Chapter 4

Verification Techniques

Wherein existing methods for building secure systems are examined and found wanting.

1. Introduction

In 1987 Fred Brooks produced his seminal and oft-quoted paper "No Silver Bullet: Essence and Accidents of Software Engineering" [1]. Probably the single most important point made in this article is one which doesn't directly touch on the field of computer software at all, but which comes from the field of medicine. Before modern medicine existed, illness and disease were believed to be the fault of evil spirits, angry gods, demons, and all manner of other causes. If it were possible to find some magic cure which would keep the demons at bay then a great many medical problems could be solved. Scientific research into the real reasons for illness and disease destroyed these hopes of magical cures. There is no single, universal cure since there's no single problem, and each new problem (or even variation of an existing problem) needs to be addressed via a problem-specific solution.

When the message in the article is reduced to a simple catchphrase, its full meaning often becomes lost: There really is no silver bullet, no rubber chicken which can be waved over a system to make it secure. This chapter examines some of the attempts which have been made to find (or decree) a silver bullet and looks at some of the problems which accompany them. The next chapter will then look at alternative approaches towards building secure systems.

As did an earlier paper on this topic which found that "proclaiming that the gods have clay feet or that the emperor is naked […] are never popular sentiments" [2] (another paper which pointed out problems in a related area found that it had attracted "an unusually large number of anonymous reviewers" [3]), this chapter provides a somewhat higher number of references than usual in order to substantiate various points made in the text and to provide leads for further study.

2. Formal Security Verification

The definition and the promise of formal methods is that they provide a means to "allow the specification, development, and verification of a computer system using a rigorous mathematical notation. Using a formal specification language to specify a system allows its consistency, completeness, and correctness to be assessed in a systematic fashion" [4]. The standard approach towards trying to achieve this goal for security-relevant systems is through the use of formal program verification techniques which make use of mathematical logic to try to prove the correctness of a piece of software or hardware. There are two main classes of tools used in this task, proof checkers (sometimes called theorem provers) which apply laws from logic and set theory to a set of assumptions until a desired goal is reached, and model checkers, which enumerate all the possible states a system can be in and check each state against rules and conditions specified by the user [5][6]. In terms of reporting problems, proof checkers (which work with symbolic logic) will report which step in a particular proof is invalid, while model checkers (which work with FSM's) will report the steps which lead to an invalid state.

Proof checkers are named thus because they don't generate the entire proof themselves but only aid the user in constructing a proof from an algebraic specification, performing many of the tedious portions of the proving process automatically. This means that the user must still know how to perform the proof themselves, there're merely assisted in this process by the proof checker. This requires some level of skill from the user, not only because they need to know enough mathematics to construct the proof and drive the checker, but also because they need to be able to recognise instances where the checker is being sent down the wrong path, in which case the checker can't complete the proof. The user can't distinguish (from the actions of the checker) the case of a proof which is still in the process of being completed, and a proof which can never be completed, for example because it's based on an invalid assumption. This can make proof checkers somewhat frustrating to use.

Another problem which arises with proof checking is with the specifications themselves. Algebraic specifications work with a predefined type of abstraction of the underlying system in which functions are defined indirectly in terms of their interaction with other functions (that is, the functions are transformational rewrite statements). Because of this they can require a fair amount of mental gymnastics by anyone working with them in order to understand them. A slightly different specification approach, the abstract model approach, defines functions in terms of an underlying abstraction (lists, arrays, and sets being some examples) which is selected by the user, as well as a set of preconditions and postconditions for each function being specified. This has the advantage that it's rather easier to work with than an algebraic specification

because it's closer to the way programmers think, but has the corresponding disadvantage that it strongly influences the final implementation towards using the same data representation as the one used in the abstract specification.

In contrast to proof checkers, model checkers operate on a particular model of a system (usually a finite-state machine), enumerating each state the system can enter and checking it against certain constraints (can the state be reached, can the state be exited once reached, and so on). A state machine is defined in terms of two things, states which have V-functions (value returning functions) which provide the details of the state, and transitions which have O-functions (observation functions) which define the transitions [7][8]. Other methodologies use the terms "state" or "variable" for V-functions and "transform" for O-functions, an exception to this is FIPS 140 which reverses the standard terminology so that "state" corresponds to the execution of a piece of code and another term has to be invented to describe what is being transformed by a "state".

An O-function works by taking a V-function and changing the details it will return about the state. In verification systems such as InaJo (which uses the "transform" terminology) the O-functions are then used to provide input and output assertions for a verification condition generator. Because the number of states grows exponentially with the complexity of the system, model checkers tend to be incredibly resource-hungry. One solution to this problem is to fall back on the use of a proof checker when the model checker can't find any problem because it has run out of memory, or to use two different, complementary formal methods in the hope that one will cover any blind spots present in the other [9]. Failing the availability of this safety device, it's unsafe to draw any real conclusions since the model checker may have found problems had it been able to search more of the state space [10]. Proof checkers have an analogous problem in that they can't detect all possible inconsistent ways to write a specification, so that with a little effort and ingenuity it's possible to persuade the system prove a false theorem [11].

An alternative approach is to apply further amounts of abstraction to try to manage the state explosion, in one example a model with a state space of 2^{87} states which would have taken 10^{12} years to search was further abstracted by partitioning the system into equivalence classes, separating the validation of portions which were assumed to be independent from one another so that they could be validated in isolation, and removing information from the model which was held to be non-germane to the validation, resulting in six validations which checked around 100,000 states each [12]. This type of manipulation of the problem domain has the disadvantage that the correspondence between the new abstraction and the original specification is lost, leading to the possible introduction of errors in the specification-to-new-abstraction mapping phase. A second potential problem area is that some of the techniques being applied (for example validating different portions in isolation) may miss faults if it turns out that there were actually interactions present between some of the portions. An example of this occurred during the analysis of the Viper ALU, which was analysed as a set of 8 4-bit slices because viewing it as a single 32-bit unit would have made analysis intractable. Since a proof used at another level of the attempted verification of the Viper CPU assumed a complete 32-bit ALU rather than a collection of 4-bit slices, no firm conclusion could be drawn as to whether one corresponded to the other [13] (the controversy over exactly what was evaluated in Viper and what constituted a "proven correct design" eventually resulted in the demise of the company which was to exploit it commercially in a barrage of finger-pointing and legal action [14][15]).

All of these approaches suffer from something called the hidden function problem, which is the inability of the system to retain any state from previous invocations. The solution to this problem is to use hidden functions which aren't directly visible to the user, but which can retain state information from previous invocations. These hidden functions manage information which isn't part of the visible behaviour of the abstract machine being specified, but which are required for its operation. Algebraic specifications in particular, whose functions are true functions in the mathematical sense that they can have no side-effects, are plagued by the need to use hidden functions. In some cases the specification can contain more hidden functions (that is, artifacts of the specification language) than actual functions which specify the behaviour of the system being modelled [16].

2.1. Formal Security Model Verification

The use of formal methods for security verification arose from theoretical work performed in the 1970's, which was followed by some experimental tools in the late 1970's and early 1980's. The belief then,

supported by the crusading efforts of a number of formal methods advocates, was that it would only be a matter of time before the use of formal methods in industry was widespread, and that at some point it would be possible to extend formal-methods-based verification techniques all the way down to the code level. It was this background which lead to the emphasis on formal methods in the Orange Book.

The formal security model which is being verified is typically based on a finite state machine model of the system, which has an initial state which is shown (or at least decreed) to be secure, and a number of successor states which can be reached from the initial state which should also be secure. One representation of the security model for such a system consists of a collection of mathematical expressions which, when proven true, verify that the state transitions preserve the initial secure state [17].

In order to perform this verification, the system's security policy (in Orange Book terms its top-level specification, TLS) must be rephrased as a formal top-level specification (FTLS) containing the security policy expressed in a mathematically verifiable form. Once the FTLS has been proven, it (or more usually the TLS, since the FTLS will be incomprehensible to anyone but its authors) is rephrased as a sequence of progressively lower-level specifications until a level is reached at which implementation becomes practical (sometimes the FTLS itself needs to be progressively decomposed in order to make analysis possible [18]). The translation from lower-level formal specification to code must then be verified in some manner, traditionally through the use of a verification system such as Gypsy or InaJo which have been designed for this stage of the process. In addition to an FTLS, the Orange Book also allows for a descriptive TLS (DTLS) which is written in plain English and which gets around the problem that no-one who wasn't involved in producing it can understand the FTLS. The Orange Book requires the use of a DTLS for classes B2 and higher and an FTLS for class A1 (B1 only requires an informal model of the security policy and was added at a late stage in the Orange Book process because it was felt that the jump from C2 to B2, then known as levels 2 and 3 [19], was too large).

After the FTLS is verified, the verification process generally stops. Specifically, there is no attempt to show that the code being executed actually corresponds to the high-level language source code it is built from, although there have been limited, mostly experimental attempts to address this problem (although at least one effort, the LOCK project, attempted to go beyond the FTLS with a formal interface level specification (FILS) [20]). Some of these included attempts to build trusted compilers using correctness-preserving transformations [21], the use of a translator from an implementation in Modula-1 (which the verification was applied to) to C (which wasn't verified) from which it could finally be compiled for the target platform [22], the use of a lambda-calculus-based functional language which is compiled into code for an experimental, special-purpose computer [23], the use of low-level instruction transformations for restricted virtual machines (one a stack machine, the other with a PDP-11 like instruction set) [24], the use of subset of the Intel 8080 instruction set (in work performed in 1988) [25], a minimal subset of C which doesn't have loops, function calls, or pointers [26],a template-like translation of a description of a real-time control system into C (with occasional help from a human) [27], and a version of Ada modified to remove problem areas such as dynamic memory allocation and recursion [28]. All of these efforts either required making a leap of faith to go from verified code to a real-world system, or required the use of an artificially restricted system in order to function (the Newspeak approach, create a language in which it's impossible to think bad thoughts), indicating that formal verification down to the binary code level is unlikely to be practical in any generally accepted formal methods sense.

3. Problems with Formal Verification

Formal methods have been described as "an example of a revolutionary technique that has gained widespread appeal without rigorous experimentation" [29]. Like many software engineering techniques covered in the next section, much work on formal methods is analytical advocacy research (characterised as "conceive an idea, analyse the idea, advocate the idea" [30]) in which the authors describe a technique in some detail, discuss its potential benefits, and recommend that the concept be transferred into practice. Empirical studies of the results of applying these methods, however, have had some difficulty in finding any correlation between their use and any gains in software quality [31], with no hard evidence available that the use of formal methods can deliver reliability more cost-effectively than traditional structured methods with enhanced testing [32]. Even in places where there has been a concerted push to apply formal methods, penetration has been minimal and the value of their use has been difficult to establish, especially where high quality can be achieved through other methods [33].

This section will examine some of the reasons why formal methods have failed to provide the silver bullet which they initially seemed to promise.

3.1. Problems with Tools and Scalability

The tools used to support formal methods arose from an academic research environment characterised by a small number of highly skilled users (usually the developers of the tools) and by extension an environment in which it didn't really matter if the tools weren't quite production grade, difficult to use, slow, or extremely resource-hungry — they're only research prototypes after all. The experimental background of the tools used often lead to a collection of poorly-integrated components built by different researchers, with specification languages which varied over time and contained overlapping and unclear features contributed by various sources, or which differed depending on which researcher's verification tool was being employed. In systems like HDM this lead to assessments by independent observers that "at present an outsider can not use HDM to design, implement, and verify a program from beginning to end" [34]. In addition the tools were tested on small problems (usually referred to somewhat disparagingly as "toy problems") which were targeted more at exercising the tools than exercising the problem. This section covers some of the issues which arose because of this.

Both of the formal methods endorsed for use with the Orange Book, Ina Jo/Ina Mod (collectively known as the Formal Development Methodology, FDM) [35][36][37] and Gypsy (as part of the Gypsy Verification Environment, GVE) [38][39][40] date from the 1970's and have seen little real development since then. Both are interactive environments, which isn't a feature but a polite way of indicating that they require a lot of tedious user intervention and manual effort in order to function. Furthermore, not only do they require extensive assistance from the user, but the difficult nature of the tools and task requires expert users to work with it, and once they're finished it requires another set of expert users to verify and evaluate the results [41][42][43][44].

Another early effort, the Boyer-Moore theorem prover, has been described as "like trying to make a string go in a certain direction by pushing it […] Proof becomes a challenge: to beat the machine at its own game (the designers and some others have chalked up some very high scores though)" [45]. Attempts to use the BM prover in practice lead to the observation that "the amount of effort required to verify the system was very large. The tools were often very slow, difficult to use, and unable to completely process a complex specification. There were many areas where tedious hand analysis had to be used" [46].

Many of the tools originate from a research environment and are of a decidedly experimental nature which contributes to the difficulty in using them. Several kernel verifications have had to be abandoned (or at least restarted so that they could be approached in a different manner) because the tools being used for the verification weren't quite up to handling the problem. This wasn't helped by the fact that they were often built using whatever other tools happened to be available or handy rather than the tools which would typically be found in a production environment, for example the Gypsy compiler which was used in some kernel validations was originally implemented as a cross-compiler into Bliss, which was only available on a limited number of DEC platforms, making it extremely difficult even to get access to the right tools for the job.

Working at a very high level of abstraction can produce a (hopefully) correct and (more or less) verifiable specification which then needs to be completely rewritten by system developers in order to make it implementable [47]. Some researchers have suggested performing the implementation directly in a specification language, however this is equivalent to a standard implementation created with an even-higher-level-language. Although this may eliminate many specification bugs, what will be left is a class of even tougher specification bugs which require an even higher-level specification system to expose [48]. In addition in order to make it workable the specification language will typically have been modified in order to remove features which are considered unsafe, but the downside of removing unsafe features such as pointer variables is that all data structures which are manipulated by a routine will need to be passed in and out explicitly as parameters, resulting in huge parameter lists for each routine and a high overhead from moving all the data around.

Another approach which has been suggested is to automatically verify (in the "formal proof of correctness" sense) the specification against the implementation as it is compiled [49]. This has the disadvantage that is can't be done using any known technology.

3.2. Formal Methods as a Swiss Army Chainsaw

Formal methods have in the past been touted as the ultimate cure for all security software assurance problems, a "panacea with unbounded applicability and potency" [50]. It has been pointed out that while it is recognised that some formal methods are more suited for use in certain areas than others, "these qualities are no more than discreetly acknowledged minimal departures from a default presumption of universal applicability. No mainstream formal method carries a maker's disclaimer that it is useful only for a small and narrowly defined class of problem" [51]. One of the reasons for their perceived failure to perform as required is the desire to apply them as a universal elixir, a silver bullet capable of slaying the security bugbear in all its forms.

Related to this problem was the desire to build a complete, general-purpose secure operating system kernel, something which is now known to be infeasible if efficiency and time/budgetary constraints are also present. The reason for this is because a general-purpose kernel is required to support any number of functions which are extremely difficult to analyse from a security point of view, for example various types of I/O devices, DMA, and interrupts all cause severe headaches for security architects, to the point where they have been disallowed in some designs because they can't be managed in a secure manner. The extra complexity of handling all of the required generality adds more overhead to the system, which drags performance down. One almost universal byline of 1980's papers on high-security kernels was some sort of lament about their lack of performance [52][53], this has also been blamed on the close correspondence between the TLS and the actual implementation since the formal specification system by its very nature is typically incapable of describing any sort of efficient implementation (all the features which make an implementation efficient also make it dangerous and/or unverifiable using formal methods). Another reason for the poor performance was that kernel design and implementation was usually driver by the verifiers, so that if the tools couldn't manage some aspect of the kernel the response was to require the kernel to be redesigned to fit what the tools could do. The performance problem was finally solved with the VAX VMM security kernel which was driven by performance rather than verification considerations (in contrast to the "A1 at any cost" of earlier efforts, the VAX VMM kernel philosophy was "A1 if possible, B3 if it would impact performance too much"), so that if the tools failed the response was to change the tools rather than the kernel [54]. A similar approach was later taken in other kernels such as MASK, where efficiency measures in the C implementation were migrated back into the formal specification [55].

The complexity of a general-purpose kernel puts a great burden on the formal verification tools. As the previous section indicated, these are often already fragile enough when faced with toy problems without having to try to cope with extremely complex, general-purpose security models and mechanisms. One attempt to mitigate this problem is by choosing a subset of the overall problem and applying formal methods only to this subset, avoiding the high cost and effort required to apply the formal methods. In security-critical systems this subset is the security kernel, but even this (relatively) small subset has proven to be very difficult to manage, and applying formal methods to even reasonable-sized systems appears to be infeasible [56].

Another factor which complicates the use of formal methods is that the mathematical methods available to software engineers are often very difficult to use (much more so than the mathematics employed in other areas of engineering) and plagued by notation which is cumbersome and hard to read and understand, with substantial effort being required to present the ideas in a manner which is understandable to non-cognoscenti [57]. As an example of the type of problems this can lead to, a medical instruments project at HP ran into problems because no-one outside the project group was willing or able to review the formal specifications [33].

One study of the comprehensibility of formal specifications found that the most common complaint among users was that the specification was incomprehensible. When asked to provide an absolute comprehensibility rating of a Z specification, subjects rated the specification at either "hard" or "incomprehensible" *after a week of intensive training in using the language* [58]. Another survey, again related to the comprehensibility of Z, found that even subjects trained in discrete mathematics who had completed a course in formal methods found it very difficult to understand any of a 20-line snippet from a Z specification, with nearly a third of the test group being unable to answer any questions relating to the specification, which they found incomprehensible [59].

Concerns about the write-only nature of many specification languages were echoed by many other groups who had tried to apply formal methods in real life. This problem arises from the fact that understandability appears to be inversely proportional to the level of complexity and formality present [60]. One survey of techniques which used as one criterion the understandability of the resulting document rated the language surveyed, PAISLey [61], as the least understandable of all the techniques covered (and PAISLey is downright comprehensible compared to many of its peers) [62]. This legibility problem isn't restricted just to tools which support program proving, for example the specification language GIST was designed to allow the specification of the states in a system and its behaviour based on stimulus-response rules [63][64], but resulted in specifications which were so hard to read that a paraphraser had to be written to translate them back into something which could be understood. In another experiment which compared the use of the formal specification language OBJ with the non-formal specification language PDL and the even less formal specification language English, one unanimous piece of feedback from users was their dislike of the formal specification language's syntax, even though it had been post-processed with a text editor in order to make it more palatable [65].

The results obtained from real-world kernel verifications are even more depressing. One paper which examined the possibility of creating a "beyond A1" system contained figures of 2-6 lines of verified code being produced per day per highly trained verification specialist, of which the entire worldwide community was estimated at around 200 individuals, of which only a small fraction are actually available for this kind of work. To put this into perspective, the verification of a portion of the system mentioned in the paper required 3 pages of specification and 200-odd pages of proof logs [66]. Another paper was even more pessimistic: "The national technology base for A1-level systems is essentially non-existent. There do not appear to be even 20 people in the world [in 1985] that have undertaken the essential steps of building an A1 system" [67]. A third makes the rather dry observation that "in order to get a system with excellent system integrity, you must ensure that it is designed and built by geniuses. Geniuses are in short supply" [68].

3.3. What Happens when the Chainsaw Sticks

A previous chapter pointed out that security kernels are generally accompanied by a collection of camp followers in the guise of trusted processes, privileged processes which can bypass the system's security policy in order to perform their intended task. This presents something of a problem in terms of formal verification since it's not really possible to verify the security of a system once these trusted processes, which exist solely to bypass the system's security, are taken into account. The workaround to this problem is to provide an informal (in the sense of it being DTLS-style rather than FTLS-style) argument capable of convincing the evaluators that the trusted process isn't really a problem, which in its defence at least forces the developer to think about the problem before they leap in and violate the formal model.

In some cases however it doesn't even take a trusted process to introduce problems into the proof process. During the SCOMP validation, the theorem prover failed to prove several formulae, which then had to be justified using English/informal explanations [69]. In another verification it was found that a significant proportion of the system's assurance argument wasn't amenable to formal specification because the specification system model was based on CSP, which wasn't capable of expressing some of the characteristics of the system and required the use of informal specification and verification methods [70] (this is a problem which seems to be common among other CSP-based models [71]). Other problems can occur when a proof has to be manually augmented with "self-evident" axioms which the prover isn't capable of deriving itself since "self-evident" truths sometimes turn out to be false and end up misguiding the prover, a problem which is explored in more detail further on.

Another problem with formal methods is the lack of allowance for feature creep (although much of this is likely to be outside the TCB). The average project goes through roughly 25% change between the point at which the requirements are complete and the first release [72], which causes severe problems for formal methods which assume that everything can be specified in advance, or even if they don't explicitly assume it at least require it, since once the formal proof process has begun a single change can entail restarting it from scratch [73][74]. This is particularly problematic in cases where multiple layers of abstraction are required to go from FTLS to implementation, since a change in any of the layers can result in a ripple effect as changes propagate up and down the hierarchy, requiring a lot of tedious re-proving and analysis which may in turn result in further changes being made. If a change manages to propagate its way into a formally proven section of the design, either the entire proof must be redone or the affected portions of the proof must

somehow be undone so that they can be re-proven, with the hope that some portion which hasn't actually been re-proven yet isn't mistakenly regarded as still being valid.

Even if the formal specification can somehow be frozen so that the implementation is based on the same specification as the one which is evaluated, the need to use trusted processes, which are almost required in order to make a system based on the Bell-LaPadula model workable [75] results in a system where it's difficult to determine the exact nature of the security rules which the system is enforcing, since what's actually being enforced isn't the same as the axioms present in the formal security model. Because the actual policy being enforced differs from the Bell-LaPadula axioms, any formal proof that the system provides certain properties (for example that it maintains a secure state) can only apply to one portion of the system rather than the system as a whole.

The assumption that a particular system will be used exactly in the manner and situation it was designed for is rather unrealistic, especially since history has demonstrated that systems will always end up in unanticipated environments, an example being the interconnection of formerly isolated systems into a single heterogenous environment [76]. The inability of formal methods to adapt to such changes means that either the systems are run in a manner which they were never evaluated for, or the evaluation is subject to increasingly tortuous "interpretations" in order to try to adapt it for each new environment which crops up (it is for this reason that the Orange Book has also been referred to as the Orange Bible, with interpretations being "ministerial treatments derived from the Orange Bible" [77]).

Once a program exists, there is an irresistible pressure to modify it, either to correct real or perceived defects or because of feature creep. This maintenance is usually done with far less care than was used when the program was originally created, and once even a single change is made to the system all bets are off [78][79]. What is running now isn't what was formally specified or what was verified. This type of problem is almost inevitable because user requirements are extremely volatile, which means the formal specification technique can't work if it assumes that the user knows exactly what they want in advance. Real-world surveys have shown that this isn't the case, with specification being an incremental, iterative process and most development being maintenance rather than so-called greenfields (starting from scratch) development [80][81]. This has lead to a constant revision of software engineering methodologies from the initial waterfall model through to the spiral model and finally "development on Internet time" or "extreme programming" (this last stage is still being worked on so it doesn't have a generic label yet).

In the case of software designed for the mass market this problem is even worse, since with custom software development there is at least some interaction with the eventual user and/or customer while with mass-market software the first chance for customer feedback on the design occurs when they install a buggy and unstable beta release on their system, or even later when they buy the finished product (possibly still in the buggy and unstable state). This is made more difficult by the fact that the developers of the applications often lack relevant domain knowledge, for example someone implementing a portion of a word processor probably hasn't had any formal training in typesetting or page layout requirements, resulting in a product which fairly promptly needs to be adapted to meet the user's requirements in a *x*.1 and *x*.2 release update. As a result of this style of development, there is a strong need to handle late changes to the design and to allow for customisation and other adaptations to the implementation late in the development cycle [82].

Unfortunately most formal methods never made it past the waterfall model, with no flexibility or provision for change later on in the development process. Although this issue isn't generally addressed in publications describing the results of applying formal methods to security system evaluations, one paper which did touch on this issue reported that the planned waterfall-model development became instead very iterative, with many update cycles being necessary in order to nail down the precise details of the model [83]. Another paper commented that "even very simple models can entail significant costs of time and effort over the verification phase. The effect of incrementally adding new modules to a stable (ie proven) body of modules introduces the obligation to integrate all new variables and data structures into the old module proofs, and thus multiply their length. As more models are integrated in this way, the effect appears to be significantly non-linear" [71]. This iterative development process mirrors real-world engineering experience in which a new product (for example a car or appliance) is almost never a greenfields development, but is very similar to its predecessors and shares the same structuring of problem and solution. The traditional engineer doesn't start with a clean slate (or monitor) but instead bases their work on successful designs which have evolved over many product generations through the contribution of a community of other engineers.

Although there have been attempts at allowing for a limited amount of design change and maintenance these attempts haven't been very successful, for example the Orange Book Rating Maintenance Program (RAMP) has been described as leading to "a plethora of paperwork, checking, bureaucracy and mistrust" being imposed on vendors who participate in it [84]. Other approaches to this problem include TCB subsetting, which involve hanging a bag on the side of the existing TCB rather than changing it in order to avoid having to go through the evaluation process again [85], and trying to combine bits and pieces evaluated at various levels for which some sort of composite rating can then be claimed [86][87], a variation of the Chinese menu approach mentioned in an earlier chapter which is bound to cause uncertainty and confusion for all involved. A more interesting proposed approach to the problem involves having new modules evaluated under ITSEC or the Common Criteria and digitally signed by the evaluators, whereupon a kernel could grant them certain privileges based on the evaluation level [88]. This approach has the downside that it requires that a high degree of confidence be placed in the efficacy of the evaluation and the evaluators.

3.4. What is being Verified/Proven?

Since current verification techniques can't generally reach down any further than the high-level specification (that is, all they can do is verify consistency between some formal model and the design), they result in large amounts of time and energy being poured into producing a design specification which by its very nature is void of any implementation detail [89]. Formal methods typically view a system as a set of operations on a state or a collection of communicating sequential processes, offering a designer almost no guidance in how to approach a particular problem. This means that a verified design doesn't ensure that the completed system is error-free or functioning correctly, merely that it meets the requirements set out in some user-defined model. Because of this it isn't terribly meaningful to claim that an implementation is "correct" (in terms of satisfying some requirement) without including as a rider the requirement which it satisfies. In its purest technical sense, correct doesn't mean "good" or "useful" or "appropriate" or any other similar approbatory adjective, but merely "consistent with its specification". As with ISO 9000, it's possible to produce an arbitrarily bad product but still claim it's correct, since it complies with the paperwork.

Determining the appropriate point at which to stop the modelling process can be difficult because in a real system information can be accessed in so many ways that don't obey the formal security model that the result is a system which can contain many apparent exceptions to the formal security model. Such exceptions can occur due to any number of hardware or software mechanisms, for example DMA or I/O device access or at a more subtle level interrupts being used as a subliminal channel all add extra complexity to a security model if an attempt is made to express the security-relevant property of each operation which can occur in a system. The choice then is either to abstract the system to a level which makes analysis tractable, or to try to model every relevant property and end up with an unworkably complex model.

Another problem with formal proofs is that although they can show with some certainty (that is, provided there are no mistakes made in any calculations and/or the supporting tools are bug-free) that a given specification is "correct", what can't be shown is that the assumptions that are being made in the specification are a correct description of the actual physical system. If the code which is supposed to implement the formal specification doesn't quite conform to it or the compiler can't produce an executable which quite matches what was intended in the code then no amount of formal proving will be able to guarantee the execution behaviour of the code. Just as Newton's laws don't work well close to the speed of light or for objects that are not in inertial frames of reference, so formal proofs have problems with issues such as arithmetic overflow and underflow (the most famous example of this being the Ariane 5 disaster), division by zero, and various language and compiler bugs and quirks [90]. Even if the compiler is formally verified, this only moves the problem to a lower level. No matter how thoroughly an application is formally verified, at some point the explanations must come to an end and users must assume that the physical system satisfies the axioms used in the proof [91].

Although the design documents for security systems are usually not made public, one of the few which has been provides a good example of how easily errors can creep into a specification. In this case the specification for the control software for a smart card as published at a security conference was presented in three locations: As a sidebar to the main text in plain English, in the main text in a formal notation with English annotations to explain what was happening, and again in separate paragraphs with more English text to provide further detail. All three versions are different [92]. In addition since the software exists as a full FTLS in InaJo, a DTLS, and finally a concrete implementation, there are likely to be at least six varying

descriptions of what the card does, of which at least three differ (the full FTLS, DTLS, and software were never published, so it's not possible to determine whether they correspond to each other).

A similar problem was found during an attempt to formally verify the SET protocol. This protocol is specified in three separate parts, a business description targeted at managers ("Book 1"), a programmer's guide ("Book 2"), and a so-called formal protocol definition ("Book 3") which in this case describes the SET protocol in ASN.1 (a data format description language) rather than in an FTLS-style language (in other words it's more a formal description of the message format than of the protocol semantics). Not only are all three books inconsistent, but the "formal definition" in Book 3 contains ambiguities which need to be explained with textual annotations, occasionally from Book 1 and 2 (in defence of the SET specification, it must be mentioned that the problem of imprecise and inconsistent specifications is endemic to Internet security protocols, a problem which has been pointed out by other authors [93]. SET probably represents one of the better-specified of these protocols). In response to one particular statement which pertains to an ambiguity in Book 3 but which itself appears in Book 2, the exasperated evaluators commented that "It is difficult to believe that such a statement could be part of the specification of a security protocol" [94]. The evaluators eventually had to assemble the protocol details from all three books, making various common-sense assumptions and simplifications in cases where it wasn't possible to determine what was intended, or where the books contradicted each other. Eventually they were able to verify some portions of their interpretation of a subset of the SET protocol, although beyond discovering a few potential problem areas it's not certain how valuable the overall results of the effort really are.

In another example of problems with the specification used to drive the verification effort, a system targeted at Orange Book A1 contained a flaw which would allow users to violate the Bell-LaPadula `*`-property, even though the verification process for the system had been completed without discovering the error. This error was present in the implementation because it faithfully followed the specification which contained the same error, although it was corrected when discovered by the implementors using a two-line fix. The same specification contained further errors in sections which had no meaning other than to guide the verification tool, one of which misguided it to the point of failing to find the flaw [95]. In another example, the AUTODIN II specification contained an internal inconsistency in the FTLS which would have allowed any arbitrary formula to be proved as a theorem [96]. In contrast when a C standards committee published an incorrect specification for the `snprintf()` function most of the implementors made sure that the function behaved correctly rather than behaving as per the specification, so the implementation was correct (in the sense of doing the right thing) only because it explicitly *didn't* comply with the specification [97]. The issue of correctness vs. correctness is examined in more detail in the next chapter.

In some cases the assumptions which underlie a security system or protocol can alter the security claims which can be made about it. One case which garnered some attention due to the debate it generated among different groups who examined it was the security proofs of the Needham-Schroeder public key protocol, which was first proven secure using BAN logic [98], then found to have a flaw under a slightly different set of assumptions using the FDR model checker [99][100], and finally found to have further problems when subject to analysis by the NRL protocol analyser [101]. It can be argued that both of the initial analyses were correct, since the first analysis assumed that principals wouldn't divulge secrets while the flaw found in the second analysis relied on the fact that if a principal revealed a secret nonce then an attacker could (depending on various other protocol details) impersonate one or either of the two principals. The third analysis used slightly different assumptions again, and found problems in cases such as one where a participant is communicating with itself (there were also other problems which were found using the NRL protocol analyser which weren't been found by the FDR model checker for slightly different reasons which are explained further on). The differences arose in part because BAN logic contains a protocol idealisation step in which messages exchanged as part of the protocol are transformed into formulae about the messages so that inferences can be made within the logic, which requires assigning certain meanings to the messages which may not be present in the actual protocol. Such an idealisation step isn't present in the FDR or NRL analysis. In addition the BAN logic analysis contained a built-in assumption that all principals are honest, while the others didn't. The situation was summed up in a later analysis of the various attacks with the observation that "The model has to describe the behaviour of principals. Protocol goals are often formalized as if agents could engage in a protocol run only by following the rules of the protocol" [102].

Even when the formal specification provides an accurate model of the physical system, real-world experience has shown that great care has to be devoted to ensuring that what is being proven is what was intended to be

proven [103][104]. In other words even if the formal specification accurately modelled the actual system, was there some way to breach security which wasn't covered by the proof? An example of this was in the SCOMP verification, where discrepancies were found during the specification-to-implementation mapping process which had been missed by the formal verification tools because they weren't addressed in the FTLS. Another system was verified to be correct and then "compiled, tested, and run with remarkably few errors discovered subsequent to the verification" [66].

These sorts of problems can arise from factors such as narrowing of the specification caused by restrictions in the specification language (for example the fact that the specification language was much more restrictive than the implementation language), widening of the specification caused by the nature of the specification language (for example some special-case conditions might be assigned a magic meta-value in the specification language which is treated quite differently from, say, an empty string or a null pointer in the implementation language), or an inability to accurately express the semantics of the specification in the implementation language. This type of specification modification arises because the specification writers aren't omnipotent and can't take into account every eventuality which will arise. As a result, an implementation of a specification doesn't just implement it, it alters it in order to fit real-world constraints which weren't foreseen by the original designers or couldn't be expressed in the specification. The resulting not-quite-to-spec implementation therefore represents a lower-level form of the specification which has been elaborated to match real-world constraints [105].

Just as it has been observed that spreading the task of building a compiler across *n* programming teams will result in an *n*-pass compiler, so the syntax and semantics associated with a formal specification language can heavily influence the final implementation. For example Estelle's model of a system consists of a collection of FSMs which communicate via asynchronous messaging, SDL's model consists of FSMs connected together through FIFO message queues, and LOTOS' model consists of multiple independent processes which communicate via events occurring at synchronisation points. None of these resemble anything which is present in any common implementation language like C.

The effects of this lack of matching between specification and implementation languages are evident in real-world experiences in which an implementation of the same concept (a layer of the OSI model) specified in the three specification languages mentioned above closely mirrored the system model used by the specification language, whether this was appropriate for the situation or not [106]. In one case an implementation contained a bug relating to message ordering which was an artifact of the specification language's view of the system and which was only discovered by running it against an implementation derived from one of the other specifications, whose system model didn't assume that messages were in any particular order. This problem arose due to the particular weltanschauung of the formal specification language rather than any error in the specification or implementation itself. In the analysis of the Needham-Schroeder public key protocol mentioned earlier, the NRL protocol analyser was able to locate problems which hadn't been found by the FDR model checker because the model checker took a CSP specification and worked forwards while the NRL analyser took a specification of state transitions and worked backwards, and because the model checker couldn't verify any properties which involved an unbounded number of executions of the protocol whereas the analyser could, allowing it to detect odd boundary conditions such as the one where the two participants in the protocol were one and the same [101].

The use of FDR to find weaknesses in a protocol which was previously thought to be secure triggered a wave of other analyses including ones which used the Isabelle theorem prover [107], the Brutus model checker (with the same properties (and limitations) as FDR but which used various reduction techniques to try to combat the state space explosion which is experienced by model checkers) [108], the Murφ model checker and typography stress tester [109], and the Athena model checker combined with a new modelling technique called the strand space model, which attempts to work around the state space explosion problem and restrictions on the number of principals (although not the number of protocol runs) which beset traditional model checkers [110][111] (some of the other model checkers run out of steam once three or four principals participate). These further analyses which confirmed the findings of the initial work are an example of the analysis technique being a social process which serves to increase our confidence in the object being examined, something which is examined in more detail in the following section.

3.5. Credibility of Formal Methods

From a mathematical point of view, the attractiveness of formal methods, and specifically formal proofs of correctness, is that they have the potential to provide a high degree of confidence that a certain method or mechanism has the properties it is intended to have. This level of confidence often can't be obtained through other methods, for example something as simple as the addition operation on a 32-bit CPU would require 2^{64} or 10^{19} tests (and a known good set of test vectors to verify the results against), which is infeasible in any real design. The solution, at least in theory, is to construct a mathematical proof that the correct output will be produced for all possible input values. However, the use of mathematical proofs is not without its problems. One paper gives an example of American and Japanese topologists who provided complex (and contradictory) proofs concerning a certain type of topological object. The two sides swapped proofs, but neither could find any flaws in the other side's argument. The paper then goes on to give further examples of "proofs" which in some cases stood for years before being found to be flawed; in some cases the (faulty) proofs are so beguiling that they require footnotes and other commentary to avoid entrapping unwary readers [112].

An extreme example of a complex proof was Wiles' proof of Fermat's last theorem, which took seven years to complete, stretched over 200 pages, and then required another year of peer-review (and a bugfix) before it was finally published [113]. Had it not been for the fact that it represented a solution to a famous problem, it's unlikely that it would have received much scrutiny, in fact it's unlikely that any journal would have wanted to publish a 200-page proof. As DeMillo et al point out, "mathematical proofs increase our confidence in the truth of mathematical statements only after they have been subject to the social mechanisms of the mathematical community". Many of these proofs are never subject to much scrutiny, and of the estimated 200,000 theorems published each year, most are ignored [114]. A slightly different view of the situation covered by DeMillo at al (but with the same conclusion) is presented by Fetzer, who makes the case that programs represent conjectures and the execution of the program is an attempted refutation of the conjecture (the refutation is all too often successful, as anyone who has used commercial software will be aware) [115].

Security proofs and analyses for systems targeted at A1 or equivalent levels are typically of a size which makes the Fermat proof look trivial by comparison. It has been suggested that perhaps the evaluators use the 1000+ page monsters produced by the process as a pillow in the hope that they absorb the contents by osmosis, or perhaps only check every 10^{th} or 20^{th} page in the hope that a representative spot check will weed out any potential errors. It's almost certain that none of them are ever subject to the level of scrutiny that the proof of Fermat's last theorem, at a fraction of the size, was.

The problems inherent in relying purely on a correctness proof of code may be illustrated by the following example. In 1969, Peter Naur published a paper containing a very simple 25-line text-formatting routine which he informally proved correct [116]. When the paper was reviewed in *Computing Reviews*, the reviewer pointed out a trivial fault in the code which, had the code been run rather than proven correct would have been quickly detected [117]. Subsequently, three more faults were detected, some of which again would have been quickly noticed if the code had been run on test data [118].

The author of the second paper presented a corrected version of the code and formally proved it correct (Naur's paper only contained an informal proof). After it had been formally proven correct, three further faults were found which, again, would have been noticed if the code had been run on test data [119].

This episode underscores three important points made earlier. The first is that even something as apparently simple as a 25-line piece of code took some effort (which eventually stretched over a period of five years) to fully analyse. The second point is that, as pointed out by DeMillo et al, the process only worked because it was subject to scrutiny by peers. Had this analysis by outsiders not occurred, it's quite likely that the code would have been left in its original form, with an average of just under one fault for every three lines of code, until someone actually tried to use it. Finally and most importantly, the importance of actually testing the code is shown by the fact that four of the seven defects could have been found immediately simply by running the code on test data.

A similar case occurred in 1984 with an Orange Book A1 candidate for which the security testing team recommended against any penetration testing because the system had an A1 security kernel based on a formally verified FTLS. The government evaluators questioned this blind faith in the formal verification

process and requested that the security team attempt a penetration of the system. Within a short period, the team had hypothesised serious flaws in the system and managed to exploit one such flaw to penetrate its security. Although the team had believed the system was secure based on the formal verification, "there is no reason to believe that a knowledgeable and sceptical adversary would have failed to find the flaw (or others) in short order" [96].

In a related case, a program which had been subjected to a Z proof of the specification and a code-level proof of the implementation in SPARK (an Ada dialect modified to remove problematic areas such as dynamic memory allocation and recursion) was shipped with run-time checking disabled in the code (!!) even though testing had revealed problems such as numeric overflows which couldn't be found by proofs (just for reference, it was a numeric overflow in Ada code which brought down Ariane 5). Furthermore, the fact that the compiler generated code which employed dynamic memory allocation (although this wasn't specified in the source code) required that the object code be manually patched to remove the dynamic memory allocation calls [28].

The saga of Naur's program didn't end with the initial set of problems which were found in the proofs. A decade later, another author analysed the last paper which had been published on the topic and found twelve faults in the program specification which was presented therein [120]. Finally (at least as far as the current author is aware, the story may yet unfold further), another author pointed out a problem in that author's corrected specification [121]. The problems in the specifications arose because they were phrased in English, a language rather unsuited for the task due to its imprecise nature and the ease with which an unskilled practitioner (or a politician) can produce results filled with ambiguities, vagueness, and contradictions. The lesson to be drawn from the second part of the saga is that natural language isn't very well suited to specifying the behaviour of a program, and that a somewhat more rigorous method is required for this task. However, many types of formal notation are equally unsuited, since they produce a specification which is incomprehensible to anyone not schooled in the particular formal method which is being applied. This issue is addressed further in the next chapter.

3.6. Where Formal Methods are Cost-Effective

Is there any situation in which formal methods are worth the cost and effort involved in using them? There is one situation where they're definitely cost-effective and that's for hardware verification. The first of the two reasons for this is that hardware is relatively easy to verify because it has no pointers, no unbounded loops, no recursion, no dynamically created processes, and none of the other complexities which make the verification of software such a joy to perform.

The second reason why hardware verification is more cost-effective is because the cost of manufacturing a single unit of hardware is vastly greater than that of manufacturing (that is, duplicating) a single unit of software, and the cost of replacing hardware is outrageously more so than replacing software. As an example of the typical difference, compare the $400 million which the Pentium FDIV bug cost Intel to the negligible cost to Microsoft of a hotfix and soothing press release for the Windows bug du jour. Possibly inspired by Intel's troubles, AMD spent a considerable amount of time and money subjecting their FDIV implementation to formal analysis using the Boyer-Moore theorem prover, which confirmed that their algorithm was OK.

Another factor which contributes to the relative success of formal methods for hardware verification is the fact that hardware designers typically use a standardised language, either Verilog or VHDL, and routinely use synthesis tools and simulators, which can be tied into the use of verification tools, as part of the design process. An example of how this might work in practice is that a hardware simulator would be used to explore a counterexample to a design assertion which was revealed by a model checker. In software development this type of standardisation and the use of these types of tools doesn't occur.

These two factors, the fact that hardware is much more amenable to verification than software and the fact that there's a much greater financial incentive to do so, is what makes the use of formal methods for hardware verification cost-effective, and the reason why most of the glowing success stories cited for the use of formal methods all relate to their use in verifying hardware rather than software [122][123][124][44] (one paper on the use of formal methods for developing high-assurance systems only cites hardware verification in its collection of formal methods successes [125]).

3.7. Whither Formal Methods?

Apart from their use in validating hardware, a task which they are ideally suited for, the future doesn't look too promising for formal methods. It's not in general a good sign when a paper presented at the tenth annual conference for users of Z, probably the most popular formal method (at least in Europe) and one of the few with university courses which teach it, opens with "Z is in trouble" [126]. A landmark paper on software technology maturity which looked at the progress of technologies initiated in the 1960's and 1970's (including formal methods) found that it typically takes 15-20 years for a new technology to gain mainstream acceptance, with the mean time being 17 years [127]. Formal methods have been around for nearly twice that span and yet their current status is that the most popular ones have an acceptance level of "in trouble" (the referenced paper goes on to mention that there is "pathetically little use of Z in industry"). Somewhat more concrete figures are given in a paper which contains figures intending to point out the low penetration of OO methods in industry [128], but which show the penetration of formal methods as being only a fraction of that, coming in slightly above the noise level.

One of the most compelling demonstrations of the conflict of formal methods with real-world practice can be found by examining how a programmer would implement a typical algorithm, for example one to find the largest entry in an array of integers. The formal methods advocates would present the implementation of an algorithm to solve this problem as a process of formulating a loop invariant for a loop which scans through the array ($\forall j \in [0 \ldots i]$, max >= array[j]), proving it by induction, and then deriving an implementation from it. The problem with this approach is that no-one (except perhaps for the odd student in an introductory programming course) ever writes code this way. Anyone who knows how to program will never generate a program in this manner because they can recognise the problem and pull a working solution from existing knowledge [129]. This style of program creation represents a completely unnatural way of working with code, a problem which isn't helping the adoption of formal methods by programmers (the way in which code creation actually works is examined in some detail in the next chapter).

This general malaise in the use of formal methods for software engineering purposes (which has been summed up with the comment that they are perceived as "merely an academic exercise, a form of mental masturbation that has no relation to real-world problems" [130]) as well as the evidence presented in the preceding sections, indicates that formal proofs of correctness and similar techniques make for a less than ideal way to build a secure system since, like a number of other software engineering methodologies they constitute belief systems rather than an exact science, and "attempts to prove beliefs are bottomless pits" [131]. A rather different approach to this particular problem is given in the next chapter.

4. Problems with other Software Engineering Methods

As with formal methods, the field of software engineering contains a great many miracle cures, making it rather difficult to determine which techniques are worthy of further investigation. There are currently around 300 software engineering standards, and yet the state of most software currently being produced indicates that either they don't work or they're being ignored (the number of faults per 1000 lines of code, a common measure of software quality, has remained almost constant over the last 15 years). This is of little help to someone trying to find techniques suitable for constructing trustworthy systems.

For example, two widely-touted software engineering panaceas are the Software Engineering Institute's capability maturity model (CMM) and the use of CASE tools. Studies are only now being carried out to determine whether organisations at level $n + 1$ of the CMM produce software which is any better than organisations at level n (in other words, whether the CMM actually works) [132]. One study which has been completed could find "no relationship between any dimension of maturity and the quality of RE [Requirements Engineering] products. […] These findings do not adequately support the hypothesised strong relationship between organisational maturity and RE success" [133]. Another report cites management's "decrease in motivation from lack of a clear link between their visions of the business and the progress achieved" after they initiated CMM programmes [134]. Of particular relevance to implementers wanting to build trustworthy systems, a book on safe programming techniques for safety-critical and high-integrity systems found only a weak relationship between the presence of faults and either the level of integrity of the code or its process certification [135].

An additional problem with methods like the CMM is the manner in which they are applied. Although the original intent was laudable enough, the common approach of using the CMM levels simply as a pass/fail

filter to determine who is awarded a contract results in at least as much human ingenuity being applied to bypassing them as is applied to areas such as tax law. Some of the tricks which are used include overwhelming the auditors with detail, or alternatively underwhelming them with vague and misleading information in the knowledge that they'll never have time to follow things up, using misleading documentation (one example which is mentioned is a full-page diagram of a peer review process which in real life amounted to "find some technical people and get them to look at the code"), and general tricks such as asking participants to carry a CMM manual in the presence of the auditors and "scribble in the book, break the spine, and make it look well used" [136]. As a result, when the evaluation is just another hurdle to be jumped in order to secure a contract, all guarantees about the validity of the process become void (in fact so much time and money is frequently invested that the belief, be it CC, CMM, or ISO 9000, often becomes an end in itself).

The propensity for organising methodologies into hierarchies with no clear indication of which sort of improvement can be expected by progressing from one level to the next isn't constrained entirely to software engineering. It has been pointed out that the same issue affects security models as well, with no clear indication that penetrating or compromising a system with a sequence of properties $P_1 \ldots P_n$ is easier than penetrating one where P_{n+1} has been added, or (of more importance to the people paying for it) that a system costing $\$2n$ is substantially more difficult to exploit than one costing only $\$n$ [137] (there have been efforts recently to leverage the security community's existing experience in lack of visible difference between security levels by applying the CMM to security engineering [138][139][140]). The lack of assurance that spending twice as much gives you twice as much security is troubling because the primary distinction between the various levels given in standards such as the Orange Book, ITSEC, and Common Criteria is the amount of money which needs to be spent to attain each level. One observer has pointed out that going to a higher level can even lead to a decrease in security in some circumstances, for example an Orange Book B1 system conveniently labels the most damaging data for an attacker to target while C2 doesn't. This type of problem was first exploited more than a decade before the Orange Book appeared in an attack which targeted classified data which was treated differently from lower-value unclassified data by the operating environment [141], the same type of attack is still possible today under Windows NT to target valuable data such as user passwords (by adding the name of a DLL to the HKEY_LOCAL_MACHINE\SYSTEM\-CurrentControlSet\Control\Lsa\Notification Packages key which is fed any new or updated passwords by the system [142]) and private keys (by adding the name of a DLL to the HKEY_LOCAL_MACHINE\-SOFTWARE\Microsoft\Cryptography\Offload\ExpoOffload key which is fed all private keys which are in use by CryptoAPI [143]).

One alternative approach to the CMM levels which has been suggested in an attempt to match the real world is the use of a capability immaturity model with rankings of (progressively) foolish, stupid, and lunatic to match the CMM levels initial, repeatable, defined, managed, and optimising, providing levels 0 to –2 of the CMM [144]. Level –1 of the anti-CMM involves the use of "complex processes involving the use of arcane languages and inappropriate documentation standards [requiring] significant effort and a substantial proportion of their resources in order to impose these" (this seems to be describing the eventual result of applying the positive-valued levels of the CMM). Level –2 mentions the hope of "automatically generating a program from the specification", which has been proposed by a number of formal methods advocates. A similar approach was taken some years earlier by another publication when it published an alternative series of levels for guaranteed-to-fail projects [145], and (on a slightly less pessimistic note) as a pragmatic alternative to existing security models which examines security in terms of allowable failure modes rather than absolute restrictions [146].

For CASE tools (which have been around for somewhat longer than the CMM), a study by the CASE Research Corp found (contrary to the revolutionary improvements claimed through the use of CASE tools) that productivity dropped markedly in the first year of use as users adjusted to whatever CASE process was in use, and then returned to more or less the original, pre-CASE level (the study found some very modest gains, but wasn't able to determine whether this arose from factors other than the CASE tools, or that it lay outside the margin of error) [147]. Another survey carried out in three countries and covering some hundreds of organisations found that it was "very difficult to quantify overall gains in the areas of productivity, efficiency, and quality arising from the use of CASE […] Currently it would appear that any gains in one area are often offset by problems in another" [148]. Some of the blame for this may lie in the fact that CASE tools, like many other methodologies, were over-hyped when it came to be their turn at being the silver bullet

candidate (as with formal methods, no CASE tool vendor would admit that there might be certain application domains for which their product was somewhat more suited than others) with the result that most of them ended up as shelfware [149] or were only used when the client specifically demanded it [150].

The reasons for the failure of these methodologies may lie in the assumptions they make about how software development works. The current model has been compared to 19th-century physics, in which energy is continuous, matter is particulate, and the luminiferous ether fills space and is the medium through which light and radio waves travel. The world as a whole works in a rational way, and if we can find the rules by which things happen we can find out which ones apply when good things happen and use those to make sure the good things keep happening [151]. Unfortunately, real software development doesn't work like this. Attempts to treat software production as just another industrial mass-production process can't work because software is the result of a creative design and engineering process, not of a conventional manufacturing activity [152]. This means that although it makes sense to try to perfect the process for reliably cranking out car parts or lightbulbs or refrigerators, the creation of software isn't a mass production process but instead is based on the cloning of the result of a one-off development effort which is the product of the creativity, skill, and co-operation of developers and users. Methodologies such as the CMM and related production-process-based techniques, which assume that software can be cranked out like car parts, are doomed to failure (or at least lack of success) because software engineering isn't like any other type of engineering process.

4.1. Assessing the Effectiveness of Software Engineering Techniques

An earlier section described formal methods as "a revolutionary technique which has gained widespread appeal without rigorous experimentation", however this problem is not unique to formal methods but extends to many software engineering practices in general. For example one independent study found that applying a variety of software-engineering techniques had only a minor effect on code quality, and none on productivity [153]. Another study, this one specifically targeting formal methods and based on a detailed record of faults encountered in a large software program, could find no compelling evidence that formal methods improved code quality (although they did find a link to the programming team size, with smaller teams leading to fewer faults) [154]. The editor of Elsevier's *Journal of Systems and Software* reports seeing many papers which conclude that the techniques presented in them are of enormous value, but very little in the way of studies to support these claims [155], as did the author of a survey paper which examined the effects of a variety of claimed-to-be-revolutionary techniques who concluded that "the findings of this article present a few glimmers of light in an otherwise dark universe" [156]. The situation was summed up by one commentator with the observation that "software engineering owes more to the fashion industry than it does to the engineering industry […] creativity is unconstrained, beliefs are unsupported and progress is either erratic or nonexistent. It is not for nothing that we have hundreds of programming languages, hundreds of paradigms, and essentially the same old problems. […] In each case the paradigm arises without measurement, subsists without analysis, and usually disappears without comment" [157].

The same malaise which besets the study of the usefulness of formal methods afflicts software engineering in general, to the extent that one standard text on the subject devotes an entire chapter to the subject of "Experimentation in Software Engineering" to alert readers to the fact that many of the methods described therein may not have any real practical foundation [121]. Some of the problems which have been identified in the study of software engineering methods are:

- Use of students as subjects. Experiments are carried out on conveniently available subjects, which generally means university students, with problems which can be solved in the available time span, usually a few weeks or a semester. In the standard student tradition, the software engineering task will be completed the night before the deadline. This produces results which indicate how the methodology applies to toy problems executed by students, but not how it will fare in the real world.
- Scale of experimentation. Real-world studies are chosen, but because of various real-world constraints like cost and release schedules, no control group is available. One of the references cited above mentions a methodology which is based on an experiment which has been performed only once, and with a sample size of one (Fleischman and Pons were not involved). An example of this type of experimentation was one which was used to justify the use of formal methods which was carried out once using a single subject who for good measure was also a student [158]. Other experiments have been carried out by the developers of the methodology being tested, or where the project was a flagship

project being carried out with elite developers with access to effectively unlimited resources, and where the process was highly susceptible to the Hawthorne Effect (in which an improvement in a production process is caused by the intrusive observation of that process). This sort of testing produces results from which no valid conclusion can be drawn, since a single positive result can be trivially refuted by a negative result in the next test.

- Blind belief in experts. In many cases researchers will blindly accept statements made by proponents of a new methodology without ever questioning or challenging it. For example one researcher who was looking for empirical data on the use of the widely-accepted principle of module coupling (ranked as data coupling, stamp coupling, control coupling, common coupling, and content coupling) and cohesion (ranging from functional through communicational, procedural, temporal, and logical through to coincidental) for software design was initially unable to identify any company which used this scheme, and after some prodding found that the ranking of five of the classes was misleading [159] (these classes have been used elsewhere as a measure of "goodness" for Orange Book kernel implementations [160]).

The problem of a lack of experimental evidence to support claims made by researchers exists for software engineering techniques other than the formal methods already mentioned above. One author who tried to verify claims made at a software engineering seminar found it impossible to obtain access to any of the evidence which would be required to support the claims, the reasons being given for the lack of evidence included the fact that the data was proprietary, unavailable, or that it hadn't been analysed properly, leading him to conclude that "as an industry we collect lots of data about practices that are poorly described or flawed to start with. These data then get disseminated in a manner that makes it nearly impossible to confirm or validate their significance" [161].

An example of where this can lead is provided by the IBM's CICS redevelopment, which won the Queen's Award for Technological Achievement in 1992 for its application of formal methods and is frequently used as a rare example of why the use of Z is a Good Thing. The citation stated that "The use of Z reduced development costs significantly and improved reliability and quality", however when a group of researchers not directly involved in the project attempted to verify these claims they could find no evidence to support them [162]. Although some papers which were published on the work contained various (occasionally difficult to quantify) comments that the new code contained fewer problems than expected, the reason for this was probably due more to the fact that they constituted rewrites of a number of known failure-prone modules than any magic worked by the use of Z. A more recent work which claims to show that Z and code-level proofs were more effective at finding faults than testing contains figures which show the exact opposite (testing found 66% of all faults, the Z proof (done at the specification stage) found 16%, and the code proof found 5¼%). The reason why the paper is able to make the claim that proofs are more effective at finding faults is because Z was more *efficient* at finding problems than testing was (even though it didn't find most of the problems) [28]. In other words, Z is the answer provided you phrase the question very carefully. The results presented in the paper, written by the developers of the tools used to carry out the proofs, haven't (yet) been subject to outside analysis. More comments on the work in this paper are given in Section 3.5 above.

Other software engineering success stories also arise in cases where everything else has failed, so that any change at all from whatever methodology is currently being followed will lead to some measure of success. One work mentions formal methods being applied to an existing design which consisted of "a hodge-podge of modules with patches in various languages that dated back to the late 1960's" [33], where it's quite likely that anything at all when used in this situation would have resulted in some sort of improvement. Just because leaping from a speeding car which is heading for the edge of a cliff is a good idea for that particular situation doesn't mean that the concept should be applied as a general means of exiting vehicles.

Another problem, not specifically mentioned above since it plagues many other disciplines as well, is the misuse of statistics, although specific complaints about their misuse in the field of software metrics have been made [163][164]. Serving as a complement to the misuse of statistics is a complete lack thereof. One investigation into the number of computer science research papers containing experimentally validated results found that nearly half the papers taken from a random sample of refereed computer science journals which contained statements which would require empirical validation contained none, with software engineering papers in particular leading the others in a lack of evidence to support claims made therein. In contrast the figure for optical engineering and neuroscience journals which were used for comparison had just over $^{1}/_{10}{}^{th}$ of the papers lacking experimental evidence. The authors concluded that "there is a

disproportionately high percentage of design and modelling work without any experimental evaluation in the CS samples […] Samples related to software engineering are worse than the random CS sample" [165].

The reason these techniques are used isn't always because of sloppiness on the part of the researchers involved, but because it is generally impractical to conduct the standard style of experiment involving control subjects, real-world applications, and testing over a long period of time. For example if a real-world project were to be subject to experimental evaluation it might require three or four independent teams (to get a reasonable sample size) and perhaps five other groups of teams performing the same task using different methodologies. This would raise the cost to around fifteen to twenty times the original cost, making it simply too expensive to be practical. In addition since the major effects of the methodology won't really be felt until the maintenance phase, the evaluation would have to continue over the next several years to determine which methodology produced the best result in the long term. This would require maintaining a large collection of parallel products for the duration of the experiment, which is clearly infeasible.

5. Alternative Approaches

Since the birth of software engineering in the late 1960's/early 1970's, the tendency has been to solve problems by adding rules and building methodologies to cover every eventuality, in the hope that eventually all possible situations would be covered and perfect, bug-free software would materialise on time and within budget. Alternative approaches lead to meta-methodologies like ISO 9000, which aren't a software engineering methodology in and of themselves but represent a meta-methodology with which a real methodology is meant to be created, the bureaucrats dream which allows the production of infinite amounts of paperwork and the illusion of progress without actually necessitating the production of an end product.

These juggernauts have now lead to backlash methodologies such as extreme programming (XP) whose principal feature is that they are everything their predecessors weren't: lightweight, easy to use, and flexible. It's instructive to take a look at XP in order to compare it with traditional alternatives.

5.1. Extreme Programming

XP is a slightly more rigorous form of an ad-hoc methodology which has been termed "development on Internet time" which begins with a general functional product specification which is revised as the product evolves and is only complete when the product itself is complete. Development is broken up into subcycles at the end of which the product is stabilised by fixing major errors and freezing certain features. Schedule slip is handled by deleting features. In addition developers are (at least in theory) given the power to veto some requirements on technical grounds [166][167].

XP follows the general pattern of "development on Internet time" but is far more rigorous [168][169][170]. It also doesn't begin with the traditional mountain of design documentation. Instead, the end user is asked to provide a collection of user stories, short statements on what the finished product is expected to do. The intent of the user stories is to provide just enough detail to allow the developers to estimate how long the story will take to implement. Each story describes only the user's needs, implementation details are left to the developers who (presumably) will understand the technical capabilities and limitations far better than the end user, leaving them with the freedom to choose the most appropriate solution to the problem.

The development process is structured around the user stories, ordered according to their value to the user and their risk to the developers. The selection of which stories to work with first is performed by the end user in collaboration with the programmers. In this way the most problematic and high-value problems are handled first, and the easy or relatively inconsequential ones are left for later. The end user is kept in the loop at all times during the development process, with frequent code releases to allow them to determine whether the product meets their requirements.

The relationship to earlier methodologies such as the waterfall model (characterised by long development cycles) and the spiral model (with slightly shorter cycles) is shown in Figure 1. This both allows the end user to ensure that it will work as required in its target environment, and avoids the "it's just what I asked for but not what I want" problem which plagues software developed using traditional methodologies in which the customer signs off on a huge, only vaguely understood design specification and doesn't get to play with the deliverables until it's too late to make any changes. The general concept behind XP is that if it's possible to

make change cheap, then all sorts of things can be achieved which wouldn't be possible with other methodologies.

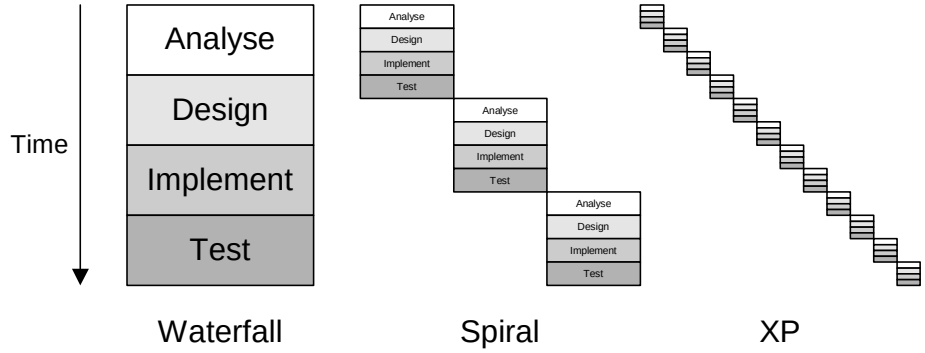

Figure 1: Comparison of software development life cycles

XP also uses continuous testing as part of the development process, actually moving the creation of unit testing code to before the creation of the code itself, so that it's easy to determine whether the program code works as required as soon as it's written. If a bug is found, a new test is created to ensure that it won't recur later.

Practitioners of "real" methodologies who are still reading at this point will no doubt be horrified by this description of XP, however it's an example of what can be done by adapting the methodology to the environment rather than trying to force-fit the environment to match the methodology. A remarkable feature of XP which arises from this is the level of enthusiasm displayed for it by its users (as opposed to its advocates, vendors, authors of books expounding its benefits, and other hangers-on), something which is hard to find for alternatives such as ISO 9000, CASE tools, and so on (the popularity of XP is such that it has its own conference and a number of very active web forums).

5.2. Lessons from Alternative Approaches

The previous section showed how, in the face of problems with traditional approaches, a problem-specific approach may be successful. Note that XP isn't a general-purpose solution (and it remains to be seen just how effective it will really be in the long term), however it addresses one particular problem, the need for rapid development in the face of constantly-changing requirements, and only tries to solve this particular problem. The methodology evolved by starting with a real-world approach to the problem of making change cheap and then codifying it as XP, rather than beginning with a methodology based on (say) mathematical theory and then forcing development to fit the theory.

The same approach, this time with the goal of developing secure systems, is taken in the next chapter.

6. References

[1] "No Silver Bullet: Essence and Accidents of Software Engineering", Frederick Brooks Jr., *IEEE Computer*, **Vol.20**, **No.4** (April 1987), p.10.

[2] "Striving for Correctness", Marshall Abrams and Marvin Zelkowitz, *Computers and Security*, **Vol.14**, **No.8** (1995), p.719.

[3] "Does OO Sync with How We Think?", Les Hatton, *IEEE Software*, **Vol.15**, **No.3** (May/June 1998), p.46.

[4] "Software Engineering: A Practitioners Approach (3rd ed)", Roger Pressman, McGraw-Hill International Edition, 1992.

[5] "A Specifier's Introduction to Formal Methods", Jeannette Wing, *IEEE Computer*, **Vol.23**, **No.9** (September 1990), p.8.

[6] "Strategies for Incorporating Formal Specifications in Software Development", Martin Fraser, Kuldeep Kumar, and Vijay Vaishnavi, *Communications of the ACM*, **Vol.37**, **No.10** (October 1994), p.74.

[7] "A Technique for Software Module Specification with Examples", David Parnas, *Communications of the ACM*, **Vol.15**, **No.5** (May 1972), p.330.

[8] "Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems", William Price, PhD thesis, Carnegie-Mellon University, June 1973.

[9] "An Experiment with Affirm and HDM", Jonathan Millen and David Drake, *The Journal of Systems and Software*, **Vol.2**, **No.2** (June 1981), p.159.

[10] "Applying Formal Methods to an Information Security Device: An Experience Report", James Kirby Jr., Myla Archer, and Constance Heitmeyer, *Proceedings of the 4th International Symposium on High Assurance Systems Engineering (HASE'99)*, IEEE Computer Society Press, November 1999, p.81.

[11] "Building a Secure Computer System", Morrie Gasser, Van Nostrand Reinhold, 1988.

[12] "Validating Requirements for Fault Tolerant Systems using Model Checking", Francis Schneider, Steve Easterbrook, John Callahan, and Gerard Holzman, *Proceedings of the 3rd International Conference on Requirements Engineering*, IEEE Computer Society Press, April 1998, p.4.

[13] "Report on the Formal Specification and Partial Verification of the VIPER Microprocessor", Bishop Brock and Warren Hunt Jr., *Proceedings of the 6th Annual Conference on Computer Assurance (COMPASS'91)*, IEEE Computer Society Press, 1991, p.91.

[14] "User Threatens Court Action over MoD Chip", Simon Hill, *Computer Weekly*, 5 July 1990, p.3.

[15] "MoD in Row with Firm over Chip Development", *The Independent*, 28 May 1991.

[16] "Formal Methods of Program Verification and Specification", H.Berg, W.Boebert, W.Franta, and T.Moher, Prentice-Hall Inc, 1982.

[17] "A Description of a Formal Verification and Validation (FVV) Process", Bill Smith, Cynthia Reese, Kenneth Lindsay, and Brian Crane, *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, August 1988, p.401.

[18] "An InaJo Proof Manager for the Formal Development Method", Daniel Barry, *ACM SIGSOFT Software Engineering Notes*, **Vol.10**, **No.4** (August 1985), p.19.

[19] "Proposed Technical Evaluation Criteria for Trusted Computer Systems", Grace Nibaldi, MITRE Technical Report M79-225, The MITRE Corporation, 25 October 1979.

[20] "Locking Computers Securely", O.Sami Saydari, Joseph Beckman, and Jeffrey Leaman, *Proceedings of the 10th National Computer Security Conference*, September 1987, p.129.

[21] "Do You Trust Your Compiler", James Boyle, R.Daniel Resler, Victor Winter, *IEEE Computer*, **Vol.32**, **No.5** (May 1999), p.65.

[22] "Integrating Formal Methods into the Development Process", Richard Kemmerer, *IEEE Software*, **Vol.7**, **No.5** (September 1990), p.37.

[23] "Towards a verified MiniSML/SECD system", Todd Simpson, Graham Birtwhistle, and Brian Graham, *Software Engineering Journal*, **Vol.8**, **No.3** (May 1993), p.137.

[24] "Formal Verification of Transformations for Peephole Optimisation", A.Dold, F.von Henke, H.Pfeifer, and H.Rueß, *Proceedings of the 4th International Symposium of Formal Methods Europe (FME'97)*, Springer-Verlag Lecture Notes in Computer Science No.1313, p.459.

[25] "The verification of low-level code", D.Clutterbuck and B.Carré, *Software Engineering Journal*, **Vol.3**, **No.3** (May 1988), p.97.

[26] "Automatic Verification of Object Code Against Source Code", Sakthi Subramanian and Jeffrey Cook, *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS'96)*, IEEE Computer Society Press, June 1996, p.46.

[27] "Automatic Generation of C++ Code from an ESCRO2 Specification", P.Grabow and L.Liu, *Proceedings of the 19th Computer Software and Applications Conference (COMPSAC'95)*, September 1995, p.18.

[28] "Is Proof More Cost-Effective Than Testing", Steve King, Jonathan Hammond, Rod Chapman, and Andy Pryor, *IEEE Transactions on Software Engineering*, **Vol.26**, **No.8** (August 2000), p.675.

[29] "Science and Substance: A Challenge to Software Engineers", Norman Fenton, Shari Lawrence Pfleeger, and Robert L.Glass, *IEEE Software*, **Vol.11**, **No.4** (July 1994), p.86.

[30] "The Software-Research Crisis", Robert Glass, *IEEE Software*, **Vol.11**, **No.6** (November 1994), p.42.

[31] "Observation on Industrial Practice Using Formal Methods", Susan Gerhart, Dan Craigen, and Ted Ralston, *Proceedings of the 15th International Conference on Software Engineering (ICSE'93)*, 1993, p.24.

[32] "How Effective Are Software Engineering Methods?", Norman Fenton, *The Journal of Systems and Software*, **Vol.22**, **No.2** (August 1993), p.141.

[33] "Industrial Applications of Formal Methods to Model, Design, and Analyze Computer Systems: An International Survey", Dan Craigen, Susan Gerhart, and Ted Ralston, Noyes Data Corporation, 1994 (originally published by NIST).

[34] "The Evaluation of Three Specification and Verification Methodologies", Richard Platek, *Proceedings of the 4th Seminar on the DoD Computer Security Initiative* (later the National Computer Security Conference), August 1981, p.X-1.

[35] "Ina Jo: SDC's Formal Development Methodology", *ACM SIGSOFT Software Engineering Notes*, **Vol.5**, **No.3** (July 1980).

[36] "FDM — A Specification and Verification Methodology", Richard Kemmerer, *Proceedings of the 3rd Seminar on the DoD Computer Security Initiative Program* (later the National Computer Security Conference) November 1980, p.L-1.

[37] "INATEST: An Interactive System for Testing Formal Specifications", Steven Eckmann and Richard Kemmerer, *ACM SIGSOFT Software Engineering Notes*, **Vol.10**, **No.4** (August 1985), p.17.

[38] "Gypsy: A Language for Specification and Implementation of Verifiable Programs", Richard Cohen, Allen Ambler, Donald Good, James Browne, Wilhelm Burger, Charles Hoch, and Robert Wells, *SIGPLAN Notices*, **Vol.12**, **No.3** (March 1977), p.1.

[39] "A Report on the Development of Gypsy", Richard Cohen, Donald Good and Lawrence Hunter, *Proceedings of the 1978 National ACM Conference*, December 1978, p.116.

[40] "Building Verified Systems with Gypsy", Donald Good, *Proceedings of the 3rd Seminar on the DoD Computer Security Initiative Program* (later the National Computer Security Conference) November 1980, p.M-1.

[41] "Industrial Use of Formal Methods", Steven Miller, *Dependable Computing and Fault-Tolerant Systems*, **Vol.9**, Springer-Verlag, 1995, p.33.

[42] "Can we rely on Formal Methods?", Natarajan Shankar, *Dependable Computing and Fault-Tolerant Systems*, **Vol.9**, Springer-Verlag, 1995, p.42.

[43] "Applications of Formal Methods", Mike Hinchey and Jonathan Bowen, Prentice-Hall International, 1995.

[44] "A Case Study in Model Checking Software Systems", Jeannette Wing and Mondonna Vaziri-Farahani, *Science of Computer Programming*, **Vol.28**, **No.2-3** (April 1997), p.273.

[45] "A survey of mechanical support for formal reasoning", Peter Lindsay, *Software Engineering Journal*, **Vol.3**, **No.1** (January 1988), p.3.

[46] "Verification Technology and the A1 Criteria", Terry Vickers Benzel, *ACM SIGSOFT Software Engineering Notes*, **Vol.10**, **No.4** (August 1985), p.108.

[47] "Verifying security", Maureen Cheheyl, Morrie Gasser, George Huff, and Jonathan Millen, *ACM Computing Surveys*, **Vol.13**, **No.3** (September 1981), p.279.

[48] "Software Testing Techniques (2nd ed)", Boris Beizer, Van Nostrand Reinhold, 1990.

[49] "Engineering Requirements for Production Quality Verification Systems", Stephen Crocker, *ACM SIGSOFT Software Engineering Notes*, **Vol.10**, **No.4** (August 1985), p.15.

[50] "Problems, methods, and specialisation", Michael Jackson, *Software Engineering Journal*, **Vol.9**, **No.6** (November 1994), p.249.

[51] "Formal Methods and Traditional Engineering", Michael Jackson, *The Journal of Systems and Software*, **Vol.40**, **No.3** (March 1998), p.191.

[52] "Panel Session: Kernel Performance Issues", Marvin Shaefer (chairman), *Proceedings of the 1981 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, August 1981, p.162.

[53] "The Best Available Technologies for Computer Security", Carl Landwehr, *IEEE Computer*, **Vol.16**, **No 7** (July 1983), p.86.

[54] "A Retrospective on the VAX VMM Security Kernel", Paul Karger, Mary Ellen Zurko, Douglas Bonin, Andrew Mason, and Clifford Kahn, *IEEE Transactions on Software Engineering*, **Vol.17**, **No.11** (November 1991), p.1147.

[55] "Formal Construction of the Mathematically Analyzed Separation Kernel", W.Martin, P.White, F.S.Taylor, and A.Goldberg, *Proceedings of the 15th International Conference on Automated Software Engineering (ASE'00)*, IEEE Computer Society Press, September 2000, p.133.

[56] "Formal Methods Reality Check: Industrial Usage", Dan Craigen, Susan Gerhart, and Ted Ralston, *IEEE Transactions on Software Engineering*, **Vol.21**, **No.2** (February 1995), p.90.

[57] "Mathematical Methods: What we Need and Don't Need", David Parnas, *IEEE Computer*, **Vol.29**, **No.4** (April 1996), p.28.

[58] "Literate Specifications", C.Johnson, *Software Engineering Journal*, **Vol.11**, **No.4** (July 1996), p.225.

[59] "Mathematical Notation in Formal Specification: Too Difficult for the Masses?", Kate Finney, *IEEE Transactions on Software Engineering*, **Vol.22**, **No.2** (February 1996), p.158.

[60] "The Design of a Family of Applications-oriented Requirements Languages", Alan Davis, *IEEE Computer*, **Vol.15**, **No.5** (May 1982), p.21.

[61] "An Operational Approach to Requirements Specification for Embedded Systems", *IEEE Transactions on Software Engineering*, **Vol.8**, **No.3** (May 1982), p.250.

[62] "A Comparison of Techniques for the Specification of External System Behaviour", Alan Davis, *Communications of the ACM*, **Vol.31**, **No.9** (September 1988), p.1098.

[63] "A 15 Year Perspective on Automatic Programming", *IEEE Transactions on Software Engineering*, **Vol.11**, **No.11** (November 1985), p.1257.

[64] "Operational Specification as the Basis for Rapid Prototyping", Robert Balzer, Neil Goldman, and David Wile, *ACM SIGSOFT Software Engineering Notes*, **Vol.7**, **No.5** (December 1982), p.3.

[65] "Fault Tolerance by Design Diversity: Concepts and Experiments", Algirdas Avižienis and John Kelly, *IEEE Computer*, **Vol.17**, **No.8** (August 1984), p.67.

[66] "Coding for a Believable Specification to Implementation Mapping", William Young and John McHugh, , *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, August 1987, p.140.

[67] "DoD Overview: Computer Security Program Direction, Colonel Joseph Greene Jr., *Proceedings of the 8th National Computer Security Conference*, September 1985, p.6.

[68] "The Emperor's Old Armor", Bob Blakley, *Proceedings of the 1996 New Security Paradigms Workshop*, ACM, 1996, p.2.

[69] "Analysis of a Kernel Verification", Terry Vickers Benzel, *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, August 1984, p.125.

[70] "Increasing Assurance with Literate Programming Techniques", Andrew Moore and Charles Payne Jr., *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS'96)*, National Institute of Standards and Technology, June 1996.

[71] "Formal Verification Techniques for a Network Security Device", Hicham Adra and William Sandberg-Maitland, *Proceedings of the 3rd Annual Canadian Computer Security Symposium*, May 1991, p.295.

[72] "Assessment and Control of Software", Capers Jones, Yourdon Press, 1994.

[73] "An InaJo Proof Manager", Daniel Berry, *ACM SIGSOFT Software Engineering Notes*, **Vol.10**, **No.4** (August 1985), p.19.

[74] "Formal Methods: Promises and Problems", Luqi and Joseph Goguen, *IEEE Software*, **Vol.14**, **No.1** (January 1997), p.73.

[75] "A Security Model for Military Message Systems", Carl Landwehr, Constance Heitmeyer, and John McLean, *ACM Transactions on Computer Systems*, **Vol.2**, **No.3** (August 1984), p.198.

[76] "Risk Analysis of 'Trusted Computer Systems'", Klaus Brunnstein and Simone Fischer-Hübner, *Computer Security and Information Integrity*, Elsevier Science Publishers, 1991, p.71.

[77] "A Retrospective on the Criteria Movement", Willis Ware, *Proceedings of the 18th National Information Systems Security Conference* (formerly the National Computer Security Conference), October 1995, p.582.

[78] "Are We Testing for True Reliability?", Dick Hamlet, *IEEE Software*, **Vol.9**, **No.4** (July 1992), p.21.

[79] "The Limits of Software: People, Projects, and Perspectives", Robert Britcher and Robert Glass, Addison-Wesley, 1999.

[80] "A Review of the State of the Practice in Requirements Modeling", Mitch Lubars, Colin Potts, and Charlie Richter, *Proceedings of the IEEE International Symposium on Requirements Engineering*, IEEE Computer Society Press, January 1993, p.2.

[81] "Software-Engineering Research Revisited", Colin Potts, *IEEE Software*, **Vol.10**, **No.5** (September 1993), p.19.

[82] "Invented Requirements and Imagined Customers: Requirements Engineering for Off-the-Shelf Software", Colin Potts, *Proceedings of the 2nd IEEE International Symposium on Requirements Engineering*, IEEE Computer Society Press, March 1995, p.128.

[83] "Validating a High-Performance, Programmable Secure Coprocessor", Sean Smith, Ron Perez, Steve Weingart, and Vernon Austel, *Proceedings of the 22nd National Information Systems Security Conference*, October 1999.

[84] "A New Paradigm for Trusted Systems", Dorothy Denning, *Proceedings of the New Security Paradigms Workshop '92*, 1992, p.36.

[85] "TCB Subsets for Incremental Evaluation", William Shockley and Roger Schell, *Proceedings of the 3rd Aerospace Computer Security Conference*, December 1987, p.131.

[86] "Requirements for Market Driven Evaluations for Commercial Users of Secure Systems", Peter Callaway, *Proceedings of the 3rd Annual Canadian Computer Security Symposium*, May 1991, p.207.

[87] "Re-Use of Evaluation Results", Jonathan Smith, *Proceedings of the 15th National Computer Security Conference*, October 1992, p.534.

[88] "Using a Mandatory Secrecy and Integrity Policy on Smart Cards and Mobile Devices", Paul Karger, Vernon Austel, and David Toll, *Proceedings of the EuroSmart Security Conference*, June 2000, p.134.

[89] "The Need for an Integrated Design, Implementation, Verification, and Testing Methodology", R.Alan Whitehurst, *ACM SIGSOFT Software Engineering Notes*, **Vol.10**, **No.4** (August 1985), p.97.

[90] "SELECT — A Formal System for Testing and Debugging Programs by Symbolic Execution", Robert Boyer, Bernard Elspas, and Karl Levitt, *ACM SIGPLAN Notices*, **Vol.10**, **No.6** (June 1975), p.234.

[91] "A Review of Formal Methods", Robert Vienneau, *A Review of Formal Methods*, Kaman Science Corporation, 1993, p.3.

[92] "Formal Specification and Verification of Control Software for Cryptographic Equipment", D.Richard Kuhn and James Dray, *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, August 1990, p.32.

[93] "A Cryptographic Evaluation of IPsec", Niels Ferguson and Bruce Schneier, Counterpane Labs, 1999, http://www.counterpane.com/ipsec.html.

[94] "Formal Verification of Cardholder Registration in SET", Giampaolo Bella, Fabio Massacci, Lawrence Paulson, and Piero Tramontano, *Proceedings of the 6th European Symposium on Research in Computer Security (ESORICS 2000)*, Springer-Verlag Lecture Notes in Computer Science No.1895, p.159.

[95] "Information Flow and Invariance", Joshua Guttman, *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, August 1987, p.67.

[96] "Symbol Security Condition Considered Harmful", Marvin Schaefer, *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, August 1989, p.20.

[97] "Re: WuFTPD: Providing *remote* root since at least 1994", Theo de Raadt, posting to the bugtraq mailing list, message-ID `200006272322.e5RNMIv18874@cvs.openbsd.org`, 27 June 2000.

[98] "A Logic of Authentication", Michael Burrows, Martín Abadi, and Roger Needham, *ACM Transactions on Computer Systems*, **Vol.8**, **No.1** (February 1990), p.18.

[99] "Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR", Gavin Lowe, *Proceedings of the 2d International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, Springer-Verlag Lecture Notes in Computer Science No.1055, March 1996, p.147.

[100] "Casper: A Compiler for the Analysis of Security Protocols", Gavin Lowe, *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, May 1997, p.18.

[101] "Analyzing the Needham-Schroeder Public Key Protocol: A Comparison of Two Approaches", Catherine Meadows, *Proceedings of the 4th European Symposium on Research in Computer Security (ESORICS'96)*, Springer-Verlag Lecture Notes in Computer Science No.1146, September 1996, p.351.

[102] "On the Verification of Cryptographic Protocols — A Tale of Two Committees", Dieter Gollman, Proceedings of the Workshop on Secure Architectures and Information Flow, *Electronic Notes in Theoretical Computer Science (ENTCS)*, **Vol.32**, 2000, `http://www.elsevier.nl/gej-ng/31/29/23/57/23/show/Products/notes/index.htt`.

[103] "The Logic of Computer Programming", Zohar Manna and Richard Waldinger, *IEEE Transactions on Software Engineering*, **Vol.4**, **No.3** (May 1978), p.199.

[104] "Verifying a Real System Design — Some of the Problems", Ruaridh Macdonald, *ACM SIGSOFT Software Engineering Notes*, **Vol.10**, **No.4** (August 1985), p.128.

[105] "On the Inevitable Intertwining of Specification and Implementation", William Swartout and Robert Balzer, *Communications of the ACM*, **Vol.25**, **No.7** (July 1982), p.438.

[106] "An Empirical Investigation of the Effect of Formal Specifications on Program Diversity", Thomas McVittie, John Kelly, and Wayne Yamamoto, *Dependable Computing and Fault-Tolerant Systems*, **Vol.6**, Springer-Verlag, 1992, p.219.

[107] "Proving Properties of Security Protocols by Induction", Lawrence Paulson, *Proceedings of the 10th Computer Security Foundations Workshop (CSFW'97)*, June 1997, p.70.

[108] "Verifying Security Protocols with Brutus", E.M.Clarke, S.Jha, and W.Marrero, *ACM Transactions on Software Engineering and Methodology*, **Vol.9**, **No.4** (October 2000), p.443.

[109] "Automated Analysis of Cryptographic Protocols Using Murφ", John Mitchell, Mark Mitchell, and Ulrich Stern, *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, May 1997, p.141.

[110] "Strand Spaces: Why is a Security Protocol Correct", F.Javier Thayer Fábrega, Jonathan Herzog, and Joshua Guttman, *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, May 1998, p.160.

[111] "Athena: a novel approach to efficient automatic security protocol analysis", Dawn Xiaoding Song, Sergey Berezin, and Adrian Perrig, *Journal of Computer Security*, **Vol.9**, **Nos.1,2** (2000), p.47.

[112] "Social Processes and Proofs of Theorems and Programs", Richard DeMillo, Richard Lipton, and Alan Perlis, *Communications of the ACM*, **Vol.22**, **No.5** (May 1979), p.271.

[113] "Fermat's Last Theorem", Simon Singh, Fourth Estate, 1997.

[114] "Adventures of a Mathematician", Stanislaw Ulam, Scribners, 1976.

[115] "Program Verification: The Very Idea", James Fetzer, *Communications of the ACM*, **Vol.31**, **No.9** (September 1988), p.1048.

[116] "Programming by Action Clusters", Peter Naur, *BIT*, **Vol.9**, **No.3** (September 1969), p.250.

[117] Review No.19,420, Burt Leavenworth, *Computing Reviews*, **Vol.11**, **No.7** (July 1970), p.396.

[118] "Software Reliability through Proving Programs Correct", *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing*, March 1971, p.125.

[119] "Toward a Theory of Test Data Selection", John Goodenough and Susan Gerhart, *IEEE Transactions on Software Engineering*, **Vol.1**, **No.2** (June 1975), p.156.

[120] "On Formalism in Specifications", *IEEE Software*, **Vol.2**, **No.1** (January 1985), p.6.

[121] "Software Engineering (2nd ed)", Stephen Schach, Richard Irwin and Asken Associates, 1993.

[122] "Acceptance of Formal Methods: Lessons from Hardware Design", David Dill and John Rushby, *IEEE Computer*, **Vol.29**, **No.4** (April 1996), p.23.

[123] "Formal Hardware Verification: Methods and systems in comparison", Lecture Notes in Computer Science No.1287, Springer-Verlag, 1997.

[124] "Formal methods in computer aided design: Second international conference proceedings", Lecture Notes in Computer Science No.1522, Springer-Verlag, 1998.

[125] "Formal Methods For Developing High Assurance Computer Systems: Working Group Report", Mats Heimdahl and Constance Heitmeyer, *Proceedings of the 2nd Workshop on Industrial-Strength Formal Specification Techniques (WIFT'98)*, IEEE Computer Society Press, October 1998.

[126] "Taking Z Seriously", Anthony Hall, *The Z formal specification notation: Proceedings of ZUM'97*, Springer-Verlag Lecture Notes in Computer Science No.1212, 1997, p.1.

[127]"Software Technology Maturation", Samuel Redwine and William Riddle, *Proceedings of the 8th International Conference on Software Engineering (ICSE'85)*, IEEE Computer Society Press, August 1985, p.189.

[128]"OO is NOT the Silver Bullet", J.Barrie Thompson, *Proceedings of the 20th Computer Software and Applications Conference (COMPSAC'96)*, IEEE Computer Society Press, 1996, p.155.

[129]"The Psychological Study of Programming", B.Sheil, *Computing Surveys*, **Vol.13**, **No.1** (March 1981), p.101.

[130]"Seven More Myths of Formal Methods" Jonathan Bowen and Michael Hinchey, *IEEE Software*, **Vol.12**, **No.4** (July 1995), p.34.

[131]"Belief in Correctness", Marshall Abrams and Marvin Zelkowitz, *Proceedings of the 17th National Computer Security Conference*, October 1994, p.132.

[132]"Status Report on Software Measurement", Shari Lawrence Pfleeger, Ross Jeffery, Bill Curtis, and Barbara Kitchenham, *IEEE Software*, **Vol.14**, **No.2** (March/April 1997), p.33.

[133]"Does Organizational Maturity Improve Quality?", Khaled El Emam and Nazim Madhavji, *IEEE Software*, **Vol.13**, **No.5** (September 1996), p.209.

[134]"Is Software Process Re-engineering and Improvement the 'Silver Bullet' of the 1990's or a Constructive Approach to Meet Pre-defined Business Targets", Annie Kuntzmann-Combelles, *Proceedings of the 20th Computer Software and Applications Conference (COMPSAC'96)*, 1996, p.435.

[135]"Safer C: Developing for High-Integrity and Safety-Critical Systems", Les Hatton, McGraw-Hill, 1995.

[136]"Can You Trust Software Capability Evaluations", Emilie O'Connell and Hossein Saiedian, *IEEE Computer*, **Vol.33**, **No.2** (February 2000), p.28.

[137]"New Paradigms for High Assurance Systems", John McLean, *Proceedings of the 1992 New Security Paradigms Workshop*, IEEE Press, 1993, p.42.

[138]"Determining Assurance Levels by Security Engineering Process Maturity", Karen Ferraiolo and Joel Sachs, *Proceedings of the 5th Annual Canadian Computer Security Symposium*, May 1993, p.477.

[139]"Community Response to CMM-Based Security Engineering Process Improvement", Marcia Zior, *Proceedings of the 18th National Information Systems Security Conference* (formerly the National Computer Security Conference), October 1995 p.404.

[140]"Systems Security Engineering Capability Maturity Model (SSE-CMM), Model Description Document Version 2.0", Systems Security Engineering Capability Maturity Model (SSE-CMM) Project, 1 April 1999.

[141]"OS/360 Computer Security Penetration Exercise", S.Goheen and R.Fiske, MITRE Working Paper WP-4467, The MITRE Corporation, 16 October 1972.

[142]"HOWTO: Password Change Filtering & Notification in Windows NT", Microsoft Knowledge Base Article Q151082, June 1997.

[143]"A new Microsoft security bulletin and the OffloadModExpo functionality", Sergio Tabanelli, posting to the `aucrypto` mailing list, message-ID `20000413102943.OGOB5378.fep03-svc.tin.it@fep11-svc.tin.it`, 13 April 2000.

[144]"A Software Process Immaturity Model", Anthony Finkelstein, *ACM SIGSOFT Software Engineering Notes*, **Vol.17**, **No.4** (October 1992), p.22.

[145] "Rules to Lose By: The Hopeless character class", Roger Koppy, *Dragon Magazine*, **No.96**, April 1984.

[146]"The Need for a Failure Model for Security", Catherine Meadows, *Dependable Computing and Fault-tolerant Systems*, **Vol.9**, 1995.

[147]"The Second Annual Report on CASE", CASE Research Corp, Washington, 1990.

[148]"An Empirical Evaluation of the Use of CASE Tools", S.Stobart, A.van Reeken, G.Low, J.Trienekens, J.Jenkins, J.Thompson, and D.Jeffery, *Proceedings of the 6th International Workshop on Computer-Aided Software Engineering (CASE'93)*, IEEE Computer Society Press, July 1993, p.81.

[149]"The Methods Won't Save You (but it can help)", Patrick Loy, *ACM SIGSOFT Software Engineering Notes*, **Vol.18**, **No.1** (January 1993), p.30.

[150]"What Determines the Effectiveness of CASE Tools? Answers Suggested by Empirical Research", Joseph Trienekens and Anton van Reeken, *Proceedings of the 5th International Workshop on Computer-Aided Software Engineering (CASE'92)*, IEEE Computer Society Press, July 1992, p.258.

[151]"Albert Einstein and Empirical Software Engineering", Shari Lawrence Pfleeger, *IEEE Computer*, **Vol.32**, **No.10** (October 1999), p.32.

[152]"Rethinking the modes of software engineering research", Alfonso Fugetta, *Journal of Systems and Software*, **Vol.47**, **No.2-3** (July 1999), p.133.

[153]"Evaluating Software Engineering Technologies", David Card, Frank McGarry, Gerald Page, *IEEE Transactions on Software Engineering*, **Vol.13**, **No.7** (July 1987), p.845.

[154]"Investigating the Influence of Formal Methods", Shari Lawrence Pfleeger and Les Hatton, *IEEE Computer*, **Vol.30**, **No.2** (February 1997), p.33.

[155]"Formal Methods are a Surrogate for a More Serious Software Concern", Robert Glass, *IEEE Computer*, **Vol.29**, **No.4** (April 1996), p.19.

[156]"The Realities of Software Technology Payoffs", Robert Glass, *Communications of the ACM*, **Vol.42**, **No.2** (February 1999), p.74.

[157]"Software failures, follies, and fallacies", Les Hatton, *IEE Review*, **Vol.43**, **No.2** (March 1997), p.49.

[158]"Applying Mathematical Software Documentation: An Experience Report", Brian Bauer and David Parnas, *Proceedings of the 10th Annual Conference on Computer Assurance (COMPASS'95)*, IEEE Computer Society Press, June 1995, p.273.

[159]"What's Wrong with Software Engineering Research Methodology", Franck Xia, *ACM SIGSOFT Software Engineering Notes*, **Vol.23**, **No.1** (January 1998), p.62.

[160]"Assessing Modularity in Trusted Computing Bases", J.Arnold, D.Baker, F.Belvin, R.Bottomly, S.Chokhani, and D.Downs, *Proceedings of the 15th National Computer Security Conference*, October 1992, p.44. Republished in the *Proceedings of the 5th Annual Canadian Computer Security Symposium*, May 1993, p.351,

[161]"The Sorry State of Software Practice Measurement and Evaluation", William Hetzel, *The Journal of Systems and Software*, **Vol.31**, **No.2** (November 1995), p.171.

[162]"Evaluating the Effectiveness of Z: The Claims Made About CICS and Where We Go From Here", Kate Finney and Norman Fenton, *The Journal of Systems and Software*, **Vol.35**, **No.3** (December 1996), p.209.

[163]"Rigor in Software Complexity Measurement Experimentation", S.MacDonell, *The Journal of Systems and Software*, **Vol.16**, **No.2** (October 1991), p.141.

[164]"The Mathematical Validity of Software Metrics", B.Henderson-Sellers, *ACM SIGSOFT Software Engineering Notes*, **Vol.21**, **No.5** (September 1996), p.89.

[165]"Experimental Evaluation in Computer Science: A Quantitative Study", Walter Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst Heinz, *The Journal of Systems and Software*, **Vol.28**, **No.1** (January 1995), p.9.

[166]"How Microsoft Builds Software", Michael Cusumano and Richard Selby, *Communications of the ACM*, **Vol.40**, **No.6** (June 1997), p.53.

[167] "Software Development on Internet Time", Michael Cusumano and David Yoffie, *IEEE Computer*, **Vol.32**, **No.10** (October 1999), p.60.

[168] "Extreme Programming Explained: Embrace Change", Kent Beck, Addison-Wesley, 1999.

[169] "Embracing Change with Extreme Programming", Kent Beck, *IEEE Computer*, **Vol.32**, **No.10** (October 1999), 70.

[170] "XP", John Vlissides, *C++ Report*, June 1999.