

An Approach to Certifying Off-the-Shelf Software Components

Jeffrey Voas

Reliable Software Technologies Corporation

21515 Ridgetop Circle

Suite 250

Sterling, VA 20166

jmvoas@RSTcorp.com

Abstract

When a software system fails, a confusing and complex liability problem ensues for all parties that have contributed software functionality to the system. This paper presents a customer-based methodology for predicting what impact on system reliability Off-the-Shelf software components will have if they were to fail. Thus this paper describes methods for assessing the likelihood that someone else's defective software components will negatively impact your system's quality. Our methods can be applied before the components are adopted and without requiring component vendors to provide proprietary information about their development process. This provides the luxury to take appropriate defensive steps against Off-The-Shelf component failures before it is too late.

Keywords

Commercial-off-the-Shelf software components, testing, fault injection, software certification, fault tolerance, software wrappers.

1 Introduction

The world software and services market is rumored to be a 300 billion US dollar industry per year. The cost of software design and development could be significantly reduced if we had a widely-used software component industry. Even the best programmers only churn out 10 lines of code per day, which for systems such as cellular phones (that now have around 300,000 lines of code in them), have made custom software development very expensive [1]. If for example, 100,000 lines of code could be commercially purchased, 10,000 programmer-days of effort could be saved, allowing products to enter markets sooner.

Besides the time savings, another “win win” situation for both the consumers and producers is possible. From the consumer’s standpoint, if a “world-class programmer” costs \$500 per day for labor, licensing 100,000 lines of code would result in a \$5,000,000 savings (minus the costs to license the functionality). From the producer’s standpoint, if \$1M was an acceptable licensing fee for these 100,000 lines of code, then after 5 licenses, the producer would break even, and on the 6th license would see profit.¹

Both government and commercial organizations are already gearing up for systems that employ OTS functionality. For example, guidelines or standards have already been put forth by the US Federal Aviation Administration, US Department of Defense, and the US Food and Drug Agency.

The idea of a software component marketplace has good precedent. Traditional manufacturing is based on the concept of components. Automobiles, radios, bridges, planes, etc., are made from hundreds or thousands of components. (Can you imagine the difficulty in trying to build a bridge as a single part?) Software development is analogous to traditional manufacturing, particularly since software systems are composed of smaller software objects. What is lacking in the software industry that is ubiquitous in other manufacturing industries is the ability to confidently swap components in and out of systems. By “confidently”, we mean knowing that a replacement component is as reliable or more reliable (in terms of logical quality) than the replaced component. If this were possible, software component commerce would flourish. This would also decrease the costs of designing and repairing systems.

Key reasons for this lack of confidence is a lack of knowledge concerning how reliable newly acquired components are as well as not knowing how well the system will tolerate the components. Even if you were omnipotent and knew that a specific component was of high quality, the component might not like the environment that you are placing it into and cause the system a host of undesirable problems.

Software components are often delivered in “black-boxes.” We say black-boxes, because software components are usually delivered as executable objects (with licensing agreements that forbid de-compilation back to source).² It is possible to be lulled into thinking that a more expensive component received more testing and thus is more reliable. But the opposite could also be true a less expensive component that has experienced more usage may actually have higher quality. Our goal is to provide a methodology for determining how much quality a OTS component has by applying a set of black-box analyses. These analyses form our methodology for qualifying OTS components.

¹Admittedly, this simple analysis has ignored many important market-based variables, but it does hint at the basic business principles that are driving consumers and producers toward a component catalogue marketplace.

²It may be possible to also license source code, but the cost of doing so may be prohibitive. Throughout this paper, we assume that source is not attainable, regardless of the reason.

Scenario Classifications	\mathcal{C} is what is Needed for \mathcal{S} ?	\mathcal{C} of High Enough Quality?	Positive Impact on \mathcal{S} ?
1.	Yes	Yes	Yes
2.	Yes	Yes	No
3.	Yes	No	Yes
4.	Yes	No	No
5.	No	N/A	N/A

Table 1 Key questions before OTS component \mathcal{C} should be embedded into system \mathcal{S} .

2 The Five Classifications

Once the decision is made to adopt OTS software rather than to build custom (*bespoke*) software, there are three key issues to consider concerning a candidate component³ (1) is the component \mathcal{C} *the one you need*? (2) is the *quality* of the software component high enough? and (3) what *impact* will the component have on your system, \mathcal{S} . To decide whether \mathcal{C} will have a positive impact on \mathcal{S} , the criticality of \mathcal{S} must be taken into account. These questions bring up an interesting set of possible scenarios (See Table 1).

These five scenarios emphasize that the so-called “OTS quality problem” is not just a component quality problem but is also an integration compatibility problem. Even a dozen highly reliable components combined together does not guarantee a highly reliable system.

Determining which scenario a set of circumstances places you in will be a subjective call. There is a broad spectrum of other categories that lie in between the five just mentioned that can confuse the issue. For example, the only difference between Scenarios 1 and 3 is component quality. If all other things are equal and we have a fairly good but not perfect component, which category should we place it into? 1 or 3? (Recognize that if deciding this were so straightforward, then why are there dozens of software reliability models that give different predictions for the same data?) The purpose of this paper is to provide a certification methodology for deciding which scenario a set of circumstances places you in. By knowing this, you can better defend yourself legally against someone else’s imperfect OTS software.

Today, there are varying approaches to certifying software. Many current approaches require that the developer take certain oaths concerning which development standards and processes were used. Our certification methodology is not based on this honor system. Our approach begins by assuming that a software component \mathcal{C} and a description of what \mathcal{C} supposedly does is all that is available to the potential buyers of the component. From there, the buyers must take responsibility for determining whether \mathcal{C} is what they want. Even if a OTS developer were to promise that exhaustive testing and formal verification had succeeded, our scheme will not use that information. Our hypothesis is that when dealing with OTS software, totally *independent* certification is the only safe approach that buyers

³We term a OTS component that is under consideration for being embedded into a system as a *candidate* component.

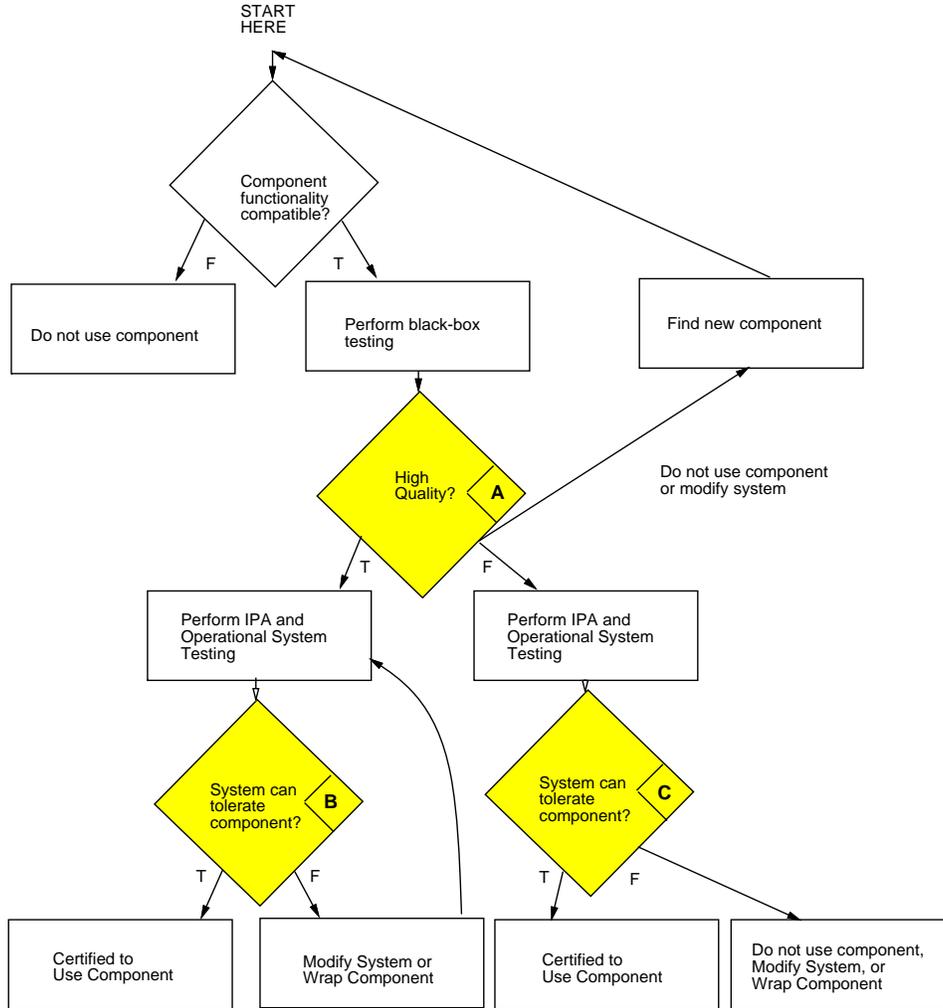


Figure 1 The OTS Component Certification Process

should trust.

Figure 1 shows the basic steps in our independent certification methodology for deciding whether a component \mathcal{C} should be adopted into system \mathcal{S} . Three pivotal decision points are labeled in Figure 1 as A, B, and C. But before we get to those decisions, we first must decide whether the component is advertised as having the functionality we need—that is the first decision point in the diagram. We can immediately ignore any component that does not meet this requirement. So for example, if we need a C++ parser but we find a Fortran-77 parser, then there is no reason to further pursue the quality and interoperability issues of this component. Next, Decision A focuses on whether component quality is sufficient. And finally, Decisions B and C focus on whether the system can tolerate the component.

Our certification methodology will employ automated technologies to differentiate between Scenarios 1-4. Scenario 4 describes components with poor quality and detrimental impacts on the system. Scenario 3 describes components of poor quality but that have a positive impact on the system. Scenario 3 highlights a weakness in common software engineering wisdom which suggests that imperfect software is *always* bad. Scenario 2 is also

intriguing; it is analogous to the medical problem of rejected transplanted organs. Here, the component has the right functionality and is of high quality, but the system cannot tolerate it. This phenomenon is an indictment against the system and not against the component. And finally, Scenario 1 is a “best of all worlds” scenario, where the component and system will, in theory, “live happily ever after.”

Determining whether a candidate component belongs to Scenario 5 requires finding out whether the specification of the component matches the requirements of the system. Although this task is prone to human error, we do not believe that mistakes here should be frequent if reasonable effort is made to assess what the component does and how it connects to its environment.

After deciding which scenario you are in, you can opt to adopt the component in its current form, find a new component, or modify the functionality of the system or component.

3 Component Certification Technologies

Three families of quality assessment techniques will be used for determining how much potential harm is afforded by a candidate OTS component. In our methodology, *black-box testing* will be employed to determine whether the component is of high enough quality, which is the decision labeled as A in Figure 1. *System-level fault injection* will be employed to determine how well a system will tolerate a failing component, and *operational system testing* will determine how well the system will tolerate the component when it is not failing; even properly functioning components can create system-wide problems. System-level fault injection and operational system testing will be used to determine the outcomes from Decisions B and C (in Figure 1). We will now discuss each of these in more detail.

3.1 Black-Box Component Testing

Black-box testing is a family of software testing techniques that selects test cases without regard for the software’s syntax. To perform black-box testing, an executable component, inputs, and an *oracle* (which decides if failure has occurred) are needed. In contrast, *white-box* testing techniques consider the code when selecting test cases. For example, one white-box testing strategy selects test cases such that each possible outcome from each decision point occurs.

For OTS software consumers, white-box testing techniques will be of little help (however nothing precludes OTS component developers from using them during development). Black-box testing (according to the *operational profile* [4] caused by \mathcal{S}) will be our technology for determining “stand alone” component quality. It may be true that the OTS supplier has already done testing of their components, but how much, and according to what “generic” profile should not be left to chance.

Black-box testing is not without criticism, however. Black-box testing can fail to exercise significant portions of the code, and from a certification perspective, that is worrisome. The value-added by black-box testing is also dependent on having accurate oracles. Faulty oracles are capable of allowing bad software to be certified. Faulty oracles can also allow quality software to not get certified. Correct oracles are of paramount importance to our certification

methodology, and correct oracles may be difficult for a component purchaser to derive. Our recommendation here is that the OTS consumer develop their own oracle according to what they want the component to do, not necessarily according to the specification from the OTS vendor. By doing this, the consumer can test the quality of the component against their requirements for what the component is expected to do, and not necessarily against the functionality claimed by the vendor.

Black-box testing of executable software components has already received attention from the work of Miller, *et al.* and their FUZZ model [9]. FUZZ works by generating random black-box inputs, feeding them to UNIX system functions, and watching for `core` dumps. FUZZ provides a low-resolution approximation of the robustness of particular UNIX utilities (*e.g.*, `ls`). Watching for a program to dump `core` (that is, watching for a program to crash) is a crude way of determining whether failure has occurred.

There is yet another serious problem with OTS software, and that is that OTS components can have unknown, *malicious* functionality. For example, a covert Trojan horse might only be exploitable by one test case. This test case is unlikely to be used during black-box testing. The likelihood of accidentally believing that a component does not contain a Trojan horse from the results of black-box testing is not negligible. In short, black-box testing plays an important (yet limited) role in assessing the quality of software components.

Admittedly, the costs of performing black-box testing by a OTS consumer cannot be overlooked. The consumer will need to generate inputs, build an oracle, and probably build a test driver. Each of these tasks can be expensive. However given the development cost savings and the time-to-market improvements provided by off-the-shelf components, we do not feel that these additional costs to demonstrate component quality in the specific environment is reasonable. After all, if a consumer is unwilling to do due diligence before adopting OTS functionality, then their defensive position with respect to liability is weakened.

3.2 Software Fault Injection

Even if we knew that a component had no Trojan horses and was of high “stand alone” quality according to our requirements, the certification process is not yet complete. The next analysis that needs to be performed is *system-level fault injection*. Quality components can still be used as the building blocks of unreliable systems. Further, low quality components may not cause a system to be unreliable. (This can occur when a component’s failures are ignored by the system.) To assess whether your circumstances match either of these unusual phenomenon, system-level fault injection is the next step in our certification process.

System-level fault injection does not explicitly state how reliable the system is, but rather provides worst-case predictions for how badly the system might *behave* in the future if components that the system depends on were to fail. There are many different forms of fault injection. The underlying ideas behind fault injection are not new, and there exists much literature describing how to employ them for hardware system validation, software testing, and hardware design validation [8, 7, 6, 5].

The particular fault injection technique that our certification methodology will use is called “Interface Propagation Analysis” (IPA) [3, 2]. IPA *perturbs* (i.e., corrupts) the states that propagate through interfaces between components. To perturb states, access to the interfaces that components use for communication is needed. We use a small software routine

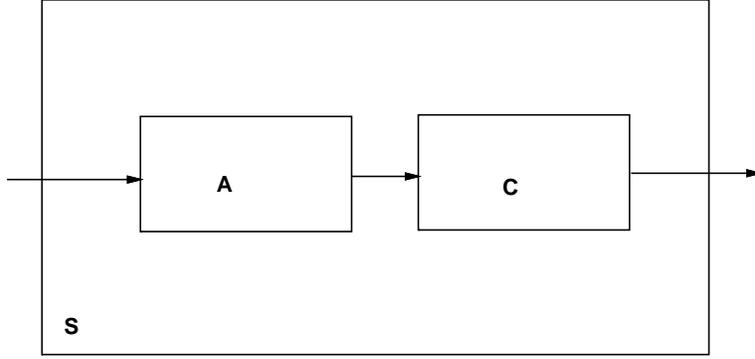


Figure 2 The system \mathcal{S} made from components A and C .

called a *perturbation function* that actually makes the replacement from the original state to a different (corrupt) state while the software executes.

IPA's role in our certification methodology is to determine the proper classification for the last column of Table 1. (Note that operational system testing, discussed in Section 3.3, will play a role in this decision as well.) By corrupting data going from one component to the successor component, failure of the predecessor component is simulated. Our justification for doing this follows

By hypothesizing that a component failed as the system executes, we can assess whether the system can tolerate anomalies originating from that component.

To observe the system-wide impact of component failures, IPA must be told what component failure modes to inject and what system failure modes to be on the lookout for. IPA then checks to see if any of these system failure modes manifest. System failure modes can be defined as faulty system output data, faulty global system data, or corrupted data flowing between successor components. Since the OTS consumer is the system builder, it is reasonable to expect that they know what constitutes as a system failure.

It is less likely though, that the system integrator will know what component failure modes to simulate during IPA. Because of this, an approach similar to FUZZ's input generation is employed, where a pseudo-random number generator modifies data in various ways, some of which are *ad hoc*. After all, if the system can tolerate *ad hoc* failures, it is more likely to tolerate the real failure modes of the component. In terms of how information is actually corrupted in the tools that we have built, most of that information is proprietary, however in [3], we provided source code to several of the generic, low-level fault injection algorithms that are used to simulate component failures.

To get a feeling for how this analysis works, consider the system \mathcal{S} composed of two components, A and C (shown in Figure 2). The output from A is the input to C . Now consider a new version of \mathcal{S} in which a OTS component B is wedged between A and C (see Figure 3). The customer for B who is considering this addition to \mathcal{S} should already have information about the quality of B from black-box testing. But how will the system react after B is added? To answer this, system-level fault injection experiments will be performed. These experiments will force corrupt information to flow in between components A and C , simulating the situation where B fails (as illustrated in Figure 4).

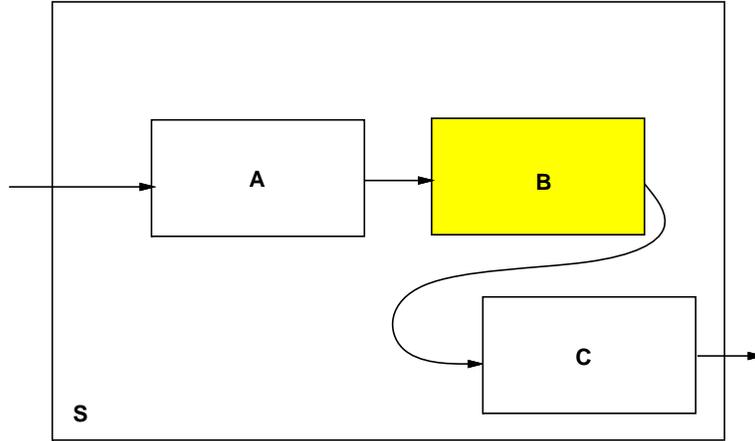


Figure 3 The system \mathcal{S} augmented with OTS component, B .

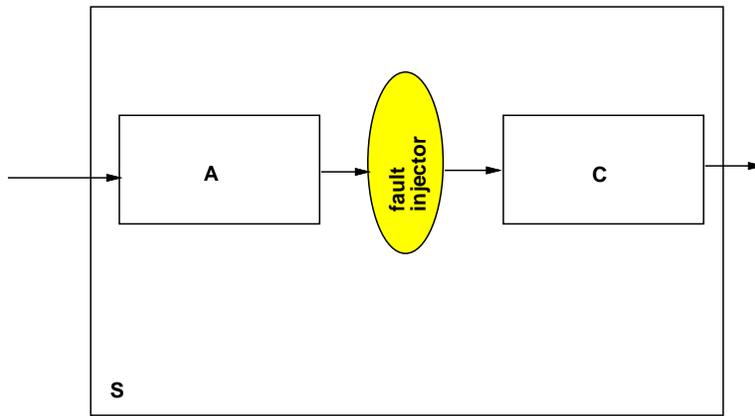


Figure 4 Simulating the failure of candidate component B .

We have applied this analysis to systems that rely on operating system utilities. Specifically, we will discuss a tool that automates IPA for AIX operating system and standard C library function calls.

The key to applying fault injection to function calls or calls to the operating system is determining how the function or operating system call returns information. The manner by which a function is declared defines what interfaces we have to work with. The output from an AIX function may be (1) a return value, (2) a changed value in an argument that is passed by reference to the AIX utility, or (3) a combination. Fault injection will be applied to the information returned to the calling application from the C library functions. By modifying returned information, you are simulating a failure of the entity that produced the information without getting wrapped up in the specific details for how the information got corrupted.

Let's illustrate using AIX's `cos()` function

```
double cos(double x)
```

This indicates that the `cos()` function receives a double integer (contained in variable `x`)

and returns a double integer. This defines the information flow into and out of the function. Because of C's language constraints, the only output from the `cos()` function is the returned value, and hence that is all that fault injection should corrupt. This declaration does not, however, specify what the function should do.

As mentioned earlier, IPA can be fine-tuned to simulate precise failure classes or simply random corruptions. For instance, based on common trigonometric mistakes, it would be reasonable to force the result of `cos()` to take on the values of NaN and Infinity. It would also be reasonable to employ 0, because this is a common incorrect result from cosine implementations.

To see how this analysis is handled inside an application that uses `cos()`, consider the following

```
if (cos(a) > THRESHOLD) {  
    // do something  
}
```

When perturbing the return value, the following instrumentation characterizes the process.

```
if (PERTURB(cos(a)) > THRESHOLD) {  
    // do something  
}
```

The fault injection instrumentation is added to the source application before the application is compiled. When the modified application is executed, a fault injection management process collects statistics on how tolerant the application was to forced corruptions.

Another example is the `strcpy()` function

```
char *strcpy(char *s1, char *s2)
```

The `strcpy()` function copies the characters contained in string `s2` into string `s1`. The value `strcpy()` returns is the pointer `s1`. To see how this OTS function is handled during fault injection, consider the following call

```
strcpy(old, new)
```

For this function, corruption is performed upon exit of the original call

```
strcpy(old, new)  
old = PERTURB(old)
```

After the call to `strcpy()`, the result contained in `old` is replaced with a different string of the same size. As with the `cos()` function, we can take advantage of information about common faults occurring in string copy implementations (*e.g.*, employing an empty string, which mimics incorrect handling of a dropout condition in the `strcpy()` logic). Since it is rare to ever use the value returned by this function, the pointer would typically not get corrupted. If, however, we did want to corrupt the pointer, we could

```
PERTURB(strcpy(old, new))
```

In summary, system-level fault injection provides a means for assessing how well an application can recover after receiving bad results from OTS functions. The beauty of this approach is that we can abstract away concerns about why the returned information is bad and concentrate on making the system more robust.

3.3 Operational System Testing

In addition to system-level fault injection, *operational system testing* with a OTS component embedded is a complementary means for determining how the system will tolerate the component. Operational system testing executes the full system with system test cases. The difference between operational system testing and fault injection is that operational system testing does not employ perturbation functions to perturb states. The output information a component produces does not get modified and the system is executed using original states.

An advantage here over fault injection is that when a component really fails, the system experiences an actual component failure. This provides a more accurate assessment of system tolerance. The downside is that if the component rarely fails, an enormous amount of system-level testing will be necessary to make that determination.

Operational system testing will, however, be valuable for system-level reliability prediction. Thus operational system testing should be performed as a “sanity check” to determine if \mathcal{C} is a good match for \mathcal{S} .

4 Building Defenses Through Wrapping

If system intolerance to a component is observed and the component is still slated for adoption, corrective measures are needed. Generally speaking, this will either involve modifying the component’s functionality or modifying the system (See Figure 1).

A software *wrapper* is a software en-casing that surrounds a component and limits what the component can do. Wrappers perform the task of detecting when undesirable outputs are going to result and then keeping those outputs from occurring. Wrappers do not modify a component’s source-code in an invasive manner, (*e.g.*, like removing lines of code from the component), but instead modify the functionality of the component indirectly, by limiting its functionality in a non-invasive (external) manner.

To accomplish this, two different wrapper approaches are used. One approach ignores certain inputs to the component and thus does not allow the component to execute on those inputs. That clearly limits the component’s output range. The other approach is to capture the output before the component releases it, check to ensure that certain constraints are satisfied, and then only allow those outputs to result that satisfy the constraints.

Wrappers are not foolproof. As shown in Figure 5, illegal outputs can sometimes trick a wrapper. For example, a call to the operating system to delete a file that the component should not be allowed to delete. If it is not known that the component is capable of deleting files, then probably any wrapper designed for this component will fail to thwart attempts at file removal. If, however, it is known that the component needs to delete temporary files that it creates, then it is wise that the wrapper ensure that any file delete requests only pertain to those temporary files.

Wrappers are a clever way to build protection against OTS components. The reliability of wrappers is directly determined by the care taken by the their developers and designers who determine what events must be protected against. In both cases, if human error is made, the value-added from a wrapper is reduced. If Trojan horses exist, wrappers will be of little value.

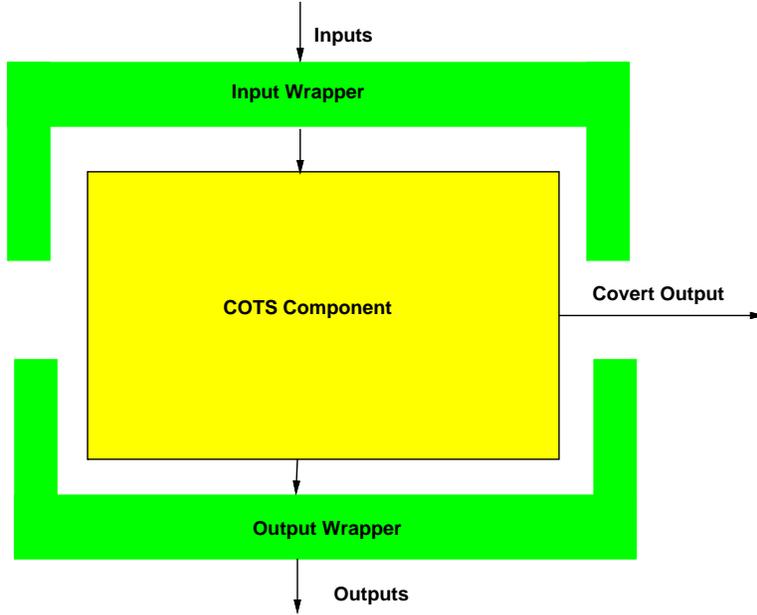


Figure 5 A “wrapper” failing to stop a covert channel.

We have discussed using fault injection to test for system reactions when incorrect information is fed to the system. Interestingly, fault injection can play a role during wrapper creation by determining what outputs need to be protected against. By determining what classes of component failures \mathcal{S} cannot tolerate, we know what outputs to filter away.

Once wrappers are built, black-box testing of the component with the wrapper should be re-applied to test their strength to ensure that they are filtering correctly. Also, once a wrapped component is embedded into a system, fault injection should be re-applied to ensure that the wrapper has improved system tolerance.

5 To Certify or Not

If after applying black-box testing, IPA, operational system testing, and defense building, we find that we are in Scenario 1 or 3, then \mathcal{C} should be certified for use in \mathcal{S} . If we are in Scenario 2, the decision is slightly more difficult, because we know that failures of the component will be infrequent, but when they occur, \mathcal{S} cannot tolerate them. And as stated, we cannot certify \mathcal{C} for \mathcal{S} if we are in Scenario 4. If you are in Scenario 4, select another component and start over.

From a design standpoint, the goal should be to only embed OTS components after it is determined that they are in Scenarios 1 or 3. If you are in Scenario 2, several options are available (1) modifications to the system, (2) finding another component that your system can better integrate with, or (3) simply accepting the risk that when the component fails (although this will be infrequent), the system is likely to also fail.

6 Summary

Commerce in off-the-shelf software components is gradually becoming a reality. Two of the greatest impediments to this occurring overnight are the issues of knowing component quality and how the system will tolerate a component. If these problems can be overcome, then the opportunity costs associated with selecting one component over the other could be better determined.

We have described a three-part process for deciding how likely systems are to fail as a result of employing off-the-shelf software functionality. The best time to perform this process is during the grace period that most vendors provide while their software can be evaluated. This process gives guidance concerning how much of someone else's mistakes you can tolerate. If it turns out that you cannot tolerate much, we have suggested several preliminary avenues for how you might better defend yourself.

Our certification methodology approach is not foolproof and it is not generic. There are aspects to it that are distasteful from a certification perspective. For example, it would be simpler if we could certify a component for use in *all* systems, thus avoiding the system dependencies needed by IPA. Our philosophy here is conservative; until we know how to assess components in a manner that accounts for all undesirable behaviors that could be forced into any system, it is prudent to only certify components for the idiosyncrasies of each system.

To foster the emerging software component marketplace, we believe that components must be traded with an unambiguous meaning of their impact (be it positive or negative). Ideally, data concerning a component's failure mode would be available before a component is traded. With failure mode data in hand, component consumers could apply appropriate static and dynamic techniques to determine system impact. Armed with this information, system builders could make better design decisions and be less fearful of liability concerns. And component vendors could expect the marketplace to grow much more rapidly.

Acknowledgements

This work has been partially supported by DARPA Contract F30602-95-C-0282, Rome Laboratory Contract F30602-97-C-0117, NIST Contract 50-DKNA-4-00119, and NIST Advanced Technology Program Cooperative Agreement No. 70NANB5H1160. I thank Frank Charron for his insights into how AIX function calls are instrumented and Lora Kassab for making suggestions on an earlier draft of this manuscript.

References

- [1] S. BAKER, G. McWILLIAMS, AND M. KRIPALANI. "Forget the Huddled Masses Send Nerds", Business Week, July 11, 1997.
- [2] J. VOAS, F. CHARRON, AND K. MILLER. Robust Software Interfaces Can COTS-based Systems be Trusted Without Them? In *Proc. of 15th Int'l. Conf. on Computer Safety, Reliability, and Security (SAFECOMP'96)*, Vienna, Austria, October 1996. Springer-Verlag.

- [3] J. VOAS, G. MCGRAW, A. GHOSH, AND K. MILLER. Glueing together Software Components How good is your glue? In *Proc. of Pacific Northwest Software Quality Conference*, pages 338–349, Portland, October 1996.
- [4] J. D. MUSA, A. IANNINO, AND K. OKUMOTO. *Software Reliability Measurement Prediction Application*. McGraw-Hill, 1987. ISBN 0-07-044093-X.
- [5] J. ARLAT, M. AGUERA, L. AMAT, Y. CROUZET, J.-C FABRE, J.-C LAPRIE, E. MARTINS, AND D. POWELL. Fault Injection for Dependability Validation A Methodology and Some Applications. *IEEE Trans. on Software Engineering*, 16(2)166–182, February 1990.
- [6] J. A. CLARK AND D. K. PRADHAN. Fault Injection A Method for Validating Computer-System Dependability. *IEEE Computer*, pages 47–56, June 1995.
- [7] J. A. SOLHEIM AND J. H. ROWLAND. An Empirical Study of Testing and Integration Strategies Using Artificial Software Systems. *IEEE Trans. on Software Engineering*, 19(10)941–949, October 1993.
- [8] R. A. DEMILLO, R. J. LIPTON, AND F. G. SAYWARD. Hints on test data selection Help for the practicing programmer. *IEEE Computer*, 11(4)34–41, April 1978.
- [9] B. P. MILLER, L. FREDRIKSON, AND B. SO. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12)32–44, December 1990.