Cryptographic Verification of Test Coverage Claims

Prem Devanbu & Stuart G. Stubblebine
Information Systems and Services Research Center,
2b417, 600 Mountain Ave,
Murray Hill, NJ 07974, USA
AT&T Labs — Research
prem,stubblebine@research.att.com

Keywords: Testing, Verification, Cryptography, Components

February 12, 1997

Abstract

The market for software components is growing, driven on the "demand side" by the need for rapid deployment of highly functional products, and on the "supply side" by distributed object standards. As components and component vendors proliferate, there is naturally a growing concern about quality, and the effectiveness of testing processes. White box testing, particularly the use of coverage criteria, is a widely used method for measuring the "thoroughness" of testing efforts. High levels of test coverage are used as indicators of good quality control procedures. Software vendors who can demonstrate high levels of test coverage have a credible claim to high quality. However, verifying such claims involves knowledge of the source code. In applications where reliability and quality are critical, it would be desirable to verify test coverage claims without forcing vendors to give up valuable technical secrets. In this paper, we explore cryptographic techniques that can be used to verify such claims. Our techniques have some limitations; however, if such methods can be perfected and popularized, they can have an important "leveling" effect on the software market place: small, relatively unknown software vendors with limited resources can provide credible evidence of high-quality processes, and thus compete with much larger corporations.

1 Introduction

As the size, functionality, and complexity of software applications increase (*e.g.* the Microsoft Office^{TM} products are in the range of $O(10^6)$ lines) vendors seek to break applications into components (spell checkers, line breakers, grammar checkers etc.). Distributed component standards such as CORBA, ubiquitous networking, portable object-oriented platforms such as Java are also additional drivers of this trend. As a result, a vibrant market for software components is growing. The cost of entry into this market is low, and small vendors can be players. As the number and types of components proliferate, and smaller, newer vendors enter the market, there is a natural concern about quality.

Traditionally, systems with stringent quality requirements undergo a rigorous verification process, often under the auspices of third party *verification agents* [1, 17, 18]. One common testing technique used is *white box* testing; The goal is to ensure that a system has been adequately exercised during testing. In this approach, an abstraction of the system (*e.g.*, a control flow graph) is used to identify the parts of the system that need to be exercised. Each of these parts is called a *coverage point*, and the entire set is the *coverage set*. When a suite of tests can exercise the entire coverage set, it is called a *covering test suite* with respect to this coverage set. One popular criterion for adequate test coverage is that all basic blocks in the control flow graph have been exercised. Typically, test coverage is verified by building an "instrumented" version of the system; the instrumentation is placed in the appropriate parts of the system; the added instrumentation keeps records when executed; thus it is possible to verify that the necessary parts of the system have been exercised. We can abstract the situation as follows:

There is a system X, with source code S^X, out of which a (shipped) binary B_s^X is built. Also, from the source S^X, we can generate a coverage set C_γ^X (for some coverage criterion γ) as follows:

$$C_\gamma^X = c_1, c_2, \ldots c_n$$

Each of the c_i's refer to a coverage point. To cover these coverage points, it is necessary to develop a covering test suite T_γ^X that can exercise the coverage set for the criterion γ:

$$T_\gamma^X = t_1, t_2, \ldots t_m$$

such that for any given coverage point c, there is a t such that the execution of test t hits c. This is verified by first building an appropriately instrumented binary B_γ^X, and running the test suite T_γ^X.

A vendor who undertakes the cost of developing an adequate set T_γ^X for some stringent γ can reasonably expect that the system is less likely to fail[1] in the field due to undetected

[1] Providing, of course, that the system passes the tests!

faults in in system X [13]. Often, in fields with exacting reliability requirements (such as transportation, telecommunications or health) software users demand high quality standards, and expect vendors to use testing processes that achieve high levels of coverage with stringent coverage criteria. In such situations, in order to establish that the requirements have been achieved, a vendor may balk at giving a customer access to the entire source code, and all the test scripts so that coverage can be verified. This can also be done via a third party who is trusted by both the vendor and customer to operate according to well defined procedures. Typically, these procedures restrict the third party from revealing the source code. In either case, there are considerable risks, delays, and/or expense involved for the vendor; these may be acceptable in some situations, and not in others. Indeed a given situation (where test coverage has to be verified) comprising of a vendor, a piece of software and a customer can be analyzed by considering the following issues.

1. How much information (source code, coverage set, test scripts etc.) is the vendor willing to reveal?
2. Is there a third party trusted by both the vendor and the customer?
3. How much time/money can be spent on test coverage verification?
4. To what level of confidence does the customer want to verify test coverage?

These questions represent different goals, which are sometimes conflicting; any practical situation will surely involve trade-offs. Different trade-offs will be acceptable under different circumstances. This paper describes one set of possible solutions that cover some typical cases; we certainly have not covered all the possibilities.

Much of the work described in this paper is aimed at reducing the amount of information that the vendor has to disclose, *viz.,* source code, symbol tables, and (particularly) the test cases. Source code is clearly the most valuable information; the symbol table, which is typically embedded in the binary, can be a valuable aid to reverse engineering. A large and exhaustive set of test scripts is also valuable information regardless of whether the source code or symbol table is known. Generating such a test set involves careful analysis of the requirements, as well as familiarity with the design and implementation of the system. While the most typical functions of the system may be widely known, a complete set of test cases would have to exercise unusual situations, create various feature interactions, cause exceptions to be raised etc. A comprehensive consideration of all these special cases is a valuable piece of intellectual property that demands protection. Indeed, there are vendors who make it their business to develop and sell comprehensive test suites [15, 25]. Our goal is to protect this information, while allowing the vendor to make credible test coverage claims.

Caveats and Assumptions. First, we assume that vendors are strongly motivated by market forces[2] to provide the highest quality software. In this context, test coverage verification protocols simply provides a way of convincing customers that the vendors have really undertaken the rigorous process of finding tests to cover all or most of the coverage points. Second, coverage testing (like any other testing method) is not perfect. All faults may not be revealed by testing every coverage point. Contrariwise, vendors may contrive to create test sets that achieve coverage while concealing faults. Coverage levels may also be artificially boosted by adding spurious code. These issues are dealt with greater detail in § 7, and some strategies for dealing with these problems are suggested; however, a significant practical disincentive to such practices is the market demand for high quality software, and the high cost of cancelled sales, refunds and/or fixing bugs in the field. Third, we assume that all parties involved in coverage verification protocols have access to a test oracle[26] that decides, at low cost, whether the output for any test case is right or not.

Finally, we have little to say about techniques for defeating reverse engineering; our focus is more to protect the secrecy of the largest possible number of test cases (which represent a significant investment by the vendor), while allowing the use of common methods for repelling reverse engineering, *viz.*, shipping only binaries without symbol tables. Without a physically protected hardware platform, a determined adversary can reverse-engineer a good deal of information about software. Techniques and tools to support reverse engineering are an area of active research. In fact, previous research [3, 5, 19, 22] demonstrates how control-flow graphs, profile information, compiler-generated binary idioms, and even slices can be derived by analyzing and instrumenting binaries. Decompilation (converting binary to source code) and binary porting (converting binaries from one machine architecture to another) are typical goals of binary analysis. We say more on this in the conclusion.

We employ several different cryptographic techniques in this paper. We begin with a brief description of these techniques; readers familiar with cryptography may skim this section

2 Summary of Cryptographic Methods

Different combinations of customers, vendors and software require different solutions. In this paper, we apply several cryptographic techniques to address some common scenarios that may arise in practice. We now briefly describe the techniques we have used in our work; more complete descriptions can be found in [23]. All of these techniques are used to build assurance in the customer ($\mathcal{C}$) that the vendor($\mathcal{V}$) has high test coverage, while attempting to protect $\mathcal{V}$'s secrets. For the descriptions that follow, we use some public/private key pairs: assume $K_{\mathcal{P}}^{-1}$ is a good private signing key for the individual $\mathcal{P}$ and $K_{\mathcal{P}}$ is the corresponding public signature verification key for the lifetime of the test coverage verification process.

[2]Different kinds of software sell in different markets, so quality needs do differ. For example, software components used in servers need to be of much higher quality than software used in clients

1. **Cryptographic Signatures** Given a datum δ, $\sigma_{K_{\mathcal{P}}^{-1}}(\delta)$ is a value representing the signature of δ by $\mathcal{P}$, which can be verified using $K_{\mathcal{P}}$. Note that $\sigma_{K_{\mathcal{P}}^{-1}}(\delta)$ is not the same as encrypting δ with $K_{\mathcal{P}}$; typically, it is just an encrypted hash value of δ. In particular, δ cannot be reconstructed from $\sigma_{K_{\mathcal{P}}^{-1}}(\delta)$ with $K_{\mathcal{P}}$; however, given δ, the signature can be verified. This is a way for $\mathcal{P}$ to "commit" to a datum in an irrevocable manner. Bids for a sealed-bid auction, for example, could be submitted as signatures of cash amounts prior to the deadline; on the day of the auction, the bids are "opened" by each participant revealing the actual amount, which can verified by comparison to the signature.

2. **Trusted Third Party**. If there is a trusted third party (denoted by $\mathcal{T}$) that can act as a "buffer" between $\mathcal{C}$ and $\mathcal{V}$, $\mathcal{T}$ can use information supplied by $\mathcal{V}$ to assure $\mathcal{C}$ about $\mathcal{V}$'s testing practices, while protecting $\mathcal{V}$'s secrets. $\mathcal{T}$ can be relied upon both to protect $\mathcal{V}$'s secrets, and operate fairly, without bias. Note that $\mathcal{T}$ can certify a datum δ by appending the signature $\sigma_{K_{\mathcal{T}}^{-1}}(\delta)$.

3. **Trusted Tools**. To verify any type of white box coverage, it is necessary to provide information extracted from source files. Tools (trusted by all parties involved) that extract exactly the information required for coverage verification, and adjoin a cryptographic signature with a published key, can be used to extract trusted coverage information (providing the tools are not tampered with). (Later, we address limitations of using trusted tools.)

4. **Random Sampling**. Consider an adversarial situation where a party $\mathcal{P}_a$ claims that a proportion ρ of the members of a set σ have a property π, and $\mathcal{P}_a$ is willing to reveal the proof that π holds for only *some* members of σ. A skeptical party $\mathcal{P}_b$ can choose a member σ_i, of this set at random, and challenge $\mathcal{P}_a$ to demonstrate that $\pi(\sigma_i)$ holds. After several such trials, $\mathcal{P}_a$ can estimate ρ subject to confidence intervals. This technique can be used to estimate test coverage levels.

5. **Autonomous Pseudo-Random Sampling**. The random challenges discussed above can be performed by $\mathcal{P}_a$ herself, using a published pseudo-random number generator $\mathcal{G}$ with a controlled seed. If $\mathcal{G}$ can be trusted by adversaries to produce a fair random sample, $\mathcal{P}_a$ can publish a credible, self-verified estimate of ρ. Notice that successive fragments of a very long hash string produced by a one-way hash function can also be used to generate samples.

3 Exploring the Problem

To explore this problem, we exhibit a series of methods that are applicable in different situations as we discussed earlier.

We start with the basic third party method, that is the currently used approach to verifying test coverage while simultaneously protecting a vendor's secrets. We then gradually introduce cryptographic techniques to protect the vendor $\mathcal{V}$'s secrets while simultaneously making it difficult for $\mathcal{V}$ to cheat. Our focus of security is not at the session layer but at the application layer. Thus we assume the testing protocols occur over secure channels such as those provided by Secure Socket Layer (SSL [12]). In the following scenarios, $\mathcal{V}$ refers to the vendor who claims to have achieved γ test coverage on a system X, and $\mathcal{C}$ refers to the skeptical customer who wants to be convinced.

Basic Third Party Method

1. $\mathcal{V}$ sends $\mathcal{T}$, a trusted third party, the source code S^X, and the test suite T^X_γ, and a description of the coverage criterion γ.

2. $\mathcal{T}$ builds B^X_γ from S^X, and constructs the coverage set C^X_γ.

3. $\mathcal{T}$ runs the test suite T^X_γ against B^X_γ and verifies that the suite hits the coverage set.

4. $\mathcal{T}$ tells $\mathcal{C}$ that the γ coverage criterion for X has been met by the test suite T^X_γ.

5. $\mathcal{V}$ ships B^X_s to $\mathcal{C}$, with the claim that $\mathcal{T}$ has verified coverage.

This approach is weakened by some questionable assumptions. First, there may not be a trusted third party acceptable to both $\mathcal{T}$ and $\mathcal{C}$, and to whom $\mathcal{V}$ is willing to to reveal the source and the test suite. There may be a commercial incentive for $\mathcal{T}$ to exploit this information in some legal or undetectable ways; a truly trustworthy $\mathcal{T}$ may be hard to find. Second, $\mathcal{T}$ has to replicate V's entire build apparatus, (*viz.,* compilers, libraries, and auxiliary tools) and execute the build (which may be complex). There is also a glaring weakness: $\mathcal{V}$ could have sent an older, well tested version to $\mathcal{T}$, and ship a newer version of X, with additional functionality (but with the same old test suite) to $\mathcal{C}$ (One way to avoid this problem is to have $\mathcal{T}$ also build an *uninstrumented* binary version of X, from the same source, sign it, and have this be the version that $\mathcal{V}$ ships to her customers). Finally, there is a subtle and inevasible attack whereby $\mathcal{V}$ can try to find "fault-hiding" test cases, which we discuss later[3] (§ 7, page 19).

The *Basic Method* requires significant effort of $\mathcal{T}$. This will probably increase $\mathcal{V}$'s costs and cause delays. However, this method can be used when there is a suitable $\mathcal{T}$, the costs are acceptable, and $\mathcal{C}$ wants an accurate estimate of test coverage. It would be better if $\mathcal{V}$ could just ship $\mathcal{T}$ the binary and not the source. This gives $\mathcal{V}$ a few opportunities to cheat, some of which can be addressed; the remaining ones may sometimes be acceptable. This is the motivation for the next section.

[3]All the protocols we describe in this paper are subject to this attack, although there are mitigating considerations.

3.1 Simple Binary Patching approaches

We now explore some approaches that use just the binary to verify test coverage claims. Note that we usually mean binaries built with full symbol tables, i.e., compiled with the "`-g`" option in most `C` compilers; the issue of eliminating (or abridging to the extent possible) this symbol table is visited in the next section.

Protocol 1

1. From S^X, $\mathcal{V}$ constructs the system B_s^X and the set of coverage points C_γ^X.
2. $\mathcal{V}$ sends $\mathcal{T}$: B_s^X, C_γ^X and the test suite T_γ^X, with the locations in the source files corresponding the coverage points.
3. $\mathcal{T}$ uses a binary instrumentation tool, either interactive (*e.g.,* a debugger, or batch-oriented (*e.g.,* ATOM [24], EEL [19]) to instrument B_s^X, using the line number/file information sent by $\mathcal{T}$, and the symbol table information embedded in B_s^X. For example, a debugger, can set break points at the appropriate locations (e.g., line numbers in files).
4. $\mathcal{T}$ runs the test suite T_γ^X against the instrumented binary, and verifies coverage level. For example, the debugger can be set to delete each break point when "hit". The coverage level is verified by the number of remaining breakpoints.
5. $\mathcal{T}$ signs the file B_s^X (perhaps after extracting the symbol table from B_s^X) and sends $\sigma_{K_\mathcal{T}^{-1}}(B_s^X)$ to $\mathcal{V}$.
6. $\mathcal{V}$ verifies the signature on $\sigma_{K_\mathcal{T}^{-1}}(B_s^X)$, using $K_\mathcal{T}$; then, $\mathcal{V}$ sends B_s^X and $\sigma_{K_\mathcal{T}}(B_s^X))$ to $\mathcal{C}$.

This method improves upon the *Basic Method* in a few ways: first, the source is not revealed to $\mathcal{T}$. Second, $\mathcal{T}$ does not recreate $\mathcal{V}$'s build environment. Third, $\mathcal{T}$ works with the shipped version of the software, so he can directly "sign" it after he has verified the coverage claims. Finally, $\mathcal{T}$'s work is reduced; rather than building instrumented and uninstrumented versions, he only has to instrument the binary, which is not harder than the link phase of a build. So presumably, *Protocol* 1 would be cheaper, faster and less error-prone than the *Basic Method.*

A major weakness in *Protocol* 1 is that $\mathcal{V}$ is trusted to build an accurate coverage set C_γ^X. $\mathcal{V}$ can cheat and build a smaller coverage set, and thus convince $\mathcal{C}$ that he has a higher test coverage than he really does. However, he may have difficulties if he ships faulty software while falsely claiming high levels of test coverage. If the software fails frequently in the field, he could be called upon to reveal the source code to a trusted third party, and prove that his coverage analysis was accurate, and that the shipped binary was built with the same source code.

Even so, $\mathcal{V}$ still has to reveal a lot to $\mathcal{T}$: the entire coverage set and the entire test set. If X is a very popular and/or difficult system to build, this information may be very valuable, and $\mathcal{T}$ may quietly sell this information to $\mathcal{V}$'s competitors. The next protocol reduces the amount of information $\mathcal{V}$ has to reveal, while not increasing $\mathcal{V}$'s opportunities to cheat; *Protocol* 3 goes further in restricting $\mathcal{V}$'s ability to construct spurious coverage sets.

Protocol 2

1. $\mathcal{V}$ builds B_s^X from S^X, creates C_γ^X, $\sigma_{K_\mathcal{V}^{-1}}(i, C_\gamma^X(i))$, $for\ i = 1 \ldots \mid C_\gamma^X \mid$. and $\sigma_{K_\mathcal{V}^{-1}}(i, C_\gamma^X(i), t_i, r_i)$, $for\ i = 1 \ldots \mid C_\gamma^X \mid$. t_i is the corresponding test case[4], and r_i is a random number inserted to confound attacks which attempt to guess the content before it is revealed.
2. $\mathcal{V}$ sends $\mathcal{T}$ all the above signatures and B_s^X.
3. $\mathcal{T}$ challenges $\mathcal{V}$ with some small number l of the coverage points.
4. For each challenge, $\mathcal{V}$ reveals $C_\gamma^X(i), t_i$, and r_i. $\mathcal{T}$ can cross-check with the signatures delivered above.
5. $\mathcal{T}$ uses the coverage point location information, instruments B_s^X and runs the supplied test case to check coverage.
6. $\mathcal{T}$ signs the binary and sends $\sigma_{K_\mathcal{T}^{-1}}(B_s^X)$ to $\mathcal{V}$.
7. $\mathcal{T}$ archives the testing procedure including all the information sent in step 2.
8. $\mathcal{V}$ ships $(B_s^X, \sigma_{K_\mathcal{T}^{-1}}(B_s^X))$ to $\mathcal{C}$.

We shall discuss the commitments in step 1 presently; for now, we focus on steps 3-5. Here $\mathcal{T}$ randomly picks a small number of challenges to $\mathcal{V}$. This method betters *Protocol* 1 in one important way: $\mathcal{V}$ reveals only *some* coverage points, and *some* tests. Since $\mathcal{V}$ cannot predict which coverage points $\mathcal{T}$ will pick, he must prepare tests to cover most of them. $\mathcal{V}$ packages the test cases with the corresponding coverage point, and a random number; this discourages $\mathcal{T}$ from brute-force searching for test cases and coverage points using the signatures from step 1; he reveals these on being challenged. With a small number of random challenges, $\mathcal{T}$ can bound $\mathcal{V}$'s test coverage. If $\mathcal{V}$'s responses cover a proportion p_s of $\mathcal{T}$'s challenges, $\mathcal{T}$ can estimate $\mathcal{V}$'s actual coverage p (using Hoeffding's version of Chernoff bounds for the "positive tail" of a binomial distribution, see [16], pp 190-191):

$$P(p_s - p \geq \epsilon) \leq e^{\frac{-n\epsilon^2}{2p(1-p)}}\ when\ p \geq 0.5 \tag{1}$$

[4] Or a string indicating the lack of a test case for this coverage point.

For a 95% confidence level, we can bound ϵ:

$$e^{\frac{-n\epsilon^2}{2p(1-p)}} \leq 0.05$$

$$\epsilon = \sqrt{\frac{2ln(\frac{1}{0.05})p(1-p)}{n}}$$

Clearly, as n goes up, $\mathcal{T}$ gains confidence in his estimate p_s. Thus, at the 95% confidence level, $\mathcal{T}$ can reasonably conclude that an estimate of $p = 0.95$ is no more than 0.09 *too high* with about 25 samples. Experimental work [20, 13] indicates that branch coverage levels in the range of 80-90% have a high likelihood of exposing faults in the software. Estimated coverage levels in this range can give a customer high confidence that the $\mathcal{V}$'s testing has exposed a good number of faults. so it is in $\mathcal{V}$'s interest to allow $\mathcal{T}$ the largest possible number of challenges, with a very high expected p. Clearly, this will incent $\mathcal{V}$ to achieve very high coverage levels. It is also important to keep in mind that this "random sampling" places limits on the accuracy of $\mathcal{T}$'s estimate of $\mathcal{V}$'s test coverage; essentially, we are trading off the amount of information disclosed for $\mathcal{T}$. In cases where this trade-off is acceptable, this technique is applicable.

There is a threat to the validity of the confidence level calculation described above: the sampling process is not really a series of independent events. Executions of coverage points (blocks, branches, or functions) are often strongly correlated. Agrawal [2] shows how to determine the set of statically independent coverage points from the control flow graph by computing the post-dominator and pre-dominator trees of basic blocks. The leaves of such trees could be used to form an indepedent set of coverage points. However, there could still be dynamic dependencies between blocks which cannot be feasibly determined by static analysis. One way to quantify these effects is to use sub-samples and statistical tests to determine if the estimates are stable. This is can be complex, may require more samples, and can enlarge the confidence intervals. In a later protocol, we commit ahead of time to a coverage level; this avoids sampling difficulties.

Now we return to the two sets of commitments (signatures) in step 1. The first set reveals a signature on each element of the coverage set and the corresponding test case; $\mathcal{T}$ needs only to know the actual coverage points for his challenges. $\mathcal{V}$ pads each signature computation with a random number to securely hide the association of the test case and the coverage point for the unchallenged coverage points. The second set of signatures, on the unpadded coverage points can be used to check that the coverage points are indeed different. If $\mathcal{V}$ dares to ship software with a spurious (smaller) coverage set, and the software fails badly, $\mathcal{V}$ may be forced to reveal the source code, (as discussed above) to $\mathcal{T}$ who can compute the coverage points, build the binary, and compare the coverage points against the signature committed to in Step 1. If $\mathcal{V}$ was cheating, he could be caught in a lie. This may be sufficient to dissuade large vendors with established brand names, but not smaller ones.

Protocol 3

1. $\mathcal{V}$ uses a trusted coverage analysis tool, which generates the covering set. This tool analyzes the source code, to find the requisite coverage points for γ and generates a set of individually signed coverage points.

2. Proceed as *Protocol* 2, with $\mathcal{V}$ sending $\mathcal{T}$ his signatures of the coverage points, and $\mathcal{T}$ randomly challenging $\mathcal{V}$.

With this approach, $\mathcal{V}$ uses a trusted coverage analysis tool to generate the coverage set C_γ^X which is cryptographically "signed" by the tool. Now, $\mathcal{T}$ can verify the signature on the coverage set, and be assured that $\mathcal{V}$ has not generated a spurious (presumably smaller) coverage set. The elements of this coverage set are sealed as in *Protocol* 2 with test cases, and random numbers, and the coverage verification proceeds with coverage point challenges from $\mathcal{T}$ to $\mathcal{V}$. The main advantage of this approach is an increased level of trust in the set of coverage points generated by $\mathcal{T}$; the disadvantage is the risk that the trusted tool may be compromised.

As noted in the beginning of this section back on page 7. The "binary-patching" approaches used in *Protocols* 2 and 3 both assume that the binary is shipped to $\mathcal{T}$ with the symbol table information intact. We now explore the implications of this, and approaches to eliminating the symbol table.

3.2 Verifying coverage claims without symbol tables

Binary instrumentation tools such as debuggers use the symbol table in the binary to locate machine instructions corresponding to source level entities. This information is used by debuggers to set breakpoints, to inspect the elements of a structure by name, etc. The symbol table has information about global symbols (functions/entry-points and variables), source files, and data structures (sizes, elements etc.). With this information, one can reverse-engineer a good deal of information about the implementation. Typically, symbol tables are stripped out of delivered software; as we discussed earlier, $\mathcal{T}$ could easily perform this stripping after verification. Sometimes, the $\mathcal{V}$ may balk at revealing this symbol table to $\mathcal{T}$. Can we verify test coverage without revealing the symbol table? *Protocol 4* addresses this issue. It uses a *pure binary patcher*, that is trusted by the parties involved. This tool, given actual binary offsets, can insert break points to verify test coverage. Interactive debuggers such as `gdb` can readily perform such a task; a command such as `break 0x89ABC` to `gdb` will set a break point at the machine address `0x89ABC` in the program. A batch-oriented tool like `EEL` [19] can also be used. Such a tool will be used by $\mathcal{T}$ to insert instrumentation at coverage points, and verify coverage. We also use *a binary location finder (blf)*, which uses the symbol table to find binary addresses for the coverage points. For example, the "`info line` *file:line*" command in `gdb`, for a given a line number in a file, calculates the corresponding binary position and size. The blf will be used by $\mathcal{V}$ to identify coverage points by physical addresses in the binary; this tool would have to be trusted by $\mathcal{T}$, and would cryptographically "sign" its output.

Protocol 4

1. $\mathcal{V}$ uses a trusted coverage analysis tool (which signs its output) and generates the coverage set C_γ^X.
2. $\mathcal{V}$ then uses a binary location finder (blf) to find binary locations corresponding to each element of C_γ^X. Call this set $f_{blf}(C_\gamma^X)$. We assume that the blf signs its output; we can also have blf verify the signature of the coverage analysis tool from the previous step.
3. The protocol proceeds as before, with $\mathcal{V}$ sending $\mathcal{T}$ signatures of coverage points, random numbers, and test cases. $\mathcal{T}$ then conducts random challenges as before; however, in this case, he uses a pure binary patcher to insert instrumentation and verify coverage.

This approach reduces the amount of information about the symbol table and the source file, that is revealed to $\mathcal{T}$. However, it increases the reliance on "trusted tools". In addition to the coverage analyzer used above, $\mathcal{T}$ needs to trust the binary location finder. This may not always be acceptable.

Trusted tools which "sign" their output are threatened by commercial incentives to cheat. The coverage analysis tool could be modified and made to sign spurious, small coverage sets; a binary location finder could be made to always point to false locations; the "secret" key could be stolen from the tool and used to sign false data. One way to avoid this problem is through the use of a physically secure co-processor. Various forms of these processors are available in tamper-proof enclosures, running secure operating systems; they are expected to become ubiquitous in the form of *smart cards* [27, 14], which are expected to become quite powerful in a few years. The physical enclosure is a guarantee of integrity; if tampered with, the processor will erase its memory and cease to function. We envision placing customizable, trusted source code analyzers [7, 6] in secure co-processors. Such a device can be installed as a co-processor at the site of the vendor; it can be sent an authenticated message which describes the type of analysis to be conducted. The smart-card resident tool can perform the analysis, and sign the results. The signature verifies that the results were created by trustworthy software resident in a secure machine, even at a potentially hostile site.

3.3 Eliminating coverage analysis tools

Such physically secured source analysis tools, however, are not yet available; we now suggest an approach to eliminate the dependence on the source altogether. However, there are complications with this approach. In the verification protocol listed below, we use basic-block coverage as an illustration; this approach can be extended to some other coverage models. The approach used here depends upon analysis of the binary, which $\mathcal{T}$ has access to. Given an instruction at an arbitrary location in the binary, and a knowledge of the instruction set of the architecture, it is possible bound the basic block containing that instruction. This property can be used for verifying basic block coverage.

Protocol 5

1. $\mathcal{V}$ sends to $\mathcal{T}$ the binary B_s^X, and $\sigma_{K_{\mathcal{V}}^{-1}}(i, C_\gamma^X(i), t_i, r_i),\ for\ i = 1 \ldots \mid C_\gamma^X \mid$.
2. $\mathcal{T}$ chooses a random location, l, within the binary.
3. $\mathcal{V}$ reveals the corresponding test case, coverage point, and random pad.
4. $\mathcal{T}$ can set a breakpoint at l using his favorite instrumentation technique, and execute the test to verify the coverage.
5. Repeat the above steps for the desired number of challenges and proceed as before.

With this approach, we don't need to analyze source code to determine the set of coverage points Here, $\mathcal{T}$ (presumably) chooses random points in the executable, and it is up to $\mathcal{V}$ to provide the coverage evidence. This can be done by $\mathcal{V}$, since he has access to both the binary symbol table and the source code. Given a machine address, $\mathcal{V}$ can identify the corresponding source line easily, using a debugger (*e.g.,* with `gdb`, the "`info line` **addr*" will translate a given machine address to a source file and line number). If the $\mathcal{V}$ has previously developed a good covering test set, and verified his coverage levels, he can readily identify the specific covering test using the file/line number and data from his coverage verification process.

However, there are several difficulties with this approach; they all have to do with "testability" of the binary. Finding test cases to cover a given location in a binary can be harder than with source code, specially when:

1. the location occurs in code generated by the compiler in response to a complex source language operator (e.g, inlined constructors or overloaded operators in C++); this code may contain control flow not present in the source, or when
2. it occurs in unreachable code generated by the compiler, or when,
3. it occurs in off-the-shelf (OTS) software incorporated by vendor in his product, for which the vendor has no tests;

There are approaches to dealing with some of these issues. Generated code often corresponds to idioms; this information can be used to find test cases. Sometimes generated code may contain additional control flow that represent different cases that can occur in the field, and

$\mathcal{V}$ can legitimately be expected to supply covering test cases. When the generated code is genuinely unreachable [21], $\mathcal{V}$ can claim it as such, and supply source code that $\mathcal{C}$ can compile to create similar binaries. Occurrences of dead code in the binary are really bugs in the compiler, and are likely to be rare.

Even when a challenge happens to fall within the bounds of an OTS binary, $\mathcal{V}$ has several options for test coverage verification. If the OTS is a well-known, reputable, piece of public domain software, he can simply identify the software, and $\mathcal{C}$ can download the software and do a byte-comparison. Even if the OTS in not public, signatures can be obtained from V_{ots}, the vendor, for comparison. If the OTS is not well known, but has been independently subject to test coverage verification, then evidence of this verification can be provided to $\mathcal{C}$. Another approach is for $\mathcal{V}$ to relay challenges to V_{ots}, who may be able to handle them, and pass her responses back to $\mathcal{C}$.

Binary-based random challenges can be performed without revealing source code, or symbol tables, and without resorting to trusted tools; the trade-off here is that the mapping to source code may be non-trivial for some parts of the binary; for this and other reasons, it may be hard to construct an exercising test case. As binary decompilation tools [4] mature and become more widely available, they can be used by customers to build confidence about areas of the binary that $\mathcal{V}$ claims to be non-testable for the reasons listed above.

4 Towards Eliminating the trusted third party

All the approaches described above rely upon a trusted third party ($\mathcal{T}$). However, it may sometimes be undesirable to use $\mathcal{T}$, for reasons of economy or secrecy.

A naive approach to eliminating $\mathcal{T}$ would be for $\mathcal{V}$ to rerun the verification protocols described above with each software buyer. This is undesirable. First, repeating the protocol with each buyer is expensive and slow. Second, $\mathcal{V}$ would reveal information to different, potentially adversarial parties who might collude to reverse engineer secrets about B_s^X. Third, since a potentially adversarial buyer is involved, there is a risk that the challenge points might be deliberately chosen to expose the most valuable information about $\mathcal{V}$'s software. For example, if $\mathcal{T}$ was forced to reveal (on a challenge from a customer c_1) some test cases that pertained to handling some unusual or difficult case in the input domain), other customers might collude with c_1 to probe other points in the same area of the binary to expose $\mathcal{V}$'s implementation/design strategies for dealing with some difficult cases.

Re-examining *Protocols* $1 \ldots 5$ listed in the previous section, it becomes clear that the main role played by $\mathcal{T}$ is choosing the challenge coverage points; we eliminate his role using autonomous pseudo-random sampling.

Protocol 6

1. $\mathcal{V}$ prepares the binary B_s^X, and $\sigma_{K_{\mathcal{V}}^{-1}}(i, C_\gamma^X(i), t_i, r_i),\ for\ i = 1 \ldots \mid C_\gamma^X \mid$.

2. $\mathcal{V}$ computes a well-known, published one-way hash function of B_s^X to yield a *location control string*, $\mathcal{L}$. Successive byte groups of $\mathcal{L}$ are used to derive locations $l^1, \ldots, l^j$.

3. For each l_i, $\mathcal{V}$ reveals the test cases, random numbers and coverage points; call each revelation $\mathcal{R}_i$.

4. After some set of challenges l_i, $\mathcal{V}$ stops, and packages the $\mathcal{R}_i$'s and the $\sigma_{K_{\mathcal{V}}^{-1}}(i, C_\gamma^X(i), t_i, r_i)$'s, along with his release of B_s^X

5. $\mathcal{C}$ verifies test coverage by repeating the generation of the location control string, and checking the corresponding revelations by $\mathcal{V}$ for coverage.

Protocol 6 offers several advantages. We have eliminated the "middleman", $\mathcal{T}$, thus saving time and money. This approach is also advantageous where secrecy is involved. Instead of $\mathcal{T}$, a one-way hash function now drives the choice of the challenges. $\mathcal{V}$ cannot control the value of the string S_{vc}; a customer can easily verify the value of the location control string using the delivered software and the public hash function. Furthermore, there is no need for $\mathcal{V}$ to repeat this process with each customer. There is no risk that customers might collude and pool information.

There is, however a plausible risk in the above scenario: since $\mathcal{V}$ has control over the input to the hash string, he could automatically repeat the following:

1. Compute $\mathcal{L} = hash(B_s^X)$.

2. If a very large subset of the resulting locations $l_1 \ldots l_n$ are not covered, stop. Otherwise,

3. Generate another binary $B_{s\,1}^X$ by padding B_s^X with null instructions. Go to step 1

This amounts to repeated Bernoulli trials drawn from the set of coverage points; after some trials, $\mathcal{V}$ could find an $\mathcal{L}$ that artificially boosts his coverage ratio. To avoid this attack, we need a way to "monitor" his trials.

Protocol 7

We introduce a trusted "general purpose" archive, $\mathcal{T_A}$ to record the fact that $\mathcal{V}$ initiates testing. We assume $\mathcal{T_A}$ archives *all* information submitted to it, and can provide signed responses to queries concerning archive contents and history.

1. When $\mathcal{V}$ is ready to verify coverage, he "registers" with $\mathcal{T}$ 1) a query of historical usage of the archive 2) identifying string I of the program to be tested 3) $\sigma_{K_{\mathcal{V}}^{-1}}(B_s^X, I)$, and 4) $\sigma_{K_{\mathcal{V}}^{-1}}(i, C_\gamma^X(i), t_i, r_i),\ for\ i = 1 \ldots \mid C_\gamma^X \mid$.

2. $\mathcal{T}_\mathcal{A}$ acknowledges $\mathcal{V}$'s registration with his registration history and a signature of the history. (For brevity, let's assume that the returned signature is random for each query response even if the query is the same e.g., a parameter of the the signature function may include a random number).

3. $\mathcal{V}$ follows a procedure like Protocol 8 except that: in step 2) $\mathcal{V}$ computes the location control string by seeding a well-known, published pseudo-random number generator using the signature returned by $\mathcal{T}_\mathcal{A}$. In step 4), $\mathcal{V}$ also packages the identifying string, and the historical usage of $\mathcal{V}$ signed by $\mathcal{T}_\mathcal{A}$. In step 5, $\mathcal{C}$ uses the historical information from the archive to determine the $\mathcal{V}$'s use of the archive. Also, $\mathcal{C}$ verifies $\mathcal{T}_\mathcal{A}$'s signature on the testing profile and the signature that seeded the pseudo-random process.

Our approach involves a string identifying the system that is also archived. When releasing a trace of the test coverage verification protocol, this identifying string is included in the release, and is input to the process that generates the location control string. $\mathcal{V}$ is free to generate many location control strings, but each time, step 1 above requires him to register with $\mathcal{T}_\mathcal{A}$. Each registration become a part of his history that is available to customers. Given a release from $\mathcal{V}$ that incorporates a system (with identifying string), and a verification trace, $\mathcal{C}$ can query $\mathcal{T}_\mathcal{A}$ to find registrations from $\mathcal{V}$. Let us assume that there are m such registrations with similar identification strings. Further assume that the location control string checks out, and that $\mathcal{V}$ presents the most favorable value of p_s and includes n challenges, of which a proportion p are found to be covered by running the test cases provided by $\mathcal{V}$. Given this data, $\mathcal{C}$ can estimate the the probability that p differs from actual coverage level p_a by ϵ, using m disjoint occurrences in the inequality 1,Page 8):

$$P(p_s - p \geq \epsilon) \leq m e^{\frac{-n\epsilon^2}{2p(1-p)}}$$

Therefore, for a 95 % confidence level, we can estimate the error ϵ as follows:

$$\epsilon = \sqrt{\frac{2ln(\frac{m}{0.05})p(1-p)}{n}}$$

Recall from Page 8 that with about 25 samples, one can achieve a 95% confidence level with an error ceiling of 0.09 on an estimate of $p = 0.95$. With 5 registrations, this error ceiling increases to 0.12. With more registrations, the $\mathcal{C}$'s faith in the actual coverage results from a particular verification trace is limited. Clearly, it is in the interests of $\mathcal{V}$ to *a*) delay registration for test coverage verification until he has already developed enough tests for a high coverage level, *b*) provide clear, distinguished identifying strings for each of his software offerings so that customers don't conflate registrations for different products and *c*) limit the number of registrations for any particular piece of software[5]. For a vendor whose development processes already achieve high levels of coverage, this not a significant burden. Finally, $\mathcal{T}_\mathcal{A}$'s services are quite limited, fully automatable, and thus should be inexpensive.

[5]Unless, of course, he reveals the verification trace for each of them, in which case the customers can get very tightly bounded estimates of his coverage level.

5 Committing and Enforcing an Upper Bound on Test Coverage

The protocols described up to this point have a weakness: $\mathcal{V}$ may get lucky and demonstrate a higher test coverage then he actually has. This is inherent to challenges based on random sampling. We now describe a technique that requires the vendor to assert an upper bound on test coverage. This technique can be used in conjunction with any of the protocols described above. With this approach, the vendor can be caught cheating if he is called upon to reveal a test case corresponding to a coverage point which he did not account for as a untested coverage point. However, the vendor is not forced to reveal potentially sensitive information about exactly which coverage points have no test cases. We present the technique here as a series of steps that can be interleaved into the protocols described above.

1. $\mathcal{V}$ commits to untested coverage points by sending $Hash(i, r_i, C_i), i = 1 \ldots N_{nt}$ for each coverage point not tested (where r_i is chosen at random by $\mathcal{V}$). Using this, the $\mathcal{C}$ or $\mathcal{T}$ can compute the upper bound on test coverage claims.
2. When the vendor is called upon to reveal a test point for which it does not have a test case, the vendor reveals r_i and i, the reference to the particular hash in the first step. The tester can recompute the hash of the tuple i, r_i, and C_i and compare it to the commitment of the untested coverage points.
3. If testing results with numbers higher than the coverage claims, the results are decreased to the upper bound.

In step 1, $\mathcal{V}$ commits to all the coverage points which are admittedly not covered by test cases. From this information $\mathcal{C}$ (or $\mathcal{T}$, as the case might be) can determine an upper bound on the actual coverage ratio. For example, in the case of Protocols 6 and 7, $\mathcal{C}$ can determine the coverage set by analysis of the binary; if the size of this set is N_{cs}, he can bound the coverage ratio as

$$\frac{N_{cs} - N_{nt}}{N_{cs}}$$

Step 1 can be done at the same time as the first step in (*e.g.,*) Protocols 6 and 7. Step 2 above is basically an extension to the random challenge step in many of the protocols. Given a random challenge, the vendor may or may not have a test case; if he does, the protocols work as described earlier. In the case where there is no test case, he reveals r_i and i, thus "checking off" one of the uncovered coverage points committed in step 1. If $\mathcal{V}$ is unable to reveal a test case, or an r_i, i pair, he is caught in a lie. Finally, in Step 3 above, the tester can compare his estimate from the random sample to the *a priori* upper bound computed above.

In this technique, $\mathcal{V}$ makes a clear claim about the proportion of tests he has coverage for, and the total number of tests. The purpose of the random trials therefore is not to narrow the confidence intervals around an *estimate* the value of the coverage ratio, but just to make sure $\mathcal{V}$ is not lying. With each trial, there is a probability that he will be caught; this increases with the number of trials. To model this analytically, assume that there are N total coverage points, and $\mathcal{V}$ is lying about l of those. i.e., for l of those he has no coverage points for, but he is trying to pretend he does. Denote the fraction $\frac{l}{N}$ by f. On any one trial, the chance that he will be caught is f, and that he will sneak through is $1 - f$. After n trials, the probability that he will escape[6] is

$$(1-f)^n$$

Let's bound this above by ϵ:

$$(1-f)^n \leq \epsilon$$

$$i.e.\;\; n\; log\; (1-f) \leq log\; \epsilon$$

$$i.e.,\;\; n\; \leq \frac{log\; \epsilon}{log\; (1-f)}$$

With $f = 10\%$, (i.e., $\mathcal{V}$ is lying about one-tenth of the coverage points), there is a 95% chance (i.e., $\epsilon = 0.05$) that $\mathcal{V}$ will be caught after roughly 28 random challenges. Again, it behooves $\mathcal{V}$ to provide many trials to build confidence that he is not lying. Note that the value of the coverage ratio is always what the vendor says it is—the confidence value refers to $\mathcal{C}$'s subjective probability of $\mathcal{V}$'s veracity, i.e., the likelihood that $\mathcal{V}$ would have been caught trying to cheat. Such an event will seriously damage a vendor's credibility; most vendors may not be willing to tolerate even a small chance of being caught, and thus would be cautious about misrepresenting the coverage ratio. If $\mathcal{V}$ tried to cheat on even 5% of the cases, with 20 trials, there is a 50% chance of exposure. For many vendors, this may be intolerable. We expect that this approach should provide $\mathcal{C}$ with a more accurate coverage estimate.

6 Disclosure Concerns

We now return to the key first issue in the list of desiderata in § 1, page 3: How much of $\mathcal{V}$'s valuable technical secrets do our techniques reveal?

At a minimum, $\mathcal{V}$ has to ship the binary B_s^X. Simply from the binary (even without a symbol table) an adversarial customer $\mathcal{C}_{\mathcal{A}}$ can construct a good deal of information by static analysis: the control flow graph, the size of the data space, the number of functions/entry points etc. A limited amount of static control & data dependency analysis is even possible. Indeed, tools like EEL can perform much of this analysis. In addition, by instrumentation, and

[6] For simplicity, we assume trials can be repeated.

dynamic analysis. $\mathcal{C}_{\mathcal{A}}$ can detect which paths of the control path are activated for different input conditions. Some recent work by Ball & Larus [3] show how it is possible to trace and profile control flow path execution using just the binary. Additional information can be gained by tracing memory references and building dynamic slices. Given the degree of information that can be reconstructed, it is important to evaluate carefully if the approaches listed above yield additional opportunities for $\mathcal{C}_{\mathcal{A}}$.

First, assume that the symbol table is stripped from the delivered binary B_s^X. During the verification process, whether driven by $\mathcal{T}$ or not, the only additional information revealed in response to challenges are the relevant coverage points and the applicable test cases. The coverage points, if based on the binary, can be independently generated by $\mathcal{C}_{\mathcal{A}}$ or $\mathcal{T}$; the only additional information is the test case, and its connection to this coverage point. *A priori*, the value of this information is difficult to estimate. Important factors include the manner in which the test cases are supplied, the resulting behavior of the system, etc. Test cases could be supplied in the form of source code or ASCII input (which might be very revealing) or in the form of binary objects or binary data (which could be more difficult to interpret). As far as the verification protocol is concerned, the only relevant behavior is that the selected challenge coverage point be exercised; however, there may be additional behaviour that reveals information to the adversary.

In the best case, if the relevant test case reflects a fairly "typical" type of usage, then the information given away is minimal; presumably $\mathcal{C}_{\mathcal{A}}$ would very likely find this out during his attacks using dynamic analysis. However, if the test case reflects an unusual circumstance, and the test case is delivered in a manner transparent to $\mathcal{C}_{\mathcal{A}}$, then some valuable information about unusual but important design and requirements details of B_s^X may be revealed. The risk of such an exposure is traded-off against the ability to verify test coverage.

The delivery of test cases to $\mathcal{C}_{\mathcal{A}}$, particularly the tests that somehow embody valuable proprietary information is an important issue that remains to be addressed: Can we deliver test cases in a way that protect $\mathcal{V}$'s secrets, while still exhibiting test coverage?

Now we relax the assumption that the symbol table is stripped out. While it is possible to verify test coverage without the symbol table, there are some difficulties (discussed above) associated with omitting it. In several of the protocols we listed above, we assumed that the symbol table was included in the binary shipped to the verifier. Clearly, the symbol table offers additional opportunities for $\mathcal{C}_{\mathcal{A}}$ to reconstruct information. Some standard obfuscation techniques such as garbling the symbolic names would be helpful. But in general, the advantages of omitting the symbol table may override the resulting difficulties.

7 Coverage Verification Limitations

There are limitations to the type of coverage that can be verified with our protocols. Any protocol that hides source code from the verifier (Protocols 1 thru 9) is limited by the available binary instrumentation facilities, and by the testability of the binary (which we discussed earlier). For simplicity, in the above discussion, we have assumed basic block coverage, since it very simple to check *e.g.,* with a debugger. If a far more stringent coverage is desired, such as *all paths* [10] then a tool based on [3] could be used to monitor the various paths during coverage verification. However, this level of coverage testing is extremely rare in practice.

Another limitation is the manner in which the challenge coverage points are selected. The discussion above again assumes "single point" type of coverage criteria (*e.g.* basic block or statement coverage). Other criteria, such as *all d-u paths*[7]. involve pairs of statements. We have not addressed *d-u path* coverage; for practical software, particularly in the context of heap memory usage, there are formidable obstacles to using this type of coverage. Our protocols can, however, be adapted to branch coverage, with suitable extensions to the instrumentation tools and to our random sampling protocol. Thus, to adapt Protocols 6 and 7 to branch coverage: first, a random location in the binary is chosen; then, the bounding basic block is found; at this point, if entry point of this basic block has a conditional branch, a coin is flipped to determine the direction of this branch; now, $\mathcal{V}$ can be challenged for a covering test case. With a suitable instrumentation facility, the binary can be instrumented and coverage verified.

It should be noted here that while coverage testing has widely used in industry, some researchers dispute the effectiveness of whitebox (or "clearbox") coverage methods. Most recent empirical work [13, 20] has found that test sets with coverage levels in the range of of 80-90% have a high chance of exposing failures. Earlier work [9] had yielded inconclusive results; however, the programs used in [9] were substantially smaller than [13, 20]. On the analytic front, rigorous probabilistic models of the relationship between increasing white-box coverage and the likelihood of fault detection have been developed [8, 11]. However, no known testing process is perfect; all known methods, white box or black box *will let some faults slip!* The best current experimental work [13, 20] suggests that high levels of white box test coverage can guarantee high levels of fault detection. However, since white box testing is not perfect, there are several complications. Given a coverage point that has faults, there may be several test cases that exercise that point. Some of these test cases will expose faults, but others may not. Consider a particular coverage point c, which has a fault f. When $\mathcal{V}$ generates a covering set of test cases, assume he finds a test τ which just happens to not expose the fault f. Since his software tests correctly, he will simply supply this test case along with the (faulty) delivered program. No malice is intended; particular case) coverage testing is imperfect, the vendor honestly delivered a program with a fault that just happened to slip by.

Now consider the case where the test case chosen by $\mathcal{V}$ happens to expose the fault. $\mathcal{V}$ now has two choices. He can fix the fault, or he can cheat: he can try to find a different test case τ^*, which covers c but does not expose a fault. The incentive for $\mathcal{V}$ to cheat depends

[7]A d-u path is a control flow path from the definition of a variable to its use, free of additional definitions of the same variable

on several factors: the likelihood the fault will occur in the field, the market conditions for the software (how many copies can he sell, for how long?), the cost of fixing the software, and the difficulty of finding a fault-hiding (but c-covering) test case τ^*. In most cases, it will probably be best to fix the fault. In the absolute worst case, assuming that the nature of the fault and the business conditions really motivate $\mathcal{V}$ to cheat, the cost finding such a τ^* depends on the distribution of failure-causing input within the subdomain of the input that causes the coverage point c to be exercised. If $\mathcal{V}$ is lucky, the failure region is small and well isolated within this input partition; he may succeed in finding a covering test case that is fault hiding. On the other hand, if the fault is provoked by many inputs from the subdomain of the input that exercises c, then $\mathcal{V}$ may have to read the code carefully (or spend a lot of time randomly sampling the input partition) to find such a τ^*; in this case, it may be easier to simply fix the problem. Finally, this method of attack (finding covering test cases that hide the fault) is not specific to the cryptographic techniques we have described; even the third party coverage certification method that is currently used is vulnerable.

We can favorably bias the situation by encouraging vendors provide (and commit to) more than one test case per coverage point. The more tests a vendor provides for a coverage point, the less likely it is that *all* these tests are fault-hiding. A random search for fault-covering cases in the corresponding input partition is not likely to work; it becomes necessary to understand the nature of the fault, and carefully construct several examples that exercise that coverage point, but hide the fault. As this effort increases, so does the incentive for $\mathcal{V}$ to simply fix the fault. The more test cases $\mathcal{V}$ can provide[8] for each coverage point, the less likely it is that he is hiding a fault in that coverage point.

Another difficulty inherent in coverage analysis is the opportunity to *pad*. Vendors can add spurious code which introduces additional control flow. Such code can be designed to be readily covered, thus artificially boosting coverage. It is not feasible to determine if this has been done. The only way to totally avoid this problem insist upon 100% coverage for some criteria; padding becomes irrelevant. For some criteria 100% coverage may be feasible for a medium-sized component; since such coverage levels are indicators of thorough testing, there may be market incentives that push vendors to achieve such a level. A series of coverage levels for increasingly stronger criteria, starting at and gradually decreasing from 100%, would be a desirable goal. Another approach to discourage padding is to demand explanations when challenge points are uncovered by tests. A vendor can voluntarily build confidence that he hasn't padded by making large number of pseudo random choices among his uncovered set and providing explanations for why they are not covered, and describing the conditions under which the points would be executed. Such conditions had better be highly unusual and difficult to create; if they are not, the vendor could be expected to provide a test case. If a large number of such disclosures are made, the vendor would be at risk of embarrassment by subsequent revelation that the coverage point was executed under less uncommon circumstances. A large number of such disclosures can build confidence that

[8]The test cases do not all have to be revealed; they could be hidden as a hash value, and revealed only upon a (random) challenge.

points do not remain uncovered simply as a result of padding. Again, another powerful disincentive to excessive padding is the vendor's inherent desire to produce high-quality software, and thus avoid costs of refunds, cancelled sales and extra update releases.

Finally, a customer may care only about one part of the vendor's claimed functionality, perhaps one that the vendor claims as a competitive advantage. In such cases, proof of an overall coverage level is less useful than a targeted demonstration of coverage of the subsystem that handles this functionality (If a 100% coverage has been achieved, this is not a concern, of course). It is not feasible for the customer to determine after the fact exactly which part of the system provides the indicated functionality. One approach to this problem is to require the vendor to identify ahead of time subsets of the coverage points that correspond to different sub-categories of the functionality of the system. In this case, a pseudo-random sample could be targeted just at coverage points in the indicated category; this can build confidence in the customer that the specific portion of the software had been tested thoroughly. If the vendor falsely makes this set of coverage points too small, the customer might find (perhaps by analyzing and instrumenting the binary) that a great many more coverage points are actually involved in the indicated functionality; the customer can then demand a pseudo random sampling of the entire set of coverage points, to try to find if the vendor can be caught lying about the other points that were claimed to be *not* involved in the indicated functionality. However, a statement such as "this piece of the code is involved in this functionality" can be difficult to prove or disprove, even if the entire system were laid wide open. If a specific sub-functionality of a system is of paramount concern, it may be best to either purchase from a vendor who has a well-tested component that factors just this functionality out, or insist on 100% coverage, or perform careful acceptance testing of the indicated functionality.

To summarize, our work rests on the assumption (again, supported by [13, 20]) that comprehensive coverage testing tends to expose faults, and on the assumption that vendors will most often find it more profitable to fix faults exposed by a covering test (rather than searching for a test that covers but hides faults). In the worst case, when the difficulty of fixing the faults exceeds the difficulty of finding test cases to hide the fault, and $\mathcal{V}$ expects the faults in question are rare enough so that he can collect enough revenue before the fault is exposed in in the field, then he may be tempted to find a hiding test case. By encouraging vendors to reveal many test cases for each coverage point, we can decrease the incentive to hide faults. But even in this worst case, the use of these techniques can provide customers with the justified belief that the vendors have every incentive and the means to find and fix all but the ones that are very unusual and/or difficult to fix. Padding is another problem; 100% coverage is the best way to preclude the chance of padding. We have suggested some ways that vendors can provide evidence that they did not pad; we are actively pursuing better ways of determining if padding has occurred.

In any case, during practical use of the protocol, it is important for both customers and vendors to be mindful of the suspected limitations of white box coverage testing. Additional black-box and/or acceptance testing will often be needed.

Protocol id	Reveals source code	Reveals symbol table	Reveals all test cases	Reveals entire coverage set	Uses trusted tools	Uses trusted third party	Comments
Basic	yes	yes	yes	yes	no	yes	
1	no	yes	yes	yes	no	yes	a
2	no	yes	no	no	no	yes	a,f
3	no	yes	no	no	yes	yes	f
4	no	no	no	no	yes	yes	b,f
5	no	no	no	no	no	yes	c,f
6	no	no	no	no	no	no	c,d,f
7	no	no	no	no	no	no	c,e,f

a Vendor can cheat by building a spurious coverage set.

b Vendor uses two trusted tools, a coverage analyzer and a binary location finder.

c Verifying test coverage on binary, rather than source, complicates finding test cases. Binary analysis tools may help build C's confidence

d Vendor can try to cheat by performing repeated trials by padding the binary.

e Involves trusted third party archive (minimally).

f Vendor's coverage level can only be estimated, subject to confidence levels; however, the "upper bound" technique described in Section 5 is applicable.

Table 1: Characteristics of the various protocols.

8 Conclusion

We have shown a set of protocols that can be used to verify test coverage, while protecting information valuable to the vendor and simultaneously reducing the vendor's ability to cheat. The results are summarized in Table 7. These protocols use various techniques such as trusted third parties, trusted tools, signatures, random challenges, and "autonomous" random challenges. These techniques can be used in different combinations, depending on the needs of the customer and the vendor. Combinations other than the ones we have presented are possible. For example, in Protocol 4, rather than using a third party to generate the challenges, the vendor could a location control string like that used in Protocol 7 to generate the challenges.

Some of our techniques are compatible with a widely used method for repelling reverse engineering, which is shipping binaries without source code or a symbol table. The only additional vendor information that our techniques need reveal are a small proportion of test cases. While it is possible to conceive of reverse engineering countermeasures that may not be compatible with our techniques, we believe that we can adjust our methods to be compatible with countermeasures that may complicate reverse engineering, such as the introduction of additional static control flow, additional dynamic control flow, or even certain other approaches that involve dynamically decrypting code prior to execution; we are actively exploring these issues.

Finally, we note that "self-validated" approaches (if they can be perfected) that verify testing effectiveness may have an important *leveling* effect on the software market. Using such methods, any vendor, without the help of a trusted third party, and at relatively low overheads, can provide a credible claim that their software quality control is stringent, while disclosing only a minimal amount of information. This enables small and unknown vendors to compete effectively (on the basis of perceived quality) with very large vendors with established brand names. It is our hope that such approaches, as they are perfected and widely adopted, will engender a creative "churn" in the software market place, to the ultimate benefit of the consumer.

References

[1] Delta Software Testing (accredited by Danish Accreditation Authority-DANAK). `http://www.delta.dk/se/ats.htm`.

[2] H. Agrawal. Dominators, super blocks and program coverage. In *Proceedings, POPL 94*, 1986.

[3] T. Ball and J. Larus. Efficient path profiling. In *Micro '96*. IEEE Press, December 1996.

[4] C. Cifuentes. Partial automation of an integrated reverse engineering environment for binary code. In *Third Working Conference on Reverse Engineering*, 1996.

[5] C. Cifuentes and J. Gough. Decompilation of binary programs. *Software Practice and Experience*, July 1995.

[6] P. Devanbu. Genoa- a language and front-end independent source code analyzer generator. In *Proceedings of the Fourteenth International Conference on Software Engineering*, 1992.

[7] P. Devanbu and S. G. Stubblebine. Building software with certified properties. *Unpublished Manuscript, available from the authors*, February 97.

[8] P.G. Frankl, R. Hamlet, B. Littlewood, and L. Strigini. Choosing a testing method to deliver reliability. In *Proceedings of the 19th International Conference on Software Engineering (To Appear)*. IEEE Computer Society, 1997.

[9] P.G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, August 1993.

[10] P.G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, August 1988.

[11] P.G. Frankl and E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, March 1993.

[12] A. Freier, P. Karlton, and P. Kocher. The ssl protocol, version 3.0 (internet draft), March 1996.

[13] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*. IEEE Computer Society, May 1994.

[14] Mondex Inc. `http:/www.mondex.com`.

[15] Plum Hall Inc. `http:/www.plumhall.com`.

[16] M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[17] National Software Testing Labs. `http://www.nstl.com`.

[18] Software Testing Labs. `http://www.stlabs.com`.

[19] J. Larus and E. Schnarr. Eel: Machine-independent executable editing. In *ACM SIGPLAN PLDI*. ACM Press, 1995.

[20] Y. Malaiya, N. Li, J. Bieman, R. Karcich, and B. Skibbe. Software test coverage and reliability. Technical report, Colorado State University, 1996.

[21] Doug McIlroy. Personal e-mail communication, 1996.

[22] N. Ramsey and M. Fernandez. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 1997.

[23] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 1995.

[24] A. Srivastava and A. Eustace. Atom: A tool for building customized program analysis tools. Technical Report 1994/2, DEC Western Research Labs, 1994.

[25] Applied Testing and Technology Inc. `http://www.aptest.com`.

[26] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[27] Bennet Yee and Doug Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of The First USENIX Workshop on Electronic Commerce*, New York, New York, July 1995.