarantees certain non-functional properties. Also, the increasing number of reports of security-relevant faults in software shows that the problem of verifying security-related properties cannot be ignored, especially in the development of high-integrity systems where safety and security are paramount.

There are several case studies proving the applicability of formal methods to security certification [17, 30]. Some practitioners are, however, still reluctant to adopt formal methods. This reluctance is mainly due to a lack of theoretical understanding, and to the misconceived perception that formal techniques are difficult to learn and apply. The detection and the prevention of faults is indeed one of the main motivations for using formal methods. Verifying a formal system specification can help to detect many design flaws; furthermore, if the specification is given in an executable language, it may also be exploited to simulate the execution of the system, making the verification of properties easier (*early prototyping*). Over the last few years software verification using formal methods has become an active research area. Special attention is being given to the verification of concurrent and parallel programs, in which testing often fails to find faults that are revealed only through the use of very specific test cases or timing windows. However, the problem of using models for checking a software product's memory-related non-functional properties, also known as *pointer analysis*, remains to be solved. To understand why, let us consider a slight variation of the `digitcount` function we introduced in Chapter 3.

```
int digitcount(char* s)
{
 int digit = 0;
 int i;
 for (i = 1; *(s+i) != '\0'; i++)
      if ('0' <= *(s+i) <= '9')
          digit++;
 return digit;
}
```

While this coding of `digitcount` is functionally equivalent to the one of Chapter 3, switching the type of the input argument from the array of characters of the original version (whose maximum size can be defined in the calling program) to a pointer to char - that is, an address pointing to a memory area whose maximum size is not known - has interesting effects. Assuming that the calling program had originally limited the size of arrays passed to `digitcount` to 256, the number of states (defined, as before, as the possible contents of the function's local variables) of our new version of the sample function may have increased dramatically as a result of this alternative coding. In this example, the state space may change in size but remains

anyway finite; in general, because of the dynamic and unbounded nature of C memory handling primitives (including allocations, deallocations, referencing, dereferencing etc.), models representing C programs with pointers must take into account an infinite state space. With infinite state spaces,[1] exhaustive searches are no longer possible, and checking the properties of models involving recursive pointer types may lead to undecidability [21].

## 4.2 Formal methods for software verification

Let us start with a survey on the categories of formal methods that are of interest for security certification. Program verification techniques fall into three broad categories: The first category involves non-formal or partially formal methods such as testing, which we have discussed in some detail in Chapter 3; the second category, known as *model checking*, involves formal verification of software systems with respect to specifications expressed in a logical framework, either using state space analysis or theorem proving; the third category includes classic *static program analysis* techniques. We shall now provide a brief introduction to both these categories.

### *4.2.1 Model Checking*

Model checking [12] is a formal verification approach for detecting behavioral anomalies (including safety, reliability and security-related ones) of software systems based on suitable models of such systems. Model checking produces valuable results, uncovering software faults that might otherwise go undetected. Sometimes it has been extremely successful: in 1998, the SPIN model-checker was used to verify the plan execution module in NASA's DEEP SPACE 1 mission and discovered five previously unknown concurrency errors. However, model checking is not a panacea. Indeed, there are still several barriers to its successful integration into software development processes. In particular, model checking is hamstrung by scalability issues; also, there is still a gap between model checking concepts and notations and the models used by engineers to design large-scale systems.

Let us focus on the scalability problem, which is a major obstacle to using model checking to verify (and certify) the security properties of software products. Methods and tools to aid design and analysis of concurrent and distributed software are often based on some form of a state reachability analysis, which has the advantage of being conceptually simple. Basically, the verifier states the non-functional properties she would like the program

[1] In some cases, there are ways of representing infinite state spaces finitely, but this would take us well outside the scope of this book.

to possess; then, by means of a model checker tool, she searches the program state space looking for *error states*, where the specified properties do not hold. If some error states are detected, the verifier removes the faults which made them reachable, and repeats the procedure. The reader may detect a certain likeness to the testing process: indeed, one can never be sure that all error states have been eliminated. Usually, state analysis is not performed directly on the code; rather, one represents the program to be verified as a state transition system, where states are values of variables, and transitions are the instructions of the program. Fig. 4.1 shows the transition system for the instruction `for (int i = 1; *(s + i) != 'n0'; i++)` in our `digitcount` example.

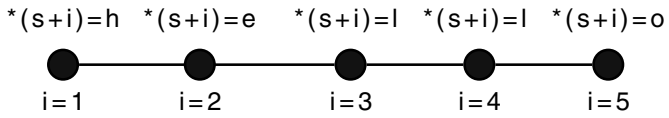


Fig. 4.1: Transition system of a `digicount` instruction

It is easy to see where the problem lies: the number of states may proliferate even for relatively simple programs, making the model checking approach computationally very expensive.

However, space search algorithms allowing more than $10^{20}$ states have been available for several years now, and today's model-checkers can easily manage millions of state variables. Also, a number of techniques have been developed to prevent state space explosion and to enable formal verification of realistic programs and designs. Here, we will only recall an important method in state space reduction, namely *abstraction*. Abstraction techniques reduce the state space of a software system by mapping the set of states of the actual system into an abstract, and much smaller, set of states in a way that preserves all relevant system behaviors. *Predicate abstraction* [20] is one of the most popular methods for systematic reduction of program state-spaces. It abstracts program data by only keeping track of certain predicates on the data, rather than of the data themselves. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. The resulting Boolean program is an over-approximation of the original program.

In practice, the verifier starts with a coarse abstraction of the system to be checked, and checks this abstraction for errors. If checking reports an unrealistic error-trace, the error-trace is used to refine the abstract program, and the process proceeds iteratively until no spurious error traces can be found. The actual steps of this iterative process follow an abstract-verify-refine paradigm.

Automated theorem proving [15] is a well-developed subfield of automated reasoning that aims at proving mathematical theorems by means of a computer program. Commercial use of automated theorem proving is mostly concentrated in integrated circuit design and verification, although many significant problems have been solved using theorem proving. Some of the fields where theorem proving has been successfully used are mathematics, software creation and verification, and knowledge based systems.

When assessing the correctness of the program, two distinct approaches using properties are in use, namely *pre/post-condition* and *invariant assertion*. Large programs should specify pre- and post-conditions for every one of their major procedures.[2] A pre-condition is a logical formula whose value is either the Boolean *true* or *false*. It is a statement that a given routine should not be entered unless its pre-condition is guaranteed[3]. Obviously, a pre-condition does not involve variables that are local to the procedure; rather, it may involve relevant global variables, and the procedure's input arguments.

Ideally, a pre-condition is a Boolean expression, possibly using the for-all ($\forall$) or there-exists ($\exists$) quantifiers.

The post-condition of a procedure is also a Boolean formula which describes the outcome of the procedure itself. The results described in a post-condition are accomplished only if the procedure is called while its pre-condition is satisfied. The post-condition talks about all relevant global variables and the input arguments and relates them to the output results. Like a pre-condition, a post-condition is a strictly Boolean expression, possibly using the for-all ($\forall$) or there-exists ($\exists$) quantifiers. Pre/post-condition -based approaches to software verification formulate the software correctness problem as checking the relationship between the pre-condition Boolean formula that is assumed to hold at the beginning of program execution and the post-condition formula that should hold at the end of program execution.

The pre/post-condition based verification process goes as follows: again, the verifier states the non-functional properties she would like the program to possess. Then, she formulates relevant pre-conditions and post-conditions known to hold for smaller parts of the program, and uses these properties and additional axioms to derive the desired non-functional properties.

One might legitimately ask where do these axioms come from and how pre- and post-conditions can be formulated. The prototype of each procedure - or better, its *signature* - is an important source of pre- and post-conditions. When a procedure declares a formal parameter as passed by value, this means that the actual argument will remain unchanged at the end of the procedure: a basic post-condition. Also, procedures may modify global variables whose names are not listed as actual arguments. It is possible to generate pre- and

[2] In this section we use the word "procedure" as a generic term for both procedures and functions. A function returns a value to the caller but has no side effects on the caller local memory, whereas the purpose of a procedure is exactly to have such a side-effect.

[3] If the procedure is entered anyway, its behavior is unpredictable

post-conditions by stating the global variables' values before and after a call to the procedure. As far as the axioms are concerned, they simply express typical pre-conditions and post-conditions of the programming language's instructions. To further clarify this issue, let us use the elegant notation of *Hoare triplets* [18], as follows: $\{P\}S\{Q\}$, where $P$ is a precondition, $S$ a statement and $Q$ a postcondition.

The meaning of a Hoare triplet is the one suggested by intuition: if $P$ holds before $S$ is executed, then after the execution of $S$ is executed, $Q$ holds. For example, $\{\mathtt{a} > \mathtt{b}\}$ `while` $(\mathtt{a} > \mathtt{b})$ $\mathtt{a}--\{\mathtt{a} = \mathtt{b}\}$ $\{(\mathtt{a}, \mathtt{b} : \mathtt{int})\}$ means that if the integer `a` is greater than `b`, the loop `while` $(\mathtt{a} > \mathtt{b})$ $\mathtt{a}--$ will make them equal. Let us now state the simplest possible axiom, the one corresponding to the assignment instruction:

$$\{Q(e/x)\}x = e\{Q\} \tag{4.1}$$

The axiom states the (rather intuitive) fact that if we have a pre-condition which is true if $e$ is substituted for $x$, and we execute the assignment $\mathtt{x} = \mathtt{e}$, the same formula will hold as a post-condition. For example, if we want to prove that $\{\mathtt{i} = 0\}$ $\mathtt{i} = \mathtt{i} + 1$ $\{\mathtt{i} > 0\}$, we apply the assignment axiom to obtain $\{\mathtt{i} + 1 = 1\}$ $\mathtt{i} = \mathtt{i} + 1\{\mathtt{i} + 1 > 1\}$, from which the thesis is derived by simple arithmetics.

Verification becomes more difficult when we consider loops, a basic control structure in nearly all programming languages. Loops are executed repeatedly, and each iteration may involve a different set of pre- and post-conditions. Therefore, verification needs to focus on assertions which remain constant between iterations rather than on pre- and post-conditions. These assertions are known as *loop invariants*, and remain true throughout the loop. To clarify this concept, let us consider the code fragment below, which computes the minimum in an array of positive integers:

```
min = 0;
int j;
for (j = 0; j <= n; j++)
{
 if(s[j] < min)
    min=s[j];
}
```

The invariant of this loop is that, at any iteration, $s[k] < min$ for $k = 0, 1, \ldots, j$.[4]. To prove that an assertion of interest still holds after a loop terminates, the verifier must start by proving that the loop does indeed terminate. The verifier needs to identify an invariant and use it together with axioms to derive the theorem that the desired assertion is true after last iteration. It is interesting to remark that identifying pre-conditions, post-

[4] The invariant holds even for $j = k = 0$, since the array is made of positive integers.

conditions and invariants is useful even if the formal verification process is not carried out. Consider the following code:

```
int digitcount (char s[])
{
 if (!precond(s))
     return -1;
 /* .. rest of the digitcount code.. */
}
```

Here, the function `digitcount` is doing something unexpected: it is checking its own precondition. If an input parameter violates the pre-condition, that is, `precond(s)` is false, the function returns a value that is outside its expected range of return values, in this example $-1$. This precaution increases the robustness of the function, preventing error conditions due to malformed inputs. [5]

A key difference between the model checking approach to software verification and the theorem proving one we just explained, is that theorem provers do not need to exhaustively visit the whole program state space in order to verify properties, since the constraints are on states and not on instances of states. Thus, theorem provers can reason about infinite state spaces and state spaces involving complex datatypes and recursion.

A major drawback of theorem provers is that they require a great deal of user expertise and effort: although theorem provers are supposed to support fully automated analysis, only in restricted cases is an acceptable level of automation is provided. This is mainly due to the fact that, depending on the underlying logic, the problem of deciding the validity of a theorem varies from trivial to impossible. For the case of propositional logic, the problem is decidable but *NP*-complete, and hence only exponential-time algorithms are believed to exist for general proofs. For the first order predicate calculus, the theorem prover could even end up in non-termination. In practice, the theoretical results require a human to be in the loop, to derive non-trivial theorems and to guide the theorem prover in its search for a proof. Despite these theoretical limits, practical theorem provers can solve many hard software verification problems.

### *4.2.2 Static Analysis*

Static program analysis aims to retrieve valuable information about a program by analyzing its code. Static analysis of programs is a proven technology

[5] Strictly speaking, here `precond(x)` is not a proper precondition of `digitcount`, because `digitcount` is executed whether `precond(x)` is true or not; but the programmer can now be confident that the "real" `digitcount` code will be executed only if `precond(s)` evaluates to true.

in the context of the implementation and optimization of compilers and interpreters. The Syntactic analysis carried out by compilers is a first step in this direction: many faults due to typing mistakes can be tracked by modifying the C syntax specification on which the compiler is based to generate appropriate warning messages. Let us consider the following code fragment:

```
int digit = 0;
if(*s = '\0')
   return digit;
else
   /* rest of the digitcount code */
```

It is easy to see that this code always returns 0; this is due to the programmer erroneously using an assignment instead of a comparison operator. The C syntax analyzer can be modified to generate a warning whenever an assignment statement appears in a conditional expression (where a comparison would be expected). In the modified syntax specification, the compiler action upon detecting an assignment in a conditional expression is specified as "print a warning message".

In recent years static analysis techniques have been applied to novel areas such as software validation, software re-engineering, and verification of computer and network security. Giving a way of statically verifying a security property has, in principle, the advantage of making the checking of the property more efficient; moreover it allows the writing of programs which are secure-by-construction (e.g., when the performed analysis is proved to imply some behavioural security properties). As most non-trivial properties of the run-time behaviour of a program are either undecidable or *NP*-hard, it is not possible to detect them accurately, and some form of approximation is needed. In general, we expect static analysis to produce a possibly larger set of possibilities than what will ever happen during execution of the program. From a practical perspective, however, static analysis is not a replacement for testing nor can it completely eliminate manual review. Static analysis is effective only when operating on the source code of programs, whereas code consumers typically deal with binary code which makes it difficult (if not impossible) for them to statically verify whether the code satisfies their policy. This is not the case with open source, though; and this remark alone would be sufficient to make static analysis an important topic for our purposes. But there is more: static analysis has a proven record of effectiveness for dealing with security-related faults. Often attackers do not even bother to find new faults but try and exploit well-known ones, such as buffer overflow, which could have been detected and removed using static code analysis. When performing static analysis of a program, the verifier uses tools like `lint` and `splint`, which perform error checking of C source, to scan the program's source code for various vulnerabilities. Such scanning involves two steps: *control flow* and *data flow* analysis.

Control Flow Analysis (CFA) [24] is an application of Abstract Interpretation technologies. The purpose of CFA is to statically predict safe and computable approximations to the dynamic behaviour of programs. The approach is related to Data Flow Analysis and can be seen as an auxiliary analysis needed to establish the information about the intra- and inter-procedural flow of control assumed when specifying the familiar equations of data flow analysis. It can be expressed using different formulations such as the constraint-based formalism popular for the analysis of functional and object-oriented languages, or the Flow Logic style. Flow Logic is an approach to static analysis that separates the specification of when a solution proposed by analysis is acceptable form the actual computation of the analysis information. By predicting the behaviour of a software system, it leads to positive information even when the system under evaluation does not satisfy the property of interest, whereas the type-system approach is binary-prescriptive (a system is either accepted or discarded). Moreover, it is a semantics-based approach - meaning that the information obtained from the analysis can be proved correct with respect to the semantics of the programming language, that is, the result reflects an appropriate aspect of the program's dynamic behaviour. The formalization of Control Flow Analysis is due to Shivers in [29], where the analysis is developed in the context of functional languages. The CFA technique has been used extensively in the optimization of compilers, but over the last few years it has also been used for verification purposes. In the case of the Flow Logic approach, there is an extensive literature, showing how it has been specified for a variety of programming language paradigms. Moreover, this technique has been used to verify non-trivial security properties, such as stack inspection and a store authorization in a broadcast process algebra. To fix our ideas, let us consider the simplest case [16], where control flow analysis involves setting up a control flow graph, i.e., a *Directed Acyclic Graph* (DAG) which represents the program's control flow. Each node in the DAG corresponds to a program instruction and the edges from one node to another represent the possible flow of control. Control flow graphs for basic instructions are shown in Figure 4.2.

Control flow graphs for more complex statements can be constructed inductively from the control flow graphs of simple statements. Function calls are also represented as nodes in a control flow graph. When traversing a program's control flow graph if one comes across a node representing a function call then the control flow graph of the corresponding function (if it exists) is also traversed.

Data flow analysis determines the different properties a variable can have through taking different paths in the program, in order to identify for potential faults. The nodes in the CFG are used to store information about certain properties of data such as initialization of variables, references to variables, and so forth.

For example, let us consider the following code fragment:

```
int digitcount (char* s)
```

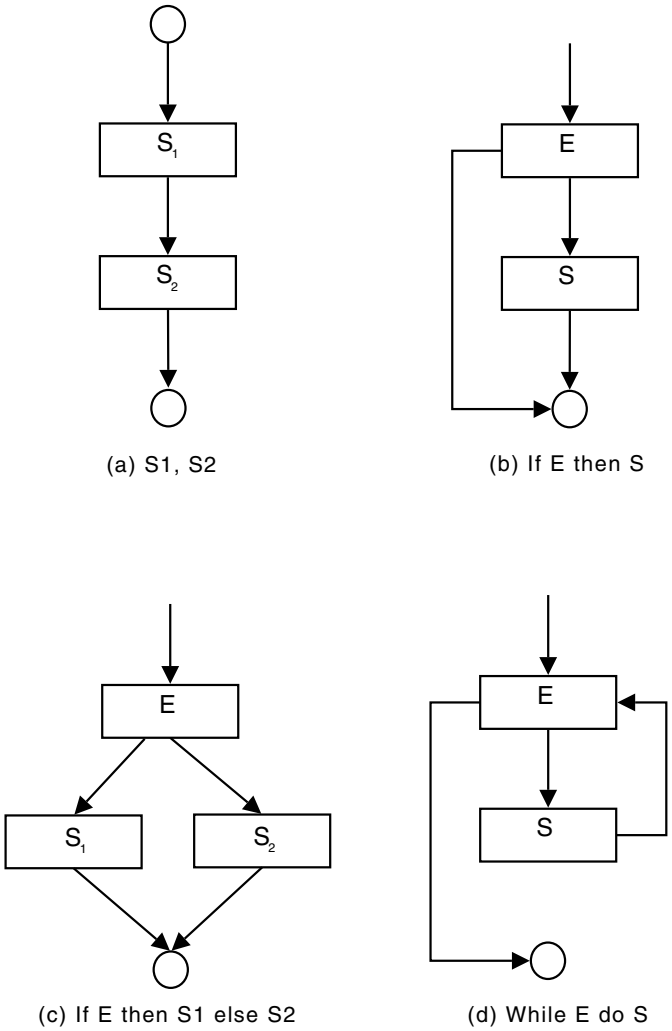


Fig. 4.2: Control flow graphs for basic instructions

```
{
 int digit;
 if (*s != '\0')
 {
  digit =0;
  for (int i = 1; *(s+i) !=' \0'; i++)
       if ('0' <= *(s+i) <= '9')
           digit++;
```

```
    }
    return digit;
}
```

Though the variable `digit` is initialized within the `if` statement, it is not always guaranteed that `digit` will actually be initialized before being accessed (when $*\texttt{s} ==' \texttt{n0}'$, `digit` is uninitialized). Static analysis will point out that variable `digit` is not initialized in one of the paths, a fact that may escape testing unless the void string is used as a test case. Analysis of the control flow graph can also be used for detecting memory-related faults like dereferencing uninitialized pointers, forgetting to free allocated memory, and so on.

To carry out this analysis, the verifier needs to use flow analysis and build the control flow graph, adding memory-related information to nodes. For each variable, such information will say whether at the execution of the instruction corresponding to the node the variable is initialized or not (remember that the variable will be considered as possibly not being initialized if there exists at least a path through the graph on which the initialization is skipped). As a formal notation, we can associate to each node $S$ on the control graph a set (let's call it *NonInit*) consisting all variables for which a path exists which terminates in $S$ and does not include initialization. Similarly, we can associate to $S$ other sets such as *NullPointer*, including all pointers whose value may be null at $S$. This way, the instruction corresponding to $S$ can be checked w.r.t. the sets associated to it. For instance if $S$ dereferences a pointer which is in *NullPointer*, or accesses a variable belonging *NonInit*, the verifier receives a warning.

A major drawback of these analysis techniques is that they may generate *false positives*. In general, the verifier has no idea of the semantics of the program, and may consider faulty some code that the programmer has written intentionally.

### *4.2.3 Untrusted code*

Significant steps forward have been made in the use of software from sources that are not fully trusted, or in the usage of the same software in different platforms or environments. The lack of OS-level support for safe execution of untrusted code has motivated a number of researchers to develop alternative approaches. The problem of untrusted binary code is solved when using certified code (both proof-carrying code (PCC) [22, 23] and model-carrying code (MCC) [27, 28]), which is a general mechanism for enforcing security properties (see Chapter 9).

In this paradigm, untrusted mobile code carries annotations that allow a host to verify its trustworthiness. Before running the guest software, the host checks its annotations and proves that they imply the host's security policy.

Despite the flexibility of this scheme, so far, compilers that generate certified code have focused on simple type safety properties rather than more general security policies. A major difficulty is that automated theorem provers are not powerful enough to infer properties of arbitrary programs and constructing proofs by hand is prohibitively expensive. Moreover the security policy needs to be shared and known a priori by both code producer and consumer. Unable to prove security properties statically, real-world security systems such as the Java Virtual Machine (JVM) have fallen back on run-time checking. Dynamic security checks are scattered throughout the Java libraries and are intended to ensure that applets do not access protected resources inappropriately. This situation is unsatisfactory for a number of reasons: *(i)* dynamic checks are not exhaustive; *(ii)* tests rely on the implementation of monitors, which are error-prone; and *(iii)* system execution is delayed during the execution of the monitor.

### *4.2.4 Security by contract*

The security-by-contract mechanism [13, 14] draws ideas from both the above approaches: firstly, it distinguishes between the notion of *contract*, containing a description of the relevant features and semantics of the code, and the *policy*, describing the contractual requirements of the platform/environment where the code is supposed to be run; secondly, it verifies at run-time if the contract and the policy match. This approach derives from the "design by contract" idea used to design computer software. It prescribes that software designers should define precise verifiable interface specifications for software components based upon the theory of abstract data types and the conceptual metaphor of a business contract.

The key idea of security-by-contract derives from the growing diffusion of mobile computing and nomadic devices. Since the demand for mobile services, which are dynamically downloaded by users carrying mobile devices, is growing, there is a need for mechanisms for certifying security properties of the downloaded application and for certifying how the application interacts with the host device. The *contract* accompanying the application specifies the relevant security actions while the *policy* defines the host's requirements i.e. the expected behaviour of applications when executed on the platform. Contracts and policies are defined as a list of disjoint rules as follows.

```
<RULE> :=
     SCOPE [ OBJECT <class> |
        SESSION |
        MULTISESSION ]
        RULEID <identifier>
        <formal specification>
```

where:

- *SCOPE* defines at which level the specified contract will be applied [13]: *(i) object,* the obligation must be fulfilled by objects of a given type, *(ii) session,* the obligation must be fulfilled by each run of the application, *(iii) multisession,* the obligation must be fulfilled by all runs of the application;
- *RULEID* identifies the area at which the contract applies, as for instance, *files* or *connections*;
- <*formal specification*> provides a rigorous and not ambiguous definition of the rule semantics based on different techniques, such as, standard process algebra, security automata, Petri Nets and the like.

Based on contract and policy definition, the matching algorithm verifies if contracts and policies are compatible. A matching will succeed if for each behaviour happening during the code execution both contract and policy are satisfied. In Table 4.1, we present a simple example of a contract/policy matching taken from [13].

| **Contract/Policy Rule** | Object can use one type of connection only | Object can use every type of connection only |
|---|---|---|
| Object can use HTTP connection only | X | X |
| Object can use HTTP and SMS connections | | X |

Table 4.1: Contract/Policy Matching.

The matching algorithm is defined in a generic way, that is, independently from the formal model used for specifying the rules. In [13] an example of how the matching algorithm can be used with rules specified as Finite State Automaton (FSA) is also provided. Differently from the model checking techniques introduced in Section 4.3.2, which use finite state automaton to define syntactic patterns the program should not contain, contracts/policies define the expected behaviour of the application.

## 4.3 Formal Methods for Error Detection in OS C-based Software

As we have seen in the previous sections, formal methods have been introduced in the past to answer a simple question: "What should the code do?'. In other words, formal methods were aimed at specifying some functional or non-functional properties the software should possess. Although the response to this question may seem trivial when looking at a toy function consisting of a few lines of code, it is much more difficult to answer in the case of huge software products (million of *Lines Of Code* (LOC)). In the previous sections, we

outlined how formal methods can be used to compare what the code should do, with what the code actually does.

The scenario described in this section is even more critical, because we need to consider open source software. The application of formal methods to open source software is somewhat difficult and it has sometimes been considered as a wrong choice. The open source paradigm is based on a cooperative community-based code development, where code changes rapidly over time and unambiguous specifications may simply not be available (see Chapter 5). The code itself, in fact, is considered the first-line specification of an open source system. This explains the problems faced by practitioners trying to apply formal methods to huge undocumented open source software, such as Linux. Usually open source software is not developed from a stable specification and is based on programming languages which do not readily support formal methods, like the C language used for coding Linux.

### *4.3.1 Static Analysis for C code verification*

Much effort has been put into designing and implementing static analysis techniques for the verification of security-critical software. The need for solution to the problem of finding potential vulnerabilities is especially acute in the context of security-critical software written in C. As we have seen, the C language is inherently unsafe, since the responsibility for checking the safety of array and pointer references is entirely left to the programmer. Programmers are also responsible for checking buffer overflows. This scenario is even more poignant in the context of an open source development community, where different developers with heterogeneous skills and profiles contribute to the software.

Software faults spotlighted in the last few years, the ones related to buffer overflows have been the most frequently exploited. A notorious attack (the so-called Morris worm) in November 1988, which infected about 10% of all the computers connected to the Internet, exploited a buffer overflow in the *finger daemon* of Sun 3 systems and VAX computers running variants of Berkeley UNIX. This attack is often quoted as the first one that caused a widespread infection, captured the attention of the world, resisted expert analysis, and finally resulted in FBI investigations and legal actions. The Morris worm had at least one positive effect: it increased the awareness of the software industry about the dangers produced by Internet-based attacks. Also, as a result of the Morris worm, the Carnegie Mellon Computer Emergency Response Team (CERT) [8] was formed. Currently, the CERT institution represents the main reporting center for Internet security problems.

Wagner et al. [31] describe an approach based on static analysis to detect buffer overflow vulnerabilities. They start from the assumption that C is insecure and developers, including expert ones, are themselves sources of

vulnerabilities. Their solution applies static analysis to identify and fix security flaws before these can be exploited by a malicious adversary. In particular, the problem of detecting a buffer overflow is modeled as an integer range constraint problem, which is solved by means of an algorithm based on graph theoretic techniques. Also the authors focus on balancing precision and scalability. The trade-off between precision and scalability introduces some imprecision in the detection software, causing the identification of wrong vulnerabilities (false positives) and the non-identification of real ones (false negatives).

Wagner's solution is based on two major ideas: *(i)* since most buffer overflows happen in string buffers, C strings are modeled as an abstract data type; *(ii)* buffers are modeled as a pair (*as*,*l*) where *as* is the allocated size for the string buffer, and *l* is the length, that is, the number of bytes used.

In summary, the authors provide a conceptual framework modeling string operations as integer range constraints and then solving the constraint system. The implementation of the framework is achieved via three main steps, which are described below.

Constraint language definition. This step carries out the definition of a language of constraints to model string operations. To this end, the concepts of *range*, *range closure*, and arithmetic operations over ranges are introduced. An integer range expression is defined as:

$$e ::= v \| n \| n \times v \| e + e \| e - e \| max(e \cdots e) \| min(e \cdots e)$$

where $n$ is an integer and $v$ is a set of range variables. From the range expression a *range constraint* is then defined as $e \subseteq v$ and an assignment as $\alpha : v \longmapsto \alpha(v) \subseteq Z^{\infty}$. An assignment $\alpha$ satisfies system of constraints if all the assertions are verified when the variables are replaced with the corresponding values $\alpha(v)$ in the assignment.

Constraint generation. In this step, after parsing the source code and traversing the obtained parse tree, a system of integer range constraints is generated. Each integer variable is associated with a range variable, whereas a string variable is associated with two variables (the string's allocated size and the actual string length) plus a safety property $len(s) \leq alloc(s)$, where $len(s)$ includes the terminating '\0'. For each statement an integer range constraint is then generated. The safety property for each string will be checked later.

Constraint resolution. Finally, an algorithm is used to find a bounding box solution to the system of constraints defined by the previous steps. This is guided by the fact that a program with $k$ variables generates a statespace $Z^k$ where the $i$-th component is the value of the $i$-th variable. The program execution is modeled as a path and the goal becomes to find a minimal bounding box including all possible paths in the $k$-dim space. To this end a graph is built where each node represents a variable and an edge between two nodes in the graph represents a constraint involving the two variables.

The constraint solver works by propagating information along the paths and finding a solution to the constraint system.

Although this technique provides a way to detect buffer overflow, it cannot handle C pointers and aliasing. Other researchers working in static analysis have focused on pointer analysis [1, 19] (see also Chapter 3). In order to understand the notion of a pointer-related software fault, let us consider the `memcpy` system call, whose prototype is:

`void memcpy(void dest, const void source, size_t n).`

`memcpy` copies the content of the memory pointed to by `source` (an address within the caller's address space) to the area, again in the caller's address space, pointed to by `dest`.[6]

Intuition suggests that a program failure may occur if the `dest` address passed to `memcpy` is invalid. This type of pointer-related faults are known to be difficult to detect through testing alone; even static analysis tends to perform rather poorly, because pointer-related faults concern specific pointer values rather than its type. Of course, one could wrap `memcpy` to check the destination address before calling it; but this might introduce a non-negligible overhead. The problem becomes more intricate if *kernel pointers* are involved [19], because user space and kernel space addresses cannot be mixed without undesirable results being produced. Let us consider the function

```
int var;
void getint(int *buf)
{
 memcpy(buf, &var, sizeof(var))
}
```

which uses `memcpy` to copy the content of the variable `var` into the buffer pointed to by `buf`. Let us assume that some malicious code initializes `buf` with a value corresponding to some user space address before calling `getint`. If the kernel blindly executes the copy using `buf` as the destination, a kernel failure may occur. For instance, if the destination address were a typical user space address like `0xbf824e60`, it would simply not be in the range of kernel space addresses (which start at `0xc0000000` and trying to write to it within the kernel would provoke a kernel `oops`. If the kernel were configured to panic on `oops`, then the machine would crash. At this point, the reader might object that `memcpy` is surely coded in such a way to prevent this attack. Unfortunately, this is not the case. Here is the implementation of `memcpy` in `linux − 2.6.24.2/lib/string.c`:[7]

---

[6] The behavior of `memcpy` is undefined when destination and source overlap

[7] Note that this implementation handles overlapping source and destination areas correctly, provided the target address is below the source address.

```
/**
 * memcpy - Copy one area of memory to another
 * @dest: Where to copy to
 * @src: Where to copy from
 * @count: The size of the area.
 *
 * You should not use this function to access IO
 * space, use memcpy_toio() or memcpy_fromio()
 * instead.
 */
void *memcpy(void *dest, const void *src,
             size_t count)
{
 char *tmp = dest;
 const char *s = src;

 while (count--)
        *tmp++ = *s++;
 return dest;
}
EXPORT_SYMBOL(memcpy);
```

The `memcpy` implementation was kept as simple as possible, and for a good reason: one cannot risk `memcpy` going to sleep, as it could happen if `memcpy` was coded in a more sophisticated way.

Since `memcpy` includes no run-time checks for pointer-related faults, we can only hope that *a priori* pointer analysis can be used to pinpoint the fact that `getint` code can be unsafe. This can be done by annotating the pointer type declarations with additional information supporting program analysis. In [19], some *qualifier*s of pointer types are used to highlight pointers which contain kernel addresses. The code calling `getint` can be annotated using two qualifiers `user` and `kernel` as follows:

```
int memcpy(void * kernel to, void * kernel from,
           int len);
int var;
void getint(int * user buf)
{
 memcpy(buf, &var, sizeof(var))
}
```

When the above code is analyzed, the analyzer notices that `getint` receives a user pointer `buf`, which is then passed to memcpy as a first parameter. A type error is then raised and the potential fault identified. Of course, an annotation-based technique like this one will generate some false positives, i.e. cases in which the function was purposefully designed to handle both kernel and user space addresses. However, the interesting experimental results

reported by [19] show that the analysis of Linux kernel 2.4.20 and 2.4.23 has identified 17 previously unknown faults due to mixing user space and kernel space addresses.

Faults like writes via unchecked pointer dereferences are often exploited by malicious code. A classic attack (often called *stack smashing* [31]) uses unchecked string copy to cause a buffer overflow.

We have described this attack in Chapter 3; however, to recall this point, let us consider the following C procedure that uses a pointer to copy an input string into a buffer stored on the stack, incrementing the pointer after copying each character without checking whether the pointer is past the end of the buffer.

```
void bufcopy(char *src)
{
 char buf[256];
 char *dst = buf;
 do
   *dst = *src;
   dst++; src++;
 while (*src != "\0")
}
```

By providing a string longer than 256, an attacker can cause the above procedure to write after the end of the buffer, overwriting other locations on the stack, including the procedure's own return address. In Chapter 3 we have seen how crafting the input string, the attacker can replace the procedure's return address with the address of malicious code stored, say, in an environment variable, so that when the procedure returns, control is transferred to the attacker's code. Context-sensitive *pointer analysis* is used to detect faults due to lack of bounds checking like the one above [1]. Two types of pointer analysis have been defined: *(i) CONServative pointer analysis* (CONS), which is suitable for C programs following the C99 standard and *(ii) Practical C Pointer* (PCP), which imposes additional restrictions that make it suitable also for programs that do not follow the C99 standard [7].

PCP includes several assumptions that model typical C usage. First of all, PCP allows arithmetic applied to pointers of an array, such as `buf++` in the example above, only if the result points to another element of the same array. When pointers to user-defined `struct` types are used, PCP applies the notion of *structural equivalence*: two user defined types are structurally equivalent if their physical layout is exactly the same. PCP allows assignments (such as `*dst = *src`) and type casts (such as `dst = (atype *)src`) only between structurally equivalent types. For this reason, PCP has been shown to provide a better accuracy in detection of format string vulnerabilities; also, pointer analysis substantially reduces the overhead produced by dynamic string-buffer overflow tools (30%-100%) [1].

More advanced techniques mark all information coming from the outside world as *tainted*. A potentially vulnerable procedure should be written to raise an error if passed a tainted parameter. Chen and Wagner [10] introduced a static analysis technique that can find taint violations. The provided solution has been tested using the Debian 3.1 Linux distribution. The experiments considered the 66% of Debian packages and found 1533 format string taint warnings, 75% of which are real faults.

Static analysis can be complemented with run-time techniques, such as white lists of memory addresses pointers are allowed to contain. An instruction modifying the value of a pointer can only be executed within a procedure which checks whether the modified value will be in the white list. Ringenburg and Grossman [25] used white lists together with static analysis for preventing format string attacks, Their solution takes advantage of the dynamic nature of white-lists of %n-writable address changes, which are used to improve flexibility and encode specific security policies.

### *4.3.2 Model Checking for large-scale C-based Software verification*

After introducing pointer analysis techniques, let us now survey some tools supporting automatic discovering of security flaws. We are particularly interested in model checking tools able to analyze huge software products such as an entire Linux distribution.

In [26], the MOPS static analyzer [9] is used to check security properties of a Linux distribution. MOPS relies on Finite State Automaton (FSA), whose state transitions correspond to syntactic patterns the program should not contain. For instance, syntactic patterns can express violations on pointer usage, structural type compatibility, and the like. The MOPS analysis process starts by letting users encode all the sequences of operations that do not respect the security properties they are interested in as paths leading to error states within FSAs. Program execution is then monitored against the FSAs; if an error state is reached, the program violates the security property. MOPS monitor takes a conservative view: potentially false positives are always reported, and then users have to manually check if an error trace is really a security vulnerability. MOPS-based experiments have been applied to the entire Red Hat Linux 9 distribution, which consists of 839 packages with about 60 millions of lines of code, and required the definition of new security properties to be model checked. To extend pattern expressiveness, [26] introduces *pattern variables*, which can describe different occurrences of the same expression. To improve scalability the concept of *compaction* is used, that is, simplifying the program to be analyzed by checking the relevant operations only, and MOPS is integrated with existing build processes and interposed with `gcc`. Error reporting in MOPS is then enhanced by dividing error traces

in groups and selecting a representative used by the users to determine if a bug has been discovered. With this extension, MOPS shows all programming errors and at the same time reduces the number of traces to be analyzed by the users by hand.

Below, we describe four main security properties defined in [26] and the results of the analysis of Red Hat Linux (see Table 4.2). The results show the feasibility of using MOPS for large scale security analysis.

| **Property** | **Warnings** | **Bugs** |
|---|---|---|
| TOCTTOU | 790 | 41 |
| Standard File Descriptors | 56 | 22 |
| Temporary Files | 108 | 34 |
| `strncpy` | 53 | 11 |

Table 4.2: Model-Checking Results

Time-To-Check-To-Time-Of-Use (TTCTTOU). This technique checks whether access rights to an object have expired at the time it is used. A classic application is to find vulnerabilities of file systems due to *race conditions*. Let us consider the classic example of a process $P$ that tries to access a file system object $O$ and, once access has been granted, passes a reference to $O$ as an argument of a system call, say, to display information in $O$. If a context switch takes place after $P$'s access rights to $O$ have been checked but before $P$ executes of the system call, permissions may be changed while $P$ is suspended. When $P$ is scheduled again, it no longer has the right to access $O$; however, it goes on to pass the reference to $O$ it already holds to the system call. Of this reference still allows displaying $O$, a vulnerability has been detected. In practice, three different types of vulnerabilities have been found: *(i) Access Checks*, discussed in the above example, *(ii) Ownership Stealing*, where an attacker creates a file where a program inadvertently writes, *(iii) Symlinks*, where the file a program is writing to gets changed by manipulating symbolic links. As shown in Table 4.2, 41 of 790 warnings (i.e., traces violating the security property) have been found to be real software faults.

Standard file descriptor. This attack uses the three standard file descriptors of Unix (i.e., *stdin*, *stdout*, *stderr*) to exploit system vulnerabilities. Attackers can be able to append data to important files and then gain privileges, or read data from files that they are not supposed to access, by manipulating the standard file descriptors. As shown in Table 4.2, 22 of 56 warnings have been found to be real faults.

Secure Temporary Files. An attacker exploits the practice of using temporary files to exchange data with other applications, writing logs, or storing temporary information. In Unix-like systems, these data are usually written to the `/tmp` directory, where each process can read/write. Also, the

functions to create temporary files are unsecure since they return a file name rather than a descriptor. An attacker guessing the file name is able to create a file with the same name and then access the information that will be stored in. As shown in Table 4.2, 34 of 108 warnings have been found to be real faults.

`strncpy`. String copying is a classic source of potential attacks leading to buffer overflow attacks. `strncpy` is not safe since it leaves to the developer the responsibility of manually appending the null character ('`n`′) that should terminate every C string. Both scenarios have been modeled with a FSA. However, this security property produced a set composed by 1378 unique warnings, which makes a complete manual analysis burdensome. An alternative to a complete analysis is selecting semi-randomly set of packages that contain at least one warning. In the experiments, 19 packages have been selected with 53 warnings, 11 of which have been identified as faults. If all warnings have the same probability of being faults, there are about 268 bugs among the 1378 unique warnings.

### *4.3.3 Symbolic approximation for large-scale OS software verification*

A recent software solution aimed at the verification of large-scale software systems is based on an approach called *symbolic approximation* [2, 3, 4, 5, 6]. Symbolic approximation mitigates the state space explosion problem of model checking techniques (see Section 4.2.1), by defining an approximate logical semantics of C programs. The approach models program states as logical descriptions of what is true at each node in the program execution graph. These descriptions are compared with a program specification in order to identify those situations in which the program may do something bad. The seminal works in this field are the ones by Peter Breuer et al. [2, 4, 6]. These works were aimed at providing a formal solution for detecting deadlock, double-free and other errors in the several million lines of code in the Linux kernel. The application of their formal analysis to the Linux code detected faults and errors never identified by thousands of developers who had reviewed the Linux code. The analyzer, written in C, is based on a general compositional program logic called NRBG (the acronym comes from "Normal", "Return", "Break", "Goto", which represent different types of control flow).[8] A program fragment analyzed in NRBG terms is considered as operating in three phases: *i) initial*, a condition $p$ holds at start of the execution of the fragment, *ii) during*, the fragment is executed, and *iii) final*, a condition $q$ holds at the end of the execution of the fragment. Based on this logic, individual program

[8] NRBG logic is also defined for the treatment of loops, conditional statements and other functions such as `lock`, `unlock`, and `sleep`. Component $G$ is used to represent the `goto` statement.

fragments are modeled in Hoare triplets. For instance, a normal exit from a program fragment is modeled as follows:

$$\texttt{p N(a;b) q} \;=\; \texttt{p N(a) r} \;\wedge\; \texttt{r N(b) q}$$

To exit normally with $q$, the program flows normally through fragment $a$, achieve an intermediate condition $r$, enter fragment $b$, and exit it normally.

A return exit ($R$) from a program fragment (i.e., the way code flows out of the parts of a routine through a "return" path) is modeled as follows:

$$\texttt{p R(a;b) q} \;=\; \texttt{p R(a) q} \;\vee\; \texttt{r R(b) q}$$

Here, two paths are possible: *(i)* return from program fragment $a$, or *(ii)* terminate $a$ normally, enter fragment $b$, and return from $b$.

A static analyzer tool allows the detail of the logic above to be specified by the user for the different program constructs and library function calls of C, giving rise to different logics for different problem analysis. The tool incorporates a just-in-time compiler for the program logic used in each analysis run. Logic specifications have the following form:

$$\texttt{ctx}\ \textit{precontext},\ \textit{precondition}\ ::\ \texttt{name(arguments)[subspecs]} = \textit{postconditions}$$

$$\textit{with ctx}\ \text{postcontext}$$

Here, *precondition* is the input condition for the code fragment, while *postconditions* is a triple of conditions applying to the standard exit paths $(N, R, B)$ for the program *name*. The *precontext* and *postcontext* contain the conditions pertaining to the additional exit paths provided by the `gotos` in the program. As an example, let us consider the forever while loop logic.

$$\texttt{ctx}\ e,\ p :: \texttt{while(1)[body]} \;=\; \textit{(b,r,F)}\ \texttt{with ctx}\ f$$

$$\texttt{where ctx}\ f,\ p :: \texttt{fix(body)} \;=\; \textit{(n,r,b)}\ \texttt{with ctx}\ f$$

A normal exit occurs when the loop body hits a break statement with the condition ($b$) holding. The normal loop body termination condition ($n$) and the associated loop body return ($r$), and the break ($b$) conditions are defined to be the fixpoint ('`fix(body)`') of the loop body, above the start condition ($p$). That is:

$$n \;\geq\; p \;\wedge\; n \;::\; \texttt{body} \;=\; \textit{(n,r,b)}$$

in the specification language terminology, or

$$p \;\Leftarrow\; n \;\wedge\; n\ \texttt{N(body)}\ n \;\wedge\; n\ \texttt{R(body)}\ r \;\wedge\; n\ \texttt{B(body) b}$$

in the abstract logic. A return exit happens when a return statement is executed from within the body of the while ($r$). The post-context $f$ contains the ways to exit the loop through a goto, as determined by the logic for the loop body.

Each logic specification for the analyser covers the full C language. Below, we briefly discuss how the solution by Breuer et al. [2, 4, 6] can be used to locate instances of a well-known fault in programming for Symmetric Multi-Processing (SMP) systems called "sleep under spinlock". A spinlock is a well-known classical SMP resource-locking mechanism in which a thread waiting to obtain a lock continuously checks if the lock is available or not (a situation called *busy waiting*). The waiting thread occupies the CPU entirely until the spinlock is released by another thread on another CPU in the SMP system. Suppose now that, in a 2-CPU SMP system, the thread holding the lock (turns off interrupts and) calls a sleepy function (one which may be interrupted and scheduled out of the CPU for some time) and then is scheduled out of its CPU. If two new threads end up busy-waiting for the same spinlock before the spinlock holder can be rescheduled, the system is deadlocked: the two threads occupy both CPUs entirely and interrupts are off. This vulnerability is critical since an adversary can exploit it to bring a *denial of service* attack [11].

To identify the calls to sleepy functions under spinlock, the logic specification in [6] provided a single unlock logic pattern for all the variants of spin-unlock calls in the Linux kernel. The logic decrements a spinlock total counter (see Figure 4.3). Similarly, the specification provided a single lock logic pattern for all the variants of spinlock calls of Linux.

$$\texttt{ctx}\ e,\ p :: \texttt{unlock(label l)} \ = \ (p[n+1/n],F,F)\ \texttt{with ctx}\ e$$
$$\texttt{ctx}\ e,\ p :: \texttt{lock(label l)} \ = \ (p[n-1/n],F,F)\ \texttt{with ctx}\ e$$

Fig. 4.3: Logic specification of `unlock` and `lock` function

A logical objective function was specified for this analysis which gauges the maximum upper limit of the spinlock counter at each node of the syntax tree. A set of trigger/action rules creates the sleepy call graph. When a new function is marked as sleepy, all callers of the function plus all callers of its aliases are marked as sleepy as well. The analysis creates a list with all the calls that may sleep under a spinlock.

| files checked | 1055 |
|---|---|
| alarms raised | 18 (5/1055 files) |
| false positives | 16/18 |
| real errors | 2/18 (2/1055 files) |
| time taken | about 24h (Intel P3M, 733 MHz, 256M RAM) |
| LoC | about 700K |

Table 4.3: Linux Kernel 2.6.3: testing for sleep under spinlock

The Linux kernel 2.6.3 was tested to find occurrences of sleep under spinlock (see Table 4.3 for more details); *1055* files of about 700K LOC were considered and *18* alarms raised. The real faults found amounted to *2* of the *18* alarms. The fact that many of the alarms were false positives should not necessarily be seen as a problem when the effort of analyzing false positives is considered in relation to the effort involved in finding faults manually.

## 4.4 Conclusion

Formal method verification is an important aspect of software security aimed at reducing risks caused by software faults and vulnerabilities. Model-based certification gathers a variety of formal and semi-formal techniques, dealing with verification of software systems' reliability, safety and security properties. In particular, model-based techniques provide formal proofs that an abstract model (e.g., a set of logic formulas, or a formal computational model such as a finite state automaton), representing a software system, holds a given property. As discussed in this chapter, much work has been done in the context of static analysis, formal methods and model checking. These solutions have gained a considerable boost in the recent years and are now suitable for software verification in critical security contexts and for verification of large scale software systems such as a Linux distribution. However, some drawbacks still need to be addressed. Firstly, these techniques produce a non-negligible rate of false positives, which require a considerable effort for understanding which warnings correspond to real faults. Secondly, model-based techniques do not support configuration evolution: it is not guaranteed that a certification obtained for one configuration will still holds when the configuration changes.

## References

1. D. Avots, M. Dalton, V. Livshits, and M. Lam. Improving software security with a c pointer analysis. In *Proc. of the 27th International Conference on Software Engineering (ICSE 2005)*, St. Louis, Missouri, USA, May 2005.
2. P.T. Breuer and S. Pickin. Checking for deadlock, double-free and other abuses in the linux kernel source code. In *Proc. of the Workshop on Computational Science in Software Engineering*, Reading, UK, May 2006.
3. P.T. Breuer and S. Pickin. One million (loc) and counting: static analysis for errors and vulnerabilites in the linux kernel source code. In *Proc. of the Reliable Software Technologies-Ada-Europe 2006*, Porto, Portugal, June 2006.
4. P.T. Breuer and S. Pickin. Symbolic approximation: an approach to verification in the large. *Innovations in Systems and Software Engineering*, 2(3–4):147–163, December 2006.
5. P.T. Breuer and S. Pickin. Verification in the light and large: Large-scale verification for fast-moving open source c projects. In *Proc. of the 31st Annual*

*IEEE/NASA Software Engineering Workshop*, Baltimore, MD, USA, March 2007.
6. P.T. Breuer, S. Pickin, and M. Larrondo Petrie. Detecting deadlock, double-free and other abuses in a million lines of linux kernel source. In *Proc. of the 30th Annual IEEE/NASA Software Engineering Workshop*, Columbia, MD, USA, April 2006.
7. *C - Approved standards.* `http://www.open-std.org/jtc1/sc22/wg14/www/standards`.
8. *Carnegie Mellon University's Computer Emergency Response Team.* `http://www.cert.org/`.
9. H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proc. of the 9th ACM Computer and Communications Security Conference (ACM CCS 2002)*, Washington, DC, USA, November 2002.
10. K. Chen and D. Wagner. Large-scale analysis of format string vulnerabilities in debian linux. In *Proc. of the 2007 Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, San Diego, California, USA, June 2007.
11. W. Cheswick and S. Bellovin. *Firewalls and Internet Security; Repelling the Wily Hacker*. Addison Wesley, 1994.
12. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*, December 1999. MIT Press.
13. N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *Proc. of the Fourth European PKI Workshop: Theory and Practice (EUROPKI 2007)*, Mallorca, Balearic Islands, Spain, June 2007.
14. N. Dragoni, F. Massacci, C. Schaefer, T. Walter, , and E. Vetillard. A security-by-contracts architecture for pervasive services. In *Proc. of the 3rd International Workshop on Security, Privacy and Trust in Pervasive and Ubiquitous Computing*, Istanbul, Turkey, July 2007.
15. D.A. Duffy. *Principles of Automated Theorem Proving*. John Wiley & Sons, 1991.
16. D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January-February 2002.
17. U. Glasser, R. Gotzhein, and A. Prinz. Formal semantics of sdl-2000: Status and perspectives. *Computer Networks*, 42(3):343–358, June 2003.
18. C. A. R. Hoare and C. B. Jones (eds.). *Essays in computing science*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
19. R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proc. of the 13th conference on USENIX Security Symposium*, San Diego, CA, USA, August 2004.
20. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.
21. W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.
22. G. Necula. Proof-carrying code. In *Proc. of the ACM Principles of Programming Languages (POPL 1997)*, Paris, France, January 1997.
23. G.C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proc. of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI 1998)*, Montreal, Canada, May 1998.
24. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, 1999.
25. M.F. Ringenburg and D. Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *Proc. of the 12th ACM conference on Computer and Communications Security (CCS 2005)*, Alexandria, VA, USA, November 2005.

26. B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West. Model checking an entire linux distribution for security violations. In *Proc. of the 21st Annual Computer Security Applications Conference (ACSAC 2005)*, Tucson, Arizona, USA, December 2005.
27. R. Sekar, C.R. Ramakrishnan, I.V. Ramakrishnan, and S.A. Smolka. Model-carrying code (mcc): A new paradigm for mobile-code security. In *Proc. of the New Security Paradigms Workshop (NSPW 2001)*, New Mexico, USA, September 2001.
28. R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D.C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, New York, USA, October 2003.
29. O. Shivers. Control-flow analysis in scheme. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1988)*, Atlanta, Georgia, USA, June 1988.
30. R.F. Staerk, J. Schmid, and E. Boerger. Java and the java virtual machine: Definition, verification, validation. *Springer-Verlag*, 2001.
31. D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. of the Network and Distributed Systems Security Symposium (NDSS 2000)*, pages 3–17, San Diego, California, USA, February 2000.