PROJECT REPORT

(6 weeks Project Training)

Function Fault Injector

Submitted by

Atishay Jain

Roll No: 10783017

Under the Guidance of

Mr. Satheesh Konidala,
Manager (T), Unified Communications,
Microsoft IDC, Hyderabad.

Department Of Computer Science and Engineering
THAPAR UNIVERSITY, PATIALA
(Deemed University)

Jun-July 2008

ANNEXURE – VIII

DECLARATION

I hereby declare that the project work entitled "Function Fault Injector" is an authentic record of my own work carried out at Microsoft IDC, Gachibowli, Hyderabad as requirements of six weeks project term for the award of degree of B.E. (Computer Science & Engineering), Thapar University, Patiala, under the guidance of Mr. Satheesh Konidala, during 2nd June to 25th July, 2008.

ATISHAY JAIN
10783017

Date: _____

ACKNOWLEDGEMENT

For the project to come out up to the standards, I would like to thank first of all my mentor, Mr. Premkumar Srinivasan, whose knowledge of the subject and guidance in the right direction helped to me not lose direction into the vast subject matter and be able to complete the project in the stipulated time. I would also like to thank my manager, Mr. Satheesh for providing me with the essential support and analyzing my work at regular intervals so that none of the requirements are left out. Besides these I would also like to thank my fellow interns for the support they provided, my teachers for guiding me to the company, my parents to allow me to go and many others, without whose support and cooperation, the project would have remained just a dream.

CONTENTS

PROJECT SUMMARY

Fault Injection – The concept

"Deliberate insertion of upsets (faults or errors) in computer systems to evaluate its behavior in the presence of faults or validate specific fault tolerance mechanisms in computers."

By Fault Injection we are testing the fault tolerant mechanisms that prevent the faults from becoming errors or bugs that cause failure.

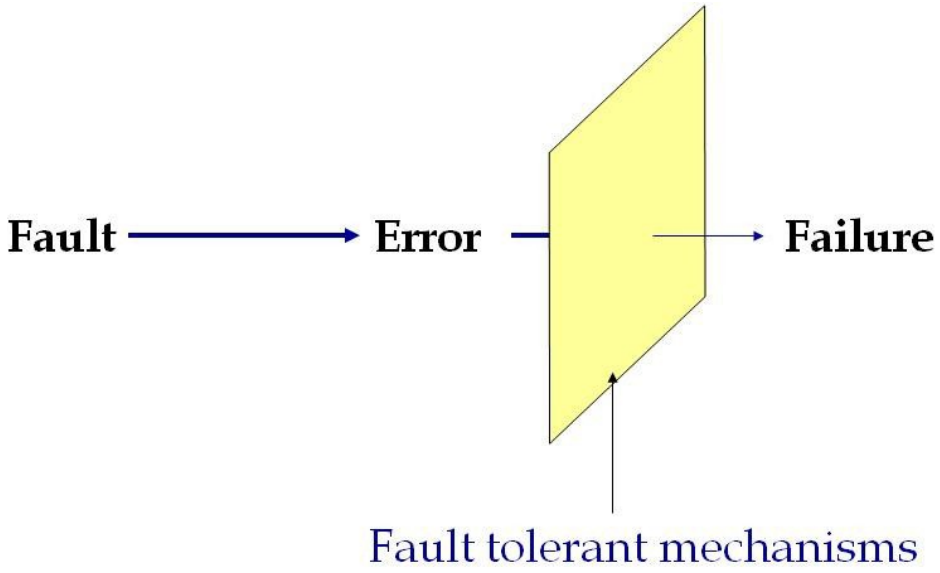


Figure 1: Conversion of fault to failure

Fault Injection is basically of two types: **Hardware and Software implemented.**

In hardware based fault injection, the basic electronic circuit introduces interrupts during key areas of program execution to analyze its effects whereas if done through software, this action is performed by altering the logic.

Basic idea of software implemented fault injection (SWIFI) is:

1) Interrupt the target application/system in some way (e.g., by inserting a trap instruction or by executing the application in trace mode).
2) Execute specific fault injection routine that emulates faults by inserting errors in different parts of the system (processor registers, memory, etc.).

3) Resume the execution of the interrupted program.
4) Collect results on faults manifestations at different levels (system level, application level, FTM, etc).

Advantages of SWIFI

- Not much affected by the complexity of the target
- Low complexity
- Low cost and development effort
- Reasonably portable
- No physical interferences

Disadvantages of SWIFI

- Do not cover faults in peripheral devices, ASICS, etc
- Limited monitoring capabilities
- Tools have great impact on the target system behavior

Function Fault Injector-The tool

Function Fault Injector (FFI) can be used to inject faults into software at the function level. By this we can perform various types of tests identifying various error and exception handlers throughout the module, by changing the return values passed by the functions present in the other dlls that this function calls. The tool provides facility of not only having alternate return value passed but also of making the value to be passed under a certain probability such that the normal function can also be run under the remaining cases. Not only this, it provides features to fill in an alternate function for the actual function, modify the produced return value by the called function, call the actual function but after modifying the parameters that were actually passed to it, all under the control of probabilities so that the fault ratio can be controlled. Besides, it can also act as the normal shim-generator, where it generates a C file that can be modified at will and compiled to act as an Application Verifier Shim.

Application Verifier (AppVerifier) is a runtime verification tool(freeware) used in testing applications for compatibility with Microsoft Windows XP/CE. This tool can be used to test for a wide variety of known compatibility issues while the application is running. It intercepts calls to system as well as non system functions within external dlls. The function fault injector uses the Application verifier as the mother software and acts as its component to aid speedy generation of shims (shims are the dlls the intercepted functions are directed to) for testing.

Also included in the FFI is the facility to save your projects and improve or modify them later. Directly added is the facility to modify the probabilities where one doesn't need to go back to the registry to modify them but can directly go to the registry editor and modify the probabilities stored as registry keys. Built for the purpose of ARMv4I environment testing, but the generated code can be compiled under any environment including x86, provided that the build window is available and there is an availability of the application verifier to run the shim.

THE PROGRAMMING AND DEVELOPEMNT ENVIRONMENT

The fault injector was designed in C# by the use of the Visual Studio 2005, IDE as the development environment. The code it called to as a part of the injector is the shimgen.exe file that was written as a part of the application verifier shim generator in Visual C++ and was kept as unmodified during the entire project. Apart from these the Microsoft 'build.exe' was used to compile and integrate the generated shims as a part of the Windows CE 6.0 code.

The C# Language

C# (pronounced C Sharp) is a multi-paradigm programming language that encompasses functional, imperative, generic and object-oriented disciplines. It is developed by Microsoft as part of the .NET initiative and later approved as a standard by ECMA (ECMA-334) and ISO (ISO/IEC 23270). Developed around the same time, C# and Java are sister languages with many of the features being similar and being added to both the languages with a huge impact from the other one.

Visual Studio IDE

Microsoft Visual Studio is the main Integrated Development Environment (IDE) from Microsoft. It can be used to develop console and Graphical user interface applications along with Windows Forms applications, web sites, web applications, and web services in both native code together with managed code for all platforms supported by Microsoft Windows, Windows Mobile, .NET Framework, .NET Compact Framework and Microsoft Silverlight.

Visual Studio includes a code editor supporting IntelliSense as well as code refactoring. The integrated debugger works both as a source-level debugger and a machine-level debugger. Other built-in tools include a forms designer for building GUI applications, web designer, class designer, and database schema designer. It allows plug-ins to be added that enhance the functionality at almost every level - including adding support for source control systems to adding new toolsets like editors and visual designers for domain-specific languages or toolsets for other aspects of the software development lifecycle.

ARM - the target environment

The ARM architecture (previously, the Advanced RISC Machine, and prior to that Acorn RISC Machine) is a 32-bit RISC processor architecture developed by ARM Limited that is widely used in embedded designs. Because of their power saving features, ARM CPUs are dominant in the mobile electronics market, where low power consumption is a critical design goal.

Today, the ARM family accounts for approximately 75% of all embedded 32-bit RISC CPUs, making it one of the most widely used 32-bit architectures. ARM CPUs are found in most corners of consumer electronics, from portable devices (PDAs, mobile phones, media players, handheld gaming units, and calculators) to computer peripherals (hard drives, desktop routers). Windows CE 6.0 supports ARM v4I and above as the machine architecture.

Windows CE-the target OS

Windows CE (also known officially as Windows Embedded Compact post version 6.0 and sometimes abbreviated WinCE) is Microsoft's operating system for minimalistic computers and embedded systems. Windows CE is a distinctly different operating system and kernel, rather than a trimmed-down version of desktop Windows.

Figure 2 Windows CE

Windows CE is a modular/componentized operating system that serves as the foundation for several classes of devices. Some of these modules provide subsets of other components' features, others which are mutually exclusive, and others which add additional features to another component. One can buy a kit (the Platform Builder) which contains all these components and the tools with which to develop a custom platform.

Platform Builder – the generator of the code to be tested

This programming tool is used for building the platform (Kernel), device drivers (shared source or custom made) and also the application. This is a one step environment to get the system up and running. One can also use Platform Builder to export an SDK (software development kit) for the target microprocessor (SuperH, x86, MIPS, ARM etc.) to be used with another associated tool set named below.

PROJECT

The major features/requirements of the project can be summarized as:

1. Intercept the program flow (via the application verifier) and pass the original function to the shimmed dll where the following can be applied:
    a. Give alternate Return Value.
    b. Fill in a stub function, to replace the original one.
    c. Change the passed parameters to the original function return the result produced henceforth.
    d. Wait for the original function to complete and modify the output or the return value after the function has ended its flow.
2. Have probabilities associated with each type of return and also with the original function.
3. Modify the probabilities dynamically, through the windows registry.
4. No injection into the original dll.
5. Can remove the shim from application via just one command.
6. Save and modify the project any time and apply the modified shim by just replacing the old one.
7. Full flexibility to write any C code and include custom headers.
8. Can also be used for API testing through the 'modify passed values or parameters' option.
9. System dlls can be shimmed to produce other types of faults by say restricting the memory available.
10. A lot of time is saved as the tool automatically generates the code and the support files which in normal shimming process take much more time.
11. Free form many leaks and flaws present in the original shimgen and hence a good alternative to produce new shims for other purposes.
12. Leaves the C files as code for any modification as the user wants.

DESIGN

The working of the fault injector is based on the working of the application verifier, which makes registry settings and based on those settings intercepts calls to a specific dll and passes it onto the shimmed dll. The function fault injector has automated the formation of the shim, by modifying the application verifier functioning.

These modifications in the shim generation process can be best described by the means of the two diagrams below:

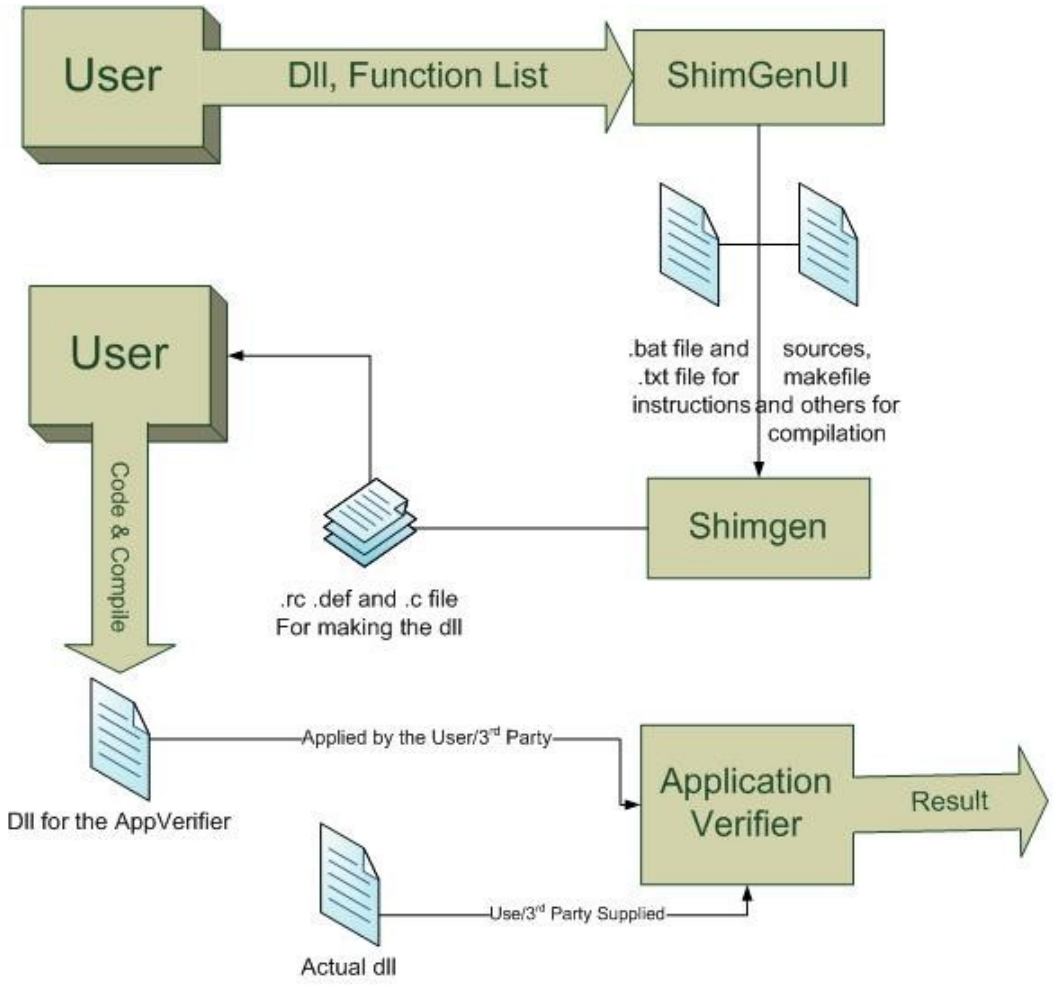


Figure 3 Shim Generation by Application Verifier

Default Shim Generation

1. User selects the Shimgen UI and identifies the functions to be shimmed. The UI lists all available unmanaged functions (Made by core C classes and C++ classes which do not

have a garbage collector and are a major part of the original windows code) present in the given dll that can be shimmed, through the use of MSDOS 'dumpbin/exports' command.

2. The UI Automatically generates the .bat file to call the old Shimgen which is a command line tool built in native C++ that actually opens the dll, finds the available debug information and generates the files for shimming.
3. User then has to manually modify the files and produce the code, which has to compiled by the WinCE build window, and applied via the Visual Studio shell prompt(Command Prompt for WinCE) into the application.

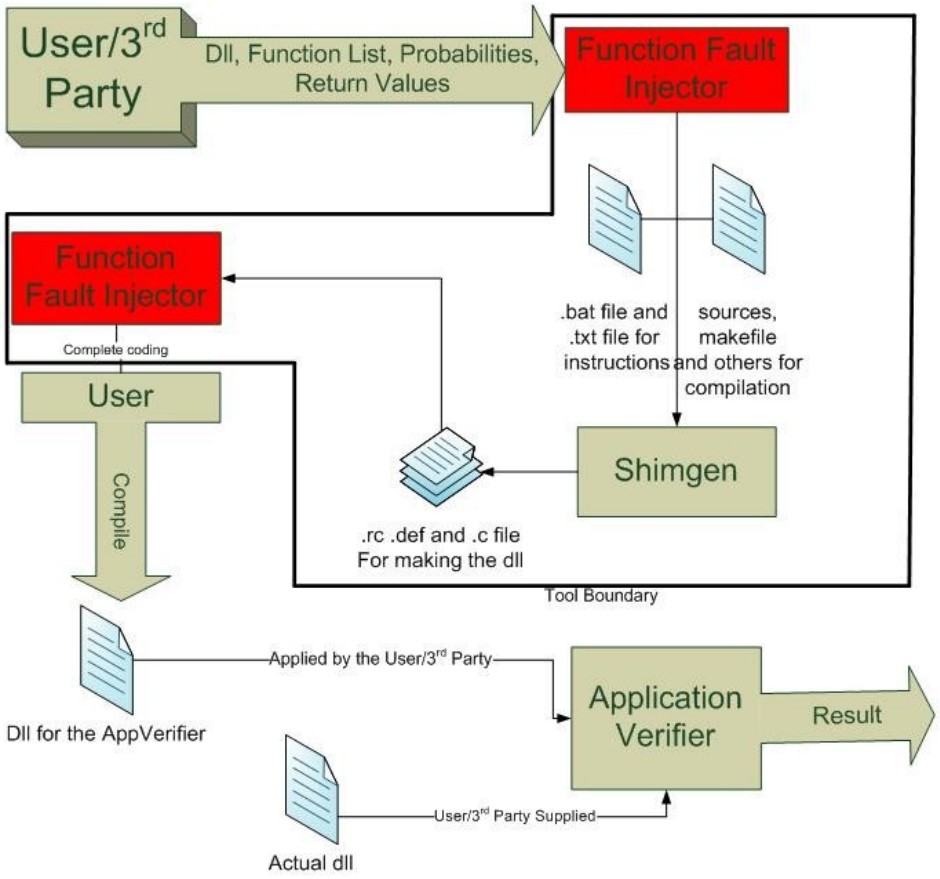


Figure 4 Shim Generation assisted by the FFI

Shim Generation assisted by the function fault injector

1. The user selects the dll as above.
2. He goes to the injection menu to select the details of the functions which have to be shimmed.
3. As he finished, he can save the details to a file for future reuse.

4. Then he is opened with the code which has to be compiled in the build window and applied as earlier.
5. No coding is required as everything is auto generated.

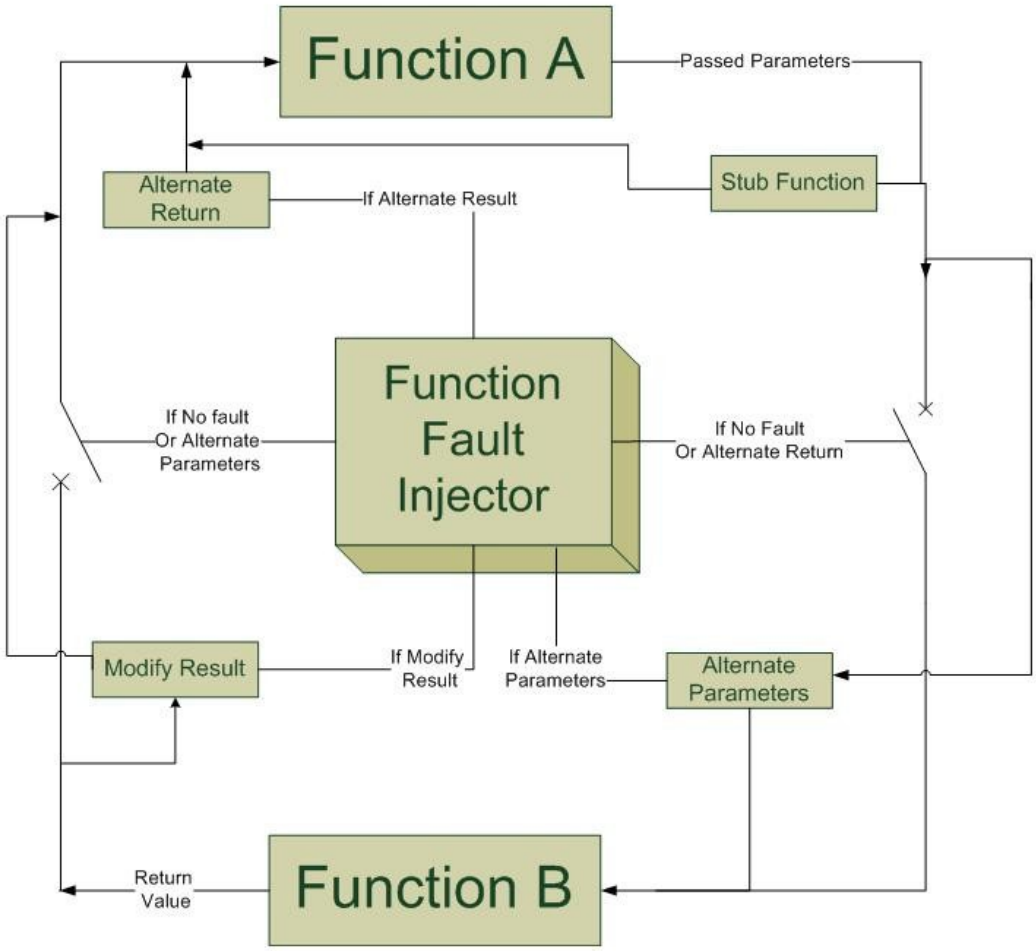


Figure 5 Interception Points

The above diagram is self explanatory as it explains at what parts of the function call when function A calls the function B, the call is intercepted as well as the names given to each of the interceptions.

Data Flow Diagrams

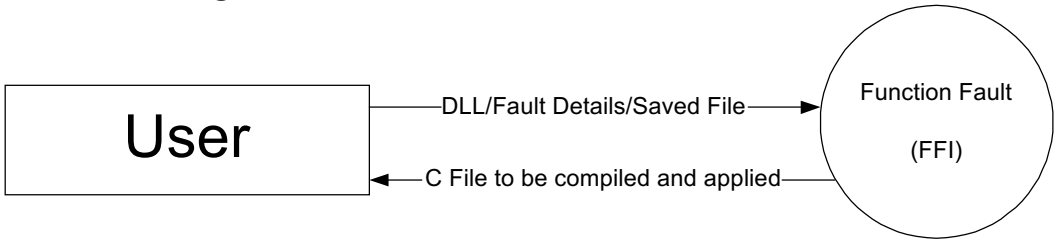


Figure 6 Context Level Diagram

- User passes the dll file that has to shimmed, shimming details, or the saved file and is generated the C file which has to be compiled and applied.
- The passed information is done through the UI that is almost similar to the old Shimgen UI.

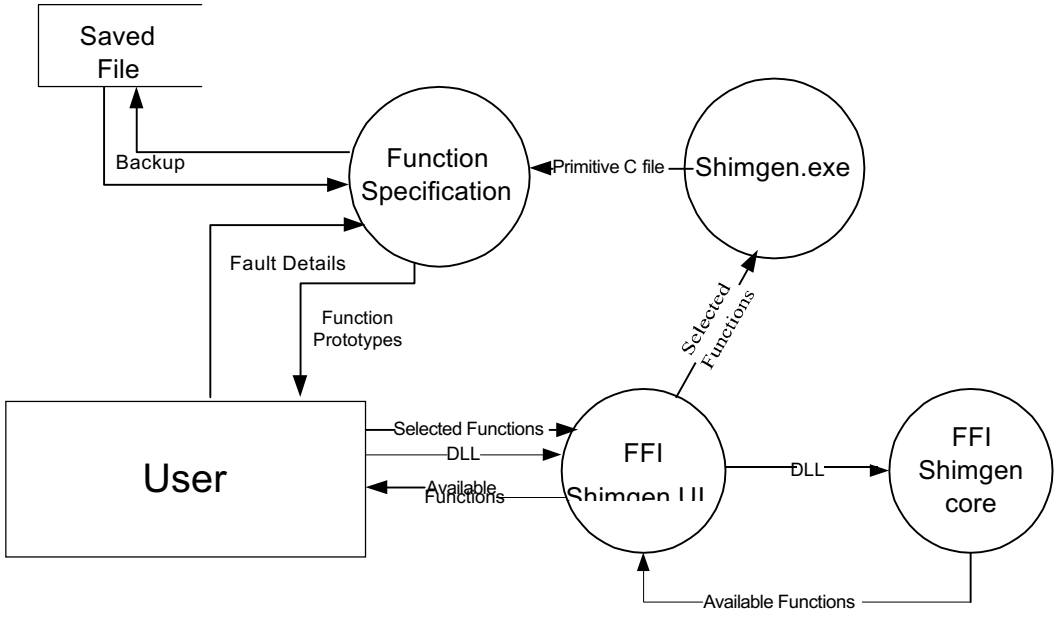


Figure 7 Level 1 DFD

The above DFD clearly describes the four components of the FFI and the interaction between them as well as the flow of data.

Work Details

Steps involved in getting the project to work:

Step 1: What you actually need?

1. To be able to actually able to use the software, you'll need the build window as well as the application verifier. Different Versions of the application verifier are available at Microsoft.com individually or as a part of the the WinCE Test Kit. The one for WinCE 6.0 can be downloaded here. The application verifier has to be strictly of the correct environment and so has to be the build window. Note that a shim created by the WinCE 6.0 build window will not run on WINCE 5.0 or windows mobile (5.0 & 6.0) which use WinCE 5.1.
2. Apart from this you will also need the required testing device, visual studio etc.
3. As far as the files are concerned, you'll need the **.dll** containing the function, its **.pdb** and **.lib** files. These files should be present in the same directory. Besides that, the regular files associated with the build window are required for the compilation process.

Step 2: Starting and selecting the functions.

1. Open the Function Fault injector.exe file.
2. Select the (...) button in front of the Original DLL and select the DLL containing the function that you wish to be shimmed i.e. the function whose return value has to be changed to apply the fault.
3. Browse to the location of the DLL.
4. You can alternatively type the location in the available text box.
5. Do the same for the Output file. Note that the location of output file should be such that the build window should be able to select the proper dlls and provide the headers for compilation.
6. As soon as you select target dll, the list of functions exported by the dll, that can be shimmed appears in the left hand box. Note that the functions in the list that appear are only the exported functions that are not a part of any class. You can view all these functions via dumpbin/exports in the command prompt.

7. If a function does not appear in the list it is probably that either the function is inside a class or not exported by the dll.
8. Select the appropriate functions and click on the add button to add them to the list of those to be shimmed.
9. You can alternatively add all the functions available by the Add All button.
10. To remove a function not wanted you can select it and click remove all.
11. You can save the project at any stage using the File>Save or pressing the Ctrl+S keys.
12. To filter the list of functions and select the functions imported by only a particular dll/exe, you can use the **imported by** button, where you can select the dll or exe and it will filter out the function by those imported by that file. again to see the list of imports you can use dumpbin/imports in the command prompt.
13. Alternatively you can apply an API filter to filter out specific functions.

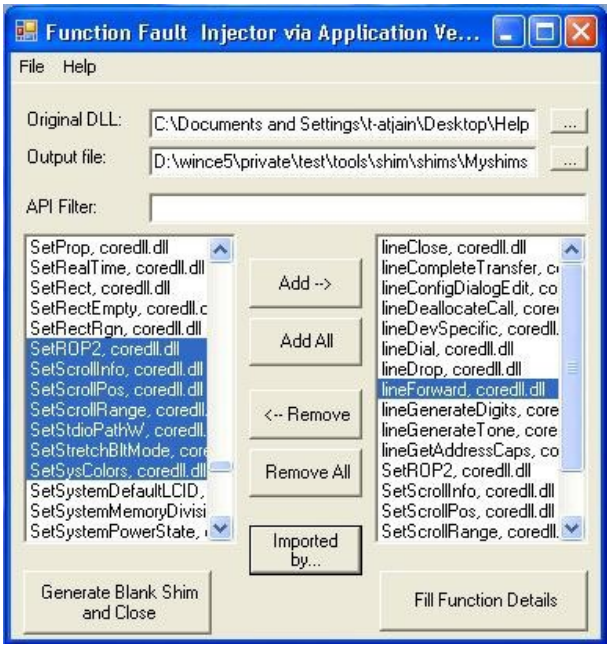


Figure 8 Selecting Functions

API Filter

1. Because of the number of functions being too large in the dlls, selecting and reaching to the correct wanted functions becomes highly difficult.
2. Use the API filter to filter out through the list of functions to search out the function wanted to search.

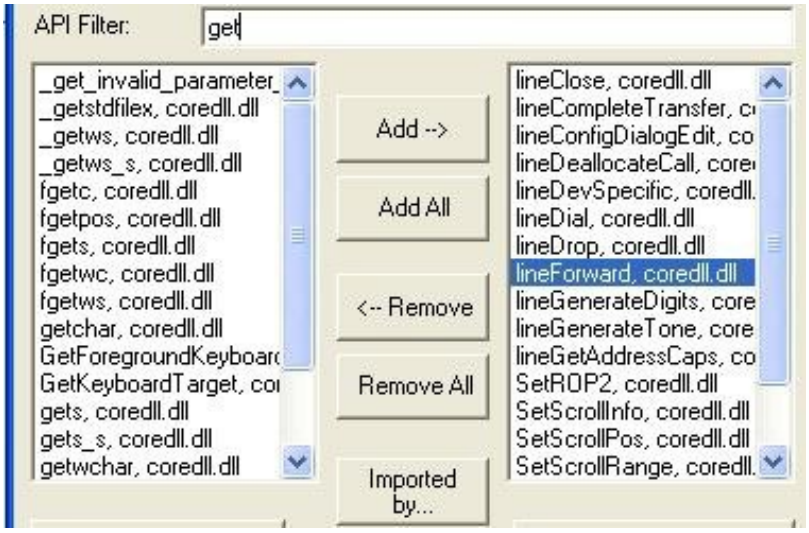


Figure 9 API filter

3. Type in any keywords to find them in a function. Note that it is case-sensitive.
4. To filter out functions beginning with a certain set of letters type '^' before the letters. Eg: ^get filters out all the functions starting with the letters get.

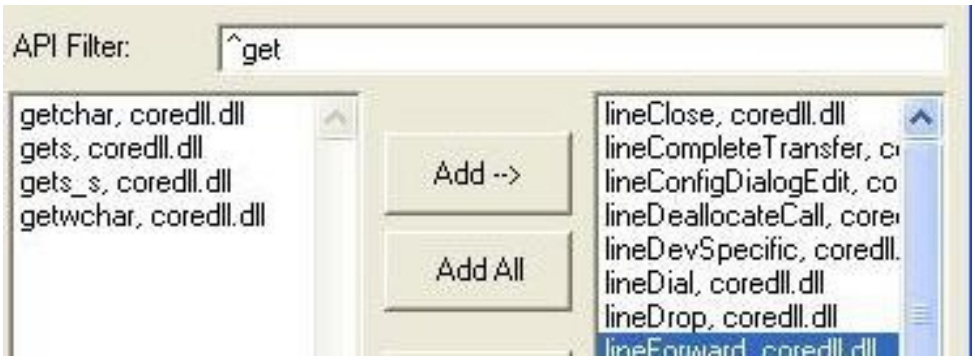


Figure 10 Start with get

5. To filter out all the functions ending with a certain set of letters type '$' sign after the letters. Eg get$ returns all the functions ending with get.

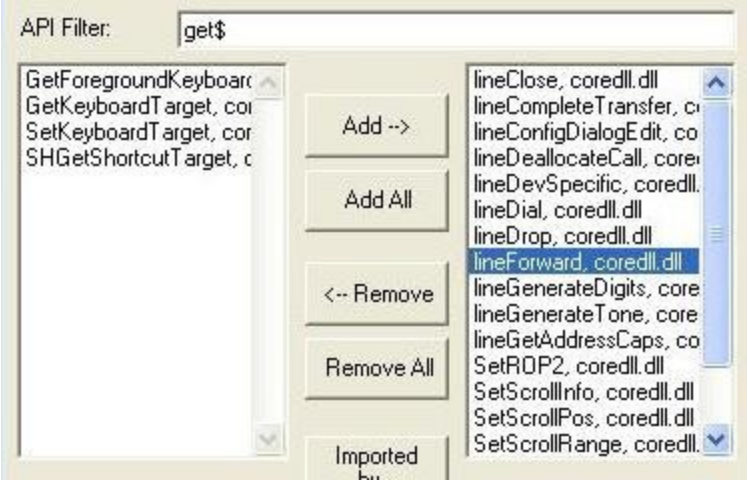


Figure 11 Ending with get

Step 3: Generating Shimgen File/moving to function details.

1. To use the Function Fault Injector as an application verifier shim generator press the Generate Blank shim and Close Button.
2. To use it as the fault injector and to move on to the function details, click on the Fill Function Details button.
3. It may take some time for some of the files to be generated.
4. Note that you lose the original C file if you had loaded the function at this stage and need to regenerate to get it back.
5. Once the files are generated, you are automatically transferred to the Fault Injection specifications Dialog.

Step 4:Filling in the Fault Specifications

1. At this stage you go to the Fault Specification Dialog Box.
2. In the dialog box you are the first function in your list. Here you can specify the details of the faults to be injected. The various types of details are described as
3. To fill in the values, check the appropriate checkbox, and fill in the probability of the specific fault. Note that one function can have at time more than one types of faults with different probabilities.
4. The probabilities can sum up to <=100, as the remaining goes to the original function being called. They cannot be negative.
5. You are allowed to write a only a valid "C" code in the columns.

6. The code should be syntactically correct and should have only 'C' elements. Note that Boolean does not exist in C and so the return type is 0 for false and 1 for true and we have to use that even if it says BOOL as the data type.
7. In the alternate return value, you can put only those values that can be returned by the return statement directly.
8. Only in the alternate return value you can have a value without a semicolon.
9. In all others you will need a semicolon in all the lines.
10. You can use all standard C++ headers or can make your own and add to the project including it in the include section.
11. The navigation buttons are self explanatory and you can use them to easily navigate between the functions.
12. You can save your project any time using File>Save or simply pressing the Ctrl+S.
13. You can build the C++ file anytime by File>Build or Ctrl+B.

Alternate Return Value

Alternate Return Value is the value that is supposed to be returned by the function in case of the probability being the match. Note that the original function is **not** called when the alternate value is returned. Also note that the value to be returned can only be which can be put down directly to the return statement as the program puts it as return <your value>;

This value is the shortest or rather the easiest to implement and among the most effective. For example say if we are shimming the functions that allocates some resources and returns a pointer to the resource like the malloc function, if would not be useful to get the resource allocated and then return, not found. it'll be better if we do not call the actual malloc so as to save resources as the fault is being generated. There this function can come as handy.

Stub Function

A stub function is an alternative function that is run when the original function had to be run using the same parameters and returning a value that is passed to the original function. Coming from the era in stub testing, when the testing is done even before the module is complete by putting in stubs in the place of incomplete blocks, it has been found that these functions come out better testing methods than just crudely changing the return value. Also called code mutation,

these functions aim at providing variety in the return by using at many times random number generators to return numbers at random.

In the FFI, you can write the stub function as any other C function whose inputs are given at the prototype.

You don't need to repeat the prototype or use the function open and close brackets but have to make sure that the function is correct. A common error faced it writing down code that is not 'C" compliant. Even though we are passed bool and string, they are the prototypes not existing in 'C' so you have to use them as what they are in 'C', bools as integers and string as character pointers. Remember to include the required headers in the bottom most column. You can return the values using the normal return function.

You will not need to include windows.h & stdlib.h as they are already included.

For example,

In the FFI, you can write the stub function as any other C function whose inputs are given at the prototype.

You don't need to repeat the prototype or use the function open and close brackets but have to make sure that the function is correct. A common error faced it writing down code that is not 'C" compliant. Even though we are passed bool and string, they are the prototypes not existing in 'C' so you have to use them as what they are in 'C', bools as integers and string as character pointers. remember to include the required headers in the bottom most column.

For example say if the function prototype is: FLOAT Diff (FLOAT Parameter1, FOAT Parameter2);

You can write a function as:

```
int param = (int) Parameter1;

Parameter2+=param;

return Parameter2;
```

or alternatively we can also write functions like:

```
return pow(Parameter1,3);
```

But we'll have to include *#include<math.h>* in the headers section.

Alternate Parameters

This is sort of an API testing technique where instead of totally removing the original function we use it but by changing the parameters passed to it. We alter the function parameters passed by the original function and pass it to the called function. This way we are testing the called function as well as the calling function by returning the output that will be produced when the modified parameters are passed which will be actually a propagated faults. Minimizing or removing propagated faults is another necessity in a program and this routine helps us to test that.

In the FFI, you can write the stub function as any other C function whose inputs are given at the prototype.

You don't need to repeat the prototype or use the function open and close brackets but have to make sure that the function is correct. A common error faced it writing down code that is not 'C" compliant. Even though we are passed bool and string, they are the prototypes not existing in 'C' so you have to use them as what they are in 'C', bools as integers and and string as character pointers. remember to include the required headers in the bottom most column. After you have modified the values, they have to be present in the same variables to be forwarded to the original function. Also note that since all the data types are not supported by 'C' you might get compilation errors for some class of data types. Those remain to be some of the limitations of the tool which will be overcome in the future.

An example of the alternate Parameter Function can be, say if the prototype is FLOAT Diff(FLOAT Parameter1, FOAT Parameter2);

You can write:

Parameter2=0;

Parameter1-=1;

Modified Return Value

Another alteration possible to the given function is a modified return value. in this case we wait for the original function's natural execution and come into action as the execution has finished. After the function has finished executing, we introduce a perturbation function that modifies its

value and send to the main program. In this way the fault values can be made dynamic. More so, if the calling of the function was necessary as it might trigger some other actions, it may prove out to be the only soln. We can still use it as alternate return by setting the result to the return value, and we have more choice.

To use this function using FFI, you can write any C code in the given block. Note that the actual result in present in a variable called result whose data type will be as per the default return type of the function. You will have to store the value to the result statement itself unless you explicitly use the return statement. You are still available with the passed parameters and among the rarest options that you can still use them.

For example say if the function prototype is: FLOAT Diff(FLOAT Parameter1, FOAT Parameter2);

You can write a function as:

result -=3;

or alternatively we can also write functions like:

result += Parameter1;

or

return Parameter1-result;

The Generated Files

The default generated files contain all the information required to be built and applied by the application verifier.

Generated C code:

This file contains the shimmed dll, where the user inputs have been plugged in at appropriate positions. Though I suggest that you use the application UI to modify the contents but alternatively, you can directly modify the contents present in the file, do bring the desired changes to the solution. it has the probability generation, application of all classes of the functions as well as the code to incorporate the registry already written.

Rest of the files is same as generated by the normal shimgen program:

DllMain.c

This file contains the implementation for DllMain. Customize this to perform any initialization or termination tasks required.

OptionsDlg.c

To add a pane for this shim in the options dialog of the settings manager, implement and export GetOptionsDialogProc. This function must return a dialog proc and resource template to be used in the property sheet. Customize the dialog resource and procedure for your specific shim.
You are not required to implement an options pane. You can also implement this in another fashion (apart from the options panel).

ParseCommand.c

To add a custom command to Platform Builder's CE Target Control, implement and export the ParseCommand function.

QueryShimInfo.c

The function QueryShimInfo is a required entry point of a shim dll. Do not change anything in this file; instead, customize ShimInfo.rc.

RemoteUI.c

Get/Set/FreeShimSettings are not required entry points for a shim dll. Implement these only if you want to send application-specific run-time settings to the device.

ShimInfo.rc

Customize this file with strings that describe your shim. The friendly name will be displayed in the right pane of the settings manager, and the description will be displayed in the lower pane.

Step 5: Compilation

If the build window is properly set up and the folder is present in the write place (where the header are accessible, ideal being: private>test>tools>shim>shims>myshim), building should come out smooth. If there are compilations errors in the c file probably our code is incorrect. You

can modify the code by opening the saved file in the UI or directly using the C file whose changes get lost once you open a FFI '.fib' file.

add.c - Notepad

File Edit Format View Help

```
//FFIBFv1.0
#include <windows.h>
#ifdef UNDER_NT
#include <tchar.h>
#endif
//FFI Generated additions to the shim
//Adding in custom headers
#include<stdlib.h>   // For the probability generation function's rand

//Adding user requesred headers - may contain errors if user typed an incorrct c header or format.
// From Function : Add
#include<math.h>

//This is the Random Number generator function which generates the random number to match the probability
int RandomNumber()
{
        return ((int)  (10000*(rand()/(RAND_MAX + 1.0))) );
}

//Now adding the prototypes of all the shimmed functions
// From Function : Add
int Add ( int a, int b);


// NOTE: Fill in the following stub functions. This code is a normal user dll, so
// you can call any code you'd like. To pass the call on to the 'original' function,
// simply call the original api; the application verifier engine will 'protect'
// shim dll's from having their imports redirected.

INT __stdcall APIHook_Add(
    INT a,
    INT b)
{
    INT result;
   int ThisTimesProbability = RandomNumber();
    DWORD dwValue;
    DWORD dwType;
    DWORD dwCount = sizeof(DWORD);
    HKEY hKey;
    DWORD dwDisposition;
    int InputtedProbability = 10000;
    int ProbabilityOfAlternateParameters=0;
    int ProbabilityOfAlternateReturn=0;
    int ProbabilityOfModifyResult=0;
    int ProbabilityOfStub=0;

    //Do not Modify, Checking values from the registry
    if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, TEXT("Software\\Microsoft\\Application Verifier\\Function Fault Injector\\AddDll.dll\\Add"), 0, KEY_ALL_ACCESS, &hKe
        {


        // The registry does not contain an entry for the function => it is first run, Creating Keys
        RegCreateKeyEx( HKEY_LOCAL_MACHINE, TEXT("Software\\Microsoft\\Application Verifier\\Function Fault Injector\\AddDll.dll\\Add"), 0, NULL, 0, 0, NULL,

        if (dwDisposition != REG_CREATED_NEW_KEY && dwDisposition != REG_OPENED_EXISTING_KEY)
                printf("\nError creating the desired subkey (permissions?).\n");

         //writing the values to the probilities in the registry

         InputtedProbability=3000;
        if (RegSetValueEx(hKey, TEXT("ProbabilityOfAlternateReturn"), 0, REG_DWORD,(const BYTE*) &InputtedProbability, sizeof(int))!=ERROR_SUCCESS)
            printf("\nThe value of the key was not set\n");

        }

     //Reading values
    dwValue = (DWORD)0;
    RegQueryValueEx ( hKey, (LPTSTR)TEXT("ProbabilityOfAlternateReturn"), NULL, &dwType, (LPBYTE)&dwValue, &dwCount );
    ProbabilityOfAlternateReturn=(int) dwValue;

     ProbabilityOfAlternateReturn = ProbabilityOfAlternateReturn + ProbabilityOfStub;
```

Figure 12 Generated C File

```
BUILD: [00:0000000153:PROGC ] Saving D:\wince5\private\test\tools\Build.dat.
BUILD: [00:0000000155:PROGC ] Done.
BUILD: [00:0000000156:PROGC ]                          Files  Warnings  Errors
BUILD: [00:0000000157:PROGC ] Midl                         0         0       0
BUILD: [00:0000000158:PROGC ] Message                      0         0       0
BUILD: [00:0000000159:PROGC ] Precomp Header               0         0       0
BUILD: [00:0000000160:PROGC ] Resource                     3         0       0
BUILD: [00:0000000161:PROGC ] MASM                         0         0       0
BUILD: [00:0000000162:PROGC ] SHASM                        0         0       0
BUILD: [00:0000000163:PROGC ] ARMASM                       0         0       0
BUILD: [00:0000000164:PROGC ] MIPSASM                      0         0       0
BUILD: [00:0000000165:PROGC ] C++                          0         0       0
BUILD: [00:0000000166:PROGC ] C                            7         0       0
BUILD: [00:0000000167:PROGC ] Static Libraries             0         0       0
BUILD: [00:0000000168:PROGC ] Exe's                        0         0       0
BUILD: [00:0000000169:PROGC ] Dll's                        2         0       0
BUILD: [00:0000000170:PROGC ] Preprocess deffile           2         0       0
BUILD: [00:0000000171:PROGC ] Resx                         0         0       0
BUILD: [00:0000000172:PROGC ] CSharp Compile               0         0       0
BUILD: [00:0000000173:PROGC ] Other                        0         0       0
BUILD: [00:0000000174:PROGC ]
BUILD: [00:0000000175:PROGC ] Total                       14         0       0
```

Figure 13 Compilation of the file

Step 6: Execution

Shim execution is the easiest step of them all. Copy the shim in the appropriate folder in the device, run appverifier and then through the command line (Target Window in case of devices) type:

appverif -m <module to be tested(not the dll> -s <shim to be applied>

In mobiles you can also create a link file to use. Or can write

<no of legal characters following the #>#appverif -m <module to be tested(not the dll> -s <shim to be applied>

Alternatively, a shim cam be applied to all calling functions as:

s appverif -m {all} -s <shim .dll>

To remove the applied shim use:

appverif -c

This removes the application of the shim.

You can change the probabilities dynamically - i.e. even when the application is under execution. All the probabilities are stored in the registry as under *HKLM>Software>Microsoft>ApplicationVerifier>FunctionFaultInjector>[dllname]>[function name]*

You can delete the above value to reset to the probabilities you had specified in the shim.

You can check whether the shim you have applied is actually applied by going to:

HKLM>Software>Shimgen>[module to be tested]>[applied shim]

```
Windows CE>appverif -m calldemo.exe -s shim_heap.dll
Verifier loader: SUCCESS
Windows CE>appverif -m calldemo.exe -s shim_calldemo.dll -opt
Verifier loader: SUCCESS
Windows CE>s calldemo.exe
Windows CE>_
```

Figure 14 Shell Prompt Loading

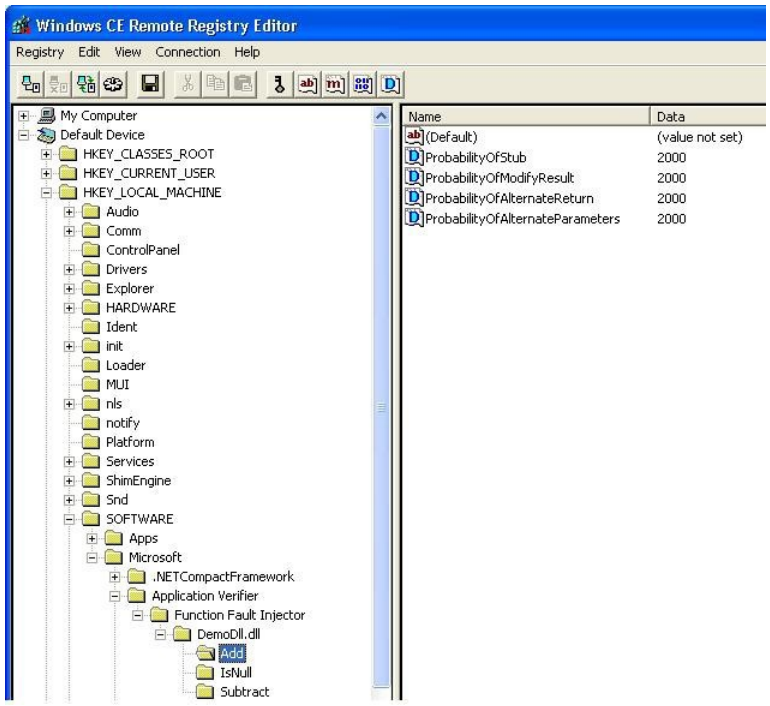


Figure 15 Values at registry

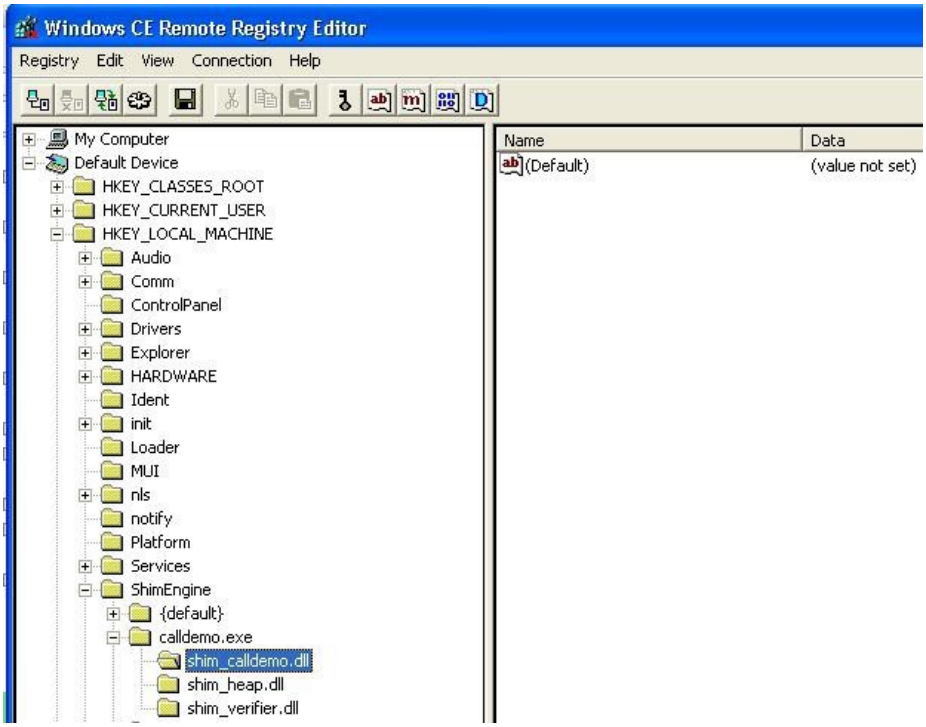


Figure 16 Loaded shim at registry

Step 7: Maintenance

The maintenance refers to changing the values of the probabilities, the functions etc. A fast method of changing the probabilities is as discussed in Execution earlier. But generally you

might want to go back to the fault specification form to add stuff or remove it. This can be done by the use of the saved Fault Injection Backup (.fib) files. Go to the fault injection tool and open then via the file menu. You can modify as you wish and rebuild. You can also add/remove functions at your will. Just remember that after compilation when you reuse the stuff you have made, you'll have to **clean up the registry** from the probabilities set, without which those probabilities will continue to be used. Not that you do not need to remove shim application to paste new. **You can paste a new shim in place of the original one even when it was being applied.** The application verifier will automatically shift to the new shim, although you will have to clean up th registry for the new probabilities to apply.

UI Details

1. **Shimgen UI**

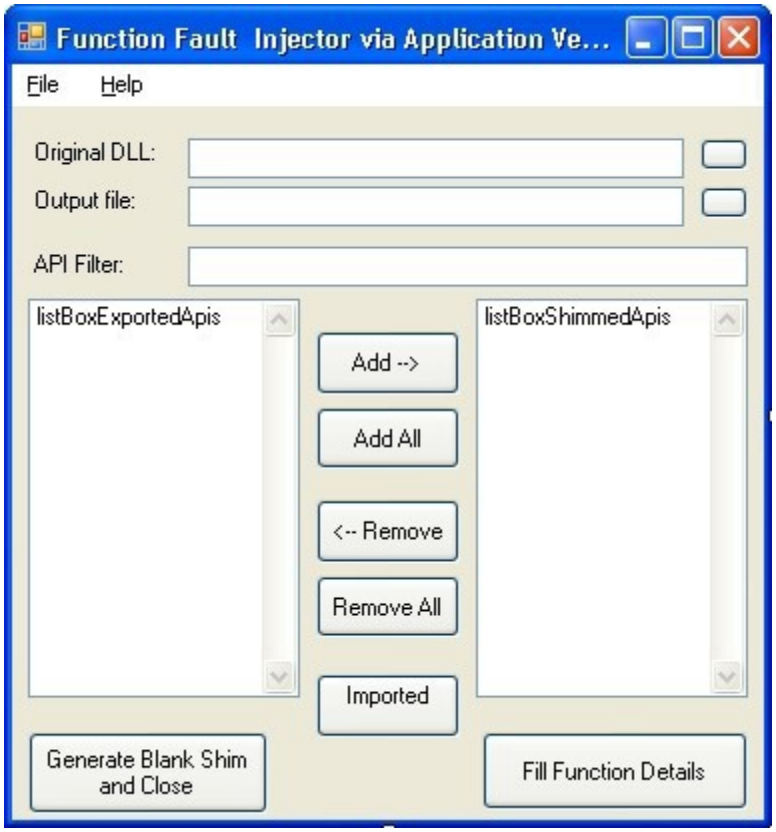


Figure 17 Shimgen UI

Menus

File

New: Create a shim project.

Open: Open an existing shim project (.fib)

Save: Save the existing project.

Exit: Terminate the application.

Controls:

Box Original Dll: Contains the location of the dll to be shimmed.

Box Shim Dll: Contains the location where the shimmed dlls C file will be located.

Box Api Filter: You can use the API filter to search the functions.

Box Exported Apis: Contains list of available functions in the dll.

Box Shimmed Apis: Contains list of functions to be shimmed.

Add: Adds the selected function/functions in To Be Shimmed list.

Remove: Removes the selected function/functions from the To Be Shimmed list

Add All: Add all the functions in the list to be shimmed.

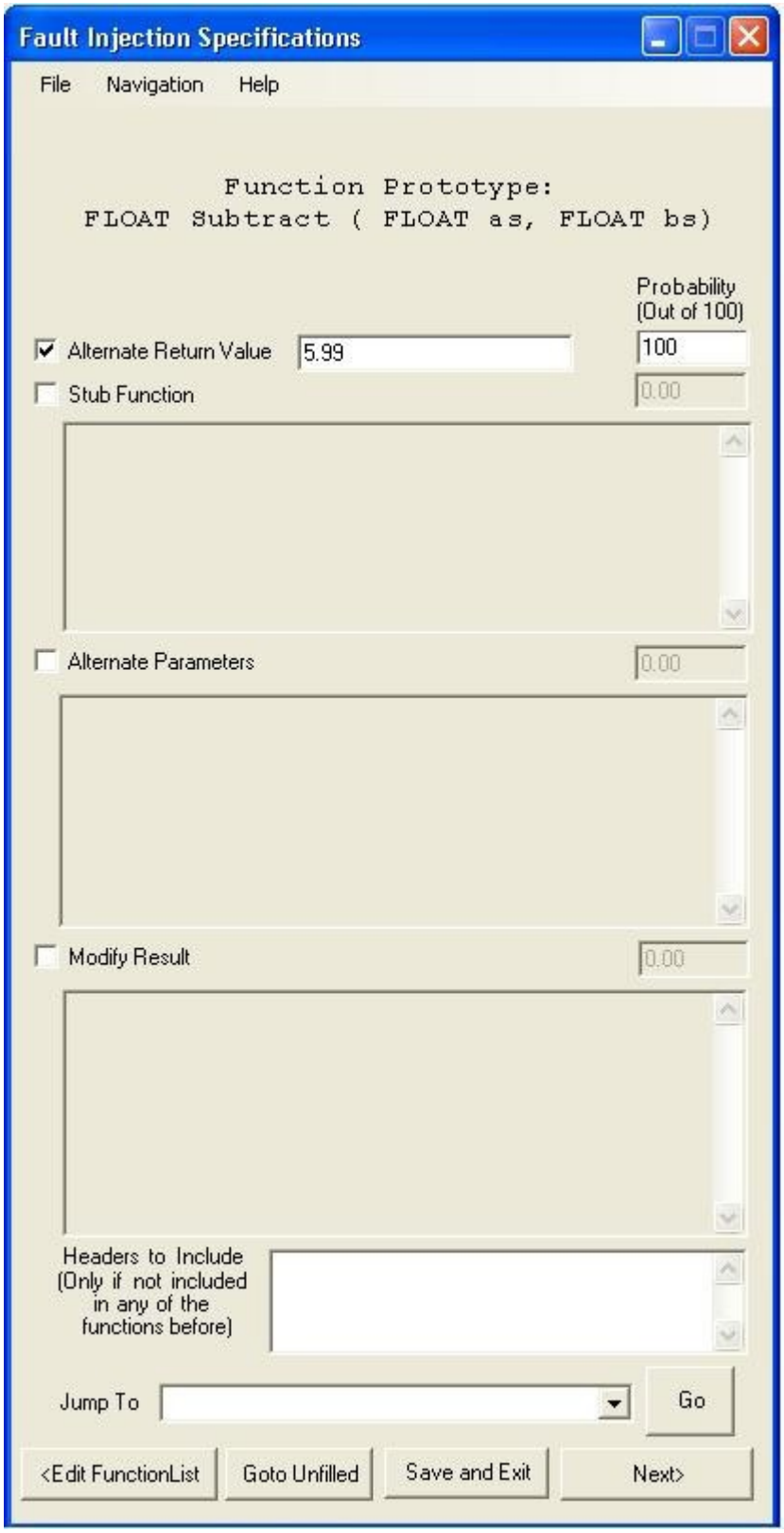
Remove All: Remove all the functions from the list of those to be shimmed.

Imported by...: Helps to minimize function list to only those imported by a dll or exe.

Generate Blank Shim and Close: Act as shimgen and close with a blank shim.

Fill Function Details: Go to the fault specifications form

2. Fault Specification UI



Menus

File

Build: Build the current Project that is filling in the code into the C file that has to be compiled.

Save: Save the current document and all its information.

Save As: Save As a new document.

Exit: Terminate the application.

Navigation

Previous: Navigate to the previous function that has to be shimmed.

Next: Navigate to the next function that has to be shimmed.

Edit Function List: Go back to the main form where you can edit the list of functions.

Goto Unfilled: Go to the next unfilled function in the list.

Controls:

Alternate Return Value: Specify the alternate return value here.

Stub Function: Specify the stub function here.

Alternate Parameters: Specify the function with alternate parameters here.

Modify Result: Specify the modify result function here.

Included Headers: Add the headers for the above that are not a part of the headers added in other functions.

Previous: Navigate to the previous function/Edit Function List if at the first one.

Save and Exit: Save the current project and exit.

Next: Navigate to the next function in the list/Build and Finish – Generate code into the C file and exit.

TESTING

- Unit testing was done for each component as it was being made.
- Use of string, Boolean and integer type as well as the advanced DWORD type fields were tested.
- The generate file was tested by the visual studio syntax checker and many of the standard statements were removed.
- The entire code was tested manually by the Team as well as on the live environment as a Beta test on the day of its first demo.
- System testing was carried out along with the check of the memory leaks with the Application Verifier itself.
- The end users of the software, the testing team carried out automated testing using various available tools.

RESULTS, CONCLUSIONS AND FUTURE SCOPE

The produced tool was found useful enough to be integrated into the system to be used whenever the need arises. Though there could be some features that can be added in the future versions/releases of the project:

- Inclusion of mangled function names and C++ functions with dlls that do not have extern 'C'.
- Ability to produce shims outside the build window and development of a build environment through a Visual Studio Project file.
- Addition of functionality to call functions within the same dll.
- Extension to produce a .CPP/.CS file instead of .C for more features and functions.
- Building in some standard test cases and integrated drop down selection from within the tool rather than filling them up manually.
- Integration into the Application Verifier UI for removal of the need of the build window and addition of ability to work directly from the device.

REFERENCES

1. Application Verifier Documentation.
2. Application Verifier Discussions group and posts.
3. MSDN article on AppVerifeir: http://msdn.microsoft.com/en-us/library/aa480483.aspx
4. Internal documentation on Application verifier.
5. Books on C# at Safari Books Online: www.safari.oreilly.com/9780596514822
6. Books on WinCE programming from the internal help files and tutorials, along with many of the MSTE courses.