Function Extraction, A New Paradigm for Producing Secure Code

RICHARD LINGER*, Assurance Labs, Inc., USA
MARK PLESZKOCH*, Institute for Defense Analyses, USA
ALI MILI*, New Jersey Institute of Technology, USA
JOHN MCHUGH*, University of South Florida, USA
WIDED GHARDALLOU*, ENISO, University of Sousse, Tunisia
JACK MCGAUGHEY*, University of Waterloo, Canada

Function Extraction (FX) is a new and evolving paradigm for the production of secure computer codes. It is, in effect, the inverse of formal verification as it analyzes code to produce a mathematical specification of its behavior. This has the potential for identifying unwanted or unexpected behaviors and has the potential for analyzing unknown or "found" code artifacts such as malware. The effort is enabled by recent developments in loop analysis that allow invariant relations to be developed for loop bodies enabling the loop function to be discovered. The paper defines program behavior as a mathematical description of the effects of program execution on the environment in which the program runs and continues with a discussion of its current status. As FX is an evolving paradigm, areas in which work remains to be done are discussed and examples of the results of two prototype analyzers are given. The paper concludes with a discussion of the path forward and the work that remains to be done.

ACM Reference Format:
Richard Linger, Mark Pleszkoch, Ali Mili, John McHugh, Wided Ghardallou, and Jack McGaughey. 2024. Function Extraction, A New Paradigm for Producing Secure Code. In . ACM, New York, NY, USA, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

JUSTIFICATION

Function Extraction (FX) is a code analysis technique that provides an as–built specification from program code. In a sense, it is an inverse of program verification, but it goes beyond that. Verification can show that a program implements a specification, but is likely to ignore the presence of seemingly irrelevant unspecified behavior. Code analysis has been performed for many years and current static analysis tools routinely perform analyses that required interactive theorem proving a few decades ago. FX, in its current form, is based on recent breakthroughs in loop analysis[10]. The authors are trying to get the technology to the point where it will be useful to coders in the security area as well as those involved in other areas requiring code with known behaviors and a high degree of dependability. As can be seen in the paper the road from theory to practice is not smooth and our experiences may be useful to practitioners and theoreticians alike.

This paper is a Regular Submission. NSPW is known as a forum in which ideas such as ours can be nudged towards maturity. At this stage, we would benefit from the type of careful scrutiny and insightful discussion for which NSPW is well known. In addition, we think that, if the community represented by NSPW finds the approach to have merit,

*All authors contributed equally to this research.

Manuscript submitted to ACM

the community would benefit from the opportunity for involvement in our efforts and we hope that we will have the opportunity to move forward with encouragement from the NSPW community.

1 INTRODUCTION

It is often asserted that testing can show the presence of errors but not their absence and that only program verification can produce error free code. This is not quite correct since what most verification produces is a program that, in an ideal implementation world, behaves in a manner that conforms to the specification against which it was verified. It is possible for a verified program to exhibit all the specified behaviors and yet have other, undesired behaviors that are nonetheless consistent with the specification. An example of this might be a program that has been verified against a specification that states that an output array must be in sorted order. The programmer might think this has guaranteed that the program sorts the input array, yet find that the program merely copies the first element of the array to all positions in the output array. Whether these behaviors can in general be considered to be errors is a matter for philosophical debate, but they can be malicious and can manifest in ways that compromise the security of the system using the software.

In this paper, we introduce a new paradigm for program characterization, function extraction (FX). Function extraction is a technique for analyzing program code and extracting from it an *as-built* specification, a mathematical function that captures the full functional behavior of the program. It is, in a sense, the inverse of conventional program verification. In function extraction, the program, or an abstract representation derived from it, is decomposed into conditional (branching) operations and linear (branch free) sequences of instructions. The linear sequences are reduced to sets of concurrent assignments that express the values of program variables at the end of the sequence in terms of their values at the start of the sequence. The branching operations provide logical conditions that are associated with the concurrent assignments for the linear sequences following the branches, resulting in sets of conditional concurrent assignments for each linear segment of the program. In our analyzers, we express the semantics of the operations using the Wolfram language and perform the subsequent analysis using Mathematica[14]. By subsuming the whole of mathematics as our behavior expression language, we take a fundamentally different approach than most other static analysis techniques, such as flow-sensitive analysis and abstact interpretation, that start out by pre-defining a limited set of behavior expressions, together with a "bottom" behavior element that is used when the program exhibits behavior outside of that limited set.

The general approach dates to the dawn of the structured programming era and is well described in works such as [8]. Using the segment behaviors and casting the results in terms of conditional equalities derived from the conditional assignments, it is trivial to develop mathematical formulations for the behavior of loop free programs without subroutines by composing the behaviors of the segments with the branching conditions following the control flow through the program.

Loops complicate the extraction process and it is only recently that techniques have been developed that can relate the conditional behavior of a loop body and its exit condition to a mathematically defined behavior [10]. Traditionally, loop analysis has involved unrolling a loop for an iteration or two and used this to approximate the loop behavior. Mili's approach involves discovering invariant relations manifest in the loop body and using these to discover the loop function. We follow that general approach and discuss it in more detail in section 4.2, below.

In principal, subroutines can be handled by simply expanding their bodies at the call site, substituting call arguments for formal parameters and dealing with naming conflicts between variables in the calling and called environment, but

we choose to characterize the behavior of subprograms[1] separately, as will be discussed further below, and incorporate the behavior of a called subprogram into the analysis at the call site. Among other things, this allows us to use, with some risk, the behaviors of subprograms for which behaviors but not source code are available. It also allows us to examine behaviors on a smaller scale where the extracted functional representation is more likely to be comprehensible and easier to compare to the intended behavior. This allows the behavior of complex programs to be composed from the behaviors of their components. This, too, will be discussed further is sections 4.4 and 4.5, below.

Function extraction has numerous roles in the development of secure and dependable code. The ability to extract an as- built specification from the code allows a straightforward comparison between intended and achieved behavior. This may obviate the need to conduct extensive unit testing and the need to develop test cases for "corner" conditions. In many cases, the behavioral analysis can identify incomplete behaviors, for example, regions of the input space for which a computation does not terminate (see figure 1). The functional form captures dependencies in somewhat more detail than the typical covert analysis approach, but it should be possible to use the resulting analysis to conduct a covert channel analysis using *e.g.*, the shared resource matrix[7] or similar tools. At the surface level, unanticipated effects on global variables or the environment may be indications of malicious behavior as may unexpected dependencies on global variables or the environment.

Function extraction can also serve in a forensic role, helping to determine the purpose and nature of malware and other "found" code artifacts. It can be applied at levels ranging from machine code up to high level source code requiring only that the semantics of the individual operations being analyzed be known and expressible in a suitable mathematical form. Function extraction, unlike formal verification techniques including proof-carrying code, does not require planning and effort during the program development phase. (However, in our conclusions we postulate that a certain synergy may result from programmers applying both formal verification and function extraction.)

In the remainder of the paper, we provide

- A discussion of program behavior. What is it? How is it characterized? How does it manifest when programs are run? What are the differences between program and mathematical characterizations? As our systems evolve, we are finding that the FX approach *may* serve as a diagnostic tool for identifying constructs in C that lead to illegal (and thus undefined) programs.
- The current status of our efforts accompanied by examples to illustrate the process. Since the system is a work in progress, this is subject to change but is accurate as of the time of submission.
- Areas requiring specialized or non-obvious approaches. These include loops where the determination of a loop function from the conditional concurrent assignments of the loop body involves both pattern searching and theorem proving activities. Calls are another area where the behavior of a called routine must be instantiated in the calling environment. Calls to library functions can be problematic as, in many cases, we do not have (or do not want to evaluate) the source code of the function. There is an obvious risk if the assumed behavior used to instantiate a call on a library routine does not match the behavior of the called code. Ultimately, we would like to have a combination of validated functionality for libraries accompanied by a provenance record that assures that the library actually used is the one for which the functionality was obtained.
- As a work in progress, function extraction is not complete but we have a plan $\cdots$.

[1]The behavior of a subprogram can be characterized as a four-tuple containing 1. the return value, 2. the effects on reference parameters, 3. the effects on global variables, and 4. the effects on the greater environment via output and other interactions.

2 PROGRAM BEHAVIOR

The behavior of a program can be seen as capturing the relationship between its initial state, e.g. the values assigned to its variables at (or prior to) the start of its execution and its final state, e.g. the values assigned to its variables at (or just prior to) the end of its execution. Although it is adequate for simple (or toy) examples, this view, if taken narrowly, is inadequate and needs to be extended for a number of reasons.

- Most useful programs interact with their environment by reading data from an outside source and writing results to an outside destination. In the case of control systems, the outputs may affect the environment in ways that influence subsequent inputs. At the very least, we need to include input and output interactions in the notion of behavior.
- Some programs do not terminate with a meaningful final state. Again, control systems are a good case in point. Such programs may be forced to terminate or may be designed to stop automatically under certain conditions, but the state of the program at termination is of little interest. The behavior of interest is the history of the program's interactions with its environment.
- Loops are problematic, especially when they capture repetitive behavior in the absence of any explicit indexing mechanism. This will be discussed further in section 4.2, below, however the appropriate behavioral characterization may depend on the intent of the computation.
- We are looking for a mathematical description of the program's behavior, but program operations are often poor shadows of the mathematical operations they seek to emulate. Mathematical *real* numbers often have little in common with their programming analog, *floating point* numbers. Given two distinct real numbers, another real number can always be found between them. This is not always the case for floating point numbers. Computer *integer* types have restricted ranges. Even a computation as seemingly simple as 13! = 6,227,020,800 cannot be performed directly using 32 bit computer arithmetic. When we try to express the behavior of code using a mathematically oriented representation such as Mathematica[14], we are constantly reminded that, while a naive translation may (or may not) capture the intent of the programmer, the observed behavior of the program often differs from its calculated behavior.

Many programs rely on the composition of results from subprograms so it is desirable to create a framework for computing the behaviors of individual subprograms. In general, a calling a subprogram affects the environment of the caller in zero[2] or more of four possible ways.

By returning a value: This value is inserted into the computation at the call point after mapping the initial formal parameter values that appear in the computed behavior back to the arguments provided in the call.

By modifying parameters passed by reference: This may happen implicitly in some languages or explicitly, by passing the address of a variable in the calling environment, in others. Expressions representing the final values of reference parameters, mapped as above, are assigned to the variables whose addresses were passed as arguments at the return point of the call.

By modifying a global variable: In many languages global variables are visible in any subprogram within a compilation and, if properly declared, may reside outside the current compilation unit. Expressions representing the new values of the affected global variables are assigned to them after mapping as defined above.

By interacting with the environment: The behavior must capture these interactions. This implies special handling of language constructs that perform these interactions. Usually, but not always, input and output are

[2]A subprogram whose only effect is to delay the execution of the calling program may have a void return and no side effects.

restricted to a limited set of operations or encapsulated within specific libraries. Inputs obtained by the subprogram are denoted by unique symbols with constant but unknown values. These may appear in any of the expression forms. Output expressions are recorded, indexed by the name of the subprogram and the line number where the output occurred. If the expression was output via a system library routine, e.g. `printf()`, the name of the routine and the positional parameter index are appended to the index. At the call site, the value expressions are mapped and the calling routine and line number prepended to the index before adding the entries to the interaction component of the called routine's behavior.

In general, we describe the behavior of a subprogram in terms of the four-tuple given above. The "values" are typically symbolic expressions couched in terms of literal constants, the initial values of parameters passed to the subprogram at the time the subprogram is called, and the values of global variables referenced by the subprogram, again using the values at the time of invocation.

Note that the "main" program is just another subprogram in many languages although it may be constrained to be of a specific return type and have a specific list of parameters to facilitate its interaction with the operating system that sets it into execution. In Unix based systems, for example, the return value is typically an integer that may be used as an error code, program global variables and reference arguments may not be accessible when the program terminates leaving environmental interactions as the only way to empirically observe the behavior. Nonetheless, we compute and report the entire four-tuple of behavior for all subprograms, including the main program.

Note that we do not include the time a subprogram takes to execute to be a component of its behavior as computed by FX. Although a timing component could theoretically be included as part of the environment, it is unclear whether programming language semantics provide enough information about timing to allow us to make useful statements about subprogram times.

3 CURRENT STATUS

We have two incomplete function extraction systems at the time of writing. The initial system analyzes a small subset of Java using the Abstract Syntax Tree (AST) produced by Javaparser[6]. The second system analyzes the LLVM [9] Intermediate Representation (LLVM IR[5] for C as produced by the Clang[2] front end).

In many ways, an AST is a better form to analyze, but there is no "universal" AST representation. Many compilers use an AST as a first step in representing parsed source code and similar representations are used in many software analysis and transformation tools. The Javaparser is intended to support a variety of applications including source code refactoring and reformatting. It preserves source code features, such as comments, that are not relevant to function extraction and includes them in its AST. Access to and processing of the AST is via a large package of Java code. Many of the AST constructs are Java specific and it is not clear that it would be feasible to process languages other than Java into this specific form. There are programs that produce ASTs for other languages, but their structures are sufficiently different so that leveraging the current code base to process them is problematic.

The Java FX prototype runs Javaparser on the input program in order to generate its Abstract Syntax tree (AST). It applies a breadth first traversal of the tree to create a partial order of the "user-defined" methods. These methods are ordered in a structure called the method table, in such a way that every called method precedes its calling methods. Thus, the behavior of each method is computed before it is needed. FX then iterates through this table in order to calculate the function of each method. The function of a method is calculated by traversing its sub-tree and by transforming each node of the sub-tree into its corresponding Mathematica equation, according to its type (assignment statement,

for/while/do loop, conditional statement, declaration statement, method call, etc). The system of equations that is generated is then sent to the Mathematica kernel which evaluates it and returns the behavior of the method. This behavior is stored in the table of methods where it is used when the calling methods are evaluated.

The LLVM project is intended to provide an intermediate representation that can be used to generate efficient machine code for a wide variety of platforms while serving as a parse target for a wide variety of source languages. In addition to the Clang front end mentioned above, front ends exist for many other languages, including Rust[12].

The LLVM IR is a static single assignment assembly language. We use the form produced by

`clang -O0 -Xclang -disable-O0-optnone -fno-discard-value-names -emit-llvm -c`,

LLVM IR is register based with an arbitrarily large number of registers. The single assignment property means that each register is assigned to only once. In the unoptimized form temporary registers are used exactly once after their initial assignment, simplifying processing. LLVM IR is not directly concerned with memory. Declarations of global / local variables result in (persistent, for the life of the program / function) registers containing pointers to the memory associated with the declared entity. The local registers are assigned pointers that are returned by `alloca` instructions and used with `load` and `store` instructions to move values between temporary registers and memory. All computations are done using temporary registers as operands. We add special persistent registers to represent memory and give us a place to hold values.

The IR code produced by the front end consists of labeled linear sequences of code representing basic blocks in the function's control flow graph (CFG). Each ends with a terminator[5, #terminators] instruction, typically a return, `ret`, branch, `br`, or switch `switch` instruction. Alternations, e.g. `if-then-else`, involve forward branches while iterations, e.g. loops, involve backward branches so we can recover the program structure. Typically, the decision logic for conditional branches is placed in a labeled basic block of its own.

The command line shown above produces a binary representation of the LLVM IR which is disassempled to produce a text file that we process using Snobol 4[4] using Phil Budne's CSNOBOL4[1]. The extractor works in a number of passes using an array of the lines from the text file. The initial passes identify the global symbols used in the program including variables, structures, and defined and declared functions and allow us to determine the evaluation order. The functions are preprocessed in this order and the basic blocks and the CFG implied by the branch instructions is determined. A local symbol table is produced for each function that contains entries for the function, its labels, any global registers (and their value entries) that it references and its local registers, again with the addition of memory surrogates to hold values. Parameter values are given stylized symbolic values, `_%p.#` where the # indicates the position of the parameter in the function's parameter list. Local variables are given unique symbolic values of the form `_%lcuv.nnn` where `nnn` is a globally unique integer of 3 or more digits. These will be overwritten when the variable is initialized and allows the presence of undefined values to be seen in the behavior (typically an error resulting in undefined behavior).

Behaviors are computed in two passes. In the first pass, the behavior of each basic block is reduced to a set of values for the registers assigned during the execution of the block. This is done by assigning to the value component of the result register of each IR operation the appropriate Mathematica expression associated with its semantics and the value components of its operands. This is straightforward except for the `call` instruction which may have side effects as discussed in sections 4.4 and 4.5, below. At the end of this process, we need only retain the registers representing memory values as the temporary registers are all dead at this point. This gives us a set of concurrent assignments for the output values of memory changed by the block in terms of the (default) values at entry to the block.

In the second pass, currently under construction for functions with more than one block, we compose block effects, starting with the `entry:` block and compute the function behavior. This is trivial for forward branches, but loops require additional processing as discussed in section 4.2, below.

4 PROBLEMATIC PROGRAMMING CONSTRUCTS

This section is focused on the LLVM IR based prototype for C. It discusses some of the C constructs for which behavior computation is difficult or impossible. Some of these are inherent in the language while others can be traced to LLVM IR. In each case we describe the construct, giving an example, discuss our current intuition about possible solutions present the outstanding questions that need to be addressed.

There are several examples of even more difficult LLVM IR constructs for which it would be premature to actively investigate until we have solutions to the issues below. For example, dynamic dispatch from object-oriented languages such as a Java or C++, or other dynamically computed control flow constructs.

4.1 Poison Values and Undefined Behavior

In their infinite wisdom, the authors and maintainers of the C (and C++) language include the notion of *undefined* behavior. This allows a compiler / optimizer to exhibit any (or no) behavior if it determines that a code construct falls into this category. The permissible behaviors are variously described in terms of "formatting the hard drive," "causing daemons to fly out your nose," and other colorful idioms[3]. Undefined behaviors arise from out of bounds structure accesses, division by zero, accessing unallocated or deallocated memory, integer underflows, overflows, and wrapping, etc.

For example, the C fragment,

```
int i = 42; while(i !=0) i = i + 1;
```

has an undefined behavior when its functionality is extracted and analyzed by Mathematica because the loop never terminates. If embedded in a program that is compiled without extreme optimization and executed, it usually compiles and executes for a few seconds before the loop terminates having iterated nearly 2^{32} times, wrapping the index from `INT_MAX` to `INT_MIN`, (producing a poison value) and eventually reaching `i == 0`, but this is not a computable behavior.

In many cases, LLVM IR has keywords that indicate the potential for this kind of behavior, and, as the work progresses, we can treat the circumstances under which poison or undefined values can occur as proof obligations. This would require us to ask Mathematica (or some other theorem prover) to show, for example, that the LLVM IR instruction,

```
%add = add nsw i32 %0, 1
```

cannot cause a "signed wrap" as indicated by the `nsw` keyword. This would happen during behavior computation and the prover would have access to information about the current state of the behavior computation. In this particular case, this is equivalent to showing that the value contained in the register `%0` is not equal to `INT_MAX`.

The problem with this is that many programmers do not provide adequate input checking in the belief that no one would input that specific value into the program (or fail to recognize the possibility) and the relatively weak type system of C may leave the conjecture unprovable. In such cases, we have no alternative to identifying the problem, declaring the behavior as undefined and giving the programmer as much help as possible in locating the source and fixing it.

[3] A thoughtful discussion of the issues can be found in the blog posts of John Regehr[11].

4.2 Loops

Loop processing is complex and we will not go into detail here but the breakthrough in loop analysis is our enabling technology and we ask the reader's indulgence in a somewhat technical description. The seminal paper is Mili *et. al.*[10] but the area is evolving rapidly and additional papers exist. We consider a loop, w, of the form `while t do b` where t is the loop condition and b is the loop body. We let $W = [w]$ be the function of the loop. In order to find a loop function, we follow the following process assuming that the loop body is a basic block or sequence of statements:

(1) Reduce the loop body, b, to a mathematical function or relation that gives the final (primed) values of all loop variables in terms of their initial (unprimed) values after one pass through the body. This can be done automatically by tracking the current values of all variables in a symbol table as we analyze the code. For example, the body `begin x = a + 5; y = x + 3; end` will be represented by the function $(a' = a) \wedge (x' = a + 5) \wedge (y' = a + 8)$, reflecting the fact that the value of `y` at the end of the body will be equal to the value of `a` at the beginning of the body plus 8. For concurrent loop bodies, e.g., `parbegin x = a + 5; y = x + 3; parend`, where each assignment statement is an atomic action, we obtain the relation $(a' = a) \wedge (x' = a+5) \wedge (y' = x+3 \vee y' = a+8)$ by taking the union of the functions stemming from all the possible execution orders of the body. In general, this form expresses the loop body as an intersection of relations which we denote as B.
(2) This form satisfies Mili's Theorem 4 and allows us to find invariant relation(s) for the loop from which the loop function W can be inferred. The invariant relation can be either creative, i.e. obtained by pattern-matching against a database of recognizers (where recognizers are made up of code patterns and invariant relation templates), or constructive, i.e. generated directly from the analysis on B. The intersection of R, an invariant relation of w and $\widehat{\widehat{T}}$ the vector representing the loop condition t) is a lower bound of W.
(3) We ask Mathematica to simplify the equations representing the join of all the lower bounds and solve the system of equations for the final (primed) state variables using the initial (unprimed) values as parameters. If W is total and deterministic, then W is necessarily the function of the loop, otherwise, it is the best approximation that we can come up with, given the invariant relations that we were able to identify in the database of recognizers.

We have recognizers for many programs, but recognizer development is an ongoing area of research, the constructive generation of invariants is in its infancy, and we have number of problems. One problem is presenting the results in forms that are familiar to a developer working in a specific domain For example, a control engineer might want to identify proportional, integral, and derivative components of the behavior explicitly while other domains might benefit from different nomenclature for similar mathematical functionality. Solving this requires the involvement of domain experts.

Most of the loop forms reduce to some form of `while` construct, and termination on equality simplifies the analysis. From a software engineering standpoint, termination on equality is undesirable as it can turn a relatively simple error into undefined behavior. Loops that solve equations involving floating point computations usually terminate when an error check shows that the result is "good enough." At this point, we do not have suitable recognizers for such cases. In cases involving searching and sorting, end point behaviors are relatively simple, but we need invariant relations that apply between arbitrary iterations of the loop and are working actively to develop them. We believe that these problems can be solved given sufficient effort, but additional theoretical breakthroughs may be required.

We realize that failure to accomodate the all the major loop based paradigms that appear in programs will limit the utility of function extraction.

4.3 Arrays and Structures

The most serious issue for arrays and structures is characterizing loop code that transforms or reorders the array or structure. This is trivial in cases where each array element is transformed by applying the same function and is tractable when each element has a straightforward relationship to preceding elements[4]. At the present time, we are unable to deal with common operations like sorting effectively. At the beginning of a sort, we know nothing about the relationship among the values. At the end, we know that the array, as a whole satisfies a predicate, say `Sorted[A]` which means, at least, that all the elements in the final array were in the initial array and that some relationship like $\geq$ applies between adjacent elements or their sort keys. Unfortunately, we need an invariant relation that describes the intermediate states of the computation and what it means for the array to become increasingly sorted as the sort progresses. This will be a subject of research for our group in the future.

Our intuition is that this is possible, but will require changing the way we characterize loop bodies, moving from the details of individual operations to properties of the structures. In the absence of this, the applicability of behavior computation in the real world will be limited. Clearly, the prognosis is even more pessimistic for more complicated behavior, such as arbitrary pointers and pointer arithmetic.

The other issues with arrays and structures is technical and tedious, but tractable. LLVM IR accesses elements of arrays and structures using the `getelementptr` instruction. This takes a pointer to a structure and a list of integer indices and returns a pointer to a specific element or sub-structure. For behavior computation, we need to reverse engineer the instruction to get a comprehensible reference to give the user. Clang treats `structs` as arrays with elements of different types and substitutes integer indices for field names. We can enable parse metadata and recover the field names making them available for our reporting. We do not anticipate any difficulties with this approach, but it may prove difficult to devlop comprehensible references for complex, nested structures. The resulting pointer must reference memory within the structure, creating another proof obligation.

4.4 Subroutine and Function Calls

Functions in C have side effects. They can alter the calling environment and interact with the outside world. Function extraction, applied to a function definition, results in a four-tuple as described in section 2, above. One effect of the call is to substitute this expression, lifted to the calling environment, as the result of the `call` instruction. Lifting an expression involves substituting the arguments given at the call site for the parameter references contained in the expression. Similarly, we lift the expressions contained in the reference and global components of the four-tuple to the calling environment and treat them as though the call was accompanied by assignments to the referenced and global variables involved. There is a potential for the program to become undefined during this process since C has strict rules about aliasing of pointers. We will attempt to detect and alert about such situations, but, it may not be possible in all cases.

The output component of the four-tuple is fairly simple. The output expressions can be lifted to the calling environment. The call site can be prepended to the history side of the relations without difficulty, providing record of where output occurs and what is output. Inputs usually occur during calls to library routines discussed below.

[4]We consider the later case as “array like” even when the array is not explicit as in section 5 where Fibonacci numbers appear in the behavior.

4.5 Declared Subprograms and Libraries

Most programs do not contain the defining code for all the subprograms that they use. C and C++ provide the `#include` statement to allow access to predefined and separately compiled libraries. Java uses the `import` statement for the same purpose and Rust uses the `crate` concept. In all cases, we can call a subprogram knowing only its interface and intended behavior without actually examining its source code or computing its behavior. For the purposes of discussion, we will refer to such subprograms as *library* subprograms. From a behavioral standpoint, libraries are a mixed bag. At the least, we need to compute, create, and organize the behavioral four tuples for any library routines that we may call from the code we are analyzing. Moreover, we need to trust these behaviors. It is easy to imagine an adversary planting a back door in a standard library used by many programs[5] with the result that the behavior described in the four-tuple would be supplemented by additional, undescribed behaviors.

In general, we don't want to have to audit and maintain all the code in our system, but this means that 1. The necessary behavior for libraries used has to be available so that call site behavior can be computed for our code and 2. the given behavior has to be complete, accurate, and trustworthy. Given that even microchip level controllers typically rely on libraries for much of their low level functionality, this can represent a substantial risk.

For our initial implementations, we ignore this problem, constructing a local repository of four-tuples for the library routines that we use or expect to use in our code. For this, we rely largely on the `man` descriptions as guides for the expected behavior and the limitations of the code. In the cases where the C and Mathematica behaviors differ for a given function e.g. `sqrt()` in C breaks for negative inputs while `Sqrt[]` in Mathematica returns a complex value, we use conditional expressions in Mathematica to capture the differences.

Input functions in C usually return values via reference parameters and we assign unique, unknown values of the form `_%inuv.nnn` to the affected variables. Input in a loop, if not done ionto an array element, requires the affected variable to be treated as though it were one.

In the long term, we will need to provide a solution to the library problem that minimizes or obviates the supply chain risks. We are not alone in facing this issue.

4.6 Global variables used locally

In C, global variables get a default value of zero and this is recognized in the LLVM IR global register declarations. When we compute the behavior of the functions in a program, we do it in a reverse dependency order so that each defined function has its behavior computed prior to computing the behavior of functions that call it[6]. In the example, below, the driver function reads values and stores them in the global variables prior to calling the function `m` which references them. In building the local symbol table used in computing the behavior of `m()`, we need to insert local references to the globals referenced and give them unique, unknown symbolic values so that the (possibly) incorrect default or declared values will not appear in the behavior. These symbols should disappear when the global effects of a call on the routine are lifted to the calling environment. Global variables have long been recognized as potential trouble spots. We expect that disciplined use of globals will be tractable, but that wide spread conditional modifications at many points in the program could result in incomprehensible value expressions appearing in the behavior. It is not clear whether this is a feature or a flaw in the approach.

[5]This would be a "supply chain" intrusion similar to the one described in [13]. A similar exploit [3] on the `xz` compression library was discovered while this paper was being written.
[6]This precludes recursion or mutually dependent functions, at present. We believe that recursion could be handled, but have not done so yet.

```
int x, y, t, i, j, k;
int f(int x){ x = 3 * x + 3; return x;}
void m() {
    t = i - j;
    if (i > j) {
        x = 0; y = f(x);
        while (i != j){
            i = i + k; k = k + 1; i = i - k; y = f(y);}
    } else {
        if (j > i){
            while (j != i){
                j = j + k; k = k - 1;
                j = j - k; y = f(y);}
        } else {
            while (t != i) {
                for(int z = 0; z != y; z = z + 1) x = x + 1;
                y = x - y; t = t + 1;}
        } }
    k = i + j; j = 2 * k;}
```

Fig. 1. The operational code from an contrived example

4.7 Arbitrary Branches

At this point, we do not support `gotos`. In some cases, these are tractable, but extensive flow analysis would be required and their infrequent use makes this a low priority. Similarly, we do not yet support the loop escapes, `continue` and `break`. These will be added at some point.

5 CONCRETE EXAMPLES

We present several examples to give the reader an idea of the process and the kinds of results it might produce. We note that using function extraction as an aid in producing a correct program and using it to analyze a mysterious code artifact such as a malware sample represent two very different approaches. In the former case, the programmer should have a pretty good idea of the intent of the program and, if the job is well done should get functionality that is in good agreement with the intent. In the later case, the user may have no idea of the intent, if any, of the code. If the code to be analyzed is derived from a memory object, the functionality may be further affected by dis-assembly and decompilation errors, transformations made during optimization, etc. Out first example lies somewhat in this region having been constructed to be incomprehensible without substantial effort.

5.1 A version using the Java prototype

The code in the example of figure 1 illustrates the ability of the prototype to extract the functionality from what is best described as a contrived example. There is no good reason to write code like this, and it would take even a competent programmer considerable effort to determine what it actually computes and the input domain for which it theoretically terminates. Not shown is a driver which reads values into the global variables, (int x, y, t, i, j, k;), prints them out, invokes `m()`, and prints the result. The C version is shown, but the java version (on which the analysis results are based is nearly identical, wrapping the code in a `class` and prefixing each declaration with `public static`. The current Java

$$
\begin{array}{l}
\quad i > j \wedge \\
\quad\quad xP = 0 \wedge tP = i - j \wedge iP = j \wedge jP = 4j \wedge kP = 2j \wedge zP = z \\
\vee \\
\quad j \geq 0 \wedge i = j \wedge y \geq 0 \wedge \\
\quad\quad xP = x\,\text{Fibonacci}(j+1) + y\,\text{Fibonacci}(j) \wedge \\
\quad\quad yP = x\,\text{Fibonacci}(j) + y\,\text{Fibonacci}(j-1) \wedge \\
\quad\quad tP = j \wedge iP = j \wedge jP = 4j \wedge kP = 2j \\
\vee \\
\quad (i \neq j \vee j < 0 \vee y < 0) \wedge i \leq j \rightarrow \\
\quad\quad \textit{Undefined}
\end{array}
$$

Fig. 2. The computed behavior of function `m()` from figure 1, modified as described.

prototype cannot handle local variables in a function that returns void, so the `for(;;)` loop has been replaced with a `while()` loop and `z` added to the global variables. The drivers differ in the methods used for the i/o operations.

The behavior of the function `m` is shown in figure 2. In this presentation, variables ending with P are the output (or "primed") values of the corresponding input variable. There are two cases of behavior given. The first occurs when $i > j$. This corresponds to the *(*true) case for `if (i > j)` at the top of the function where the while loop has the effect of reducing `i` by 1 in each iteration and repeatedly applying `f()` starting with an initial value of 3. The second occurs only when the *false* case holds for the `if (j > i)` which implies that $i = j$. If we view the behavior as conditional assignments for the two cases[7], $i > j$ and $j \geq 0 \wedge i = j \wedge y \geq 0$ we note that the full input domain for the variables, i, j, and y is not covered. What about the remainder, $(i \neq j \vee j < 0 \vee y < 0) \wedge i \leq j$?

In the mathematical world, the while loop guarded by the predicate `j != i` does not terminate as the net effect of its body is to increase j which is already larger that i. Similarly, the final `else` reached when $i = j$ starts with $t = i - j = 0$ contains a while loop that terminates when $t = i$ which is also $t = j$ and t is incremented in the loop, so it cannot terminate if j (i) is negative. Even if the outer `while()` terminates, the inner `for(;;)` cannot terminate if $y < 0$ since incrementing z from 0 cannot produce a negative value. Mathematica cannot compute behaviors for infinite loops, so it simply omits these cases from its analysis. We identified these cases negating the conjunction of the cases for which solutions are provided and added them to the behavior table as *Undefined*.

[7]The $j \geq 0$ term might make more sense if it were $i \geq 0$ as the outer loop in this section terminates on `while (t != i)`

```
#include <stdio.h>
static int a = 1, b = 2, c = 3, d = 4;
int foo(int w, int* x, int** y, int*** z)
{   w = w + 1; *x = *x + 1; **y = **y + 1; ***z = ***z + 1;
    return w + *x + **y + ***z; }
int main(int argc, char ** argv)
{   int e; int* ca = &c; int* da = &d; int** daa = &da;
    printf("1: a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);
    e = foo(a, &b, &ca, &daa);
    printf("2: a = %d, b = %d, c = %d, d = %d, e = %d\n", a, b, c, d, e);
    return -e; }
```

Fig. 3. Call by reference and global modification

5.2 An example using the LLVM C prototype

The example shown in figure 3 is intended to show the handling of reference parameters at several levels of indirection and the modification of global variables. It also illustrates the recording of output behaviors. The function, `foo`, adds 1 to each of its arguments and returns their sum. The `main()` function creates a pointer, `ca` to the global `c` and an indirect pointer, `daa` to `d`. `a`, the address of `b`, the address of the pointer to `c`, and the address of the indirect pointer to `d` are passed to `foo()` after printing the values of the globals. Their values on return are also printed, along with the value returned by `foo()`, `e`. Finally, `main()` returns `-e`. Executing the example produces:

```
1: a = 1, b = 2, c = 3, d = 4
2: a = 1, b = 3, c = 4, d = 5, e = 14
```

reflecting the incrementation of `a` in the function that is not reflected back into the calling environment.

The LLVM IR produced by processing this is some 111 lines and space precludes including it, but the signature for `foo()` is "`define i32 @foo(i32 noundef %w, ptr noundef %x, ptr noundef %y, ptr noundef %z)`." Note that the pointers are opaque indicating neither the level of indirection nor the type of the target. These issues are addressed in the code generated by the `clang` front end. For example, `***z = ***z + 1;` becomes

```
1: %9 = load ptr, ptr %z.addr, align 8
2: %10 = load ptr, ptr %9, align 8
3: %11 = load ptr, ptr %10, align 8
4: %12 = load i32, ptr %11, align 4
5: %add3 = add nsw i32 %12, 1
6: %13 = load ptr, ptr %z.addr, align 8
7: %14 = load ptr, ptr %13, align 8
8: %15 = load ptr, ptr %14, align 8
9: store i32 %add3, ptr %15, align 4
```

which reflects both the single assignment and single use properties of the LLVM IR. Note that `%12` and `%15` contain the address of the memory being modified and an optimization pass could easily recognize this.

The behavior is captured in a pair of states, represented as symbol tables. We need only retain persistent values as the temporary registers are dead upon use. To save space, we only show entries for the values of `w` and `z`, but similar entries apply for `x` and `y`. Note that the initial value of `w` is represented as a scalar while that of `z` indicates a pointer. When `z` is computed, we dereference the pointer and use a value expression.

```
Initial persistent variable values
  Symbol          Kind  Type            References                    Value
  _%w.addr        lvar  i32             a16,s20,l24,s26,l48           _%p.1
  _%z.addr        cvar  ptr             a19,s23,l39,l44,l56           =>_%p.4
Final persistent variable values
  Symbol          Kind  Type            References                    Value
  _%w.addr        lvar  i32             a16,s20,l24,s26,l48           ( _%p.1 + 1 )
  _%z.addr        cvar  ptr             a19,s23,l39,l44,l56           ( _%p.4 + 1 )
```

Memory is not explicitly represented in LLVM IR, so in "`%9 = load ptr, ptr %z.addr, align 8`," `%z.addr` points to the memory allocated to hold the value of `z`. We denote this memory using the notation `_%z.addr`. and use the value field in the symbol table entry to track the value. Rather than use variable names here, we denote the initial values of parameters with a positional notation, `_%p.4` to indicate the value of the 4th parameter to the routine. This simplifies the lifting of results from then called to the calling environment. Values of type `ptr` will be reflected into the calling environment while values of scalar types will not. Thus. the changes to parameters 2-4 will be seen in the calling environment while the changes to parameter 1 will not.

`foo()` shows both return and reference behaviors (again omitting some reference parameters to save space).

```
Behavior for function @foo
```

```
    Return value behavior
      4 + _%p.1 + _%p.2 + _%p.3 + _%p.4
    Reference parameter behavior
      Reference       Value
        =>_%p.4         1 + _%p.4
```

Similarly, `main` shows return. global, and output behaviors samples of which are seen below:

```
Behavior for function @main
   Return value behavior
      -4 - _%gbuv.019 - _%gbuv.020 - _%gbuv.021 - _%gbuv.022
   Global variable behavior
      Global          Value
      _@d             1 + _%gbuv.022
   Output behavior
      Output                    Value
      @main(93),@printf(1)      c"2: a = %d, b = %d, c = %d, d = %d, e = %d\0A\00"
      @main(93),@printf(2)      _%gbuv.019
      @main(93),@printf(5)      1 + _%gbuv.022
      @main(93),@printf(6)      4 + _%gbuv.019 + _%gbuv.020 + _%gbuv.021 + _%gbuv.022
```

At this point, behaviors are computed on a per function basis, capturing the behavior in terms of values available at call time. Thus, although globals in `C` must have values, in general, we cannot simply use the initial values to capture the behavior of functions in which global variables are changed and we use a symbolic notation, `_%gbuv.#` to capture the effect of modifications. In this case, the only modifications happen when `main()` passes references to globals to `foo()` where they are modified. These modifications are reflected in the behavior of the main program. The output behavior is a work in progress. The `@printf()` references indicate the positional argument to the output function. What is being printed can be determined by looking at the format string given as `@printf(1)`. Here, we show the format and the final values of `a` `d` and `e`, the return from `foo()`. The `@main()` references indicate the line in the LLVM IR file that called `@printf()`. This would be expanded to include a full calling path for more complicated programs and the values would be lifted to the top level calling environment. At this point, we can substitute the initial global values for the symbolic ones, simplify, and get a program behavior that is in agreement with the output results shown on page 13.

6 THE EVOLUTION OF FUNCTION EXTRACTION AND LONG TERM GOALS

At present, we are moving towards a system that will compute the behavior of many `C` programs and, we hope, others that use a front end to produce unoptimized LLVM IR. We recognize that a single mathematical function will not be useful for large, complex, programs. After all, even if you had such an expression for, say, Microsoft Word, what would you do with it? At the level of computational subroutines and modest subsystems we believe that FX will be useful for finding errors as well as for finding unexpected additional behaviors. Both of these are important aspects of security and behavioral analysis may be an attractive alternative to formal verification as well as obviating the need for extensive low level testing. As we go forward, we plan to look at the ability of the approach to discover situations in which the program being examined has the potential for becoming *undefined*. Many existing vulnerabilities arise from such situations and our analysis appears to have the potential to deal with many of these by identifying cases where actual program behavior differs from the mathematical ideal.

In this age, no paper would be complete without, at least, a mention of Large Language Models and programs like Chat GPT. We have had several encounters. Last fall, students working for one of the authors asked Chat GPT to provide code samples for several examples similar to ones that we hoped to analyze. The results were mixed. We had hoped to analyze multiple versions of code for computing simple statistics. We had at hand at least three versions, naive direct implementations of the definitional algorithms from statistics textbooks (notoriously bad numerically in many cases), versions from scientific subroutine libraries that avoid or minimize many of the numeric issues and an incremental version, also numerically sound, intended for the efficient computation of mean, variance, skew and kurtosis of incrementally arriving time series data. The AI suggested versions tended towards the naive versions, and in some cases had errors that prevented compilation. This is a rapidly evolving area and a potential area where our analysis could be useful is in determining whether or not AI generated code actually has the desired functionality. A second effort performed by one of the authors, asking GPT4 to analyze examples including the contrived example shown in figure 1. The analysis is superficial, being at the level of a beginning student who sees trees, but has no concept of a forest. It completely misses the non-termination of the loop in the $j < i$ case and the non-termination of part of the $j = i$ case. Its analysis of this case also misses the computation of Fibonacci numbers.

> If i equals j: A while loop runs as long as t is not equal to i. Within this loop, a for-loop increments x by 1, y times. Then, y is recalculated as x - y, and t is incremented. This block seems designed to perform operations a certain number of times based on the initial difference between i and j.

We find this unsurprising as nothing in the training of AIs on masses of text might be expected to teach the AI how to apply the knowledge contained in its training material. There is a big difference between knowing that a program might exhibit a behavior and asserting that a never before seen program has that behavior.

In addition, we believe that the routine use of function extraction tools can support the teaching of programming. Typical computer science curricula defines programmer education in three broad steps: years one and two, introduction to programming concepts; years two and three, algorithms, data structures, and discrete mathematics; years three and four, programming language concepts and semantics, and (possibly) an introduction to program verification. Students are generally taught how to write programs before being taught how to write correct programs, or even what a correct program is. Software engineering is the only engineering discipline burdened by this educational paradox.

By introducing function extraction tools early in the educational process, we can help students specify and verify programs as they are written. It is easy to imagine a student starting a session with a vaguely formulated specification and an incorrect program, and concluding the session with a valid specification and a correct program. Instilling program verification principles will serve students well over their entire professional careers. Although security is not explicit in this approach, we believe that learning to think in terms of correct program behavior will go a long way towards avoiding many of the programming "blunders" that manifest as vulnerabilities. It is important to note that this will widen the knowledge gap between students who have this education and those who, for whatever reason (e.g., socio-economic), are not exposed to this approach. As can be seen from figure 2, the format of the computed behavior from FX is not intuitively obvious.

The possibilities of integrating FX and formal verification continue after schooling into employment as a programmer. FX can be used as a check of correctness after formal verification has taken place, thus providing useful process feedback to the development life cycle. Note that even without formal verification, the use of FX can help detect problems in developed code, leading to higher quality programs with fewer security issues.

REFERENCES

[1] Phil Budne. 2024. *CSNOBOL4.* https://www.regressive.org/snobol4/csnobol4/curr/

[2] Clang. 2024. *Clang: a C language family frontend for LLVM.*

[3] Dan Goddin. 2024. *What we know about the xz Utils backdoor that almost infected the world.* https://arstechnica.com/security/2024/04/what-we-know-about-the-xz-utils-backdoor-that-almost-infected-the-world/

[4] Ralph E. Griswold, James F. Poage, and Ivan P. Polonsky. 1971. *The SNOBOL4 Programming Language (2nd ed.).* Prentice Hall.

[5] LLVM IR. 2024. *LLVM Language Reference Manual.* https://llvm.org/docs/LangRef.html

[6] Javaparser. 2024. *Tools for your Java Code.* https://javaparser.org/

[7] Richard A. Kemmerer. 1983. Shared resource matrix methodology: an approach to identifying storage and timing channels. *ACM Trans. Comput. Syst.* 1, 3 (aug 1983), 256–277. https://doi.org/10.1145/357369.357374

[8] Richard C. Linger, Harlan D. Mills, and Bernard I. Witt. 1979. *Structured Programming: Theory and Practice.* Addison Wesley Publishing Company.

[9] LLVM. 2024. *The LLVM Compiler Infrastructure.* https://llvm.org/

[10] Ali Mili, Shir Aharon, and Chaitanya Nadkarni. 2009. Mathematics for reasoning about loop functions. *Science of Computer Programming* 74, 11 (2009), 989–1020. https://doi.org/10.1016/j.scico.2009.09.009

[11] John Regehr. 2010. *Embedded in Academia: A Guide to Undefined Behavior in C and C++, Parts 1–3.* https://blog.regehr.org/archives/213

[12] rust lang.org. 2024. *Rust Compiler Development Guide: Overview of the compiler.* https://rustc-dev-guide.rust-lang.org/overview.html

[13] Dina Temple-Raston. 2021. *A 'Worst Nightmare' Cyberattack: The Untold Story Of The SolarWinds Hack.* https://www.npr.org/2021/04/16/985439655/a-worst-nightmare-cyberattack-the-untold-story-of-the-solarwinds-hack

[14] Wolfram Research, Inc. 2024. *Mathematica, Version 14.0.* https://www.wolfram.com/mathematica, Champaign, IL.