Function Extraction Technology: Computing the Behavior of Malware

Rick Linger, Kirk Sayre, Tim Daly, Mark Pleszkoch
CERT, Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA
rlinger, sayre, daly, mpleszko@cert.org

Abstract

Current methods of malware analysis are increasingly challenged by the scope and sophistication of attacks. Recent advances in software behavior computation illuminate an opportunity to compute the behavior of malware at machine speeds, to aid in understanding intruder methods and developing countermeasures. The behavior computation process helps eliminate certain forms of malware obfuscation and computes the net effects of the remaining functional code. This paper describes behavior computation technology and provides an example of its use in malware analysis.

1. A malware vulnerability

Malware often exhibits a fundamental vulnerability that can be exploited by defenders. No matter how a malware package is obfuscated, and no matter what attack strategy it implements, it must ultimately execute on a target machine to achieve its objectives. That is, the intended behavior of a malware package must be realized through ordinary execution of instructions and manipulation of memory, just as must the intended behavior of legitimate software. A potential Achilles heel of malware is literally its functional behavior which must achieve a purpose intended by the attacker. This paper describes application of software behavior computation to help eliminate certain forms of obfuscation in malware and derive the net behavior of the remaining functional code.

This malware vulnerability is being exploited through research and development carried out by the CERT organization of the Software Engineering Institute at Carnegie Mellon University. The result is an emerging technology named Function Extraction (FX). The objective of behavior computation is to produce the net functional effect of the sequential logic of a program in all circumstances of use with mathematical precision to the maximum extent possible. This process is subject to theoretical limitations, for example in loop behavior computation. Research has shown how to reduce the effects of these limitations for practical application. Function Extraction has been successfully employed in malware analysis [12,13].

A specialization of FX technology is being developed in the FX/MC (Function Extraction for Malicious Code) system. Intruders often obfuscate malware packages with complex control flow (spaghetti logic) and blocks of no-op code (code with no functional effect), in an effort to make analysis difficult or impossible. The FX/MC system eliminates obfuscation caused by spaghetti logic and no-op blocks, and computes the net behavior of the remaining functional code.

2. Foundations of behavior computation

Software behavior computation is enabled by the Structure Theorem and the Correctness Theorem.

The Structure Theorem guarantees the sufficiency of single-entry-single-exit sequence, alternation, and iteration control structures to represent any sequential program. The constructive proof of the Theorem defines an algorithm for transforming arbitrary control flow containing jumps into function-equivalent form expressed as an algebraic structure of nested and sequenced control structures. This structure is a necessary precondition for behavior computation. Application of the Structure Theorem eliminates arbitrary branching logic as is found in control flow obfuscation of malware packages. The proof of the theorem is given in [8].

The Correctness Theorem defines the transformation of procedural control structures, including sequence, ifthenelse, and whiledo, into procedure-free functional forms. The functional forms represent the behavior signatures of the control structures. They can be obtained through function composition and case analysis as described below (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, and define the mathematical starting point for behavior calculation.

The behavior signature of a sequence control structure

P: g; h

can be given by

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

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

The behavior signature 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 signature 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 representation of loop behavior that is readily understandable. The roots of the Correctness Theorem are found in the mathematics of denotational semantics [1,8,9,11,14]. Proofs of the Correctness Theorem and a related Iteration-Recursion Lemma are given in [8].

The functional behavior defined by the Correctness Theorem is identical to that of the control structure from which it is computed, that is, the computed behavior and corresponding control structure are function-equivalent mappings of inputs into outputs. Thus, computed behaviors can be freely substituted for corresponding control structures. Such substitution defines a stepwise process of behavior computation, whereby the algebraic control structure hierarchies produced by the Structure Theorem are traversed from bottom to top. At each step, net effects of control structures are composed and propagated while procedural details are left behind. Behavior computation involves mathematics beyond the Structure and Correctness Theorems, but it would be impossible without them.

3. The FX/MC system

Substantial research in mathematical foundations and algorithm design for behavior computation has been required to develop the technology to its present state. To see how the FX/MC system works, consider the architecture diagram of Figure 1. FX/MC operates on malware coded in or compiled into Intel assembly language. The algorithmic process of behavior computation requires four principal steps as follows.

Step 1: Transform instructions to functional semantics. Behavior computation operates at the level of functional semantics of programs, not syntactic representations. Each instruction in an input program is transformed into a functional form that defines the net effect of the instruction on the state of the system. For example, an add instruction operating on registers not only produces a sum, but also changes the values of certain flag registers on the processor. The instruction transformation is driven by a pre-defined repository of instruction semantics as shown in the figure. There are over 1100 op codes on the processor. Build-out of this repository is an ongoing task.

Step 2: Transform program to structured form. The true control flow of the input program, including any computed jumps and branching logic, is determined by deterministic reachability analysis in a frontier propagation algorithm. The program is then transformed into structured form, guided by the constructive proof of the Structure Theorem. This step expresses the program in an algebraic structure of single-entry, single-exit control structures including sequence, ifthenelse, and whiledo. Control flow obfuscation caused by arbitrary jumps in the code, often inserted by intruders using commonly available tools, is eliminated by the structuring process.

Step 3: Compute program behavior. Behavior computation can now be carried out, guided by the Correctness Theorem that defines transformations from procedural structures to non-procedural behavior expressions. A significant amount of mathematical processing is required for this step. Research has shown how to accommodate theoretical limitations on loop behavior computation.

Step 4: Reduce behavior to final form. The computations of step 3 account for all behavior, even taking machine precision into account. This initial behavior must now be reduced to final form. In analogy, recall high school algebra and the need to reduce expressions such as $3x^3 + 2x^2 - x^3 + 4x^2$ to $2x^3 + 6x^2$. This process is driven by a repository of Semantic Reduction Theorems (SRTs) as shown in the figure. These micro-theorems encapsulate information required to reduce terms in computed behavior to simpler form. The theorems are very general and widely applicable. For example, the library of SRTs for finite arithmetic provides reductions for arithmetic expressions will not require modification unless the processor architecture is modified. Build-out of this repository is an on-going task. In addition, computed behavior can exhibit structural relationships useful for organization and presentation. For example, behavior expressions often contain repeated substructures that can be factored and abstracted.

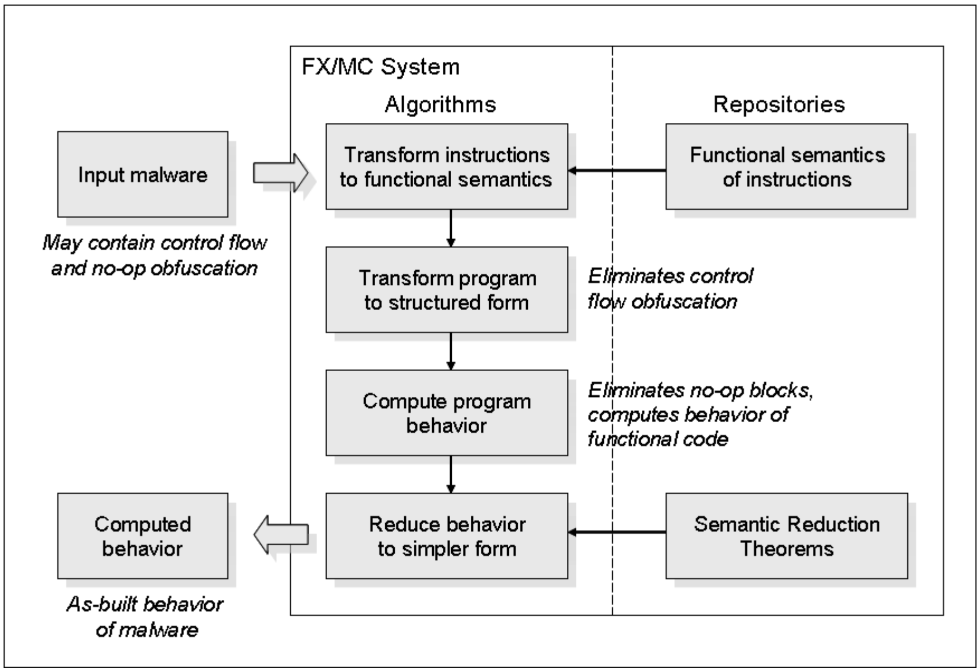

Figure 1: FX/MC system architecture

4. Properties of behavior computation

Consider the miniature illustration of behavior calculation in Figure 2. The three-line program in the upper left is expressed in design language form, and operates on small integers x and y (":=" is the assignment operator). It is not immediately obvious what the program is doing, but its effect can be calculated with the trace table shown in the Figure. The table contains a row for each assignment and a column for each variable assigned. Each row shows the effect of its assignment on variables x and y (in the first row, "0" signifies "old value," 1 signifies "new value, and similar for the other rows). The derivations apply algebraic substitutions and reductions in a function composition process to arrive at output values for the program expressed in terms of input values, with intermediate operations abstracted out. This computation reveals that the program is a swap that exchanges the initial values of x and y.

The behavior is expressed in terms of a conditional concurrent assignment (CCA). The condition is true (the sequence is always executed since it contains no branching logic), and the assignments to final x and y are carried out concurrently, that is, all expressions on the right of the assignment operators are assigned to all targets on the left at the same time. This CCA structure is the only statement form required in the FX/MC behavior expression language. It is an important structure for understanding the examples that follow.

Suppose the program of Figure 2 contained an error, say, for example, that the addition in the first assignment had been mistakenly coded as a subtract operation. The trace table and derivations would reveal the computed behavior as the following concurrent assignment, and the error is apparent:

true →

x := y
y := x – 2y

This miniature example can be used to point out two important properties. The first is the “computing, not searching” property. Behavior computation does not search for things in code at the syntactic level, as is the case with many methods of analysis. Rather, it applies the semantics of instructions and the mathematics of function composition to compute net effects of programs. Thus, both the correct and error results of the computation above are produced by the same algorithm, with no special cases of analysis required. The computation simply “follows it nose” to produce whatever behavior is present, whether intended, unintended, or malicious.

The second is the “many implementations, one behavior” property. There are many possible ways to implement a given specification. For example, the swap could be implemented with a temporary variable, t,

t := x
x := y
y := t

or with “exclusive or” instructions:

x := x xor y
y := x xor y
x := x xor y

Each of these implementations would result in the same computed behavior on x and y, namely, a swap of their initial values. When behavior is computed, specifics of procedural implementations are replaced by net behavior that can represent a variety of algorithmic strategies. This property will prove useful in identifying and analyzing malware families.

Of course, orders of magnitude more mathematical processing are carried out by the FX/MC system in computing behavior for real programs. This simple example nevertheless depicts generation of behavior knowledge through function composition and illustrates key properties of the process. The next section provides a more substantial example.

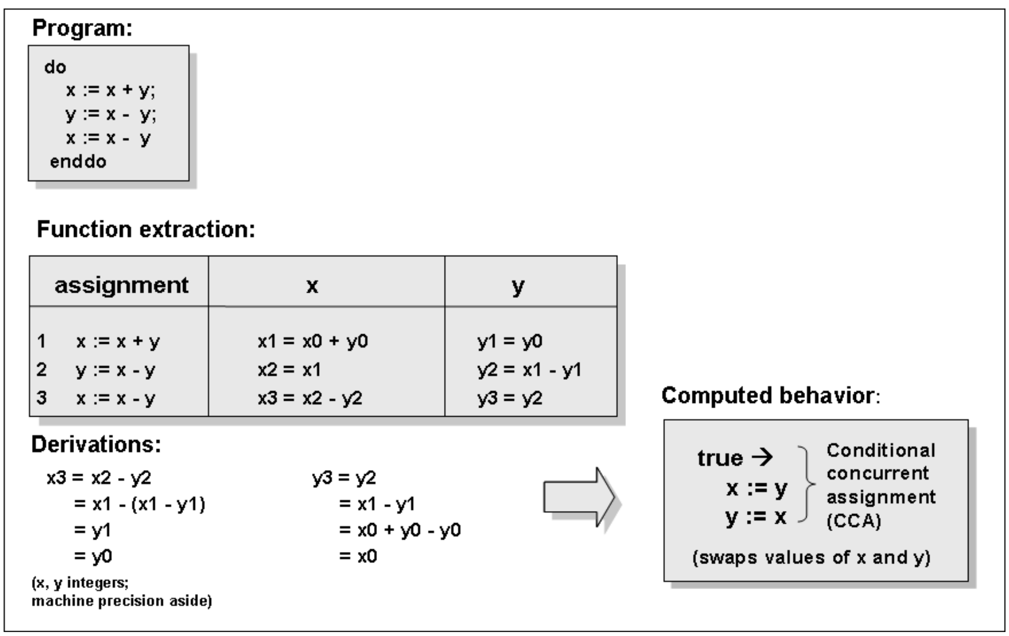

Figure 2: A miniature example of behavior computation

5. An example of malware obfuscation removal and behavior computation

Figure 3 depicts the first two screenshots from IDA Pro of a nine-part display of a malware program containing about 340 lines of Intel assembly language. Only the first two screenshots are shown to save space. The others exhibit similar complexity. The malware has been intentionally obfuscated by a tool that added complex, spaghetti-logic control flow as shown by the many red arrows on the left (the IDA Pro displays do not show all of this obfuscation). In addition, no-op blocks of code that have no functional effect have been inserted, all of which makes analysis very difficult. However, as observed earlier, any obfuscation by an intruder must not perturb the intended functional effect of the malware, or risk defeating its purpose. The FX/MC system is designed to eliminate such obfuscation and compute the behavior of the remaining functional code of the malware.

Figure 4 depicts the input malware program of Figure 3 after transformation to structured form and elimination of control flow obfuscation. To save space, only the first two parts of a four-part display are shown. The others simply continue the sequential logic. The constructive proof of the Structure Theorem and other mathematics was employed to create a function-equivalent version of the program expressed in an algebraic structure of nested control structures.

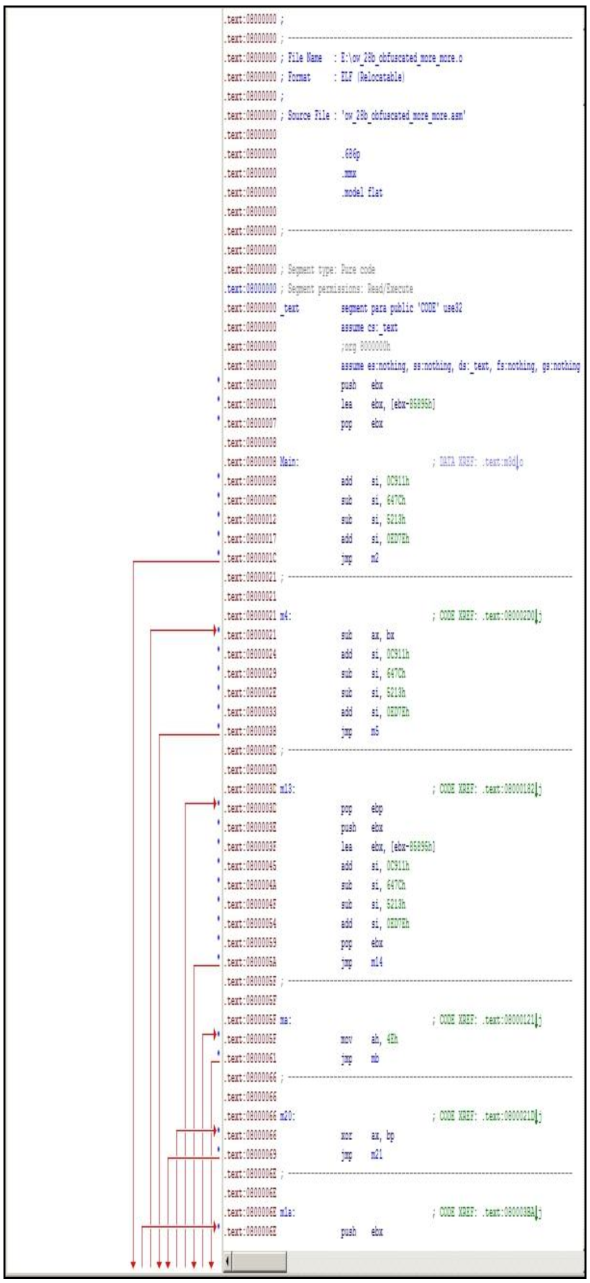

Figure 3. First two screenshots of an obfuscated malware program

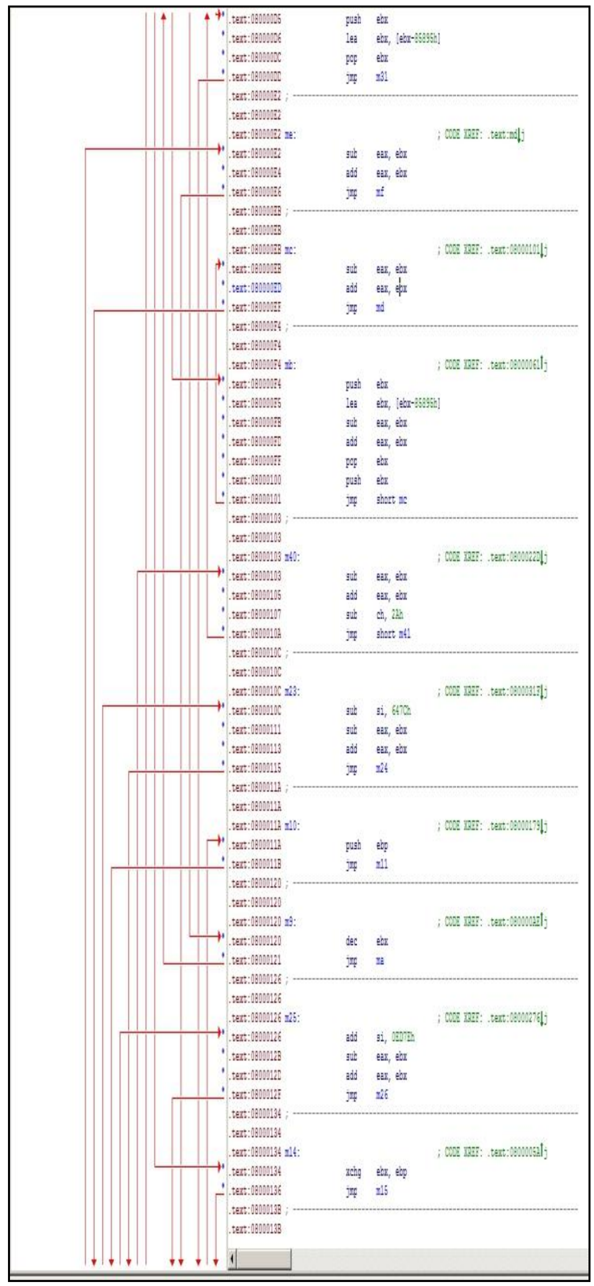

Figure 3 continued.

In this case, the structuring transformation reveals that, despite the addition of so much control flow obfuscation by the intruder, the program is in reality a simple sequence structure with no branching or looping logic present. Jump statements are left in the program for traceability, but have no effect on control flow and can be regarded as comments. The program is smaller with control flow obfuscation removed. The elimination of arbitrary control flow jumps seen here is an intrinsic property of the structuring mathematics that works no matter what particular configuration of spaghetti logic may be present. No special cases or heuristics are employed in this process.

The simple control flow of this structured version of the malware, produced in seconds at machine speeds, can now be read and understood by analysts, a virtually impossible task for the initial spaghetti-logic version of Figure 3. The structuring process helps reduce the effectiveness of this type of control flow obfuscation as a weapon for intruders.

While the logic of the malware program is now understandable, it still contains embedded no-op blocks of code (code with no functional effect) that can complicate the analysis process and must be eliminated. The next step is to compute the behavior of the structured program of Figure 4.

```
FLOW
    // Reference Count: 0
    top:
        push ebx
        lea EBX, [EBX-546965]
        pop ebx
        add SI, 51473
        sub SI, 25724
        sub SI, 21011
        add SI, 60798
        jmp 0x000003F2
        jmp 0x000002BD
        push ebx
        lea EBX, [1265]
        push ebp
        pop ebx
        lea EBP, [ebp+ebx*8+3803040]
        pop ebp
        xchg EBX, EBP
        jmp 0x00000021
        sub AX, BX
        add SI, 51473
        sub SI, 25724
        sub SI, 21011
        add SI, 60798
        jmp 0x0000024D
        add AX, BX
        xor AX, 0x0002
        xor AX, 0x0002
        xor AX, BP
        xor AX, BP
        jmp 0x000003C4
        jmp 0x0000015E
        sub AX, BX
        add AX, BX
        sub EAX, EBX
        add EAX, EBX
        jmp 0x000000A5
        inc ebx
        push ebx
        lea EBX, [EBX-546965]
        pop ebx
        jmp 0x00000120
        dec ebx
        jmp 0x0000005F
        mov AH, 78
        jmp 0x000000F4
        push ebx
        lea EBX, [EBX-546965]
        sub EAX, EBX
        add EAX, EBX
        pop ebx
        push ebx
        jmp 0x000000EB
        sub EAX, EBX
        add EAX, EBX
```

Figure 4. Start of the program after structuring and eliminating control flow obfuscation

```
        jmp 0x000003D9
        jmp 0x000000E2
        sub EAX, EBX
        add EAX, EBX
        jmp 0x0000016D
        lea EBX, [1265]
        sub AX, BX
        add AX, BX
        jmp 0x0000011A
        push ebp
        jmp 0x00000329
        pop ebx
        add SI, 51473
        sub SI, 25724
        sub SI, 21011
        add SI, 60798
        jmp 0x0000017B
        lea EBP, [ebp+ebx*8+3803040]
        jmp 0x0000003D
        pop ebp
        push ebx
        lea EBX, [EBX-546965]
        add SI, 51473
        sub SI, 25724
        sub SI, 21011
        add SI, 60798
        pop ebx
        jmp 0x00000134
        xchg EBX, EBP
        jmp 0x00000144
        sub EAX, EBX
        add EAX, EBX
        jmp 0x000003F7
        jmp 0x00000194
        jmp 0x000000A0
        jmp 0x000003B2
        btc EAX, 74
        btc EAX, 74
        jmp 0x0000006E
        push ebx
        lea EBX, [EBX-546965]
        pop ebx
        jmp 0x00000301
        jmp 0x000003D4
        jmp 0x00000290
        xor AX, 0x0002
        xor AX, 0x0002
        jmp 0x000001B9
        lea DX, [1020]
        sub AX, BX
        add AX, BX
        sub AX, 0x0001
        add AX, 0x0001
        jmp 0x0000020C
        sub AX, 0x0001
        xor AX, BP
```

Figure 4 continued.

Behavior computation, which ultimately produces the net functional effect of the program, traverses the control structures in a stepwise process of function composition. If an intermediate composition produces a state seen previously, the intervening code is a no-op block and can be eliminated. This process results in the display of Figure 5.

```
mov AH, 78
lea DX, [1044]
int 33
mov AH, 60
mov DX, 0x009E
int 33
mov BH, 22
add BH, 42
xchg bx, ax
lea DX, [8]
mov CX, 0x0410
int 33
(ret)
label = exit
```

Figure 5. Malware program with embedded no-op blocks eliminated

With no-op blocks removed, the 340-line malware program reduces to just 14 lines of code, a better than 20:1 reduction. It turns out that nearly all of the code of Figure 4 had no functional effect at all, and was present solely to make analysis more difficult. Compare these 14 lines of functional code to Figure 3 which depicts the original obfuscated version of the program. The reduction in size and complexity of malware illustrated here can be easily expressed in metrics that provide objective measures of system performance.

In determining the purely functional instructions in the malware program, the FX/MC system computes their net behavior. The results of the computation are depicted in Figure 6. It turns out that this small malware program exhibits four possible cases of behavior, three of which result from programming errors that produce nothing more than incidental effects. The first of these is shown in the Figure, the others are similar. The fourth case in the figure, however, reveals the full malicious capabilities of the malware.

The cases of behavior in Figure 6 each represent a conditional concurrent assignment (CCA). If the condition on a case is true, that case defines the behavior that the program will produce. The cases are disjoint, so only one case of behavior will occur on a given execution of the program.

As shown, the concurrent assignments can involve updates to registers, memory, the file system, and flags. All of the assignments in these categories occur at once, essentially a vector assignment from right-hand to left-hand sides of the ":=" assignment operator. The cases represent an as-built specification of the malware program.

Consider the behavior defined by Case 1. The condition contains two predicates highlighted in italics, namely, create_file_failed and write_file_failed, both of which take arguments involving file names and attributes. It is clear that the malware is attempting to create a file and write it. However, because a case of behavior only occurs if its condition is true, this case will only occur if both the create and the write have failed. As a result, the behavior produced by this failure case involves only incidental effects that are not shown in the Figure. Case 1 represents a programming error, a mistake in the malware that produces no malicious effect at all.

Cases 2 and 3, not shown, exhibit similar outcomes. In these cases either the create or the write fails, and the result is similar: incidental behavior is produced for registers, memory, and flags, an empty file is created in case 2, and bytes are written to an unintended file in Case 3. In either case, the malware again fails to achieve the desired effect. These are error cases as well, revealing more coding mistakes.

In Case 4 both the create_file_succeeded and write_file_succeeded predicates are true. The File System is updated with a file starting at byte 0 and of size 39 bytes (shown in italics). This represents the location of the malware itself, which is exactly 39 bytes long. The computed behavior reveals that malware carries out a self-replication by writing itself into the File System of the host machine.

6. Future research

A need exists to provide better tools for malware analysis. The functional semantics of malware is a resource available for this task. Automated behavior computation taps this resource in a new approach to the problem. FX can provide analysts with knowledge of malware structure and behavior that is not currently available and can be used in a variety of ways:

- Understand the function of malware.
- Gain insight into how malware spreads.
- Reveal vulnerabilities and attack strategies.
- Evaluate intruder skill levels.
- Compare malware based on computed behavior.
- Develop defenses and countermeasures.

Beyond malware analysis, FX technology can be applied to other areas, including software development and verification [3,7], embedded system validation [2], software testing [5,6], analysis of security attributes [15,16], and malware detection [10]. Controlled experiments have shown significant improvements in programmer productivity and program quality for small programs when computed behavior is available [4]. As the build-out of FX technology continues and experience with behavior computation accumulates, additional research opportunities and application areas will emerge.

CASE 1 (first of three cases resulting from programming errors, the other two are not shown)

Condition:

```
create_file_failed(
            file_name_addr = 158,
            file_attribute = (word at (40 + (dword at (4 + ESP)))))
and
write_file_failed(
 file_handle = create_file_error_code(
           file_name_addr = 158,
           file_attribute = (word at (40 + (dword at (4 + ESP))))),
buffer_to_write = 0,
num_bytes_to_write = 39)
```

Registers, Memory, File System, Flags: *(Incidental behavior not shown)*

CASE 4 (successful self-replication case)

Condition:

```
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=39)
```

Registers, Memory, Flags:

(Incidental behavior not shown)

File System:

```
FILES :=

create_file_and_truncate(
            file_name_addr = 158,
            file_attribute = (word at (40 + (dword at (4 + ESP)))))
and
write_file(
            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 = 39)
```

Figure 6. Computed malware behavior

A key objective for future research is comparison of behavior computation with other methods for malware analysis in controlled experiments. Research is currently underway to evaluate computed behavior as a means to augment or replace certain forms of software testing for embedded systems. Another area of future research is application of computed behavior to functional understanding and documentation of legacy software.

7. References

[1] Allison, L., A Practical Introduction to Denotational Semantics, Cambridge Computer Science Texts 23, Cambridge University Press, 1986.

[2] Bartholomew, R., L. Burns, T. Daly, R. Linger, and S. Prowell, "Function Extraction: Automated Behavior Computation for Aerospace Software Verification and Certification," Proceedings of 2007 AIAA Aerospace Conference, Monterey, CA, May, 2007, Vol. 3, pp.2145-2153.

[3] Burns, L. and T. Daly, "FXplorer: Exploration of Computed Software Behavior: A New Approach to Understanding and Verification," Proceedings of Hawaii International Conference on System Sciences (HICSS-42), IEEE Computer Society Press, Los Alimitos, CA, 2009.

[4] Collins, R., G. Walton, A. Hevner, and R. Linger, The CERT Function Extraction Experiment: Quantifying FX Impact on Software Comprehension and Verification, CMU/SEI-2005-TN-047, Software Engineering Institute, Carnegie Mellon University, 2005.

[5] Linger, R. and T. Daly, "Advanced Technology for Test and Evaluation of Embedded Systems," CERT 2009 Research Report (R. Linger, Ed.), Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2010.

[6] Linger, R., . Pleszkoch, and R. Hevner, "Introducing Function Extraction into Software Testing," The Data Base for Advances in Information Systems: Special Issue on Software Systems Testing, ACM SIGMIS, New York, NY, 2008.

[7] Linger, R., M. Pleszkoch, L. Burns, A. Hevner, and G. Walton, "Next-Generation Software Engineering: Function Extraction for Computation of Software Behavior," Proceedings of Hawaii International Conference on System Sciences (HICSS-40), Hawaii, IEEE Computer Society Press, Los Alamitos, CA, 2007.

[8] Linger, R., H. Mills, and B. Witt, Structured Programming: Theory and Practice, Addison-Wesley, Reading, MS, 1979.

[9] Mills, H. and R. Linger, "Cleanroom Software Engineering." *Encyclopedia of Software Engineering, 2nd ed.* (J. Marciniak, ed.). John Wiley & Sons, New York, NY, 2002.

[10] Pleszkoch, M. and R. Linger, "Improving Network System Security with Function Extraction Technology for Automated Calculation of Program Behavior." Proceedings of Hawaii International Conference on System Sciences (HICSS-37). Hawaii, IEEE Computer Society Press, Los Alimitos, CA, 2004.

[11] Prowell, S., C. Trammell, R. Linger, and J. Poore, *Cleanroom Software Engineering: Technology and Practice,* Addison Wesley, Reading, MA, 1999.

[12] Prowell, S., M. Pleszkoch, and C. Cohen, "Applying Function Extraction (FX) Techniques to Reverse Engineer Virtual Machines," CERT 2009 Research Report (R. Linger, Ed.), Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2010.

[13] Sayre, K., M. Pleszkoch, T. Daly, R. Linger, and S. Prowell, "Function Extraction for Malicious Code Analysis," CERT 2009 Research Report (R. Linger, Ed.), Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2010.

[14] Smullyan, R. Recursion Theory for Metamathematics, Oxford Logic Guides 22, Oxford University Press, 1993.

[15] Walton, G. and R. Linger, "Security Requirements as Functional Behaviors for System Analysis," Proceedings of 9th Annual Security Conference, April 7-8, 2010, Los Vegas, NV.

[16] Walton, G., T. Longstaff, and R. Linger, Technology Foundations for Computational Evaluation of Security Attributes, Technical Report CMU/SEI-2006-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2006.