Using Binary Code Rewrite to Bypass License Checks

Tevfik Kosar* Mihai Christodorescu Rob Iverson
Barton P. Miller

Computer Sciences Department
University of Wisconsin, Madison
1210 W. Dayton St.
Madison, WI 53706-1685 USA
{kosart,mihai,riverson,bart}@cs.wisc.edu

April 8, 2003

Abstract

A common method of enforcing software license terms is for a program to contact another program, called a license server, and ask for permission to run. This study attempts to bypass these license checks in a commercial product through runtime code modification, using the DynInst library.

The programs chosen as victims for this study are Adobe FrameMaker, the Purify family of programs, and MatLab. We successfully bypass the FrameMaker licensing checks, allowing full use of the product when the license server is unavailable. Limitations in DynInst prevent similar results with Purify or MatLab. A set of powerful tools has been developed and used in the process, and their generality should simplify similar license bypassing efforts on other software products.

Key words: *System security, intellectual proporty protection, cyber cime, dynamic instrumentation, bypassing licence checks, binary code rewriting.*

1 Introduction

Most modern commercial products use some form of checking that the user is legally authorized to run the program. Such checks prevent piracy, and enforce the software vendor's product licensing terms. The widespread implementation involves getting some data from an external source (such as a protected file, or a secured network server) and verifying it for validity.

This study tested the strength of such algorithms, by trying to bypass them using dynamic instrumentation [5] and attain full program functionality when the license data cannot be obtained. We developed several tools that can help analyze a running program, without any prior information about the executable and without access to source code. We used a binary rewriting[1] library called DynInst [1] that allowed us to keep a high-level view of the target program. All of the work described in this paper was performed on an UltraSPARC IIi machine running Solaris 2.6.

*Contact Author. Tel:+1(608)262–5945

[1]Binary rewriting is the process of modifying compiled binary code without knowledge of the source code.

We started from the premise that a legally licensed product is available, and that product downtime due to non-product related failures is an unfortunate possibility. In the case of a network-based license verification, the machine on which the license server runs can go down, but it should not affect the users' ability to run the licensed product.

Licensed checking data can be obtained from a file, or from a license server. We focused in this study on products that use a license server, but our results can be easily extended to products using local license files. This is due to the fact that the network communication and the local file system have similar access Application Programming Interfaces (APIs).

2 DynInst

DynInst is an API that allows insertion of code and modification of subroutine calls inside a running program. It also extracts program information that allows the recreation of the call graph, of function types and arguments (when debugging information is present in the target program). The code insertion is performed by displacing instructions and inserting appropriate jumps [2] to and from the inserted code, which are called *snippets* in the DynInst parlance. DynInst can attach to the process to be modified (called the *mutatee*) if the process is already running, or can start the mutatee. The controlling program (called the *mutator*) uses the DynInst functionality to insert code into the mutatee (see Figure 1). For further information about DynInst, please see [3].

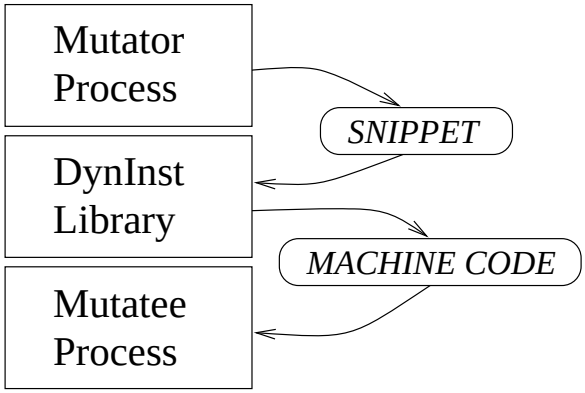

Figure 1: Overall architecture of a DynInst-based application.

DynInst preserves program behavior, although a slowdown was noticed in our experiments, most likely due to the fact that cache locality properties are destroyed. We use all the functionality available in the API, and at points we had to supplement it with our own code. We hope that some of our work will be incorporated back into DynInst. A short list of limitations we encountered in DynInst and of suggestions for further development is part of section 6.

3 Attack Methods

We approached the problem of bypassing license checking from several perspectives. We attempted to see the program as a black box, from which we capture the I/O for later replay. Also, we have traced the program to understand where the license checking is performed and to modify those sections of code. The combination of these two methods helped us understand the program behavior, a necessary step in modifying the license checking. Both methods are detailed in the following subsections.

3.1 I/O Replay

The problem of bypassing license verification is similar to cracking a security protocol. They both attempt to gain access to otherwise denied resources (data or functionality) by modifying the expected behavior of the authentication engine. This suggests that some of the attack methods in the cryptography world are valid candidates in our case. We developed an attack technique using I/O replay, which involves capturing the communication data and feeding it back to the validation module on a later run.

Our implementation involves a module which uses DynInst to attach itself into the mutatee at any place which performs I/O, thus making it applicable in analyzing the behavior of programs which communicate with license servers. The module replaces the `open()` library function, among others, with a custom version of `open()`. This new `open()` sets up a mirror file for later use by `read()` and `write()`, which are modified to copy their data into the mirror file. In this way we can save the contents of any temporary files, any socket activity, any data worth analyzing. The idea is to find successful client-server transactions and save them for later replay (perhaps with slight modification if some sort of time stamp or machine identification is included in the transaction). These processes are illustrated in Figures 2, 3 and 4.

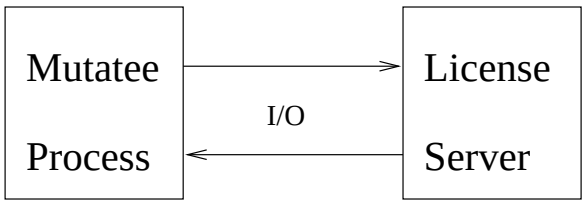

Figure 2: Normal communication pattern.

We have succeeded in evolving the module to the point where it can track most I/O performed by the mutatee. This attack method provides a large amount of data from the mutatee, but processing this information and replaying it becomes complicated by timing issues. We turned to full program tracing in order to gain some context information necessary for a successful replay attack.

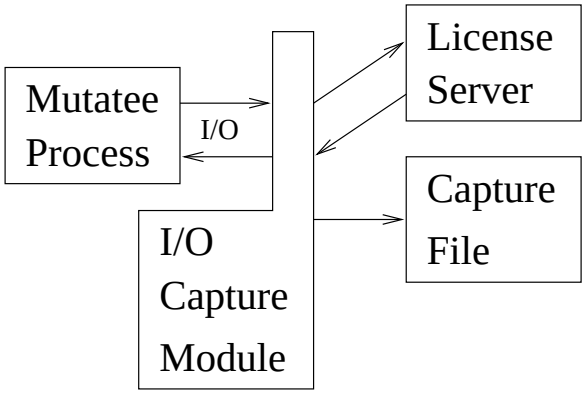

Figure 3: The process of capturing the I/O traffic.

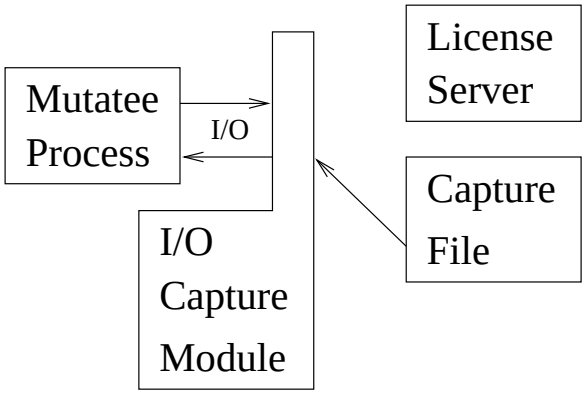

Figure 4: The process of replaying previously captured I/O traffic.

3.2 Program Tracing

Full program tracing is another method we tested in our experiments. By tracing the execution of the target program in case of success and in case of failure and by comparing the two traces, one can locate where the license checking occurs. More importantly, functions to be skipped or replaced to avoid the failure of the license check can be determined..

Using DynInst, tracing is easily implemented: we insert code at the beginning and at the end of a function in the mutatee. The code will trigger output which the mutator can interpret. Depth in the call stack and order of calls can be determined, as can the return values from each function. Although highly intrusive, this method preserves the program semantics, and provides us with an understanding of the mutatee inner workings.

We attempted to perform tracing by inserting the trace snippets in every function. This method is not successful with large programs. Since a lot of code is inserted, DynInst increases the size of the target program, and at some point the virtual memory system is overloaded and starts thrashing. When the available physical memory fills up, the trace program crashes, possibly due

to the memory handling code. We improved our algorithm to perform incremental tracing.

The benefits of incremental tracing are multiple: limited memory impact, ability to trace only the functions of interest, and ease of stopping and restarting the tracing on demand, while the mutatee is running. A quick outline of the algorithm follows:

1. On entry to a function: insert breakpoints before and after all the calls made by that function.
2. Before a call to a function: insert breakpoints on entry to and on exit from that function.
3. After a call to a function: remove the breakpoints on entry to and on exit from that function.
4. On exit from a function: remove the breakpoints before and after all the calls made by that function.

As one can easily determine, now the code insertions closely follows the call stack. There are a couple of problems generated by this approach: calls made through function pointers cannot be followed using only DynInst functionality, callbacks cannot be detected (unless **all** the calls, including library calls, are traced - quite expensive in our experience), and the execution time increases almost quadratically. We have developed a solution for the first problem, which we present in the next section. Callbacks are not completely handled, but we are looking into possible workarounds. Execution time cannot be improved too much, but our belief (backed by experimental observations) is that during tracing, information is gathered that leads to a converging set of functions to trace, thus improving execution time.

3.2.1 Dynamic Calls Through Function Pointers

One of the obstacles to the successful analysis of a program has been that many call points in the program are dynamic, i.e. calls through function pointers. These function pointers show themselves through DynInst returning a NULL pointer when requesting information about the called function. Since the target of the call cannot be determined until reaching that point in the code, we have developed a different method to determine which function is being called than we have for static calls.

The functions pointers take the form of the SPARC "jump-and-link" (`jmpl`) instruction. This instruction causes an unconditional branch to either a register plus another register or a register plus a constant. The most common occurrence of this instruction is the two-register version, with the second register being `%g0`, which always holds the value 0 on SPARC.

This lead us to the desire to instrument these call points to find out which functions were being called in order to produce a more accurate callgraph. The method we developed, known as *redirection*, is clean and conceptually simple. During initial instrumentation, the dynamic call is replaced with a static call to a library function. The value in the register is passed as an argument to another library function, `DC_redirect`, which communicates that value to the mutator and stops on a breakpoint. The mutator does the desired instrumentation or recording and wakes up the mutatee. `DC_redirect` then calls the function normally. This process is shown in Figure 6.

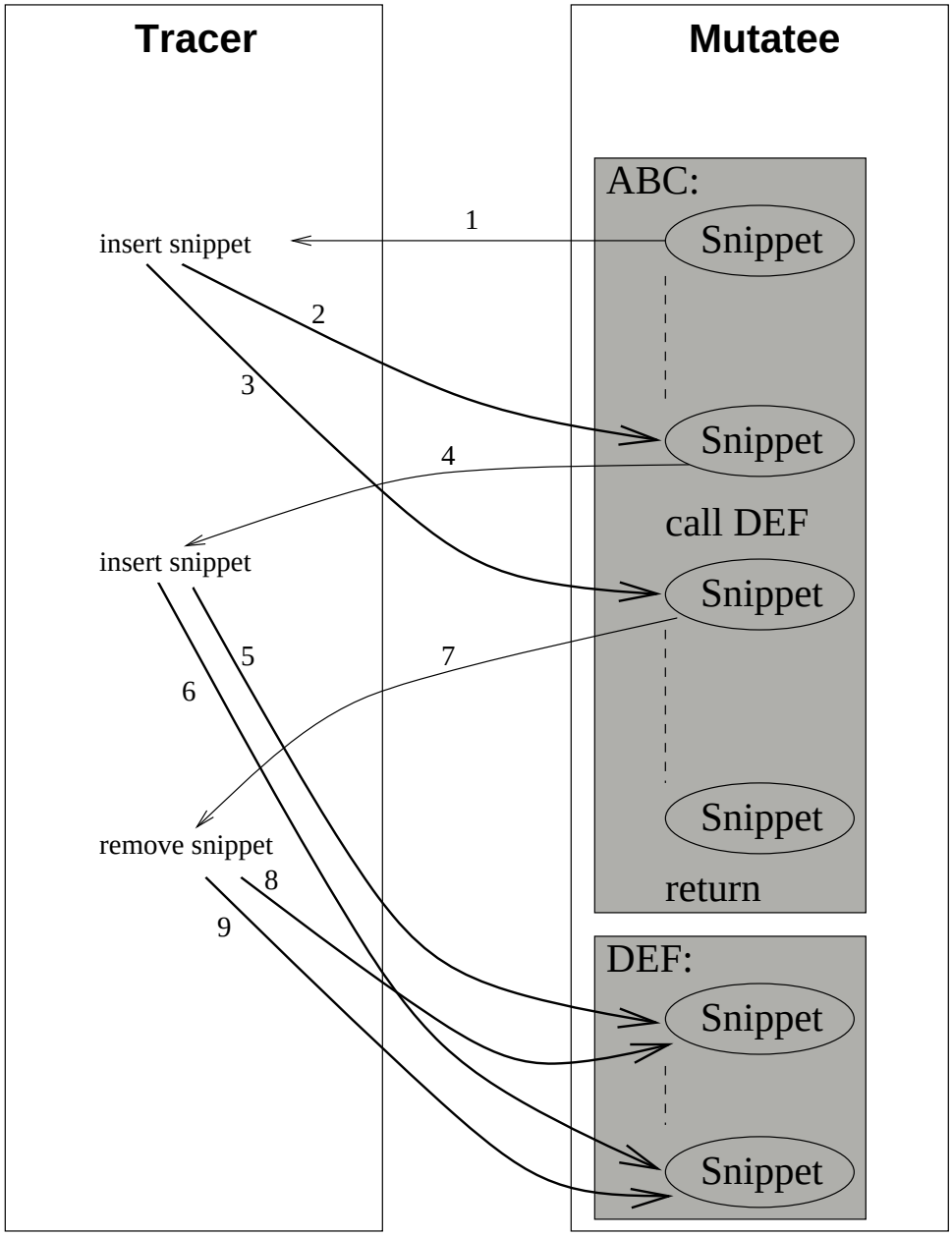

Figure 5: The interaction between the tracer and the mutatee for one function.

Since this sort of redirection is not supported by DynInst, we had to deal with many interesting complications. First of all, there is no functionality to read data or code from an arbitrary address in the mutatee's address space; provisions are only made for reading the values of *variables* in this address space. Through the extremely unwholesome, but strangely fulfilling action of hacking the DynInst header files, we are able to attain this functionality, and thus are able to analyze the instruction. At this point, we create a customized static call instruction to the initial redirector.

This method currently works for function calls made through 24 of the 27 possible register

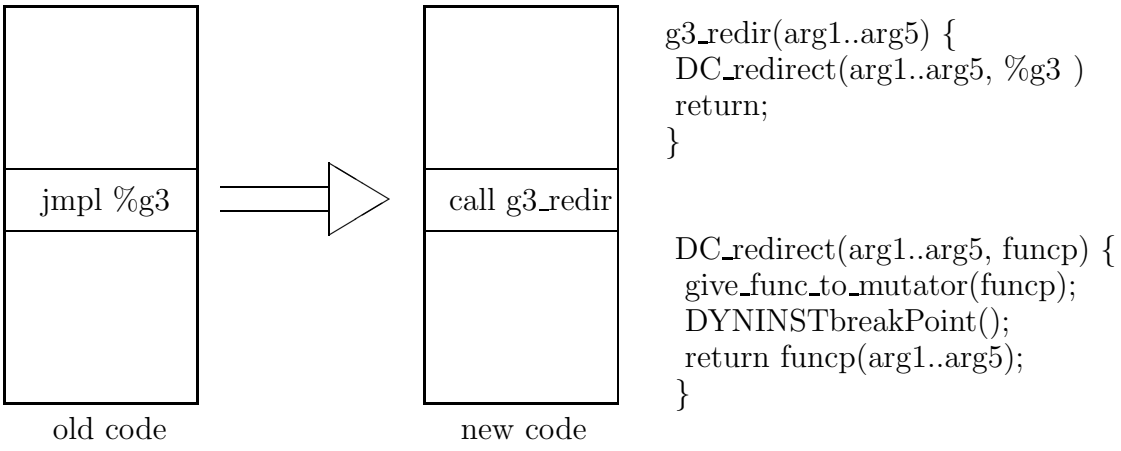

Figure 6: Dynamic Function Pointers

targets, as long as the functions do not take more than 5 parameters. This limitation, which affects few functions, is currently being removed. Return values are preserved in all cases. We believe this method is useful and it can greatly improve the power of our techniques.

4 Target Programs

We applied the presented methods to three common applications that perform license checking: Adobe's FrameMaker, Rational's Purify, and MathWorks' MatLab. The following subsections describe in more detail our efforts to bypass the license checks in these applications.

4.1 Adobe FrameMaker

FrameMaker is a complex software product, with numerous callbacks and special code that complicates analysis. The default installation uses a shell script to start the program. The script detects the architecture of the machine it is run on, reads some configuration data, sets environment variables, and finally calls the correct executable. In spite of the fact that the script sets more than 10 environment variables, we managed to limit the number of relevant ones to 4.

The FM_FLS_HOST variable provides the FrameMaker executable with the information necessary for the license check functionality. If the variable is set correctly, the executable performs the license checking and runs successfully. If the variable is empty, the executable will pop up a dialog box asking the user whether to run in demo mode or exit.

Having a sure, reproducible way of controlling the behavior of the license checking module, we then traced FrameMaker. There are 86 subroutines that perform some kind of license-related function, and, by transitivity, the set of functions calling any of the license-related functions contains 1,181 entries. The total number of functions in FrameMaker is 16,943 (including dynamic library functions), and by using our tools we have cut down the number of functions to be instrumented to less than 10%.

In the next few paragraphs we describe the license checking mechanism employed by FrameMaker, and after we explain how this mechanism was bypassed to provide full program functionality in the absence of license server connectivity.

During the initialization phase, FrameMaker's `main()` calls `NlOpenLicenses()`, which contacts the license server, retrieves the license data, and stores it in memory for later use. If the FM_FLS_HOST is not set, `NlOpenLicenses()` fails, and no license data is obtained. In either case, the program continues initializing, going through all the steps for setting up X windows, reading the user defaults, the Most-Recently-Used document list, and other configuration settings.

At the end of the initialization phase, `main()` calls `NluiCheckLicense()`. This function checks the license data in memory. When the license data is missing, FrameMaker pops up a dialog box asking the user whether to go into demo mode or exit. It then calls `ChangeProductToDemo()` or `FmExit()` (depending on the user input).

The program continues execution and displays the standard FrameMaker toolbar. The license checking validation is performed once more, when the user moves to open or create a document. An X library callback verifies the license data, and calls the appropriate functions if everything is OK. If the license data is missing or invalid, text warning about the lack of a license is displayed, and access to the full functionality is denied. Otherwise, full functionality is enabled, and the user can proceed with creating / editing / saving documents. The license checking validation code is called from time to time (at random intervals), while the user edits a document, even though an initial license check validation was done.

The behavior exposed above was discovered using our DynInst-based tools. We are working on more complete control over function calls, using dynamic call disambiguation (as presented in section 3.2.1), and over the data transfer through network interfaces.

We successfully bypassed the licensing checks, by the following procedure:

1. Allow the retrieval of the license data to fail.
2. Prevent FrameMaker from entering demo mode, by modifying the functionality of `ChangeProductoDemo()`.
3. Bypass the first license data validation, by skipping over the sequence of code that performed it.
4. Modify all later license data validations to always succeed, regardless of the presence of the license data in memory.

Using this *controlled failure* mode, we managed to run a successful FrameMaker editing session, without a license being checked out from the license server.

4.2 MatLab

Another piece of software that we tried to instrument was MatLab. MatLab is invoked by a Bourne Shell script which first sets the environment variables, then determines the machine

architecture, and finally starts the appropriate executable.

MatLab maintains a local license file, given by the environment variable `$(LM_LICENSE_FILE)` (by default set to `$(MATLAB)/etc/license.dat`) on startup. The MatLab startup script tests for the existence of this file and the syntax of the contents. If the test fails, it exits with an error message without invoking the MatLab executable. If the test succeeds, and all other environment variables are set correctly, the MatLab executable is invoked by the script.The license check is performed just at the start of the executable, and if it fails, the executable exists with an error message. If the license check succeeds, the application continues execution.

We were not be able to instrument MatLab using the DynInst library, since the standard DynInst call to create a process fails and exits without creating the process and without any error messages. We were able to attach to a running MatLab process using an alternate API call, but we still could not instrument it since, at this point, DynInst gives a warning message:

```
function _start has call to same location as call, NOT instrumenting
```

and does not let us to proceed any further. We have reported this bug to the developers of DynInst, and postponed our work on MatLab until it gets fixed.

4.3 Purify / Quantify / PureCoverage

The Purify, Quantify, and PureCoverage programs are software development tools which all follow the same pattern of execution. They run on a program's object code, modify the code in some way, and create a new executable that is linked to similarly modified libraries. We quickly determined that the license checking is not done by the Purify programs themselves, so we assumed that it was done by the new executable. This proved to be only partially true, as it did not seem possible to instrument the program before the license checking happened. For example, attempts to stop the program (even with gdb) at `main()` or `_start()` showed that the Purify window would be displayed *before* the break point. This means that the most likely place to find the license-checking code is in a dynamic library initialization routine.

Since the Purify license checking code is not present in the executable but instead in a dynamic library, we need to be able to insert the instrumentation before the library is initialized. The DynInst system automatically loads the target program's dynamic libraries during the process creation step, which begins the instrumentation session. Because all modifications to the program must be done after this step, there is no hope of ever controlling these programs with DynInst as it stands now.

To successfully bypass the license checking of the Purify programs, a different approach would need to be taken. One way would be to modify DynInst to prevent the automatic loading of a program's libraries. It may be possible to modify it to allow the partial loading of the libraries, just enough to allow DynInst to find and modify the functions in those libraries, but without calling the initialization routines.

Another way which might work would be to modify the libraries before loading the program. This cannot be done with DynInst, because DynInst is more suited to modifying running programs rather than object code on disk. Due to time constraints, we chose to focus on other licensed programs instead.

5 Tools

In the course of this study, we experimented with numerous methods and techniques, and developed numerous tools. Many of these tools are useful for general DynInst programming, as well as specifically for our study of license checking methods. Some of these tools are listed here. They all act as mutators using DynInst to modify/control the mutatee.

5.1 Callgraph Analyzer

We created static callgraph-related tools to help us perform some of the necessary tracing tasks. We have tools to find all parents and children of a given function, as well as all possible descendants and ancestors. Of course these do not reflect calls through function pointers, but they are useful nonetheless.

Some general utility routines have been developed to manage the problem of dealing with more than 16,000 core functions. Some of these tools allow us to add and remove functions from a list based on a regular expression, and to take the intersection and union of lists.

5.2 Java to DynInst Compiler

In order to specify snippets to be inserted, DynInst requires the construction of a hierarchy of objects similar to an Abstract Syntax Tree (AST) for the desired piece of code. This method is cumbersome and error-prone for any code sequence containing more than one instruction.

In order to simplify this programming task, we implemented a compiler supporting all arithmetic expressions, function calls and if statements which are necessary for snippet insertion. As the interface language for our compiler, we selected Java due to our previous experience with Java compilers and its similar syntax to C/C++. Our Java to DynInst compiler performs all necessary operations to:

- find the correct type and allocate variables
- create assignment and expression statements
- find the corresponding function
- allocate a vector of snippets for its parameters
- create constant expressions for parameter values

- create a procedure call using the function found

Figure 7 shows a simple Java input file for our compiler, and Figure 8 shows the corresponding DynInst code, generated by it. As you can see, this tool greatly simplifies code generation in DynInst.

5.3 GUI for License Bypasser

A major goal in this study was development of a tool which will simplify the process of tracking down and bypassing the license checking step in any software product. For this purpose, we developed a graphical user interface (GUI), which would enable the user to instrument the executable without any knowledge of DynInst.

In the implementation of the GUI, we could either use FLTK (The Fast Light Tool Kit, pronounced "fulltick") which is a C++ graphical user interface toolkit, or we could use Java Foundation Classes (JFC) Swing components. We implemented a simple callgraph GUI using both tools, and compared their performances. We observed that FLTK (C++) was more than 10 times faster compared to Swing (Java). In addition to the performance, when we considered the integration of the GUI with the utility functions that we have already developed in C++, we decided to use FLTK to implement the graphical user interface of our tool.

The License ByPasser GUI allows the user to walk through the callgraph of the target application, search for certain keywords which maybe the sign of possible license check points, list all functions calling a specific function, list all modules and get the list of all license-related functions in the application, load user-defined libraries, replace function calls and continue execution without writing a single line of code. Figure 9 is a screenshot of the GUI in action.

5.4 `dynit` - DynInst-based shell

All of our results up to now, as well as the various algorithms we developed, were integrated in `dynit` , which provides a Unix shell-like interface, similar in many aspects to `gdb`.

`dynit` is the top of the line tool for dynamic program analysis and control. Backed by DynInst, it provides access to various run-time information. It offers access to low-level DynInst functionality together with higher level analysis tools. Although developed for this project, we have attempted to keep it general, and it should be useful in many other cases.

`dynit` implements a simple, yet powerful, scripting language meant to drive a running application. It is similar to a debugger, but with a higher level list of features. The language specification is still under heavy development, but the initial version of the tool is surprisingly flexible and easy to use. We plan to extend it to allow for more powerful binary rewriting.

The GUI for the License ByPasser (5.3) will drive `dynit` , providing yet another easy to use interface and level of indirection. For a complete description of `dynit` , please see Appendix **??**.

```
class X
{
  public int open( String path, int flag, int mode ) { }

  public static void main( String argv[] )
  {
    int test;
    test = 1;
    if( test != 0 )
    {
      open( filename, O_WRONLY | O_CREAT, 0666 );
    }
  }
}
```

Figure 7: Simple Java input file for our compiler.

```
BPatch_function *openFunc = $IMAGE->findFunction("open");
BPatch_Vector<BPatch_snippet*> openArgs;

BPatch_variableExpr *test = $THREAD->malloc(*$IMAGE->findType("int"));

BPatch_arithExpr $ArithExpr(BPatch_assign, *test,
                            BPatch_constExpr(1)); //<-Arith. Expr.

BPatch_constExpr path(filename);
BPatch_constExpr flag(O_WRONLY|O_CREAT);
BPatch_constExpr mode(0666);

openArgs.push_back(&path);
openArgs.push_back(&flag);
openArgs.push_back(&mode);

BPatch_funcCallExpr openCall(*openFunc, openArgs); //<-function call

BPatch_Vector<BPatch_snippet*> StatementsInsideIF;
StatementsInsideIF.push_back(&/*pointer to the statement goes here*/);
BPatch_sequence IFSequence(StatementsInsideIF);
BPatch_boolExpr boolExpr(BPatch_eq, *test, BPatch_constExpr(0));
BPatch_ifExpr IFExpr(boolExpr,IFSequence); //<-If Epression
```

Figure 8: DynInst snippet creation code for the Java code in Figure 7.

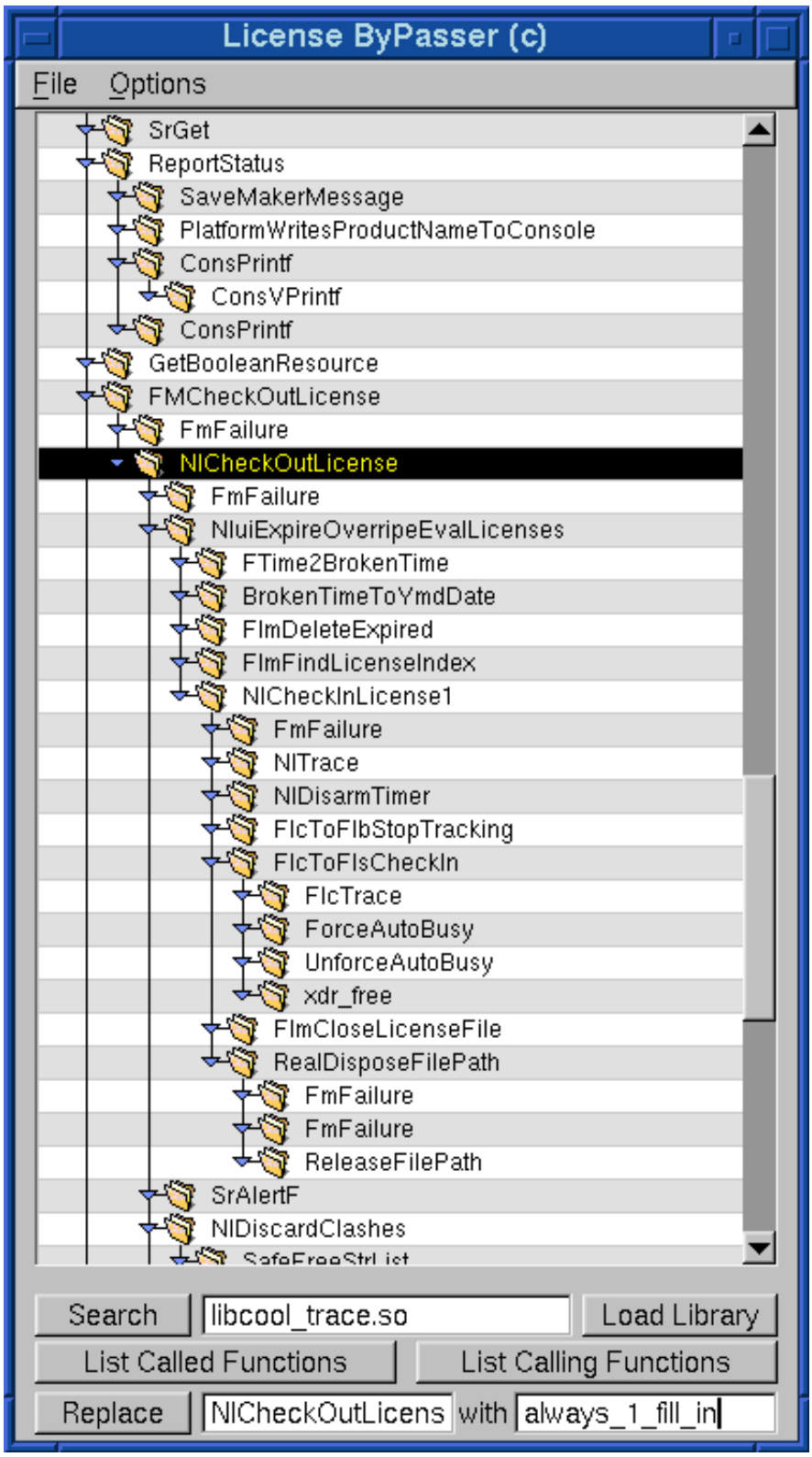

Figure 9: Screenshot of the License ByPasser GUI.

6 DynInst Limitations and Suggested Extensions

In the process of developing the tools necessary for our analysis of license-verification modules, we encountered various limitations and inconsistencies within DynInst. As DynInst is still under heavy development, this is understandable. We compiled a list of suggestions for the DynInst team.

Problems:

- Call points to small functions which cannot be instrumented by DynInst are set to NULL, preventing the driver program from constructing a complete callgraph.
- Too many return-value snippets can crash DynInst.
- Instrumentation at entry point to `main` is not always executed.

Suggestions:

- A finer-grain Control over the program is desirable (instrumentation can only be inserted at certain points in the current DynInst implementation).
- There should be a way of refreshing the callgraph information (the calls made by a function). Since the mutator can change function calls to go to other functions, DynInst information should be more dynamic.
- A `BPatch_image::findModule` function would be useful.
- The `BPatchSnippetHandle` should provide more information about the snippet it represents. This oversight forces programs using DynInst to store information which is obviously internal to DynInst. Extra information in `BPatchSnippetHandle` would simplify the programming task, and reduce the memory footprint of the driver program.
- Along the same lines, when the program is stopped, DynInst should maintain state information about where the stop is (at function entry or exit, for example).
- The behavior of the instrumented program is undefined when removing a snippet that contains the current point of execution (PC). Functionality to insert at the execution point would solve this problem.
- DynInst should allow for access to machine-level resources, such as registers, in the snippet generation code, as well as insertion of assembly code directly into the target program.

7 Conclusions

The goal of this sudy was to test the strength of the licence checking algorithms and question difficulty of bypassing them using dynamic instrumentation. The DynInst library enabled us to make arbitrary changes in the behaviour of Adobe Framemaker and bypass its licence cheks, allowing full use of the product without a licence. But the limitations in the current version of DynInst API prevented us getting similar results with the other tested software. With some enhacements to the DynInst library, it is not hard to get successful results also in other software. As a result, our study showes that by dynamically instrumenting binary code on the fly, the licence checks of most of the commercial software can be easily bypassed.

References

[1] B. R. Buck and J. K. Hollongsworth, *"An API for Runtime Code Patching"*, Journal of High Performance Computing Applications **14**, 4, Winter 2000, pp.317-329

[2] J. G. Ganssle, *"A Look Back"*, Embedded Systems Programming, **12**, 13, December 1999.

[3] J. K. Hollingsworth, "DynInstAPI Programmer's Guide - release 2.0", Computer Science Department, University of Maryland, Oct. 1999.

[4] J. K. Hollingsworth, B. P. Miller, M. J. R. Gonçalves, O. Naim, Z. Xu, L. Zheng *"MDL: A Language and Compiler for Dynamic Program Instrumentation"*, 1997 International Conference on Parallel Architectures and Compilation Techniques, November 1997, San Francisco, California

[5] B. P. Miller, M. Christodorescu, R. Iverson, T. Kosar, A. Mirgorodskii, F. Popovici, *"Playing Inside the Black Box: Using Dynamic Instrumentation to Create Security Holes"*, Parallel Processing Letters, **11**, 2 & 3, 2001, pp 267- 280.