

# Dynamic Systems Security Testing using Function Extraction

Alan R. Hevner  
College of Business  
University of South Florida  
Tampa, FL 33620  
[ahevner@usf.edu](mailto:ahevner@usf.edu)

Richard C. Linger  
Cyberspace Sciences and Information Intelligence Research Group  
Oak Ridge National Laboratory  
Knoxville, TN 37831  
[lingerr@ornl.gov](mailto:lingerr@ornl.gov)

**Abstract**— We describe an approach for applying Function Extraction (FX) technology to the dynamic testing of security in large-scale, operational software systems. FX is used proactively as an intrusion detection and prevention system (IDPS) within a security infrastructure surrounding the operation of a critical software system. An innovative aspect of the FX approach is the concept of computational security attributes (CSA). The CSA approach to software security analysis provides theory-based foundations for precisely defining and computing security attribute values. The translation of a static security property expressed as an abstraction of external data to the dynamic behavior of a program expressed in terms of its data and functions is a key to the CSA approach for verification of behaviors that meet specific security properties. The paper concludes with a discussion of future research and development directions for applying FX to dynamic security testing.

**Keywords**— Systems security testing, Intrusion detection, Intrusion prevention, Function extraction, Computational security attributes

## I. INTRODUCTION

Recent statements by the FBI’s top cyber cop, Shawn Henry, have highlighted the immense challenges of security in critical computer systems and data networks (Barrett 2012). Current public and private security approaches are unsustainable. Henry states, “I don’t see how we ever come out of this without *changes in technology or changes in behavior*, because with the status quo, it’s an unsustainable model.” (emphasis added)

Our research on the emerging technologies of Function Extraction (FX) for software system understanding and analysis provides a paradigm change that can result in new ways of thinking about the security testing of software systems. The objective of FX is to compute the behaviors of software systems to the maximum extent possible with mathematical precision. FX presents an opportunity to move from the current range of slow and costly security testing processes to fast and cheap computation of system behaviors, including behaviors related to security, at machine speeds. Because a principal objective of testing is to validate system behaviors and qualities, automated

computation can be expected to streamline testing processes and permit increased focus on system-level issues such as security and sustainability (Pleszkoch et al. 2008).

In this paper, we propose an approach for applying FX technology to the dynamic testing of security in large-scale, operational software systems. FX is used proactively as an intrusion detection and prevention system (IDPS) within a security infrastructure surrounding the operation of a critical software system. An innovative aspect of the FX approach for IDPS is the concept of computational security attributes (CSA). The CSA approach to software security analysis provides theory-based foundations for precisely defining and computing security attribute values (Walton et al. 2009). The translation of a static security property expressed as an abstraction of external data to the dynamic behavior of a program expressed in terms of its data and functions is a key to the CSA approach to verification of behaviors that meet specific security properties.

The paper concludes with a discussion of future research and development directions for applying FX to dynamic security testing.

## II. FUNCTION EXTRACTION

Software behavior computation begins with the observation that sequential programs can be regarded as rules for mathematical functions or relations. That is, programs can be treated as mappings from domains to ranges, and mapping functions can be computed through methods of function composition (Linger et al. 1979).

For illustration, consider the following simple program fragment operating on small integers. It takes in values for  $a$ ,  $b$ , and  $c$  (used but never set) and produces values for  $r$ ,  $s$ ,  $t$ ,  $w$ ,  $x_1$ , and  $x_2$ :

```
r := b*b
s := a*c
t := 2*a
w := sqrt(r - 4*s)
x1 := (-b + w)/t
x2 := (-b - w)/t
```

Final values for  $r$ ,  $s$ , and  $t$  are self-evident, and final values for  $w$ ,  $x_1$ , and  $x_2$  can be easily composed through successive algebraic substitution:

$$\begin{aligned}
w &= \sqrt{r - 4*s} \\
&= \sqrt{b*b - 4*a*c} \\
x1 &= (-b + w)/t \\
&= (-b + \sqrt{r - 4*s})/t \\
&= (-b + \sqrt{b*b - 4*a*c})/2*a \\
x2 &= (-b - w)/t \\
&= (-b - \sqrt{r - 4*s})/t \\
&= (-b - \sqrt{b*b - 4*a*c})/2*a
\end{aligned}$$

The final step in each derivation gives the net behavior as:

$$\begin{aligned}
\text{true} \rightarrow \\
w &:= \sqrt{b*b - 4*a*c} \\
x1 &:= (-b + \sqrt{b*b - 4*a*c})/2*a \\
x2 &:= (-b - \sqrt{b*b - 4*a*c})/2*a
\end{aligned}$$

Thus, the behavior of the fragment is to compute the familiar formula for the two roots of a quadratic equation defined by coefficients a, b, and c, and assign them to variables x1 and x2, with the square root of the discriminant captured in the final value of w. Note that these expressions are procedure-free. The right-hand-sides are assigned to the left-hand-side variables concurrently. This is a conditional concurrent assignment (CCA), the general form of expression for computed behavior. In this case, the condition is “true” because the program is a sequence structure that always executes. This is the as-built specification of the program. If the program contained an error or unintended content, it would similarly be revealed in the composition of its behavior.

Programmers engage in this sort of reasoning to determine net effects of programs all the time. They have learned the semantics of individual instructions, that is, how each instruction transforms the state of a program, and must mentally compose these functional effects to arrive at an understanding of what an entire program written by themselves or others does in execution. This is not an easy task. As software systems are developed and evolve over time, large quantities of composite semantic content are continuously created, whether correct or incorrect; secure or not secure.

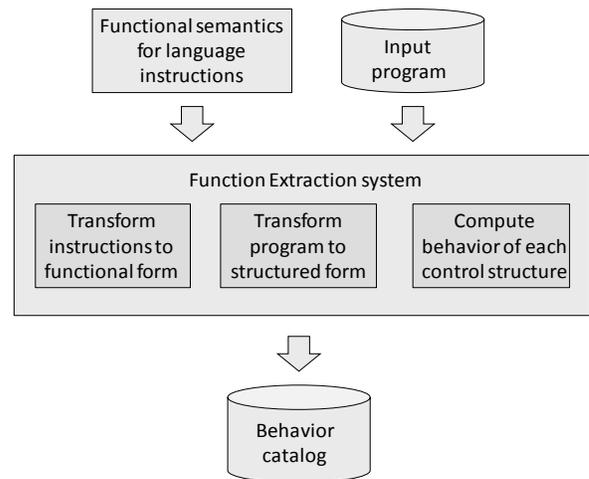
Effective system development and evolution depends on how well these behavioral semantics are understood. The problem is compounded by the massive numbers of possible paths through programs and the necessity to understand them all. It is one thing to do this for small programs, but quite another for large ones, where complexity can easily overwhelm human comprehension. Because human fallibility in mental program composition is a source of errors and vulnerabilities in software development, it is reasonable to ask if this task can be offloaded to machine computation to help free human intelligence and attention for more creative tasks in software engineering.

This question is being addressed in research and development being carried out on Function Extraction (FX) technology by Oak Ridge National Laboratory (ORNL).

The objective of FX is to compute the behavior of software with mathematical precision to the maximum extent possible. As noted, sizable programs can contain a massive number of execution paths; however, they are constructed of a finite number of nested and sequenced control structures, each of which makes a knowable contribution to overall behavior. These structures correspond to mathematical functions that can be computed in a stepwise process that traverses the finite control structure hierarchy. At each step, procedural details are abstracted out, while net effects are preserved and propagated in extracted behavior.

FX technology is initially being implemented for programs written in or compiled into Intel assembly language, with a current emphasis on malware analysis (Pleszkoch and Linger 2004) and use of computed behaviors to augment or replace certain forms of testing (Pleszkoch et al. 2008). The focus of malware analysis is on the use of computed behavior to unravel complex program logic and remove control flow and no-op block (code with no functional effect) obfuscation inserted by intruders, followed by computation of the behavior of the remaining functional code.

The overall architecture of an FX system is depicted in Figure 1. The starting point is a definition of the functional semantics of the programming language. The Function Extraction process generally proceeds from transformation of an input program into functional form, followed by structuring and behavior computation. There is a lot more to the process than this, but the figure provides a notional view of the major steps involved.



**Figure 1: The Function Extraction process**

Function Extraction has potential for widespread application across the software engineering life cycle, as discussed in (Collins et al. 2011). At this point, FX is an emerging technology that can be built out for many wide-ranging evaluation and operational applications.

### III. FX FOR INTRUSION DETECTION AND PREVENTION

With the use of FX technology, an opportunity exists for systems testing and customer acceptance testing to shift from defect detection to certification of fitness for use. In particular, we highlight the application of FX for the purposes of intrusion detection and prevention in operational software systems. Intrusion detection is the process of monitoring dynamic events in a system and analyzing them for indications of violations or threats of violations of security policies, acceptable use policies, or standard operating procedures for security practices (Scarfone and Mell 2007). An intrusion prevention system works with the detection mechanism to proactively react and attempt to stop security violations. Here we outline FX as an element of an intrusion detection and prevention system (IDPS). A key aspect is use of FX for the analyses of Computational Security Attributes as described in (Walton et al. 2006)

### IV. COMPUTATIONAL SECURITY ATTRIBUTES (CSA)

Fast and reliable analysis of security attributes is vital for every sector of our software-dependent society. For example, access to enterprise applications and data must be restricted to those who can provide appropriate proofs of identity. Applications and data must be protected so that attempts to corrupt them are detected and prevented. Healthcare systems must protect personal data while allowing controlled access by authorized personnel. Enterprises must be able to demonstrate that every accounting change is auditable. The flow of data through enterprise applications and the flow of transactions that drive the data must be logged and reported as proof of what actually happened.

In the current state of practice, security properties of software systems are typically assessed through labor-intensive evaluations by security experts who accumulate system knowledge in bits and pieces from architectures, specifications, designs, code, and test results. Ongoing program maintenance and evolution limit the relevance of even this hard-won but static and quickly outdated knowledge. When systems operate in threat environments, security attribute values can change very quickly. To further complicate matters, security strategies must be sufficiently dynamic to keep pace with organizational and technical change.

A fundamentally different approach recognizes and leverages the fact that the problem of determining the security properties of programs comes down in large measure to the question of how the software behaves when invoked with stimuli intended to cause harmful outcomes. Because security properties have functional characteristics amenable to computational approaches, it is appropriate to focus on the question “What can be computed with respect to security attributes?” The computational security attribute approach provides a step toward a computational security

engineering discipline. The ultimate goal is to develop and describe mathematical foundations and their engineering automation to permit:

- rigorous specification, evaluation, and improvement of the security attributes of software and systems during development,
- specification and evaluation of the security attributes of acquired software,
- verification of the as-built security attributes of software systems, and
- real-time evaluation of security attributes during system operation.

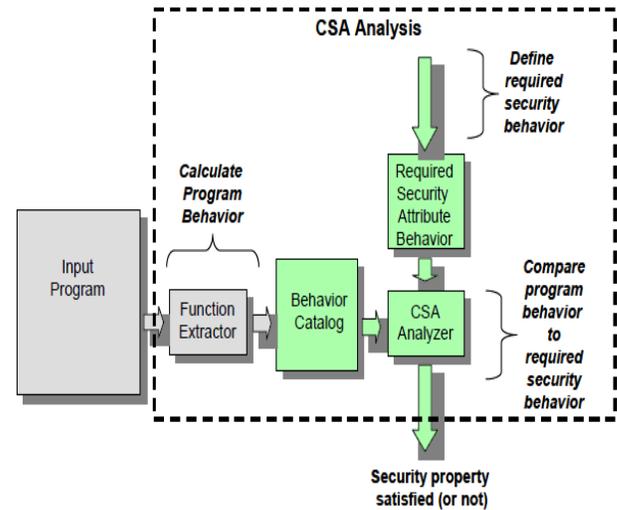


Figure 2: Use of FX for CSA Analysis

While analysts have often characterized many security attributes as “non-functional” properties of programs, it turns out that they are in fact fully functional and thereby subject to FX-style automated analysis. Complete definitions of the required behavior of security attributes of interest can be created based solely on data and transformations of data. These definitions can then be used to analyze the security properties of programs. Thus, as illustrated in Figure 2, computational security attribute (CSA) analysis consists of three steps (Walton et al. 2009):

1. **Define required security behavior.** Specify security attributes in terms of required behavior during execution expressed in terms of data and transformations on data.
2. **Calculate program behavior.** Apply function extraction to create a behavior catalog that specifies the complete “as built” functional behavior of the code.
3. **Compare program behavior to required security behavior.** Compare the computed behavior catalog with required security attribute behavior to verify whether it is correct or not.

Requirements for security attribute behavior must explicitly define expected behavior of code in all

circumstances of interest. Thus, the requirements for security attribute behavior must include a minimal definition of required behavior for all inputs of interest to the security attributes, including desired inputs (for example, an authenticated user id) and undesired inputs (for example, an unknown user id). Usage environment conditions related to security attributes are specified in the same manner as inputs to the system. For example, availability of the network might be specified by a Boolean value that indicates whether or not the network is currently available. Security successes and failures are also specified in terms of data. For example, system control data can be used to indicate whether the current user has been authenticated using a trusted authentication mechanism.

Verification that a security property is satisfied requires verification of both the data at rest (i.e., the control data values) and the data in motion (i.e., the mechanisms used to perform the data transformations). Some common tasks to verify data at rest include checking to make sure that a specific task (for example, an audit task) will always be carried out to validate the contents of a specific control data structure. Advantages of this approach to security attribute verification include the use of constraints and boundary conditions that can make any assumptions explicit. People and process issues can be handled by the CSA approach by using assumptions and constraints as part of the behavior catalogs. Behaviors can embody requirements for a given security architecture. The attribute verification process will expose security vulnerabilities, making it easier to address evolution of code, environment, use, and so forth.

The CSA verification process can provide important opportunities for improved acquisition and third-party verification. A “user” of a system might be a person, a device, or a software component. The user may be the intended user or may be an unexpected and/or hostile user. An issue that must be considered with commercial off-the-shelf (COTS) products and reuse is that the definition of “user” embodied in the security behavior requirements may not be the same definition that was employed in the COTS or reused component. The same issue occurs when unknown components are employed as “black boxes” in systems of systems. If, in the composition of components or systems, it doesn’t matter what a specific “black box” component does with respect to security attribute requirements, then that component can be used. However, if the behavior of a component does matter, it cannot be used until its security attributes have been verified. In this case, a behavior catalog can be calculated for the component using its executable, even if documentation and source code are not available. Only externally observable behaviors are of interest to security attribute analysis. Thus, while the behavior catalog will have to be produced for the entire system in order to extract the externally observable behaviors, there is no need to expose the algorithm or source code, and there’s no need to understand the entire state space.

## V. CSA EXEMPLARS

Security properties are fully functional and are dependent on the execution behavior of software. We briefly describe seven security attributes to illustrate the range of CSA analyses that can be performed via the use of FX. Three of these attributes (confidentiality, integrity, and availability) are important to information. The other four attributes (authentication, authorization, non-repudiation, and privacy) relate to the people who use that information. The behavioral requirements for each of these attributes can be completely described in terms of data items and constraints on their processing. The processing can be expressed, for example, as logical or quantified expressions or even conditional concurrent assignments, which can be mechanically checked against the calculated behavior of the software of interest for conformance or non-conformance with CSA requirements. A fuller discussion of these CSA analyses can be found in Walton et al. (2006, 2009).

- *Authentication:* Authentication requires that a trusted user has been bound to the behavior. That is, the system will only allow the program to be executed if the user has previously been determined to be a trusted user. To verify authentication, one must examine the net effects on the control data related to authentication: verify the data that provides evidence that the binding took place, and verify that this evidence data was not changed before completion of any operation that required authentication.
- *Authorization:* Authorization requires that a user has the right to perform the requested process. To verify that an authorized operation took place, one must examine the net effects on the control data to verify that it provides evidence that authorization occurred before the operation, and that the evidence data for the authorization was not changed before that operation completed.
- *Non-Repudiation:* Non-repudiation of data transmission requires that neither the sender nor the recipient of the data can later refute his or her participation in the transaction. Non-repudiation of changes to a dataset requires that the means for authentication of changes cannot later be refuted. For the purposes of this discussion we treat data change as a special case of data transmission, where receipt of the data transmission includes making and logging the requested change to the dataset. To verify non-repudiation one must examine the net effects on the control data related to non-repudiation.
- *Confidentiality:* Confidential data access or confidential data transmission requires that unauthorized disclosure of one or more specific data items will not occur. Confidentiality is often described in terms of a security policy that specifies the required strength of the mechanisms that ensure that the data cannot be accessed outside the system. For example, the security policy may require

verification that approved encryption mechanisms are used for the output. To verify confidentiality, one must examine the net effects on the control data related to confidentiality.

- *Privacy*: Privacy requires that an individual has defined control over how his/her information will be disclosed. To verify privacy, one must examine the net effects on the control data related to privacy.
- *Integrity*: Integrity requires that authorized changes are allowed, changes must be detected and tracked, and changes must be limited to a specific scope. Integrity is defined as a property of an object, not of a mission. To verify integrity, one must examine the net effects on the control data related to integrity. That is, one must be able to: isolate the object, isolate all the behaviors that can modify the object, detect any modifications to the data, and ensure that all transformations of the data across the object are within the pre-defined allowable subset.
- *Availability*: Availability requires that a resource is usable during a given time period, despite attacks or failures. To verify availability, one must examine the net effects on the control data related to availability. To avoid having to consider temporal properties, one can specify non-availability rather than availability (i.e., specify under what conditions the program's behavior catalog do not apply).

## VI. RESEARCH STATUS AND FUTURE DIRECTIONS

Computational security attribute (CSA) analysis is a step toward a computational security engineering discipline. It can potentially transform systems security engineering by rigorously defining security attributes of software systems and replacing or augmenting labor-intensive, subjective, human security evaluation. Advantages of the CSA approach include the following:

- A rigorous method is used to specify security attributes in terms of the actual behavior of code and to verify that the code is correct with respect to security attributes.
- The specified security behaviors can provide requirements for security architectures.
- Traceability capabilities can be defined and verified outside of the automated processes.
- Vulnerabilities can be well understood, making it easier to address evolution of code, environment, use, and users.
- The use of constraints provides a mechanism for explicitly defining all assumptions.

CSA technology addresses the specification of security attributes of systems before they are built, specification and evaluation of security attributes of acquired software, verification of the as-built security attributes of systems, and real-time evaluation of security attributes during system operation.

Our future directions include the development of prototype automation to support application of CSA technology. This automation will be based on a vision of

human-computer interaction that would complement and amplify human capabilities for reasoning about software security attributes during systems development and for real-time evaluation of a system's security attributes during operation. These tools will be constructed in accumulating increments to maximize earned value and minimize risk.

CSA supports a usage-centric evaluation of security attributes that can explicitly consider the objectives and constraints of specific execution environments. Such an approach will support modeling, analysis, and evaluation of the security attribute values of software, as constrained by the policies of specific execution environments. In order for this approach to be widely used, tools are needed to support user input and query of security requirements, including automatic mapping of the model of user-specified acceptable function calls and safe behavior to the code's behavior catalog. The ORNL FX project is developing tools that will be used to compare behavior catalogs. These FX tools, combined with the CSA approach and proposed CSA tools, will support security analysts in the comparison of security attribute requirements and constraints with behavior catalogs, thus providing a mechanism for automated security attribute analysis.

## ACKNOWLEDGMENT

We gratefully acknowledge our collaborators at the Oak Ridge National Laboratory in this research.

## REFERENCES

- D. Barrett, "U.S. Outgunned in Hacker War," *The Wall Street Journal*, March 28, 2012.
- R. Collins, A. Hevner, and R. Linger, "Evaluating a Disruptive Innovation: Function Extraction Technology in Software Development," *Proceedings of the 44<sup>th</sup> Annual Hawaii International Conference on System Sciences (HICSS44)*, Hawaii, January 2011.
- R. Linger, H. Mills, and B. Witt, *Structured Programming: Theory and Practice*. Reading, MA: Addison Wesley, 1979.
- M. Pleszkoch and R. Linger, "Improving Network System Security with Function Extraction Technology for Automated Calculation of Program Behavior," *Proceedings of the 37<sup>th</sup> Annual Hawaii International Conference on System Sciences (HICSS-37)*. Big Island, Hawaii. Los Alamitos, CA: IEEE Computer Society Press, 2004, pp. 20299c.
- M. Pleszkoch, R. Linger, and A. Hevner, "Introducing Function Extraction into Software Testing," *The Data Base for Advances in Information Systems*, Vol. 39, No. 3, August 2008, pp. 41-50.
- K. Scarfone and P. Mell, *Guide to Intrusion Detection and Prevention Systems (IDPS): Recommendations of the NIST, NIST Special Publication 800-94*, Gaithersburg, MD, 2007.
- G. Walton, T. Longstaff, and R. Linger, "Technology Foundations for Computational Evaluation of Software Security Attributes," SEI Tech Report CMU/SEI-2006-TR-021, 2006.
- G. Walton, T. Longstaff, and R. Linger, "Computational Evaluation of Software Security Attributes," *Proceedings of the 42<sup>nd</sup> Hawaii International Conference on System Sciences (HICSS-2009)*, Hawaii, 2009.