Notes on Function Extraction Technology

Mark Pleszkoch, Tim Daly, Rick Linger, Kirk Sayre

1. Motivation and Strategy

Determining the full functional behavior of software is a difficult and resource-intensive task. The problem is compounded for malicious software, where analysis may have been rendered difficult or even impossible by obfuscation techniques. More broadly, much software today is out of intellectual control, in that no one knows for sure what it does in all circumstances of use. Unknown behavior can contain defects and vulnerabilities that compromise functionality and security.

The FX project utilizes the semantics of programming languages in a new way, to compute net effects of programs based on the composed semantics of their instructions. Today programmers mentally compose the effects of instructions they read or write to determine if they do what is intended. As programs grow in size this task can quickly spin out of intellectual control. FX automates this analysis to augment human performance .

The FX project is developing foundations for computing the behavior of software with mathematical precision to the maximum extent possible. We apply solid mathematical foundations to maximize capabilities up to theoretical constraints on behavior computation. At the same time, the project seeks to avoid other limitations. For example, syntax-based methods depend on predefined signatures and can be thwarted by obfuscation, and execution-based methods can do no more than sample the massive number and variety of possible executions that programs can produce.

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 non-procedural functional form. In brief, these transformations are defined by the Correctness Theorem [4], and are localized to individual single-entry, single-exit control structures nested and sequenced in an algebraic structure. 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 [4] defines a method for this transformation. 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, etc. Finally, initial computed behavior typically exhibits opportunities for simplification and abstraction. Taken together, these requirements prescribe a necessary five-part 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 understand and analysis.

2. Theoretical Basis

Function Extraction takes as input a program 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.

Function Extraction is rooted in denotational semantics as defined by Dana Scott and Christopher Strachey [1,2] and applied to computing by Harlan Mills [3,4]. 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 of programming languages, which instead works by defining a concrete model of computation and then mapping programs into that computational model. Function Extraction 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 [5,6] and then automating the extraction of the semantics from actual input programs. Thus, Function Extraction could be viewed as "computational denotational semantics."

Mills' functional verification uses the particular domain of set-theoretic functions (with set-theoretic relations for non-deterministic 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 non-procedural functional form. This theorem is discussed in more detail below.

Functional verification as defined by the Correctness Theorem provides a practical, scalable 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 their 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 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 [7], which forms the key basis for determining the true control flow (the H-Chart algorithm) of input spaghetti logic prior to structuring.

3. Computability

Rice's theorem states that any non-trivial 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, by taking a particular example, this can be seen not to be the case at all. Consider two programs, P1 and P2. Program P1 takes an integer N as input, and always returns 0. Program 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. Program P2 returns 0 if no counterexample is found and returns 1 otherwise. It is an open problem in present-day mathematics as to whether programs 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, we 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 can't 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 non-iterating behavior expression. Processing of loops is an important element of FX technology and is discussed in more detail in the FX Development section below.

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.

We have seen that 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 is the run-time performance of such a translation process. 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 don't 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, described below, to translate procedural input into procedure-free output. 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 blowup in size and thus complexity. To avoid this, a common subexpression factoring and abbreviation technique is being added to the FX system. This abbreviation technique will completely solve the problem, and will permit 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 multiply would be expressed, say, in terms of vector operations and dot products. Such abstractions are easily expressed in Semantic Reduction Theorems (SRTs, discussed below), and can be defined once and for all. For example, the FX system contains a library of SRTs for finite arithmetic that would require modification only if the Intel processor architecture were changed.

The objective of initial behavior extraction is mathematical correctness in transforming from procedural logic to non-procedural functional form. After initial extraction, which (after the aforementioned abbreviation technique is applied) is linear in the number of instructions, symbolic behavior expressions still must be simplified as much as possible for best use by both human users and follow-on automated analysis. These post-extraction simplification steps will be discussed below. 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. 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 useful for providing 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.

4. 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 means that we are assuming 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, so that 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 significantly more complicated than the smallest equivalent behavior expression. However, it is always possible for FX to provide a symbolic behavior expression that is 100% 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 below.

5. Problem Classes and Boundaries

- **Analysis and Simplification**

As mentioned, 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 one that 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 above, 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. Thus, 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, they can turn off the dirty stack simplification behavior and reanalyze the program.

- **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 extraction 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 an exponential blowup 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 that 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 the human user can frequently determine whether aliasing is present and rerun the FX system with the appropriate options to obtain a better expression for the program's behavior.

6. What FX Can Do

As noted above, 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 most if not all FX applications.

It is important not to define the FX system solely by limitations present from 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, the SRTs can encapsulate the analysis skills 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 described below, automated behavior computation can be applied to many tasks across 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 FX/MC system under development for AFRL 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. Phase I of the system has been delivered, and Phase II is scheduled for delivery in September. Millions of lines of code were processed in testing the deobfuscation capabilities of the Phase I 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 [16].

7. Extant Work

We are not aware of any other work in software behavior computation. In terms of syntax-based methods, many tools exist for looking for particular strings and operations in code. These tools are useful in debugging and analysis at the procedural level. In terms of execution-based methods, symbolic execution in various forms has been applied to understanding what a program does on generalized input variables. Symbolic execution differs from FX in that it traces through a single execution path, rather than computing the full behavior of a program. In particular, symbolic execution of loops requires the user to specify the number of times loops should be symbolically executed.

Related software tool efforts include the following. None of these tools compute program behavior.

IDA Pro (commercial)

IDA Pro is the leading machine code disassembly and exploration tool used by malware analysts. One of its strengths is its extensive interactive capability under the direct control of the user. It is a proprietary tool, and several of its key algorithms are based on hidden heuristics.

Rose (LLNL)

Rose is an open-source platform for machine code and higher-level language transformation and processing. It has the capability to read input programs into an internal abstract syntax tree (AST) format, and to output the abstract syntax tree format back to executable or compilable files.

BitBlaze (Berkeley)

BitBlaze is a machine code analysis platform that combines static and dynamic analysis. The emphasis of BitBlaze is the detection and analysis of malicious code. Parts of BitBlaze are open source.

CodeSurfer (commercial)

CodeSurfer is a machine code browsing tool that incorporates leading-edge technology in pointer and indirect function call analysis. The GrammaTech company offers a wide variety of C/C++ browsing tools.

Related Literature

Symbolic Execution

Symbolic execution [14] was an approach originated in the 1970’s to understanding what a program does on generalized input expressed as symbolic variables. It differs from FX in that symbolic execution traced through a single execution path, rather than computing its behavior as a static artifact. In particular, symbolic execution of loops required the user to specify the number of times that the loop should be symbolically executed.

Abstract Interpretation

Abstract interpretation [15] is an application of denotational semantics where a scaled-down, computationally tractable semantic domain is used to rigorously approximate a fuller, mathematically defined but computationally intractable semantic domain. In cases where the scaled-down domain of abstract interpretation is still suitable for an application, the extraction algorithms of abstract interpretation are generally more efficient than FX. However, if it turns out that the scaled-down domain is not suitable, the abstract interpretation algorithms typically return no information about an input program.

Model Checking

Model checking is fundamentally different from the FX system. Whereas FX works directly from source code, model checking works on a user-specified abstract automata, leaving the user with the responsibility for guaranteeing conformance of the source code to the automata. In addition, model checking typically focuses on answering specific questions about software, and models are tailored to those questions, whereas FX focuses on full behavioral analysis of software which can then be used to answer specific questions .

8. FX Development

- **Key Concepts:**

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 represent behavior from individual instructions up to control structures up 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.

Function Equations

The Correctness Theorem defines transformations from procedural logic to non-procedural functional forms as follows (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 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.

Trace Tables

Trace tables contain rows for instructions and columns for conditions and variables assigned new values, with cells recording results of successive compositions. Trace tables represent the implementation of the function equations of the Correctness Theorem.

Meta-Programming Language (MPL)

The syntax that FX uses for behavior expressions is called MPL. The language incorporates concepts from type theory and lambda calculus [5,6]. MPL provides a Turing-complete symbolic language for behavior expressions. The specific operations used in MPL are defined in an external MPL file, to enable extension of FX to new instructions and new input languages without requiring recoding of the internal Java code of the system.

Semantic Reduction Theorems (SRTs)

One method for simplification of MPL expressions is the use of rewriting rules based on theorems regarding the equivalence of MPL operation applications. 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.

True Control Flow (H-Chart) Computation

The H-Chart algorithm employs a frontier propagation strategy to create 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. For self-modifying code, a different partition between code and data state can theoretically be used, but this does not result in a useable result.

Loop Behavior Computation

The unique loop behavior computation component of FX is designed to analyze loops in input programs, to determine whenever possible whether the loop terminates, and if so compute its end-to-end functional behavior. Loop computation is implemented in conjunction with the SRT portion of the MPL subsystem, so that new loop recognizers can easily be added. These recognizers represent entire classes of loops and are broadly applicable.

- **Principal Operations in Behavior Computation:**

1. Input program transformation

   The first step in behavior computation is transformation of instructions in the input program into functional form expressed in CCAs. This transformation is supported by a library of functional definitions of Intel assembly language instruction semantics that is being built out. Instruction frequencies have a long flat tail. Semantics are being defined in frequency order.

2. True control flow determination

   The H-chart algorithm employs a frontier propagation method for determining the true control flow of spaghetti logic in the input program. The resulting flowgraph is closed under semantic reachability.

3. Transformation to structured form

   This step transforms the true control flow of an input program into a function-equivalent algebraic structure expressed in single-entry, single-exit sequence, ifthenelse, and whiledo prime control structures. The algorithm is based on the constructive proof of the Structure Theorem.

4. Behavior computation

   Stepwise computation and propagation of the procedure-free function of each prime control structure in an algebraic structure is accomplished through function composition. The starting point is the Correctness Theorem mapping from prime control structures into functional form, with additional transformation of the recursive form for whiledo structures. The computed behavior is represented in terms of CCAs.

5. Behavior reduction and abstraction

   This step applies Semantic Reduction Theorems (SRTs) to initial computed behavior to reduce it to simpler form. SRTs are general and widely applicable micro-theorems, and libraries of SRTs which are being built out. Referentially transparent factoring and hierarchicalization of computed behavior is being added to the system. In addition, an opportunity exists for user naming of factored expressions which represent common data processing and subject-matter operations.

9. IP Generation

A significant amount of IP has been generated in FX research and development, including the following:

- Functional semantics definitions for Intel assembly language instructions.

- Behavior expression language with a single statement (CCA) that scales from individual instructions up to control structures up to entire programs.

- Deterministic frontier propagation algorithm for determining true control flow.

- Structuring engine for Intel assembly language for gaining intellectual control over spaghetti logic code.

- Unique new method for loop behavior computation.

- Method for reduction and abstraction of computed behavior using Semantic Reduction Theorems.

- Method for keeping computed behavior expressions linear in the size of the input program using factoring, hierarchicalization, and encapsulated behavior tables with context-sensitive propagation.

10. Potential FX Applications

Malicious code detection

All cases of behavior, both legitimate and potentially malicious, are computed by FX for analysis. Even malicious code that is scattered throughout a host program is aggregated and coalesced into cases of behavior.

Malware classification

In machine learning approaches to malware classification, for example, Concordia [12], FX can help categorize code fragments into classes with equivalent behavior to improve the performance of algorithms such as K-means clustering.

Code deobfuscation

FX can eliminate control flow obfuscation and no-op blocks of code. This is the principal functionality of the FX/MC system being developed for AFRL.

Offensive code obfuscation

Offensive obfuscation is limited by methods to ensure that core functionality remains intact. FX can compute core functionality in heavily obfuscated code to permit confident use of more complex and effective methods of obfuscation.

Dynamic analysis for malware

FX can help detect one-off malicious content in streaming executables in network systems through whitelisting methods and fast, concurrent operation. A collaboration is in the planning stage with APL and ORNL to demonstrate this capability.

Software development

FX can provide feedback on the functionality of code as it is being written, and produce as-built specifications of code for verification, maintenance, and evolution [8].

Software verification

FX produces the as-built behavior of a program which can be verified against specifications if they exist and are authoritative, or otherwise verified by inspection against user requirements [9].

Software testing

FX can augment or substitute for functional testing at the unit and subsystem level [10].

Reengineering legacy software

FX can be used to reverse engineer legacy code to provide a basis for understanding and modernization. A collaboration is planned with MITRE to demonstrate this capability for legacy IRS software that has presented maintenance and modernization problems for decades.

Security properties

FX can help reveal vulnerabilities and security properties of software [11].

Research tool

FX provides an instrument for researchers to investigate unforeseen properties and uses of computed behavior.

Compiler optimization

FX can detect behavior of known functions and replace their implementations with optimized versions, and also eliminate no-op code (in testing, FX/MC is finding no-op blocks in code generated by the Visual C++ compiler and TCC.)

Secure coding analysis

With reference to the work of Robert Seacord [13], an opportunity exists to detect the behavior of vulnerabilities such as buffer overflows and replace them with functionally equivalent secure code.

Scientific subroutine validation

Numerous scientific subroutine libraries (GMP, MPRL, BLAS, LAPACK, ATLAS, etc.) are used worldwide. These subroutines have exact specifications for their semantics. FX could be used to extract the actual semantics of an implementation on a given platform to validate that the code computes the specified values.

Debugging

A failing application can provide state information about an error, but no way to reproduce that state. Computed behavior can help classify the failing state and find the code that produced it.

Precise, current documentation

Computed behavior can help provide precise documentation for confident use of acquired software, and provide a means to catalog software based on documented behavior requirements for acquisition and reuse.

Other platforms

Malicious content is affecting many networked devices, including cell phones, PDAs, etc. FX technology can be applied to these platforms.

11. References

[1] Scott, Dana S., "Logic and Programming Languages", Comm. of the ACM, Sept. 1977, v. 20, n. 9, pp. 634-641.

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

[3] Mills, Harlan D., "The New Math of Computer Programming", Comm. of the ACM, Jan. 1975, v. 18, n. 1, pp. 43-48.

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

[5] Barendregt, Hendrik P., "The Lambda Calculus: Its Syntax and Semantics," *Studies in Logic and the Foundations of Mathematics,* Volume 103, North-Holland, 1985.

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

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

[8] Burns, L. and Daly, T., "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.

[9] Bartholomew, R., Burns, L., Daly, T., Linger, R., and Prowell, S., "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.

[10] Linger, R., Pleszkoch, M., and Hevner, R., "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.

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

[12] Burn, L. and Daly, T., Concurrent Architecture for Automated Malware Classification, *Proceedings of Hawaii International Conference on System Sciences (HICSS-43),* IEEE Computer Society Press, Los Alimitos, CA, 2010.

[13] Seacord, R. Secure Coding in C and C++. Addison-Wesley, 2005

[14] King, James C., "Symbolic execution and program testing", Comm. of the ACM, v. 19, n. 7, 1976, pp. 385-394.

[15] Cousot, P. and Cousot, R., "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", by 4th Conference on the Principles of Programming Languages (POPL), Los Angeles, CA, 1977, pp. 238-252.

[16] Pleszkoch, M., Prowell, S., Cohen, C., and Havrilla, J., "Applying Function Extraction (FX) Techniques to Reverse Engineer Virtual Machines," *Cert Research Annual Report 2009,* (R. Linger, ed.) Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2010.