In Situ Reuse of Logically Extracted Functional Components [*]

Craig Miles Arun Lakhotia
Center for Advanced Computer Studies
University of Louisiana at Lafayette
craig@craigmil.es arun@louisiana.edu

Andrew Walenstein
School of Informatics and Computer Science
University of Louisiana at Lafayette
walenste@ieee.org

Abstract

Programmers often identify functionality within a compiled program that they wish they could reuse in a manner other than that intended by the program's original authors. The traditional approach to reusing pre-existing functionality contained within a binary executable is that of physical extraction; that is, the recreation of the desired functionality in some executable module separate from the program in which it was originally found. Towards overcoming the inherent limitations of physical extraction, we propose in situ reuse of logically extracted functional components. Logical extraction consists of identifying and retaining information about the locations of the elements comprising the functional component within its original program, and in situ reuse is the process of driving the original program to execute the logically extracted functional component in whatever manner the new programmer sees fit.

1. Introduction

There exists many possible situations where a programmer may need to reuse capabilities embedded within a binary executable for which the source code is not available. Such a need arises when porting applications from older architectures and operating systems to newer environments. There are also times when a compiled component may contain certain capabilities that are desired to be used in another context, outside of the original system. Similar needs arise when performing security audits of third party applications to determine the existence of undesired behaviors or during forensic analysis of a potentially hostile program to exercise its capabilities under a different control environment.

When the need arises to implement a new system that will include one or more functionalities that have equivalent semantics to those in a previously existing binary executable for which the source code is not available, those functionalities are generally redeveloped ex novo. However, such ex novo redevelopment is unnecessary and inefficient because code that performs the desired functionality already exists within the original binary executable. In lieu of ex novo redevelopment, we propose a system for reusing pre-existing functionalities without separating them from the binary executables in which they were originally found.

The term *functional component* has been defined as a collection of programmatic constructs (instructions, data structures, etc.) that accomplish a particular function [4]. Extraction of such functional components is generally thought of in the "physical" sense; that is, to extract a functional component from a compiled program, the code and data that comprise that functional component are identified and separated from the original program into a standalone executable module. However, the physical model is not the only paradigm available for the extraction and subsequent reuse of functional components. Rather than physically extracting a functional component from its original disk image or process space, a functional component may be extracted "logically". *Logical extraction* is the process of making a functional component available for reuse *in situ*; that is, within its original context. Such a logical extraction is achieved not by separating the functional component from the program in which it resides, but rather by identifying and retaining the locations of all of the elements within the original program that comprise the functional component. We refer to that retained information as the "descriptor" of the logically extracted functional component (LEFC). Reuse consists of programming to the exported interface described by the descriptor. Given a descriptor containing the relevant information about a functional component within a compiled program, that program may be loaded into memory and driven in some manner so as to execute the desired functional component.

We propose a taxonomy of LEFCs: cold, hot, warm, and truly hot. The category to which a LEFC is classified indicates the manner in which it may be reused by a programmer or reverse engineer. In order to reuse a hot LEFC in situ, the program in which the functional component resides must first be put into a particular state prior to running the LEFC. Cold LEFCs, on the other hand, are those that may be run regardless of the program's state. Warm LEFCs are a special subset of hot LEFCs which may be converted into cold LEFCs, and truly hot LEFCs are hot LEFCs which are not warm.

To demonstrate the usage of LEFCs, we have designed and implemented a proof-of-concept software system capable of facilitating their in situ reuse. The system provides the ability to reuse both hot and cold LEFCs in situ. The software system, called LEFC Reuser, reads in descriptors of LEFCs and provides a programmatic interface whereby they may be executed. The user provides argument values to be passed to the LEFC when appropriate, and LEFC Reuser facilitates the return of computed information back to the caller after the LEFC has finished executing.

The goal we envision for logical extraction and in situ reuse of functional components is the ability to treat any compiled executable as a library of exportable functional components in the same manner that Windows DLLs are libraries of exported functions. A programmer should be able to quickly and easily identify and extract the interesting functional components from a compiled program such that they then may be called by his or her own program. The present work is the first step in that direction.

Through this work, we make both theoretic and empiric contributions relating to logically extracted functional components and their in situ reuse. The primary contributions of this work follow:

[*] The definitive version was published in Journal in Computer Virology, Volume 8, Number 3 (2012), 73-84, DOI: 10.1007/s11416-012-0167-y. The final publication is available at www.springerlink.com. URL: http://www.springerlink.com/content/j9015921584h07g7/

- We propose in situ reuse of logically extracted functional components.
- We formally define logical extraction of functional components.
- We formally define categories into which logically extracted functional components may be classified, and we discuss how the process for reusing logically extracted functional components differs depending on their categories.
- We further the state of the art by developing and describing the implementation of a software system, called LEFC Reuser, that allows for in situ reuse of logically extracted functional components.

2. Motivation

In this section, we describe a real scenario experienced by the authors in which we found ourselves wanting to use functionality of a program in a manner unavailable to us through the program's UI. We first detail the steps we followed to "physically extract" the functionality, and later we show how we may more easily and efficiently achieve our reuse objective by logically extracting the functionality and reusing it in situ.

The authors were recently asked to help reverse engineer a Windows PE. The executable was provided as a reverse engineering challenge at a Capture the Flag type contest. In order to proceed in the competition, contestants were required to recover and submit a password embedded within the program. The user interface of the application consists of a single form with a Get My Fortune! button and a text field. Upon clicking the button a random fortune cookie message is displayed in the text field, and a delay is set such that the button may not be clicked again for a few seconds. New fortune cookie messages continue to be displayed, even after clicking the button many times.

The organizers of the competition indicated that one of the randomly displayed fortune cookie messages would contain the password. With luck, a contestant could repeatedly click the button and possibly have the password displayed to him. However, with the appearance of so many unique fortune cookie messages, it seemed evident that such an approach would take much longer than would be desirable. A better approach is to obtain all of the fortune cookie messages at once.

Through much reverse engineering, it was determined that all of the possible fortune cookie messages are embedded in the PE in an encrypted form. Each time the button is pressed, one of the encrypted messages is decrypted by the Windows API function CryptDecrypt() and displayed to the user. Figure 1 shows the display of the Windows debugger OllyDbg[1] with the execution of the Fortune Cookie program stopped one instruction past the call to CryptDecrypt(). The arguments to the CryptDecrypt() system call have been added as comments. We see that the CryptDecrypt() function takes both encrypted data pointed to by the register EDI and a descriptive hKey structure which contains information about the encryption algorithm employed and the decryption key to be used. Finally, we see that the previously encrypted data pointed to by EDI has been decrypted in place, as the location now contains the decrypted fortune cookie message that will be displayed to the user.

In order to read all of the embedded fortune cookie messages at once, we first extracted the entire block of encrypted fortune messages from the binary executable to a file. With the block of encrypted strings in hand, we were left to re-implement the decryption algorithm. By consultation of the MSDN, we found that the construction of the hKey structure occurs by calling several Windows cryptographic functions sequentially. CryptAcquireContext() creates our cryptographic environment, and CrypteCreateHash() inserts a hash object into that environment. Once the object is created, a plain-text decryption key is hashed by CryptHashData() and is inserted into the hash object. Finally, the actual hKey decryption key structure is generated from the hash object by CryptDeriveKey(). With the knowledge of how to generate the proper hKey decryption key structure and how to call CryptDecrypt(), we re-implemented the decryption routine and made it decrypt each of encrypted fortune cookie messages we had extracted. The Fortune Cookie program and the physically extracted C++ source code of our decryption program is available online[2]. Amongst the decrypted messages was found the password for the challenge: "YoU g0t It!! This 1s d4 K3Y :p".

The tedious process of reconstructing the Fortune Cookie program's decryption code that we have just detailed is an example of physical extraction. Some functionality was located within a program that we wished to leverage in a way unintended by the original authors; thus the semantics of the functionality were understood and reimplemented externally. Though in this case the functionality was reimplemented in C++, it also could have been physically extracted by separating the actual X86 assembly code that performed the functionality we desired (the set up for and calls to the five Windows encryption functions) from the executable, and organizing it into a standalone executable module.

The question now becomes, why must we care how the decryption key structure is set up? Why must we create a new program with the ability to decrypt the encrypted messages when all of that functionality is already present within the Fortune Cookie program itself? These questions form the crux of our argument for logical extraction and in situ reuse of functional components. Such logical extraction allows for a functional component to be reused without extracting it from its original context. If we could logically extract the functional component that does just the actual decryption, then we could start up the Fortune Cookie program, let it construct its hKey decryption key structure as it normally would, and finally take control of its execution and cause it to run the decryption functional component on each of encrypted messages.

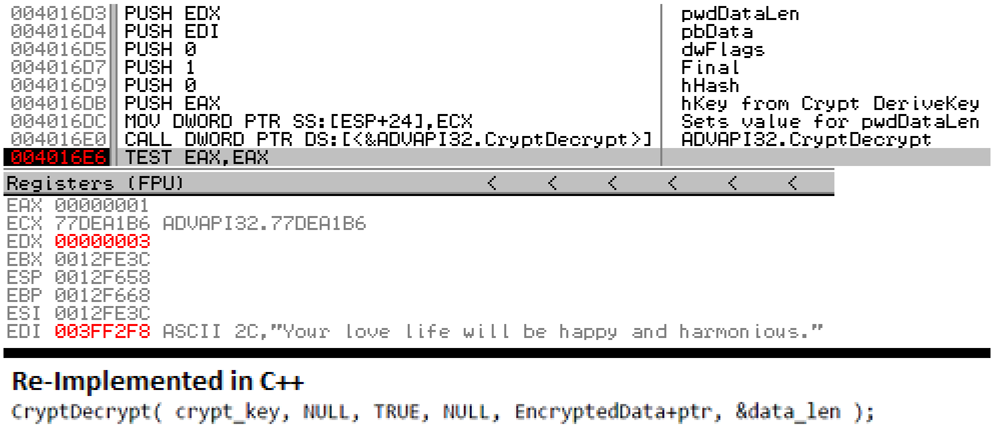
3. Logically Extracted Functional Components

A *functional component* is a collection of programmatic constructs (instructions, data structures, etc.) that accomplish a particular function [4]. Such a functional component may be logically extracted from a binary executable (the *target program*) by identifying and retaining information about it, thereby creating a logically extracted functional component (LEFC). The information that must be retained includes the address of the functional component's entry point within its target program, and the address(es) of the exit point(s). Also necessary are the parameters of the functional component, the locations where the functional component's return values are stored, and possibly the state that the target program must be in prior to running the LEFC.

From this definition, it is evident that a functional component is not the same thing as a C-like function. The first instruction of a functional component need not correspond to the entry point of a compiled C-like function. For example, the entry point of a functional component might be in the middle of some loop body. Furthermore, a functional component does not necessarily end on a *ret* instruction, but rather it ends on any instruction that has been specified by the definer of the functional component to be an exit instruction. In our view, a functional component is a collection of instructions, with one specified as the entry point, through which control flows until an instruction specified as an exit point is executed and/or an exit condition is met.

[1] http://www.ollydbg.de/

[2] `http://www.cacs.louisiana.edu/~csm9684/Fortune_Cookie_Program_and_Decrypter.zip`



```
004016D3  PUSH EDX                                        pwdDataLen
004016D4  PUSH EDI                                        pbData
004016D5  PUSH 0                                          dwFlags
004016D7  PUSH 1                                          Final
004016D9  PUSH 0                                          hHash
004016DB  PUSH EAX                                        hKey from Crypt DeriveKey
004016DC  MOV DWORD PTR SS:[ESP+24],ECX                   Sets value for pwdDataLen
004016E0  CALL DWORD PTR DS:[<&ADVAPI32.CryptDecrypt>]    ADVAPI32.CryptDecrypt
004016E6  TEST EAX,EAX
Registers (FPU)
EAX 00000001
ECX 77DEA1B6 ADVAPI32.77DEA1B6
EDX 00000003
EBX 0012FE3C
ESP 0012F658
EBP 0012F668
ESI 0012FE3C
EDI 003FF2F8 ASCII 2C,"Your love life will be happy and harmonious."
```

Re-Implemented in C++

```
CryptDecrypt( crypt_key, NULL, TRUE, NULL, EncryptedData+ptr, &data_len );
```

Figure 1. Display of OllyDbg showing the Fortune Cookie program paused after the call to CryptDecrypt(), and "physically extracted" C++ source code of the same call in our physically extracted decryption program.

In terms of how they may be reused, particularly differentiating amongst LEFCs is whether or not their target program must first be put into some state prior to being able to run the LEFC. Referring back to Section 2, the functional component that does the actual decryption of the encrypted fortune cookie messages (and nothing more) is dependent on the prior construction of the hKey decryption key structure. Prior initialization of the hKey structure constitutes the state of the target program on which the decryption functional component depends. A LEFC may be classified into two broad categories depending on whether or not its target program must first be put into some state before the LEFC may be executed.

A LEFC that does not depend on its target program first setting up some state is a *cold* LEFC. An example of a cold LEFC is a procedure that takes two integer arguments and returns their sum. Such a LEFC is cold because no state need be set up prior to running the functional component in order for it to perform the desired action; it must simply be provided the two integers to be summed.

A *hot* LEFC is a LEFC that depends on its target program being put into some state before the LEFC may be run; in other words, the LEFC may depend on data initialized by code of the target program that is not part of the LEFC. For example, consider again the decryption functional component of the Fortune Cookie program. The functional component has a data dependency on the prior initialization of the hKey decryption key structure. In order to make use of the decryption LEFC, the Fortune Cookie program in which it resides must first be driven to some state where it is known that the key structure initialization has already taken place.

We may further partition the universe of hot LEFCs into those that are *warm* and those that are *truly hot*.

A warm LEFC is one that depends on its target program being put into some state which must always be exactly the same in order for every execution of the LEFC to exhibit the desired behavior. Once more, consider the fortune cookie decryption LEFC. If it can be shown that the initialized hKey decryption key structure on which it depends is always initialized in the same way (every initialization of it results in an hKey data structure comprised of the exact same bytes as the previous initialization), then the decryption LEFC is a warm one.

Hot LEFCs that are not warm are said to be *truly hot*. A truly hot LEFC depends on its target program being put into a state that is not always the same. For example, consider a LEFC that sends an encrypted message to a recipient. Assume that the transmission of the encrypted message must be preceded by a cryptographic handshake between the sender and the recipient, thus the LEFC that sends the encrypted message depends on a state of the target process in which the cryptographic handshake has already taken place. A LEFC of this nature is truly hot because the result of the cryptographic handshake, which is the primary component of the state on which the LEFC depends, differs each time the handshake takes place.

Warm LEFCs are differentiated from truly hot LEFCs because warm LEFCs can be converted into cold LEFCs. To do so, the initialized data comprising the state on which the warm LEFC depends can be identified and retained in the descriptor of the LEFC. Prior to running the LEFC, that state can be artificially constructed within the target program's process space, rather than relying on the target program's code to set it up. In this manner, the warm LEFC would have been converted into a cold LEFC, as it no longer relies on the target program's code to construct the state on which it depends.

The question may arise as to why the code that sets up the state on which an LEFC depends would not also just be included within the LEFC. Because LEFCs are defined as starting from a singular entry point from which execution flows until an exit point or condition is reached, no ability to jump from one arbitrary chunk of code to another is present. As such, if execution does not flow from the code that sets up the state to the code that comprises the rest of the functional component, then there is no way to combine those two segments of code into a single LEFC. In order to meaningfully include two segments of code within a single LEFC, control must flow from the first to the second of its own accord. Referring once again to the Fortune Cookie program to provide an example, through our analyses we determined that the code which initializes the hKey decryption key structure is executed only once when the Fortune Cookie program is initially loading, whereas the code the performs the actual decryption of encrypted message executes each time the Get My Fortune! button is pressed. As control does not flow from the hKey initialization code to the message decryption code, the two sections of code cannot be extracted as a single LEFC. They could, of course, both be extracted into two individual LEFCs.

4. In Situ Reuse

In situ is a Latin phrase that, when literally translated, means "in place". In archeology, in situ refers to an artifact that has not been moved from its original place of deposition. We employ the phrase in a manner similar to how it is used by archaeologists. In particular, we use *in situ* to qualify the manner in which we programmatically reuse LEFCs; in our work, we execute (reuse) LEFCs without separating them from their original places. In contrast and as was described earlier, a functional component may instead be physically extracted from it's original program into some standalone executable module. We call execution of such a physically extracted functional component to be a case of *ex situ* reuse.

For some LEFC extracted from a target program, in situ reuse of the LEFC is accomplished by driving the execution of the target program via some mechanism such that the instructions specified by the LEFC are executed. The described driver must have the ability to modify the instruction pointer of the target program so that it can be set to the LEFC's entry point, and it must also be able to monitor the target program's subsequent execution such that it can be stopped upon reaching an exit point and/or meeting a specified exit condition. Additionally, the driver must be able to write to and read from memory within the target program's process space in order to pass argument values, artificially construct states on which the LEFC depends, and read values from return locations to be sent to the user once the LEFC has finished executing.

As will be discussed in much greater detail in Section 7, we have chosen to construct our driver on top of a debugger. A debugger has all of the previously discussed requisite abilities necessary to implement a LEFC in situ reuse driver. Specifically, modern debuggers for the the X86 platform allow for (1) direct modification of the EIP register (the X86 instruction pointer), (2) continued execution until a specific address is reached (via breakpoints), and (3) for read/write access to the debugee's memory. We have also speculated that such a driver could be constructed by injecting a new thread directly into the target program's process space, however that line of research has not yet been explored.

5. Formalization

A logically extracted functional component (LEFC) is a collection of information about a functional component. This collection of information is referred to as the LEFC's descriptor. A descriptor for an LEFC consists of the following (uncommon terms are formally defined shortly hereafter):

- Entry Point - The address of the first instruction of the LEFC within the target program.
- Exit Points and Conditions - A set of addresses within the target program through which the LEFC may exit and/or conditions which, when met, indicate that the LEFC should exit.
- State Elements - A set of 2-tuples of {location, value}, where the locations must be initialized to the corresponding values in order to artificially construct some or all of the state on which the LEFC depends. State Elements are used to decouple the LEFC from the target program.
- Parameters - A set of the locations into which values should be placed such that the behavior of the LEFC may be applied on those values.
- Return Locations - A set of locations whose contents should be returned to the caller after the LEFC has exited.

The LEFC metadata (name, description, etc.) and the specific low-level information that must be retained concerning the programmatic items just described (data types of parameters and return values, accessing locations by address versus stack pointer offset, etc.) is further detailed in Section 7.

In order to formalize the definitions of cold, warm, and hot LEFCs, we first appropriate some terms from data flow analysis and the lambda calculus. Though the lambda calculus terminology maps well to the current context when viewed at a slightly abstract level, we stipulate that our use of it is perforce an abuse of jargon. Let M be a LEFC consisting of instructions, at least some of which refer to one or more locations. A location is simply a place where bits are stored; that is, the registers, stack locations, heap addresses, and the locations to which any of those point (and so on, recursively). Let $REF(M)$ be the set of locations accessed in M, and let $LIVE(M)$ be the set of locations used in M before being defined. Specifically, a location x is in $LIVE(M)$ if there exists a path from the entry point of M over which x is used without first being assigned to. Let $PARAM(M)$ be the union of the parameters and the locations in the state element 2-tuples of M. From the lambda calculus terminology, it is illustrative to let $FREE(M) = LIVE(M)$ and $BOUND(M) = REF(M) \setminus LIVE(M)$. M is then said to be *closed* if $FREE(M) = \emptyset$, and M is a *closure* if $FREE(M) \subseteq PARAM(M)$.

A cold LEFC is a LEFC that is either closed or a closure. A hot LEFC is a LEFC that is neither closed nor a closure. A hot LEFC M is also a warm LEFC if $\forall\ l \in LIVE(M) \setminus PARAM(M)$, l must always be initialized to the same value in order for in situ reuse of the LEFC to result in the desired behavior. A truly hot LEFC is a hot LEFC that is not warm.

6. Example LEFCs

We now detail the logical extraction of a functional component from a contrived sample program. Consider the following C++ program:

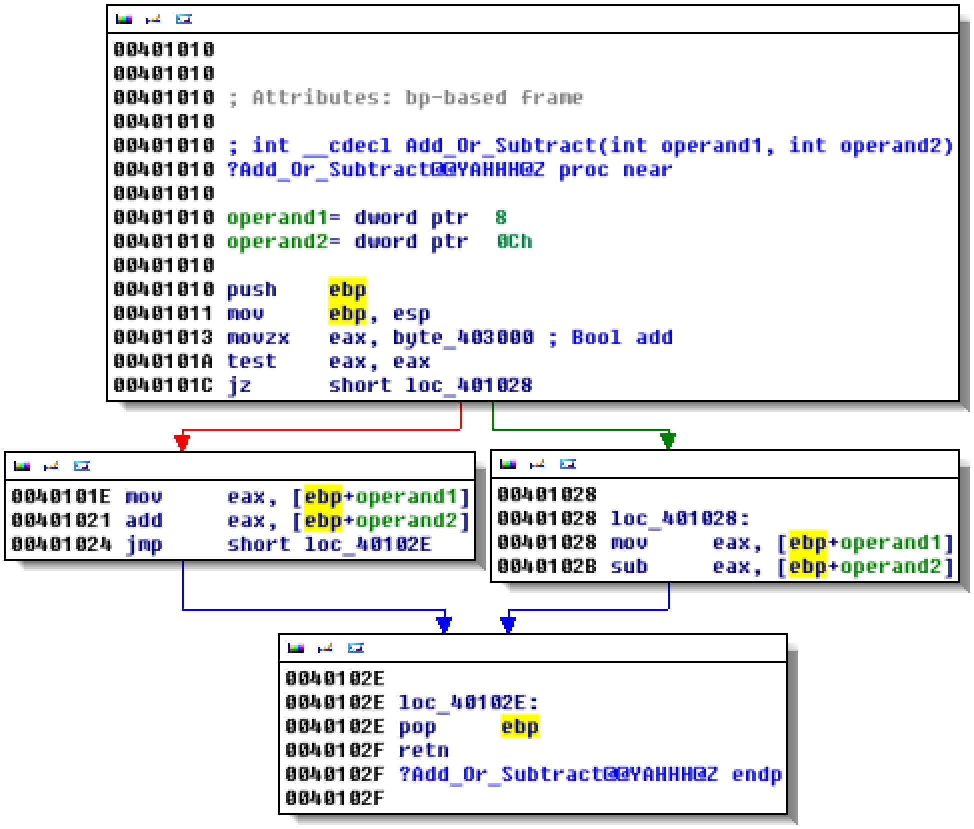
```
bool add;

int _tmain(int argc, _TCHAR* argv[])
{
  add = true;
  Add_Or_Subtract(10, 20);
  return 0;
}

int Add_Or_Subtract(int operand1, int operand2)
{
  if(add)
    return operand1 + operand2;
  else
    return operand1 - operand2;
}
```

An IDA Pro disassembly of the Add_Or_Subtract() function, compiled with no optimizations enabled and linked with debug information, is shown in Figure 2.

Suppose we wish to extract from this contrived program a functional component that computes and returns the sum of two integers. To do so, we may construct a LEFC with entry point 0x401010, exit point 0x401024, parameters at esp+8 and esp+C, and no state elements. Such an LEFC is hot because it depends on a state (the prior initialization of the Boolean global variable *add* to true), but that state information is not available in the LEFC's descriptor. In order to run this LEFC, the target program must first be driven to execute the statement "add = true;" in _tmain(). We also note that this LEFC is warm because the state on which it depends is always the same for every execution of the LEFC. Because it is warm, we can convert it into a cold LEFC by adding a Boolean state element 2-tuple of {0x403000, true} to the descriptor. The state on which the LEFC depends, which is now fully described in the LEFCs descriptor, can be artificially constructed by the LEFC Reuser driver rather than relying on the target program's code to construct it. The modified LEFC is now an example of a cold LEFC because the target program itself is no longer used to set up the state on which the LEFC depends.



```
00401010
00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 ; int __cdecl Add_Or_Subtract(int operand1, int operand2)
00401010 ?Add_Or_Subtract@@YAHHH@Z proc near
00401010
00401010 operand1= dword ptr  8
00401010 operand2= dword ptr  0Ch
00401010
00401010 push    ebp
00401011 mov     ebp, esp
00401013 movzx   eax, byte_403000 ; Bool add
0040101A test    eax, eax
0040101C jz      short loc_401028
```

```
0040101E mov     eax, [ebp+operand1]
00401021 add     eax, [ebp+operand2]
00401024 jmp     short loc_40102E
```

```
00401028
00401028 loc_401028:
00401028 mov     eax, [ebp+operand1]
0040102B sub     eax, [ebp+operand2]
```

```
0040102E
0040102E loc_40102E:
0040102E pop     ebp
0040102F retn
0040102F ?Add_Or_Subtract@@YAHHH@Z endp
0040102F
```

Figure 2. Disassembly of the Add_Or_Subtract() function.

7. Implementation

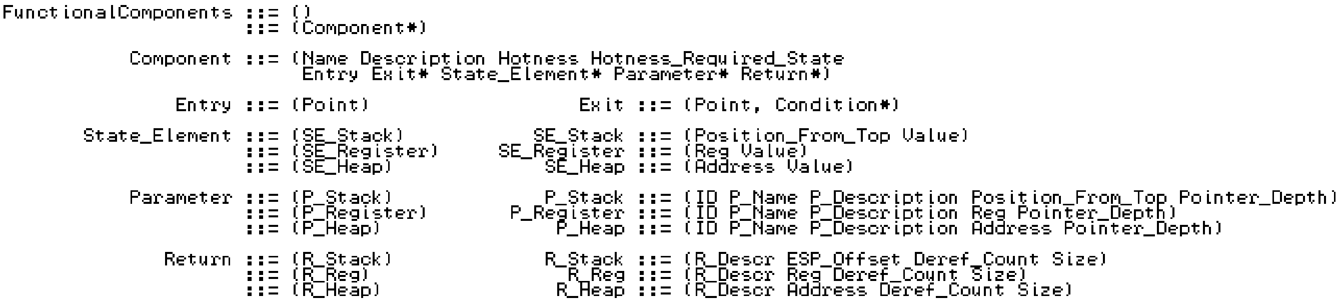
LEFC Reuser is a software system that provides for the in situ reuse of LEFCs. Three major architectural components comprise LEFC Reuser:

1. LEFC Descriptors - XML files which contain descriptors of LEFCs and conform to LEFC Reuser's schema.
2. LEFC Descriptor Compiler - A program that compiles the high-level XML descriptor of a LEFC into a set of low-level commands that instruct the LEFC Executing Debugger how to execute the LEFC.
3. LEFC Executing Debugger - We interface with the OllyDbg debugger via a modified version of the ODBGScript OllyDbg plugin in order to drive execution of the target program.

7.1 LEFC Descriptors

For each target program from which functional components have been logically extracted, an XML file conforming to LEFC Reuser's schema is generated and retained. Each such XML file contains one or more LEFC descriptors, where a descriptor contains all of the information required to reuse a LEFC *in situ*. A grammar for describing LEFCs, which is both derived from and maps directly to LEFC Reuser's schema, is shown in Figure 3. The terminals of the grammar are defined as follows:

- Name - User specified name for the LEFC.
- Description - User specified description of the LEFC.
- Hotness - Either "Hot" or "Cold".
- Hotness_Required_State - If Hotness is "Hot", a description of how to put the target program into the state on which the LEFC depends; otherwise, the field is unused.
- Point - An address of an instruction in the target program's process space.
- Condition - A predicate to be evaluated when the Point is reached. If all Conditions associated with the Point evaluate to true (or if there are no Conditions), then the LEFC exits.
- Position_From_Top - An integer specifying the position from the top of the stack in which to store a Value, where the top is position 0, the next location is position 1, and so on.
- Value - The State_Element's value.
- Reg - An X86 register ("eax", "ebx", "esp", etc.).
- Address - An address of a memory location within the target program's process space.
- ID - A unique integer identifier for each parameter. Used to identify the parameter being referred to when multiple sets of argument values to run the LEFC against are provided via a parameter input file.
- P_Name - User specified name for a Parameter.
- P_Description - User specified description for a Parameter.
- Pointer_Depth - An integer specifying the depth of the pointers pointing to the Parameter. For example, if the Parameter is of type int, then Pointer_Depth should be 0, however if the Parameter's type is **int, then Pointer_Depth should be 2.
- R_Descr - User specified description of the return value.
- ESP_Offset - An integer *offset* to be added to esp, such that esp + *offset* contains a value to be returned to the user (or a pointer to it).
- Deref_Count - An integer specifying the number of times the pointer stored at the location of the Return (either its esp + ESP_Offset, Reg, or Address) should be dereferenced to access the actual return value; should be 0 if it is not a pointer.
- Size - The size in bytes of the value to be retrieved from the specified location.



```
FunctionalComponents ::= ()
                     ::= (Component*)

           Component ::= (Name Description Hotness Hotness_Required_State
                          Entry Exit* State_Element* Parameter* Return*)

               Entry ::= (Point)                     Exit ::= (Point, Condition*)

       State_Element ::= (SE_Stack)             SE_Stack ::= (Position_From_Top Value)
                     ::= (SE_Register)       SE_Register ::= (Reg Value)
                     ::= (SE_Heap)               SE_Heap ::= (Address Value)

           Parameter ::= (P_Stack)               P_Stack ::= (ID P_Name P_Description Position_From_Top Pointer_Depth)
                     ::= (P_Register)         P_Register ::= (ID P_Name P_Description Reg Pointer_Depth)
                     ::= (P_Heap)                 P_Heap ::= (ID P_Name P_Description Address Pointer_Depth)

              Return ::= (R_Stack)               R_Stack ::= (R_Descr ESP_Offset Deref_Count Size)
                     ::= (R_Reg)                   R_Reg ::= (R_Descr Reg Deref_Count Size)
                     ::= (R_Heap)                 R_Heap ::= (R_Descr Address Deref_Count Size)
```

Figure 3. Grammar for LEFC descriptors derived from LEFC Reuser's schema.

7.2 LEFC Descriptor Compiler

The LEFC Descriptor Compiler, implemented in C# .NET, takes an LEFC Descriptor as input and compiles it into a set of low-level commands that instruct the LEFC Executing Debugger how to execute the LEFC. The LEFC Descriptor Compiler consists of: (1) A parser that reads in LEFC Descriptors and validates them with respect to LEFC Reuser's schema, (2) A mechanism to compile the high-level XML into low-level commands to be sent to the LEFC Executing Debugger, and (3) A communication mechanism that allows the compiler to send its commands to and receive return values from the LEFC Executing Debugger.

A user may provide the LEFC Descriptor Compiler its input LEFC Descriptors either manually, by way of the Compiler's GUI, or programmatically. Providing a LEFC Descriptor to the Compiler programmatically consists of dropping the descriptor XML file and a file containing the desired argument values (see the ArgumentSet example in Section 8) into a pre-specified directory. In this manner, a programmer may make reuse of LEFCs in situ directly from within his or her own program.

The parser is implemented using the XML parsing and validating functionality provided by .NET. Once parsed, the high-level XML descriptor is compiled into low-level debugger commands which instruct the LEFC Executing Debugger how to execute the LEFC. An example illustrating the compilation of a high-level XML descriptor into low-level debugger commands is given in the following section. Finally, the LEFC Descriptor Compiler opens the target program in the LEFC Executing Debugger and sends the compiled commands to it. When the LEFC Executing Debugger is done running the commands (when the LEFC has exited), the return value(s) are communicated back to the LEFC Descriptor Compiler which subsequently returns them to the caller. Communication between the LEFC Descriptor Compiler and the LEFC Executing Debugger is facilitated by dropping files containing debugger commands or return values at pre-determined locations.

7.3 LEFC Executing Debugger

The LEFC Executing Debugger is comprised of the OllyDbg Win32 debugger and a modified version of its ODBGScript plugin. OllyDbg[3] is a 32-bit assembler level analyzing debugger for Microsoft Windows. ODBGScript[4] is a plugin for OllyDbg that provides a scripting interface to the debugger. It is into ODBGScript's low-level scripting commands that the LEFC Descriptor Compiler compiles XML descriptors.

The ODBGScript plugin provides all of the requisite functionality necessary to reuse a LEFC in situ. Specifically, it provides direct access to the X86 instruction pointer EIP, the ability to drive the target program until a specified exit point is reached and to evaluate if exit conditions are met, and the ability to read from and write to memory addresses, stack locations, and registers. Furthermore, it provides the ability to allocate memory within the target program's process space, which we often make use of when we artificially create a state specified by the State_Elements of an LEFC descriptor.

In order to facilitate communication between the LEFC Descriptor Compiler and the LEFC Executing Debugger, the ODBGScript plugin was modified such that it may receive scripting commands programmatically; in the unmodified version, a user must open a script manually via OllyDbg's menu system. The modified ODBGScript plugin was originally written in C++, and the necessary modifications were made therein. The modified ODBGScript plugin listens for commands sent by the LEFC Descriptor Compiler and runs them upon receipt. After ODBGScript finishes running them, it communicates the requested return values back to the LEFC Descriptor Compiler by dropping a file containing them at a pre-determined location.

7.4 Summary of LEFC Reuser Work-flow

Figure 4 illustrates the work-flow of logical extraction and in situ reuse using our LEFC Reuser system. Descriptions of each of the numbered edges follow:

1. The Target Program is analyzed by the LEFC Identifier and Extractor which identifies functional components in the Target Program and logically extracts them into the collection of LEFC Descriptors. No such LEFC Identifier and Extractor software or algorithms have yet been developed. In the present work, all functional components have been identified and extracted manually. Creation of an LEFC Identifier and Extractor that requires only minimal human interaction forms the basis for our future work.
2. The User, either a human or a program, obtains a LEFC descriptor from the collection of LEFC Descriptors.
3. The User sends to the LEFC Descriptor Compiler (1) the selected LEFC descriptor and (2) a collection of arguments for the functional component to be run against.
4. The LEFC Descriptor Compiler (1) compiles the LEFC descriptor and the arguments into ODBGScript commands, (2) opens the Target Program in the LEFC Executing Debugger, and (3) sends the compiled commands to the LEFC Executing Debugger.
5. The LEFC Executing Debugger runs the compiled commands, thereby driving the Target Program to execute the functional component described by the selected LEFC descriptor against the provided arguments.
6. Once all of the compiled ODBGScript commands have been run, the values at the return locations specified by the LEFC descriptor are communicated back to the LEFC Descriptor Compiler.

[3] http://www.ollydbg.de/

[4] https://github.com/epsylon3/odbgscript/

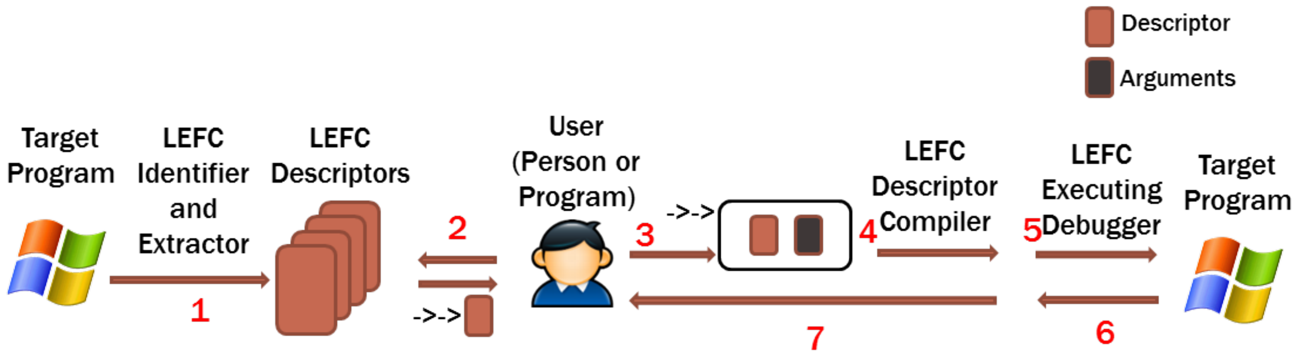


Figure 4. Work-flow of the LEFC Reuser system.

7. The LEFC Descriptor Compiler conveys the return values on to the User.

7.5 Evaluation of LEFC Reuser

The described LEFC Reuser software system has been successfully tested on a set of LEFCs that collectively make use of all of the various elements made available by LEFC Reuser's LEFC descriptor schema. A mixture of both hot and cold LEFCs have been tested successfully. Such tested LEFCs included those with multiple exits, those whose parameters, state elements, and returns make use of all three of the location types (heap, stack, and register), and those whose return values are obtained by dereferencing pointers. Some of the sample LEFCs also required that their arguments and state elements be stored in newly allocated memory within the target program's process space.

For a LEFC with descriptor D, LEFC Reuser's parsing and compilation algorithms run in $O(n)$ where n is the number of XML elements in D. Of course, the complexity of actually running (in situ reusing) a LEFC is determined by the instructions that comprise it.

8. Fortune Cookie Revisited

Having described the LEFC Reuser system, we now show how it may be applied to the purpose of decrypting all of the encrypted messages in the Fortune Cookie program. Recall from Section 2 that we do not want to waste time figuring out how to construct the necessary hKey decryption key structure ourselves. Because the functionality to properly construct the hKey is already present within the Fortune Cookie program, we wish to let it do so for us. Once the Fortune Cookie program has constructed the hKey, we want to hijack the Fortune Cookie program's control in order to make it iterate over and decrypt all of the encrypted messages within itself.

Figure 5 shows one possible LEFC Reuser descriptor for the decryption functional component of the Fortune Cookie program. The structure of the LEFC descriptor maps perfectly to the BNF grammar of Figure 3. Referring back to Figure 1, we see that the call to CryptDecrypt() is at 0x4016E0, which we specify as the decryption LEFC's entry point in Figure 5. 0x4016E6, the address of the instruction immediately after the call to CryptDecrypt(), is specified as the LEFCs exit point with no associated exit conditions. The State_Elements given in the descriptor partially describe (less the Parameters) the setup of the stack necessary for the call to CryptDecrypt() to correctly decrypt the encrypted fortune cookie messages. Regarding the State_Element with Position_From_Top=0, on our test system 0x15D660 is always the user-space handle into the kernel-space hKey structure constructed by the Fortune Cookie program and pushed onto the stack (via EAX) at line 0x4016DB of Figure 1. Of course, the location at which the hKey structure is constructed might be different on another system or if ours were rebooted, but the address is easily re-determinable by observing the call to CryptDecrypt() when the Get My Fortune! button is pressed. The given Parameters specify that two arguments, a pointer to an encrypted fortune cookie message and a pointer to the size of that encrypted message, complete the decryption LEFC's setup. Finally, the single Return specifies that the decrypted fortune cookie message will be pointed to by ESP-8 after the LEFC has exited.

Following is an abstracted snippet of the input file to be provided, along with the decryption LEFC's descriptor, to LEFC Reuser:

```
ArgumentSet:
  Arg_0: {ID: 0, IsAddressInTarget: true,
          Value: 0x542760, Size: 40}
  Arg_1: {ID: 1, IsAddressInTarget: false, Value: 40}
ArgumentSet:
  Arg_0: {ID: 0, IsAddressInTarget: true,
          Value: 0x5427A4, Size: 30}
  Arg_1: {ID: 1, IsAddressInTarget: false, Value: 30}
...
```

The input file contains all of the sets of arguments (addresses of the encrypted fortune cookie messages within the Fortune Cookie program and their respective sizes) for the decryption LEFC to iterate over and decrypt.

As specified by the decryption LEFC's descriptor, an ArgumentSet in this instance consists of two Arguments corresponding to the two given Parameters. The Argument with ID=0 (referred to as Arg_0) corresponds to the "Encrypted Data" Parameter in Figure 5 and Arg_1 corresponds to the "Size of Encrypted Data" Parameter. The IsAddressInTarget field of the input file is a Boolean which specifies whether or not the subsequent Value of the Argument is the actual value or the address of the value within the target program's process space. Finally, if IsAddressInTarget for an Argument is true, then Size specifies the number of bytes at that address which comprise the argument's value. We see that the first encrypted fortune cookie message, 40 bytes in size, is located at address 0x542760 in the Fortune Cookie program. One such ArgumentSet for each encrypted fortune cookie message is present in the actual input file.

Once LEFC Reuser is provided the decryption LEFC's descriptor and the corresponding input file, the LEFC Descriptor Compiler compiles the descriptor into ODBGScript commands and opens the Fortune Cookie program in the LEFC Executing Debugger. Notification is provided to the user that the present LEFC is hot, and that in order to construct the state necessary for the LEFC to run correctly the Fortune Cookie program must first be allowed to run un-

```
                 Name: "Decrypt"
          Description: "Decrypts an encrypted fortune cookie message."
              Hotness: "Hot"
Hotness_Required_State: "The state is constructed once the GUI is displayed."
                Entry: 0x4016E0
                 Exit: 0x4016E6
             SE_Stack: {Position_From_Top: 0, Value: 0x15D660}
             SE_Stack: {Position_From_Top: 1, Value: 0}
             SE_Stack: {Position_From_Top: 2, Value: 1}
             SE_Stack: {Position_From_Top: 3, Value: 0}
              P_Stack: {ID: 0, P_Name: "Encrypted Data",
                        P_Description: "Encrypted fortune cookie message.",
                        Position_From_Top: 4, Pointer_Depth: 1}
              P_Stack: {ID: 1, P_Name: "Size of Encrypted Data",
                        P_Description: "Size in bytes of Encrypted Data.",
                        Position_From_Top: 5, Pointer_Depth: 1}
              R_Stack: {R_Descr: "Decrypted message.", ESP_Offset: -8,
                        Deref_Count: 1, Size: 0x48}
```

Figure 5. Descriptor for Fortune Cookie program decryption LEFC.

til its GUI is displayed to the user (see the Hotness_Required_State field in Figure 5). Once the user notifies LEFC Reuser that the Fortune Cookie program GUI is displayed, the compiled commands are sent to the debugger where they cause the Fortune Cookie program to iterate over and decrypt all of the encrypted fortune cookie messages. Once all of the commands have been run, the debugger returns the location where the decrypted results may be found and LEFC Reuser displays that location to the user. The password for the Fortune Cookie Challenge, “YoU g0t It!! This 1s d4 K3Y :p”, is found in the decrypted results.

Using the LEFC Reuser system, we have shown how all of the encrypted fortune cookie messages within the Fortune Cookie program can be decrypted in a single shot with only a minimal understanding of how the actual decryption takes place.

9. Related Work

Reuse has long been considered to be a desirable alternative to developing new code. Efforts toward providing reuse of functionalities from compiled programs have included work in the field of COTS (commercial off-the-shelf) software integration. Such integration involves the inclusion of some other party’s commercial software within your own software system. COTS software integration has often been limited due to a lack of provided interfaces (APIs) to the COTS software functionality. Egyed and Balzer [5] make the case for reusing what we call functional components from COTS software, however they only target reuse for functional components which already have interfaces provided by the original authors. Their major contribution is the description of a wrapper for provided interfaces which extends those interfaces to allow for better synchronization between the caller and the callee. In general, the theme of this past work in COTS software integration has been to make usage of provided COTS software interfaces (APIs) less messy. Our work, on the other hand, can be characterized as creating and exporting interfaces for functionalities to which no interfaces previously existed.

Instrumentation is a technique for inserting extra code into an application to observe its behavior, and dynamic instrumentation is simply the application of the instrumentation technique on a running process. Dynamic instrumentation tools such as Pin[5] [8] and DynInst[6] [1, 9, 11] provide C++ API’s that allow a user to insert snippets of code into a target program to be executed when specified points are encountered during its execution. The chosen instrumentation framework places trampoline code at the points specified by the user. When one of these points is encountered during the target program’s normal flow of execution, the trampoline code stores the state of the target process and then transfers control to the snippet. When execution of the snippet has concluded, the trampoline code restores the process to its prior state and transfers control back to the next instruction of the target program. Profilers, cache simulators, trace analyzers, and memory bug checkers have all been implemented as dynamic instrumentation snippets. Dynamic instrumentation and in situ reuse are similar in that they both allow for execution of a target program in ways unintended by the target program’s original author, however they go about that goal in quite different ways. While dynamic instrumentation relies on the target program’s normal flow of control to initiate execution of newly inserted snippets of code, in situ reuse takes complete command of the target program’s control in order to drive it to execute code that was already present in new ways. However, we recognize that the aforementioned dynamic instrumentation tools already contain many primitive functionalities that, if recombined with LEFCs in mind, could provide much of the basis for a system capable of providing in situ reuse. Specifically, the Dyner [11] add-on to DynInst could quite possibly be extended to work in a fashion similar to our LEFC Reuser. Furthermore, the ability to construct conditional breakpoints within the target program’s process space using dynamic instrumentation [1, 11] would allow for a much more efficient implementation of LEFC exit point and exit condition checking than is currently available to us via our OllyDbg back-end.

Cifuentes and Fraboulet [3] and Kiss et al. [6] adapted Weiser’s high-level language slicing techniques [10] (and those technique’s descendants) to the purposes of inter- and intra-procedural slicing on compiled binaries. However, such slices are taken with respect to either an individual register or a set of registers at a given instruction, and therefore they are not analogous to the functional components with which we are concerned. While these slicing techniques may aide in the identification of state elements, they cannot directly be used for the purpose of logically extracting executable functional components. Slicing on binaries also presents other severe problems, including the need for accurate procedure and call information.

Kolbitsch et al. [7] extract externally observable behaviors, rather than functional components, from compiled executables. After selecting some externally observed behavior reported by a run-time monitor, their technique uses dynamic slicing to extract a ‘gadget’ that comprises all of the code and a memory snapshot

[5] http://www.pintool.org/

[6] http://www.dyninst.org/

needed to recreate the selected behavior. The gadget can then be replayed by a gadget player, thereby recreating the exact behavior of the originally observed program. Because these techniques are primarily concerned with behavior replay rather than component reuse, conditional branches in the extracted code are modified in order to force the flow of execution to always follow the originally observed path. As such, the extracted behavior is fixed to correspond to a single set of inputs, and cannot generally be used as a reusable component. For example, sometimes a behavior is only triggered under specific conditions, such as an update mechanism that only runs on a specific day of the week. Because their technique is able to identify and subsequently extract only behaviors whose executions are observable by their dynamic analysis component, if they are not observing the program on the day that it conducts its update check, then they cannot identify and subsequently extract a gadget that replays that behavior.

Caballero et al. [2] proposed techniques to identify and physically extract callable functions using interface abstraction methods. Primary contributions of their work include the development of a mechanism to identify the prototype of a binary code fragment and a technique for extracting its code and data dependencies. Extracted functional components consist of a C function that contains the extracted code as inline assembly and a header file that contains the required data. Limitations of Caballero et al.'s approach include: (1) components with parameters that are recursive structures, such as trees, cannot be extracted, (2) in order to extract a function, it must first be observed executing natively at least once (if some specific input is needed for control to reach the function, they assume it is provided), (3) functions with variable-length parameter lists such as printf cannot be extracted, and (4) a component to be extracted may not contain any code that explicitly makes use of knowledge of its own location.

Logical extraction, in lieu of physical extraction, is able to overcome each of the aforementioned limitations under certain conditions. While it may be difficult to identify the exact structure of a complex argument, if the structure is known a priori and because we leverage functional components within their original context, then there is no reason that such a component could not be made externally reusable. Furthermore, while the ability to observe a functional component execute in its native context is certainly advantageous when attempting to identify the elements comprising its would-be descriptor, it is not absolutely necessary; the requisite information could alternatively be ascertained via static analysis. Even if a functional component's parameter list is identified as being of variable length, because it will be executing within its native context there is no reason why a system using logical extraction could not interface with it if the structure of that parameter list were known. Finally, since our LEFCs are reused in situ, there are never limitations on their ability to make use of the knowledge of their own locations within the target program.

10. Directions for Future Work

Throughout this work, the process of identifying functional components that may be logical extracted and reused in situ has taken place manually. This included manually searching for interesting segments of code within compiled programs that could be logically extracted as useful and reusable functional components, as well as manually determining those functional components' state elements, parameters, and returns via both static and dynamic analysis. Towards the goal of making the present work more useful, many directions for future work exist. Primarily, automation of the process for identifying and learning about functional components is a fertile and interesting research area. Such work might entail usage of novel combinations of machine learning and classification techniques to automatically identify code that may comprise interesting or useful functional components, program slicing and data flow analysis techniques to automatically determine a functional component's state elements and parameters, and variable identification and type inferencing techniques to more accurately determine the prototypes of functional components. Another possible interesting direction for future work would be to prove that the set of elements included in LEFC Reuser's LEFC descriptor schema comprises the sufficient set of such elements required to adequately describe all possible functional components.

11. Conclusion

Prior approaches for extracting functional components from compiled programs have been physical in nature; that is, they generally relied on physically separating the code and data comprising the functional component from the target program in which it originally resided. We have shown that not only does another paradigm exist, that of logical extraction, but that extracting functional components in this new manner overcomes some of the limitations of past approaches.

We have described the process of logical extraction and the necessary information that must be stored in order to logically extract most (and possibly all) functional components, and we have both described how such a LEFC may be reused in situ and have developed a proof-of-concept implementation of this process. Formal definitions have been presented for the different types of LEFCs, cold, hot, warm, and truly hot, and the process for converting a warm LEFC into a cold LEFC has been described. Results from using our implementation, LEFC Reuser, have been positive; it has been shown to be capable of in situ reuse of many logically extracted functional components of varying complexity.

With the present work, we have taken a step towards meeting the final goal of being able to treat any compiled executable as a library of exportable functional components that are reusable by a programmer within his own programs.

12. Acknowledgments

The authors gratefully acknowledge the contributions of Chris Parich. This research was sponsored in part by the Air Force Research Laboratory and DARPA (FA8750-10-C-0171) and the Air Force Office of Scientific Research (FA9550-09-1-0715).

References

[1] B. Buck and J. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[2] J. Caballero. Binary code extraction and interface identification for security applications. Technical report, DTIC Document, 2009.

[3] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *Software Maintenance, 1997. Proceedings., International Conference on*, pages 188–195. IEEE, 1997.

[4] Defense Advanced Research Projects Agency. Research Announcement - Binary Executable Transforms (BET) - DARPA-RA-11-56. Technical report, Defense Advanced Research Projects Agency, 2011.

[5] A. Egyed and R. Balzer. Unfriendly COTS integration-instrumentation and interfaces for improved plugability. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 223–231. IEEE, 2001.

[6] A. Kiss, J. Jász, G. Lehotai, and T. Gyimóthy. Interprocedural static slicing of binary executables. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*, pages 118–127. IEEE, 2003.

[7] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector Gadget: Automated extraction of proprietary gadgets from malware binaries.

In *2010 IEEE Symposium on Security and Privacy*, pages 29–44. IEEE, 2010.

[8] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[9] G. Ravipati, A. Bernat, N. Rosenblum, B. Miller, and J. Hollingsworth. Toward the deconstruction of Dyninst. Technical report, Computer Sciences Department, University of Wisconsin, Madison (ftp://ftp.cs.wisc.edu/paradyn/papers/Ravipati07SymtabAPI.pdf), 2007.

[10] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[11] C. Williams and J. Hollingsworth. Interactive binary instrumentation. *Intl. Works. on Remote Anal. and Measurement of Softw. Sys*, pages 312–327, 2004.