Carnegie Mellon University
Research Showcase

Software Engineering Institute

2-1-2011

Function Extraction (FX) Research for Computation of Software Behavior: 2010 Development and Application of Semantic Reduction Theorems for Behavior Analysis

Richard C. Linger
Carnegie Mellon University, rlinger@sei.cmu.edu

Tim Daly

Mark G. Pleszkoch
Carnegie Mellon University, mpleszko@sei.cmu.edu

Follow this and additional works at: http://repository.cmu.edu/sei

Recommended Citation
Linger, Richard C.; Daly, Tim; and Pleszkoch, Mark G., "Function Extraction (FX) Research for Computation of Software Behavior: 2010 Development and Application of Semantic Reduction Theorems for Behavior Analysis" (2011). *Software Engineering Institute.* Paper 664.
http://repository.cmu.edu/sei/664

This Technical Report is brought to you for free and open access by Research Showcase. It has been accepted for inclusion in Software Engineering Institute by an authorized administrator of Research Showcase. For more information, please contact research-showcase@andrew.cmu.edu.

Software Engineering Institute

Function Extraction (FX) Research for Computation of Software Behavior: 2010 Development and Application of Semantic Reduction Theorems for Behavior Analysis

Research Report for the Air Force Office of Scientific Research Mathematics and Information Science Directorate

Richard Linger
Tim Daly
Mark Pleszkoch

February 2011

TECHNICAL REPORT
CMU/SEI-2011-TR-009
ESC-TR-2011-009

CERT® Program
Unlimited distribution subject to the copyright.

http://www.sei.cmu.edu

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2011 Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.

External use. This document may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about SEI publications, please visit the library on the SEI website (www.sei.cmu.edu/library).

Table of Contents

Executive Summary **v**

1 The Behavior Computation Process **1**
1.1 The Architecture of Behavior Computation 1
1.2 An Illustration of Behavior Computation for Reverse Engineering 3

2 Research Results for Determining True Control Flow of Malware Programs **7**
2.1 H-Chart Algorithm 7
2.2 Using Behavior Computation for Semantic Analysis of Control Flow 9
2.3 Semantic Reduction Theorems in True Control Determination 9
2.4 Meta-Programming Language (MPL) 11

3 Behavior Computation Technology Transition **12**

4 Next Steps in the Research Program **13**

Appendix: Mathematical Foundations of Software Behavior Computation **15**

References **25**

List of Figures

Figure 1: Behavior Computation System Architecture 2

Figure 2: Behavior Computation for As-Coded Matrix Multiplication 4

Figure 3: Behavior Computation for Matrix Multiplication Containing an Error 5

Figure 4: Behavior Computation for Matrix Multiplication with Obfuscation and Embedded Malware 6

Executive Summary

For several years, the Software Engineering Institute (SEI) at Carnegie Mellon University has been engaged in a project to compute the behavior of software with mathematical precision to the maximum extent possible. Air Force Office of Scientific Research (AFOSR) sponsorship has played a key role in this effort. The general thrust of the research for AFOSR has been in technology for (1) overcoming difficult aspects of behavior computation and (2) analyzing and manipulating computed behavior. In 2009, the research focused on computing the behavior of loops, a process subject to theoretical limitations. This resulted in practical methods for loop computation that minimize the effects of these constraints. The 2010 research focused on foundations and implementations of algorithms that employ computed behavior and semantic reduction theorems to determine the true control flow of malware programs as an essential first step in computing overall malware behavior. Determining the true control flow of a program in the presence of computed jumps and jump table operations has been a difficult problem for some time. Syntactic methods of control flow analysis exhibit limitations that reduce their effectiveness. The semantic methods employed by behavior computation can produce improved results. The findings of this research have been implemented in a system for malware analysis and have improved capabilities for behavior computation in other applications. At the same time, the research has revealed a potential new approach to both reverse engineer and forward engineer software based on rigorous specification and verification in the context of behavior computation.

1 The Behavior Computation Process

The Software Engineering Institute (SEI) at Carnegie Mellon University has been engaged for several years in a project to automate the computation of software behavior with mathematical precision to the maximum extent possible. Since its inception, the research, sponsored by the Air Force Office of Scientific Research (AFOSR), has focused on solutions to difficult problems in behavior computation and on analysis of computed behavior for human understanding.

In 2009, the emphasis was on developing theory and implementation for loop behavior computation, a difficult problem subject to theoretical constraints, as expressed in the Halting Problem. This effort was successful and resulted in methods that help to limit the effects of these constraints. The technology of Semantic Reduction Theorems (SRTs) developed in the AFOSR project played a major role in this work. SRTs are predefined microtheorems that can be used to analyze computed behavior for a variety of purposes, including

- loop behavior computation
- reduction of intermediate computed behavior to simpler form
- abstraction of computed behavior to specification-level definitions in reverse engineering
- specification and verification of intended behavior in new software development

SRTs are generally applicable and can be defined. For example, a repository of SRTs for finite arithmetic will never change unless the processor architecture is changed, an unlikely possibility. SRTs are used within a behavior computation system for a variety of purposes in supporting the computation itself. From the user perspective, SRTs also play a role in reverse engineering, the understanding of existing behavior in legacy or acquired software, and in specifying and verifying intended behavior in new software development. SRTs have been a major focus of the AFOSR research program.

In 2010, the research focused on SRT application to determine the true control flow of input programs as a necessary initial step in behavior computation. This has been a difficult problem for some time. Syntactic methods of control flow analysis exhibit limitations that reduce their effectiveness. The semantic methods embodied in behavior computation permit more extensive analysis of control flow than otherwise possible.

1.1 The Architecture of Behavior Computation

The architecture of a behavior computation system provides a useful context for describing this research and how true control flow determination supports the computational process. Control flow is often obfuscated by intruders to mask malicious operations. Behavior computation, or any other software analysis technology, requires knowledge of true control flow to be effective. Referring to the architecture diagram of Figure 1, the instructions of an input Intel assembly language program in binary form are first transformed into functional form using a predefined repository of instruction semantics. The repository definitions of instruction semantics account for all effects

each instruction can have on the state of the hardware, including all register, memory, and flag settings, plus effects of the finite nature of machine precision.

Next, because the input program may contain complex control logic, including computed jumps and jump table operations, it is necessary to determine the true control flow produced by this logic as a preliminary step. As noted, this is a difficult problem that was a major focus of the 2010 AFOSR effort. The solution employs behavior computation itself as a local, internal process, as well as the application of SRTs to help determine true control flow.

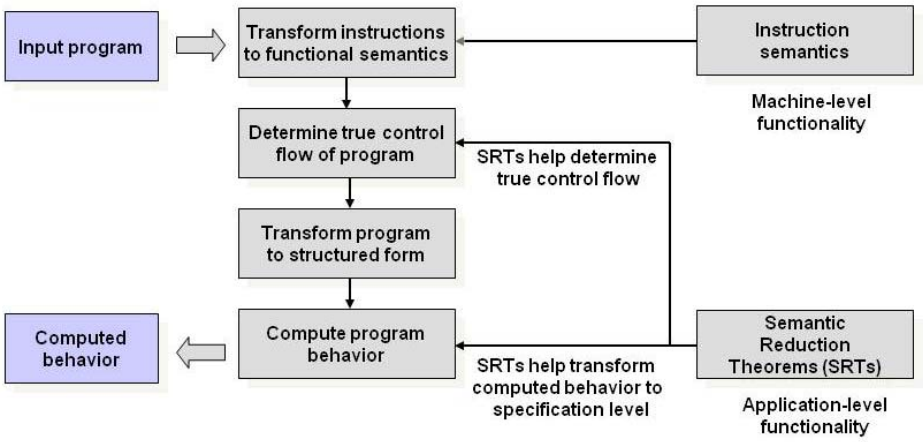

Figure 1: Behavior Computation System Architecture

Once the true control flow is known, the input program, possibly containing complex "spaghetti" logic representing that control flow, is transformed into structured form through the application of the Structure Theorem. The constructive proof of this theorem defines a process for transforming arbitrary program logic into structured form, expressed in the basic control structures of `sequence`, `ifthenelse`, and `whiledo` [Linger 1979]. These control structures are nested and sequenced in an algebraic hierarchy with all arbitrary jumps eliminated.

Each control structure is a single-entry, single-exit structure whose functional effect can be expressed as a mathematical function, that is, as a mapping from domain to range, or input to output. This structured form of the input program is the basis for behavior computation. The Correctness Theorem is applied to compute program behavior by composing the functional effects of instructions to produce the output computed behavior database [Linger 1979]. The theorem defines a mapping of procedural logic expressed in `sequence`, `ifthenelse`, and `whiledo` structures into procedure-free functional form, essentially, the as-built specification of each structure. The mapping for looping (`whiledo`) structures is expressed in a recursive equation of limited value for human understanding. The 2009 AFOSR-sponsored research resulted in methods to produce a more useful functional form for expressing loop behavior. The behavior computation process is compositional in nature, combining the net functional effects of instructions embedded in leaf node control structures in the algebraic hierarchy of the structured code, and then propagating their functional effects to the next level of control structures. This process continues until the behavior of an entire program has been computed. A valuable side effect of this process is the avail-

ability of the intermediate behavior of every control structure in the hierarchy for analysis and understanding.

The SRT technology developed through AFOSR research plays a key role in behavior computation. At each step, initial computed behavior is reduced to simpler form through the application of SRTs. For example, as noted above, SRTs dealing with finite arithmetic operations can be applied to simplify arithmetic behavior operations prior to propagating them to the next level. Libraries of SRTs dealing with common data processing operations can be defined using the internal meta-programming language (MPL) of the behavior computation system.

1.2 An Illustration of Behavior Computation for Reverse Engineering

The following example, produced by the SEI as part of an internal study, illustrates the application of SRT technology as it has continued to evolve under AFOSR sponsorship, leading to new perspectives on its use in both reverse engineering and new development. In this example, SRTs are applied to the reverse engineering of existing software. Consider the computations required for spatial maneuvering by a robot control arm. These computations depend on the correct operation of matrix multiplications (cross products of the form A x B → C) that compute rotation and translation movements. Code that performs matrix multiplications was selected for reverse engineering and analysis of behavior, and four operators for use in SRTs were defined to specify the mathematical process:

- Vector memory shape. Matrix multiplication operates on vectors of elements. This operator has parameters to define the vector start location, number of elements, and stride, that is, the size of steps to use when reading a vector from memory, in number of bytes.
- Dot product. Matrix multiplication can be expressed using the dot products of vectors. The dot product operator has parameters representing the two vectors for which to compute the dot product.
- Matrix memory shape. Matrix multiplication requires that the vectors form matrices. A matrix is represented as an operator whose parameters represent the starting address of the matrix in memory and the number of rows and columns in the matrix.
- Matrix multiplication. Matrix multiplication must produce a matrix of dot products. The matrix multiplication operator has parameters representing the two matrices to multiply together.

SRTs were developed using these operators to first recognize dot products of vectors and then recognize matrix multiplications. Three versions of the matrix multiplication code were then analyzed.

Behavior of Code in Original Form

For this version, no changes were made to the as-coded assembly language. The results of the behavior computation are depicted in Figure 2. This is a procedure-free conditional concurrent assignment produced by the Function extraction (FX) system, accounting for effects of the code on processor registers, memory, and flags. It is important to note that all the assignments in the figures are concurrent and represent “vector assignments” of right-hand-side expressions on initial

state into left-hand-side variables in the final state. This represents the net functional effect of the code.

```
CONDITION

    TRUE →

REGISTERS

EAX := DOT_PRODUCT of
        VECTOR(4) at (12 + (dword at (8 + ESP))) by stride 16
        and
         VECTOR(4) at (48 + (dword at (4 + ESP))) by stride 4

ECX := DOT_PRODUCT of
        VECTOR(3) at (12 + (dword at (8 + ESP))) by stride 16
        and
        VECTOR(3) at (48 + (dword at (4 + ESP))) by stride 4

EDX := dword at (60 + (dword at (4 + ESP)))

EIP  := dword at ESP

ESP := 4 + ESP

MEMORY

M :=
MATRIX of dwords at (dword at (12 + ESP) :=
    MATRIX_MULTIPLICATION  of
        MATRIX(4,4)  of dwords at (dword at (4 + ESP))
        and
        MATRIX(4,4)  of dwords at (dword at (8 + ESP)))

label := "exit"

FLAGS

AF := false
CF := arb_bool_val
OF := arb_bool_val
PF := is_even_parity_lowbyte(48  + (dword at (8 + ESP)))
SF := is_neg_signed_32(48  + (dword at (8 + ESP)))
ZF := (4294967248  == (dword at (8 + ESP)))
```

Figure 2: Behavior Computation for As-Coded Matrix Multiplication

The condition on the behavior is "true" because the code always executes. Memory `M` contains a `MATRIX` of double words (`dword`) starting at location `12 +` the stack pointer (`ESP`), specifically, a `MATRIX_MULTIPLICATION` of a `4x4 MATRIX` starting at the location defined by (`4 + ESP`) and a `4x4 MATRIX` starting at the location defined by (`8 + ESP`). Thus, the computed behavior for the original code satisfies the SRTs that define a matrix and a matrix multiplication, both of which depend on SRTs for vector and dot product definition. Given that these SRTs are themselves correct (SRTs can be verified by theorem provers), the code correctly implements a matrix multiplication. The computation also reveals that the registers contain residual values involving dot products and vectors, and the flags contain residual values, as well.

Behavior of Code with Error Inserted

Figure 3 depicts computed behavior for the matrix multiplication code with an error inserted; specifically, values in the target matrix (C matrix) are no longer initialized to zero as in the correct version. In this case, the computed behavior reveals that the code no longer computes a matrix multiplication. Memory `M` now contains a series of double words, each the sum of itself and a dot product of vectors. The code still satisfies SRTs for dot products and vectors, but not the SRT for matrix multiplication. The fact that each of these locations is a sum of itself and a computation that should ultimately contribute to a matrix multiplication reveals the problem. The sum is present because the location is not initialized to zero, and any value present on entry will contribute to the final value on exit. Initializing the target matrix will solve the problem.

```
CONDITION

    TRUE →

(Flags not shown)

REGISTERS

EAX := (dword at (60 + (dword at (12 + ESP))) +
DOT_PRODUCT of
        VECTOR(4) at (12 + (dword at (8 + ESP))) by stride 16
        and
        VECTOR(4) at (48 + (dword at (4 + ESP))) by stride 4

ECX := (dword at (60 + (dword at (12 + ESP))) +
DOT_PRODUCT of
        VECTOR(3) at (12 + (dword at (8 + ESP))) by stride 16
        and
        VECTOR(3) at (48 + (dword at (4 + ESP))) by stride 4

EDX := dword at (60 + (dword at (4 + ESP)))

EIP := dword at ESP

ESP := 4 + ESP
```

```
MEMORY

M :=
dword at (dword at (12 + ESP)) :=
dword at (dword at (12 + ESP)) +
DOT_PRODUCT of
   VECTOR(4) at (dword at (4 + ESP)) by stride 4
   and
   VECTOR(4) at (dword at (8 + ESP)) by stride 16

dword at (4 + (dword at (12 + ESP)) :=
dword at (4 + (dword at (12 + ESP))) +
DOT_PRODUCT of
   VECTOR(4) at (dword at (4 + ESP)) by stride 4
   and
   VECTOR(4) at (4 + (dword at (8 + ESP))) by stride 16

dword at (8 + (dword at (12 + ESP)) :=
dword at (8 + (dword at (12 + ESP))) +
DOT_PRODUCT of
   VECTOR(4) at (dword at (4 + ESP)) by stride 4
   and
   VECTOR(4) at (8 + (dword at (8 + ESP))) by stride 16
...
```

Figure 3: Behavior Computation for Matrix Multiplication Containing an Error

Behavior of Code with Malware Inserted

Figure 4 shows computed behavior for the matrix multiplication obfuscated with a large number of arbitrary jumps, interleaved instructions, and malware inserted with instructions dispersed throughout the code. In this case, the condition is no longer simply "true." A condition has been revealed involving SRTs named `create file succeeded` and `write file succeeded`. Since this code should not be creating and writing files, something is obviously wrong. The behavior is now affecting the file system and the operating system because the code is writing a file beginning at buffer byte 0 and ending at byte 6738. This is the exact length of the code. The inserted malware is embedded within the code and is writing that code into the file system of the machine. It is a self-replicating virus. Note that the matrix multiplication is itself unaffected by the malware, but that hardly matters at this point. In addition, this behavior computation produced three other cases of behavior (not shown), all of which represented errors in the malicious code that prevented the self-replication.

```
CONDITION 1

create_file_succeeded(
   file_name_addr = 158,
   file_attribute = (word at (40 + (dword at (4 + ESP)))))
and
write_file_succeeded(
   file_handle = get_new_file_handle(
         file_name_addr = 158,
         file_attribute = (word at (40 + (dword at (4 + ESP))))),
   buffer_to_write = 0,
   num_bytes_to_write=6738)
```

```
FILE SYSTEM

FILES :=

create_file_and_truncate(
   file_name_addr = 158,
   file_attribute = (word at (40 + (dword at (4 + ESP)))))
then
write_file(
   file_handle = get_new_file_handle(
      file_name_add r= 158,
      file_attribute = (word at (40 + (dword at (4 + ESP))))),
   buffer_to_write = 0,
   num_bytes_to_write = 6738)
```

```
MEMORY

M :=

MATRIX of dwords at 100 :=
   MATRIX_MULTIPLICATION of
      (MATRIX(3,4) of dwords at (dword at (4 + ESP)))
      and
      (MATRIX(4,4) of dwords at (dword at (8 + ESP)))

MATRIX of dwords at (dword at (12 + ESP)) :=
   MATRIX_MULTIPLICATION of
      (MATRIX(4,4) of dwords at (dword at (4 + ESP)))
      and
      (MATRIX(4,4) of dwords at (dword at (8 + ESP)))
```

```
OPERATING SYSTEM

DOS :=

mark_file_handle_as_writing(
   file_handle = get_new_file_handle(
      file_name_addr=158,
      file_attribute = (word at (40 + (dword at (4 + ESP))))),
   buffer_to_write=0,
   num_bytes_to_write=6738)
```

(Flags not shown)

Figure 4: Behavior Computation for Matrix Multiplication with Obfuscation and Embedded Malware

This example illustrates the use of SRTs in reverse engineering to validate (or not validate) the behavior of existing code. Imagine, however, a development scenario in which the code does not yet exist. In this case, the SRTs assume the role of functional specifications that can guide development of the code and validate (or not validate) its behavior. Behavior computation can also be performed on the developed code with little effort at any time after deployment to ensure that tampering has not occurred. Note also that SRTs can themselves be proven to be correct (or not correct) through use of theorem-proving systems, such as ACL2. As seen in this example, the AFOSR research has resulted in a view of SRTs as specifications of functional behavior for either reverse engineering of existing code or forward engineering of new code, with code verification carried out through behavior computation. This is an important outcome of the work that can impact the development of certified software. It is of particular interest for potential application to the verification of embedded software in autonomous systems, a need that has been identified by the USAF.

2 Research Results for Determining True Control Flow of Malware Programs

An initial step in behavior computation is to determine the true control flow of the input program. A primary focus of AFOSR research in 2010 was on the development of foundations and algorithms for this process. The H-Chart algorithm employs a frontier propagation strategy, which is the stepwise incorporation of control flow effects of successive flowchart nodes based on local computed behavior and application of SRTs. This creates an unstructured flowchart corresponding to a machine code program that is closed under semantic reachability. It works by partitioning the machine state space into code and data portions and by creating different flowchart nodes for each reachable code state. In practice, the instruction pointer (EIP register for Intel x86 programs) is used for the code portion so that each flowchart node corresponds to a different value of the instruction pointer. SRT technology plays a prominent role in this solution, and the build-out of the SRTs was directed to support the types of semantic analyses required by the algorithm.

2.1 H-Chart Algorithm

The H-Chart algorithm improves upon the simple syntactic construction of a flow chart by only looking at individual machine code instructions. The algorithm can be characterized as computing the minimal semantically closed subgraph of a given input graph. The algorithm works by starting with a subgraph consisting of just the entry point of the input graph and incrementally growing the subgraph under the condition of semantic reachability. When the subgraph can no longer be grown, it becomes the output of H-Chart algorithm and is semantically closed. Since the subgraph started off as the entry point, it is contained in any semantically closed subset and is, therefore, minimal. The algorithm performs several types of semantic analyses on machine code to determine and apply information that is not apparent from an analysis of single machine-code instructions. For example, these semantic analyses can

- determine that apparent conditional transfers of control do not actually occur (i.e., dead points)
- determine possible addresses of the targets of control transfers that are not immediately apparent (i.e., star points)
- determine if a putative subroutine call structure actually implements proper subroutine semantics (i.e., dummy calls and dummy returns)

These capabilities depend on the availability of computed behavior employed to make localized decisions on correct control flow.

Dead Points

A dead point occurs when an apparent conditional transfer of control does not actually occur. Dead points can occur due to intentional code obfuscation, often encountered in malware programs, or as a result of insufficient compiler optimization. For example, the following conditional jump is never taken:

```
xor eax, eax    ; set EAX to zero, ZF (zero flag) to true
jnz LABEL       ; jump on non-zero is never taken, because result is
                     always zero
```

Similarly, the following conditional jump is always taken:

```
xor eax, eax    ; set EAX to zero, ZF (zero flag) to true
jz LABEL        ; jump on zero is always taken, because result is
                      always zero
```

In either example, the H-Chart algorithm creates a flow chart of the conditional jump using only the transfer of control that is actually taken, as determined by the computed behavior to that point, as opposed to a syntactic flowcharting approach that always creates 1-in, 2-out predicate nodes from conditional jump instructions.

Star Points

A star point occurs when the target of a transfer of control is given by a variable or an expression, as opposed to a constant value, and, thus, the target address is not immediately apparent, for example

```
mov eax, LABEL  ; LABEL is some address in the program
xor ebx, eb     ; set EBX to zero
cmp ecx,  0     ; set ZF to true if ECX is 0, set ZF to false otherwise
setz bl         ; change EBX to 1 if ZF is true, keep EBX at 0 otherwise
add eax, ebx    ; add EBX to EAX
jmp eax         ; this jumps to either LABEL or LABEL+1, depending if
                  ECX was 0
```

Star points typically occur from jump tables that are implementing higher-level language case statements or virtual method invocations in object-oriented languages. A syntactic flowcharting approach is of little value in the presence of star points. In this case, the H-Chart algorithm performs a behavior computation of the program up to the computed jump. The algorithm then analyzes the expression corresponding to the value of the jump target to determine the set of values that it can possibly take on as a function of the initial program state.

Dummy Calls and Dummy Returns

Dummy calls and dummy returns typically occur due to code obfuscation like those found in malware programs, although a dummy call is occasionally used to obtain the current program address:

```
call $+5        ; relative call to next instruction: push address of
                  next instruction, then jump to it
pop eax         ; now EAX contains the address of this instruction
```

An example of a dummy return is

```
push LABEL      ; push LABEL on the stack
ret             ; this is really an unconditional jump to LABEL and not a
                  subroutine termination
```

Syntactic flowcharting approaches that assume all subroutine calls and returns, to include calls to dummy subroutines, are valid, can produce incorrect flow charts. Code obfuscators occasionally employ dummy calls to trick the disassemblers, such as IDA Pro, into creating incorrect function boundaries.

2.2 Using Behavior Computation for Semantic Analysis of Control Flow

The H-Chart algorithm makes use of internal calls to FX behavior computation functions to perform its semantic analyses. This is advantageous because improvements made to behavior extraction capabilities will immediately feed back into corresponding improvements in H-Chart results.

Dead Point Processing

To examine whether a given branch from a conditional jump instruction is live, a function node with semantics `ISLIVE := true` is placed after the given branch, and function nodes with semantics `ISLIVE := false` are placed at all other exits from the program, including the other branch from the conditional jump. Here, `ISLIVE` is a Boolean variable added to the state space. The resulting program is then structured and its behavior extracted. If the final value of the `ISLIVE` variable can be simplified to "false" using SRTs, then the given branch is not live, and the H-Chart algorithm will not create a predicate node for the conditional jump instruction.

Star Point Processing

To examine the possible target addresses of a given computed jump or a computed call instruction, a function node with the semantics `TARGET_ADDRESS := jump-expression` is placed after the given instruction, where `jump-expression` is the expression used to compute the jump target address by the instruction. The resulting program is then structured and its behavior extracted. After simplification using SRTs, the final value of `TARGET_ADDRESS` is examined to see if it was simplified to a constant value or to a conditional expression that selects between various constant values. Also, if memory locations in the initial state occur in the right-hand side of the behavior, these are substituted with the initialized contents of those locations from the loaded executable file.

Dummy Calls and Dummy Returns Processing

To examine whether a given call target implements proper subroutine semantics, each of the return instructions reachable from the putative subroutine entry point are analyzed as in the star point situation. If the targets of all the return instructions simplify to a memory access expression that retrieves the value from the stack that was placed by the call instruction, then the subroutine is found to have proper subroutine semantics. Otherwise, any calls to that target address are treated as dummy calls.

2.3 Semantic Reduction Theorems in True Control Determination

SRTs are crucial in performing the semantic analyses and subsequent simplifications required by the H-Chart algorithm. For example, consider the following machine code:

```
cmp eax, ebx
```

```
jbe LABEL       ; conditional jump on "below or equal"
```

Performing behavior computation at the instruction level yields the following for `cmp eax, ebx`:

```
[ ZF := int_equal(EAX, EBX)
: SF := is_neg_signed_32(add_32(EAX, negate_32(EBX)))
: PF := is_even_parity_lowbyte(add_32(EAX, negate_32(EBX)))
: CF := carry_flag_sub(EAX, EBX)
: OF := overflow_flag_sub_32(EAX, EBX)
: AF := auxiliary_carry_flag_sub(EAX, EBX) ]
```

And the following for `jbe LABEL`:

```
[ or(CF,ZF) ->
     goto LABEL
| and(not(CF),not(ZF)) ->
     IDENTITY ]
```

Composition through trace table analysis of these two behaviors gives the following overall behavior:

```
[ or(carry_flag_sub(EAX, EBX), int_equal(EAX, EBX)) ->
   [ ZF := int_equal(EAX, EBX)
   : SF := is_neg_signed_32(add_32(EAX, negate_32(EBX)))
   : PF := is_even_parity_lowbyte(add_32(EAX, negate_32(EBX)))
   : CF := carry_flag_sub(EAX, EBX)
   : OF := overflow_flag_sub_32(EAX, EBX)
   : AF := auxiliary_carry_flag_sub(EAX, EBX) ]
   goto LABEL
| and(not(carry_flag_sub(EAX, EBX)),not(int_equal(EAX, EBX))) ->
   [ ZF := int_equal(EAX, EBX)
   : SF := is_neg_signed_32(add_32(EAX, negate_32(EBX)))
   : PF := is_even_parity_lowbyte(add_32(EAX, negate_32(EBX)))
   : CF := carry_flag_sub(EAX, EBX)
   : OF := overflow_flag_sub_32(EAX, EBX)
   : AF := auxiliary_carry_flag_sub(EAX, EBX) ]
]
```

The conditions for this behavior are extensive, reflecting the fact that the Intel chip design makes use of a sequence of a first instruction to set the status flags followed by a second instruction to interpret those flags. Therefore, mathematical identities are required to verify that the condition code tests implement the desired behavior. Inside the FX system, these identities are expressed as SRTs. For example, the SRTs for the `jbe` check are

```
SRT: or(carry_flag_sub(EAX, EBX), int_equal(EAX, EBX))
==> less_or_equal(EAX,EBX)and
SRT: and(not(carry_flag_sub(EAX, EBX)), int_not_equal(EAX, EBX))
==> greater_than(EAX,EBX)
```

These SRTs, together with removal of the status flag settings that are no longer in scope, simplify the behavior of the Intel code sequence `cmp eax, ebx; jbe LABEL` to a more understandable form:

```
  [ less_or_equal(EAX,EBX) ->
     goto LABEL
  | greater_than(EAX,EBX) ->
     IDENTITY
 ]
```

Each of the condition code checks architected in the Intel chip gives rise to SRTs that reflect the usage pattern of that conditional jump preceded by the corresponding flag-setting instruction.

Other SRTs required for the H-Chart algorithm come from common patterns of code obfuscation used in malware, common patterns of jump table access from various compilers, and so on. In summary, this research has revealed the value of semantic computational methods for determining the true control flow of malware code.

2.4 Meta-Programming Language (MPL)

In the FX system, the MPL provides the syntax and semantics of the right-hand sides of the conditional, concurrent assignment statements that express the extracted behavior of input programs. The MPL was designed to be customizable and extendable, as the FX system is applied to different domains of programming.

The key underlying component of the MPL is the micro-operations. For example, some micro-operations from the behaviors above are `int_equal()`, `is_neg_signed_32()`, `add_32()`, `negate_32()`, and `carry_flag_sub()`. A micro-operation in MPL is declared by giving information about its arguments, their types, the micro-operation's return type, the definition body, and a list of properties (such as commutativity and associativity) possessed by the micro-operation.

An SRT expresses the equivalence of two MPL expressions and provides that occurrences of the first expression should be simplified by rewriting them into the second expression. For example, the following SRT expresses the fact that subtraction is equivalent to addition after negating the second operand and provides that occurrences of `sub_32()` should be rewritten in terms of `add_32()` and `negate_32()`:

```
SRT: sub_32(x,y) ==> add_32(x,negate_32(y))
```

SRTs are the primary mechanism whereby MPL expressions are simplified. Such simplification can be applied after the behavior of a sequence of instructions is formed from composing the behaviors of the individual instructions.

3 Behavior Computation Technology Transition

The research and development in behavior computation supported by AFOSR is being implemented in the Function Extraction for Malicious Code (FX/MC) system. This system, based on Application Programming Interface (API), applies structuring and behavior computation to help eliminate

- control flow obfuscation created by massive amounts of intruder-inserted complex branching logic superimposed on malware to make analysis difficult or even impossible
- no-op blocks of code (code with no functional effect) inserted by intruders to increase the size and complexity of malware

The system also helps identify subroutine boundaries in portable executable (PE) files of malware code. The H-Chart algorithm is part of this system.

Oak Ridge National Laboratory (ORNL) is initiating a Department of Energy-sponsored project to apply behavior computation to verification and vulnerability analysis of embedded software in Smart Grid components. This two-year project will implement SEI-developed behavior computation capabilities on High-Performance Computing (HPC) clusters at ORNL and apply them to critical components in the grid network.

In addition, behavior computation has the potential to be applied in a number of areas that have been examined and documented through studies and publications, including program comprehension [Collins 2008], software verification [Bartholomew 2007, Burns 2009, Linger 2006, Linger 2007], software test and evaluation [Pleszkoch 2008], and security analysis [Linger 2010, Pleszkoch 2004, Pleszkoch 2010, Walton 2006, Walton 2010].

4 Next Steps in the Research Program

Building on 2009–10 results, the AFOSR research could be used to continue development of theory and implementation for SRTs in the behavior computation process. SRT research could also be applied to improvements in loop behavior computation (2009 focus) and true control flow determination (2010 focus).

Most importantly, as noted above, the 2010 research revealed a potential new approach to both reverse engineering of legacy and acquired software and to the development of new software. This rigorous approach treats SRTs as specifications (a posteriori for reverse engineering, a priori for new development) that permit verification of software through behavior computation. This emerging potential could be investigated and reported upon as a key next step in development of the technology. Behavior computation research and development could help address challenging problems identified by the United States Air Force, for example, the need for better verification and validation (V&V) technology to certify embedded software in autonomous systems, as articulated by Dr. Werner Dahm in the *Technology Horizons* report.

Appendix: Mathematical Foundations of Software Behavior Computation

Because the research by the Air Force Office of Scientific Research (AFOSR) has contributed so much to the understanding of the foundations of behavior computation, it is appropriate to include an appendix that summarizes some theoretical aspects of the work.

Behavior Computation Strategy

Viewing programs as rules, or implementations, for mathematical functions or relations permits automated computation of behavior by providing foundations for transformation from procedural logic to nonprocedural functional form. In brief, these transformations are defined by the Correctness Theorem and are localized to individual single-entry, single-exit control structures nested and sequenced in an algebraic structure [Linger 1979]. The computed functional forms for each control structure can be propagated within the algebraic structure in a stepwise composition process. For completeness, composition requires that instructions be expressed in terms of their full functional semantics. To create the initial algebraic structure, it is necessary to transform any spaghetti logic in an input program into structured form. The constructive proof of the Structure Theorem defines a method for this transformation [Linger 1979]. The structuring process itself takes as input the true control flow of the input program, which must first be determined in the presence of computed jumps, jump tables, and so on. Finally, initial computed behavior exhibits opportunities for simplification and abstraction. Taken together, these requirements prescribe a general process for behavior computation as follows:

transformation of instructions into functional semantics →

determination of true control flow of input spaghetti logic →

structuring of spaghetti logic into control structures in an algebraic structure →

computation of non-procedural behavior expressions in a stepwise process →

simplification and abstraction of initial computed behavior

There is a lot more to behavior computation, as described below, but this framework provides an anchor for understanding and analysis.

Theoretical Basis

Function extraction (FX) takes, as input, a program expressed in a Turing-complete language with precisely defined semantics, such as Intel machine code, and produces, as output, a symbolic representation of the program's behavior. The symbolic behavior expression can be used either by humans to help understand and verify what the program does or as input to further automated analysis.

FX is rooted in denotational semantics as defined by Dana Scott and Christopher Strachey [Scott 1977, Stoy 1977] and applied to computing by Harlan Mills [Mills 1975, Linger 1979]. Denotational semantics provides a basis for programming language semantics by, first, defining a mathematical domain of semantic objects and, then, defining a mapping from the set of all programs into the domain of those semantic objects. Thus, denotational semantics operates at a higher level of abstraction than operational semantics, which, instead, works by defining a concrete model of computation and then mapping programs into that computational model. FX extends the practical application of denotational semantics by defining a referentially transparent symbolic representation for the domain of semantic objects based on lambda calculus and type theory [Barendregt 1985, Sambin 1998] and then automating the extraction of the semantics from actual input programs. Thus, FX could be viewed as "computational denotational semantics."

Mills' functional verification uses the particular domain of set-theoretic functions (with set-theoretic relations for nondeterministic programs) to provide a mathematical basis for not only program semantics, but also program verification and top-down program development. The key result for its use in program verification is the Correctness Theorem, which defines function-equivalent transformations from prime control structures into nonprocedural functional form: for control structure labeled `P`, operations on data labeled `g` and `h`, predicate labeled `p`, and program function labeled `f`. These function equations are independent of language syntax and program subject matter, defining the mathematical starting point for behavior calculation as follows:

The behavior of a sequence control structure

```
P: g; h
```

can be given by

```
f = [P] = [g; h] = [h] o [g]
```

where square brackets denote the behavior signature of the enclosed program and `o` is the composition operator. That is, the program function of a sequence can be calculated by ordinary function composition of its constituent parts.

The behavior of an alternation control structure

```
P: if p then g else h endif
```

can be given by

```
f = [P] = [if p then g else h endif]
  = ([p] = true → [g] | [p] = false → [h])
```

where `|` is the `or` symbol. That is, the program function of an alternation is given by a case analysis of the "true" and "false" branches.

The behavior of an iteration control structure

```
P: while p do g enddo
```

can be expressed using function composition and case analysis in a recursive equation based on the equivalence of an iteration control structure and an iteration-free control structure (an `ifthen` structure):

```
f = [P] = [while p do g enddo]
  = [if p then g; while p do g enddo endif]
  = [if p then g; f endif]
```

This recursive functional form must undergo additional transformations to arrive at a non-recursive representation of loop behavior. For FX, the Correctness Theorem guides stepwise computation and propagation of the procedure-free function of each prime control structure in turn.

Functional verification, as defined by the Correctness Theorem, provides a practical approach to program correctness. In contrast to axiomatic verification, which is context-dependent, functional verification is context-free. For example, there could be dozens of individual statements in a large program that say `x := x + 1`. In functional verification, each of these statements is verified in exactly the same way; whereas, in axiomatic verification, each of these statements would have a different, unique precondition and postcondition based on its context in the larger program. Functional verification localizes context, an important benefit for behavior computation. This localization is a fundamental property of prime control structures, whose single-entry, single-exit flowcharts permit nesting and sequencing in algebraic structures.

Because functional verification methods operate directly on structured programs expressed in structured flowcharts, it is important to be able to transform other program representations into structured flowchart form. The Structure Theorem provides a constructive proof for transforming spaghetti logic into a function-equivalent algebraic expression in single-entry, single-exit prime control structures, including `sequence`, `ifthenelse`, and `whiledo`. Programs may exhibit an essentially infinite number of execution paths but only a finite number of control structures, each of which can be analyzed, in turn, in a finite, stepwise process. This algebraic structure permits stepwise application of the Correctness Theorem for computing behavior. In addition, a method for transforming machine code into flowcharts is described in Mills. This method forms the key basis for determining the true control flow (the H-Chart algorithm) of input spaghetti logic prior to structuring.

Computability

Rice's Theorem states that any nontrivial functional property of Turing machines is undecidable. Since termination is a functional property, Rice's Theorem can be thought of as a generalization of the undecidability of the Halting Problem. Applied to behavior computation, Rice's Theorem means that FX is theoretically unable to reduce all function-equivalent input programs to an identical, normal-form behavior expression.

At first thought, it may appear that Rice's Theorem implies failure of the FX approach. However, this is not the case at all. Consider this example of two programs, P1 and P2. P1 takes an integer N as input and always returns 0. P2 also takes an integer N as input and searches for counterexamples to Riemann's Hypothesis that are less than distance N from the complex origin. P2 returns

0 if no counterexample is found and returns 1 otherwise. It is an open problem in present-day mathematics as to whether P1 and P2 compute the same function. Thus, it is not realistic to expect FX to output the same symbolic behavior expression for both P1 and P2. Instead, a realistic behavior expression for P1 would show that the program always outputs 0, and a realistic behavior expression for P2 would show that the computational content of the program is to search for a counterexample to Riemann's Hypothesis and report on the results.

For real-world programs, although there are many different ways a person or compiler could implement the specification of always returning 0, the authors have yet to see an implementation where the coder first proved Riemann's Hypothesis and then coded P2. Thus, the theoretical limitations of Rice's Theorem prove not to be that important in practice.

Another common example of Rice's Theorem in practice is the termination of loops. Imagine a loop that keeps on looking for a counterexample for Riemann's Hypothesis until it finds one and, thus, loops forever if Riemann's Hypothesis is true. How can FX extract the behavior of such a loop when the best mathematicians in the world cannot figure out if the loop terminates or not? The answer is easy: The behavior of the loop is given as a conditional rule, based on the condition of Riemann's Hypothesis being true or false. However, given more realistic programs, the FX system can often prove the termination of given loops and express the end-to-end loop behavior in a single noniterating behavior expression.

In summary, it is theoretically and practically possible for FX to compute normal-form behavior expressions on a wide range of real-world input programs, even though complete coverage is unobtainable due to Rice's Theorem. And non-normal-form behavior expressions can still be of significant use in most, if not all, FX applications.

Despite Rice's Theorem, it is theoretically possible for FX to compute a mathematically correct symbolic behavior expression for every input program. This is because, from one theoretical perspective, the behavior computation process can be viewed as a straightforward translation from one Turing-complete language into another. The next important question becomes what the run-time performance of such a translation process is. It turns out that the computational complexity of such a program transformation is typically linked to how much the size of the output expands compared to the size of the input. Input features that exist in the output language can typically be translated directly, in linear space and time. Input features that do not exist in the output language must be represented in some other way and may experience greater than linear growth in translation.

For FX, the output language is the symbolic language of behavior expressions. Following the lead of denotational semantics, these expressions are both referentially transparent and procedure-free. Referential transparency is a key property of denotational semantics, which provides that every behavior expression and subexpression can be understood directly by examining it by itself, without any context information being required. Procedure-free means that behavior expressions describe the output of a program directly in terms of its input, without defining the program's procedure that implements that computation.

The FX system uses the trace table algorithm to translate procedural input into procedure-free output. Trace tables contain rows for instructions and columns for conditions and variables that

are assigned new values, with cells recording results of successive compositions. Trace tables represent the implementation of the function equations of the Correctness Theorem. The resulting computed behavior is expressed in terms of conditional concurrent assignments (CCAs). CCAs are procedure-free, vector assignments from input state to output state guarded by conditions on input state. CCAs define disjoint partitions of the behavior space and can represent behavior from individual instructions, to control structures, to entire programs. The right-hand sides of CCAs contain expressions in terms of program variables in the state space and operations on those variables. This is the only language structure required for representing computed behavior of sequential logic.

Because the trace table algorithm performs the substitution of intermediate value expressions into later value usage, using the trace table algorithm by itself can result in an exponential increase in the size of behavior expressions. This can be avoided by a common subexpression factoring and abbreviation technique. This abbreviation approach will completely solve the problem, permitting behavior computation in linear time and space under all circumstances. In particular, this technique allows intermediate program conditions to be localized to a behavior abbreviation and, thus, not propagate up to higher-level behavior expressions. It is also important to note that, as behavior computations move to higher levels in the algebraic structure, it is often the case that local behavior drops out of expressions, essentially becoming out of scope. In addition, behavior at higher levels in the algebraic structure can often take advantage of symbolic operations that express higher levels of abstraction. For example, the top of an input procedure might easily be expressed as performing a matrix multiplication, whereas lower levels of that procedure that perform individual computations in the matrix multiplication would be expressed, say, in terms of vector operations and dot products. Such abstractions are easily expressed in Semantic Reduction Theorems (SRTs) and can be defined.

The objective of initial behavior computation is mathematical correctness in transforming from procedural logic to nonprocedural functional form. After initial extraction, which is linear in the number of instructions once the abbreviation technique has been applied, symbolic behavior expressions still must be simplified as much as possible for best use by both human users and follow-on automated analysis. In understanding the run-time complexity of these simplification processes, it is important to realize that worst-case complexity is not always the best measure of performance. For example, the Simplex method for linear optimization exhibits an exponential worst-case run-time complexity but, in practice, always finds the linear optimum faster than more complicated optimization algorithms that only exhibit polynomial worst-case complexity. The simplifications used in FX have been chosen for their performance against typical, real-world input programs, regardless of worst-case complexity.

Note that, at a conceptual level, the run-time complexity of the behavior simplification processes are dependent on the degree of simplification that is performed. Thus, it is possible to give the FX user a measure of control over the amount of resources that should be allocated to this step.

The primary method of behavior simplification in FX is the use of rewriting rules that are based on equivalence theorems for various symbolic operations. These theorems, known as SRTs, are expressed in meta-programming language (MPL), the internal language that FX uses for behavior expressions. MPL incorporates concepts from type theory and lambda calculus [Barendregt 1985,

Sambin 1998]. It provides a Turing-complete symbolic language for behavior expressions. The specific operations used in MPL are defined in an external MPL file to extend FX, which enables new instructions, new input languages, and new SRTs to be added without requiring recoding of the internal Java code of the system. Concepts from the rewriting tool Maude were used in the development of the SRT component of MPL. SRTs are used for reducing, simplifying, and abstracting computed behavior expressions and are also applied to loop behavior computation.

The complexity of applying rewrite rules depends on many factors, including the number of rules that need to be applied. The current FX system has a rewrite limit that cuts off the application of rewrite rules after a preselected count.

Additional simplification methods are planned to be added to the system for improving human-readable results. Binary decision diagrams (BDD) simplify Boolean expressions and overcome the limitation that perfect Boolean simplification is NP-complete. Presburger arithmetic quantifier elimination provides a method for simplifying integer expressions involving addition, constant multiplication, and inequalities, which often occur in compiler-generated code. Common expression folding collapses instances of identical sub-behaviors, often in large numbers, that tend to occur in compiled code. Hierarchical organization provides levels of abstraction emphasizing nesting behavior found in real programs. These and other techniques are planned for future releases of FX.

Assumptions and Limitations

One absolute assumption of FX is that the semantics of the input programming language are known precisely. For the case of Intel machine instructions, this includes the assumption that the Intel specification manuals, as augmented by information available on the Internet, provide a correct description of the x86 processor semantics.

One absolute limitation of FX is given by Rice's Theorem: FX cannot always provide a symbolic behavior expression in absolute normal form. More advanced results from recursion theory indicate that there will inevitably also be input programs for which the FX output behavior expression is more complicated than the smallest equivalent behavior expression. However, it is always possible for FX to provide a symbolic behavior expression that is 100 percent mathematically correct relative to the assumption that the input programming language semantics are precisely known.

In practice, the real-world assumptions of the FX system come from situations where we seek to provide a "better" answer than pure mathematical theory will allow. These are discussed in the Problem Classes and Boundaries section.

Problem Classes and Boundaries

Analysis and Simplification

There will be cases where the FX simplification algorithms will not be able to produce the simplest possible behavior expression. However, in particular cases where a simpler form is known, it is possible to augment the rewriting theorems (SRTs) as needed. For example, given Andrew Wiles' proof of Fermat's Last Theorem, it is now possible to add a simplification rule that will

recognize that a program written to find a counterexample to Fermat's Last Theorem will never succeed in finding one. Note that such a rule has not been needed so far.

Because of the lack of complete simplification, there will be times when the FX system will not be able to determine something about the input program and will be forced to assume the worst case. For example, in considering a conditional jump instruction, such as the `JZ jump if the zero flag is set` instruction, the H-Chart algorithm for defining true control flow will attempt to determine which outgoing branches of the conditional jump are live. There will be many cases where the FX system will be able to determine that both branches are, indeed, live. There will also be many cases where the system will be able to determine that one particular branch is definitely dead. However, there will inevitably be cases where the system cannot definitely determine whether a given branch is live or dead and so must conservatively assume that both branches are live, thus exploring additional parts of the input program.

Self-Modifying Code

The H-Chart algorithm for true control flow definition deals with self-modifying code in a mathematically correct manner, but it is not very useful in practice. Thus, when self-modifying code can be detected, the appropriate response is to reject the input program as not analyzable. However, as mentioned in the Analysis and Simplification section, there will also be cases where the location of a memory write cannot be completely determined to be a data location and not a code location. In those cases, a reasonable response is to continue processing the input program with a warning that, if the code is self-modifying, the analysis will not be correct.

Disciplined Stack Usage

In some situations involving code that has undergone automated obfuscation, the sequence of a push followed by a pop is treated as a no-operation instruction by the obfuscation. However, from a mathematical perspective, this code sequence has the effect of doing a write to memory at a location that is now in the "dirty" part of the stack. In order to undo the effects of the obfuscation, the FX system has been augmented with the capability to assume disciplined stack usage and identify such sequences as no-ops. Thus, for programs that do not have disciplined stack usage, an incorrect analysis may result. Note that, in the FX configuration file, there is an option for turning the dirty stack simplification behavior on or off. If turned off, FX will continue to track all updates to the portion of memory representing the stack, regardless of the current value of the stack pointer. So, if an analyst thinks the program is reading already-popped items from stack memory, the dirty stack simplification behavior can be turned off, and the program can be reanalyzed.

Concurrency

The present version of the FX system uses only sequential semantics for Intel machine code. For example, the `lock` prefix for Intel instructions is completely ignored. The motivation for this choice is that such semantics are easier to understand and very useful for many FX applications. An FX system that computes the concurrent semantics of machine code is certainly possible based on the same basic principles as the current system. However, it will be important to understand the desired applications before making an appropriate choice from the many different versions of denotational semantic domains that have been developed to model concurrency.

Finite Arithmetic

Precise behavior computation requires dealing with the finite precision of machine computations. The FX system deals with finite machine arithmetic exactly. For example, the extracted semantic behavior of an add instruction contains both overflow and non-overflow cases. In the planning stages of the FX system, it was envisioned that a simplified semantics containing just the non-overflow case might be useful in certain FX applications where the emphasis was on the mainline calculation for an input program. However, it turned out that including the overflow semantics was more useful in all situations.

Memory Aliasing

Memory aliasing occurs if two addresses in a program could refer to the same memory location. This is a difficult problem in computer science, and Rice's Theorem guarantees that there will always be some programs for which the answer is not known. For cases where occurrence of memory aliasing cannot be ruled out, there are three possibilities for how the FX system can handle the situation. The first is to use conditional semantics to describe all possible cases of aliasing; however, this can lead to rapid growth in the size of the extracted behavior. The second is to use heuristics to select some subset of all potential memory aliases to consider in detail, with the other cases assumed to not overlap. The third is to use an extended memory model that keeps track of the order in which memory accesses are made; however, this can lead to more complicated semantic behavior expressions because the order-dependent memory operations are not commutative. In real-world situations where the FX system was producing less than optimal behavior expressions due to the potential for memory aliasing, we have found that a human user can frequently determine whether aliasing is present and rerun the FX system with appropriate options to obtain a compact expression for the program's behavior.

Behavior Computation Capabilities

FX can compute normal-form behavior expressions on a wide range of real-world input programs, even though complete coverage is unobtainable due to Rice's Theorem. And non-normal-form behavior expressions can still be of significant use in FX applications.

It is important not to define the FX system solely by limitations present in computability theory. For example, computer algebra systems cannot solve every equation, but they nonetheless provide invaluable assistance to human users by accurately keeping track of more details than human minds can manage and by performing symbolic algorithms that have been developed by the some of the best minds from throughout history. Similarly, the FX system can effortlessly keep track of the values of many variables across pages of program text that would leave an unaided human analyst in the dust. In addition, SRTs can encapsulate the skills and insights of the best human analysts, making that knowledge available to every FX user.

The internal algorithms of the FX system have been designed to be language-independent, permitting additional languages to be processed through new front ends that translate instructions into functional semantics. As a result, automated behavior computation can be applied to many tasks in the software engineering life cycle.

Malicious code analysis is a particular strength of FX technology. FX can help deobfuscate and determine the functionality of malware in native, metamorphic, and polymorphic forms. In this connection, malware exhibits a fundamental vulnerability: no matter how it is obfuscated by an intruder, the malware functionality must remain intact to execute on the target machine using its language and facilities. That is, obfuscation must not impair the intended malicious effect. The Function Extraction for Malicious Code (FX/MC) system, currently under development, uses computed behavior to remove obfuscation caused by insertion of spaghetti control flow and no-op blocks of code to reveal the core functional instructions and compute their behavior. Millions of lines of code have been processed in testing the deobfuscation capabilities of the system.

One of most complicated types of metamorphic malware is virtualized malware, where the malware is implemented on top of a customized software-based virtual machine so that the malware behavior is expressed in a novel manner that resists traditional analysis. FX techniques have proven to be an important tool to defeat this type of obfuscation. By examining the overall functional effect of sequences of virtual machine instructions, it proved to be possible to detect when a sequence of virtual instructions accomplished the same effect as some x86 instruction and then to record that decoded x86 instruction. In this manner, we were able to undo most of the virtual machine encoding that had been performed, obtaining a standard x86 program suitable for reverse engineering with existing tools and techniques [Pleszkoch 2010].

References

URLs are valid as of the publication date of this document.

[Barendregt 1985]
Barendregt, Hendrik P. *The Lambda Calculus: Its Syntax and Semantics. (Studies in Logic and the Foundations of Mathematics 103)*. Elsevier Science, 1984.

[Bartholomew 2007]
Bartholomew, R.; Burns, L.; Daly, T.; Linger, R.; & Prowell, S. "Function Extraction: Automated Behavior Computation for Aerospace Software Verification and Certification." *Proceedings of 2007 AIAA Aerospace Conference.* Monterey, CA, May 2007. AIAA, 2007.

[Burns 2009]
Burns, L. & Daly, T. "FXplorer: Exploration of Computed Software Behavior: A New Approach to Understanding and Verification," 1-10. *Proceedings of Hawaii International Conference on System Sciences (HICSS-42)*, Big Island, HI, Jan. 2009. IEEE Computer Society Press, 2009.

[Collins 2008]
Collins, R.; Linger, R.; Walton, G.; & Hevner, A. "The Impacts of Function Extraction Technology on Program Comprehension: A Controlled Experiment." *Journal of Information & Software Technology 50*, 11 (October 2008): 1165-1179. Edited by M. Shepperd; C. Wohlin; & S. Elbaum.

[Linger 1979]
Linger, Richard C.; Mills, Harlan D.; & Witt, Bernard I. *Structured Programming: Theory and Practice*. IBM Systems Programming Series, Addison-Wesley, 1979.

[Linger 2006]
Linger, R.; Pleszkoch, M.; & Prowell, S. "Automated Calculation of Software Behavior with Function Extraction (FX) for Trustworthy and Predictable Execution." *Proceedings of NIST Static Analysis Conference.* Gaithersburg, MD, June 2006. National Institute of Standards and Technology, 2006.

[Linger 2007]
Linger, R.; Pleszkoch, M.; Burns, L.; Hevner, A.; & Walton, G. "Next-Generation Software Engineering: Function Extraction for Computation of Software Behavior." *Proceedings of the Hawaii International Conference on System Sciences (HICSS-40).* Big Island, HI, Jan. 2007. IEEE Computer Society Press, 2007.

[Linger 2010]
Linger, R.; Pleszkoch, M.; & Sayre, K. "Computing the Behavior of Malware." *Proceedings of the 6th Annual Cyber Security and Information Intelligence Workshop.* Oak Ridge National Laboratory, Oak Ridge, TN, April 2010. Association for Computing Machinery Press, 2010.

[Mills 1975]
Mills, Harlan D. "The New Math of Computer Programming." *Communications of the ACM 18*, 1 (Jan. 1975): 43-48.

[Mills 1980]
Mills, Harlan D. "Function Semantics for Sequential Programs," 241-250. *Proceedings of the IFIP Congress 80*. Amsterdam, North-Holland, 1980.

[Pleszkoch 2004]
Pleszkoch, M. & Linger, R. "Improving Network System Security with Function Extraction Technology for Automated Calculation of Program Behavior." *Proceedings of Hawaii International Conference on System Sciences (HICSS-37)*. Big Island, HI, Jan. 2004. IEEE Computer Society Press, 2004.

[Pleszkoch 2008]
Pleszkoch, M.; Linger, R.; & Hevner, A. "Introducing Function Extraction into Software Testing." *The Data Base for Advances in Information Systems: Special Issue on Software Systems Testing 39*, 3 (August 2008): 41-50.

[Pleszkoch 2010]
Pleszkoch, M.; Prowell, S.; Cohen, C.; & Havrilla, J. "Applying Function Extraction (FX) Techniques to Reverse Engineer Virtual Machines." *2009 CERT Research Annual Report*. ed. R. Linger. Software Engineering Institute, Carnegie Mellon University, 2010. http://www.sei.cmu.edu/newsitems/cert_researchreport2009.cfm

[Sambin 1998]
Sambin, Giovanni & Smith, Jan M., eds. *Twenty-five Years of Constructive Type Theory*. Oxford University Press, 1998.

[Scott 1977]
Scott, Dana S. "Logic and Programming Languages." *Communications of the ACM 20*, 9 (Sept. 1977): 634-641.

[Stoy 1977]
Stoy, Joseph E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*. MIT Press, 1977.

[Walton 2006]
Walton, G.; Longstaff, T.; & Linger, R. *Technology Foundations for Computational Evaluation of Security Attributes* (CMU/SEI-2006-TR-021). Software Engineering Institute, Carnegie Mellon University, 2006. http://www.sei.cmu.edu/library/abstracts/reports/06tr021.cfm

[Walton 2010]
Walton, G. & Linger, R. "Security Requirements as Functional Behaviors for System Analysis." *Proceedings of the 9th Annual Security Conference*. Las Vegas, NV, April 2010. Springer, 2010.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. **AGENCY USE ONLY** (Leave Blank)	2. **REPORT DATE** February 2011	3. **REPORT TYPE AND DATES COVERED** Final
4. **TITLE AND SUBTITLE** Function Extraction (FX) Research for Computation of Software Behavior: 2010 Development and Application of Semantic Reduction Theorems for Behavior Analysis		5. **FUNDING NUMBERS** FA8721-05-C-0003
6. **AUTHOR(S)** Richard Linger, Tim Daly, Mark Pleszkoch		
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-2011-TR-009
9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. **SPONSORING/MONITORING AGENCY REPORT NUMBER** ESC-TR-2011-009
11. **SUPPLEMENTARY NOTES**		
12A **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS		12B **DISTRIBUTION CODE**

13. **ABSTRACT (MAXIMUM 200 WORDS)**

For several years, the Software Engineering Institute (SEI) at Carnegie Mellon University has been engaged in a project to compute the behavior of software with mathematical precision to the maximum extent possible. Air Force Office of Scientific Research (AFOSR) sponsorship has played a key role in this effort. The general thrust of the research for AFOSR has been in technology for (1) overcoming difficult aspects of behavior computation and (2) analyzing and manipulating computed behavior. In 2009, the research focused on computing the behavior of loops, a process subject to theoretical limitations. This resulted in practical methods for loop computation that minimize the effects of these constraints. The 2010 research focused on foundations and implementations of algorithms that employ computed behavior and semantic reduction theorems to determine the true control flow of malware programs as an essential first step in computing overall malware behavior. Determining the true control flow of a program in the presence of computed jumps and jump table operations has been a difficult problem for some time. Syntactic methods of control flow analysis exhibit limitations that reduce their effectiveness. The semantic methods employed by behavior computation can produce improved results. The findings of this research have been implemented in a system for malware analysis and have improved capabilities for behavior computation in other applications. At the same time, the research has revealed a potential new approach to both reverse engineer and forward engineer software based on rigorous specification and verification in the context of behavior computation.

14. **SUBJECT TERMS** function extractions, binary code analysis, object code analysis, deobfuscation, computed behavior, malware analysis, verification, validation	15. **NUMBER OF PAGES** 35
16. **PRICE CODE**	

17. **SECURITY CLASSIFICATION OF REPORT** Unclassified	18. **SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	19. **SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	20. **LIMITATION OF ABSTRACT** UL