Manipulating Program Functionality to Eliminate Security Vulnerabilities

Martin Rinard
Department of Electrical Engineering and Computer Science
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract Security vulnerabilities can be seen as excess undesirable functionality present in a software system. We present several mechanisms that can either excise or change system functionality in ways that may 1) eliminate security vulnerabilities while 2) enabling the system to continue to deliver acceptable service.

1 Introduction

We discuss several automatic techniques for changing program behavior in ways that may eliminate security vulnerabilities. We take the perspective that vulnerabilities are undesirable functionality and therefore focus on techniques that change or even eliminate some of the functionality that the system offers to users.

One of the observations motivating our approach is that many software systems provide substantially more functionality than users require, desire, or are even aware of. There are several reasons for this phenomenon:

- **General-Purpose Software:** Because of the high cost of developing software systems and the consequent need to amortize this cost over many users, many software systems are designed to contain functionality for a wide range of users. Because users have such varying needs, each user winds up using only a small fraction of the total functionality.
- **Feature Accretion:** As software systems go through their life cycle, developers almost always preserve existing features (to ensure backwards compatability) while adding new features. Over time the software accumulates more and more functionality, much of it obsolete and designed for operating contexts that have changed since the introduction of much of the functionality.

Martin Rinard
EECS Department, CSAIL, Massachusetts Institute of Technology, e-mail: rinard@mit.edu

- **Subsystem Reuse:** It is often quicker to build systems by incorporating existing subsystems than by building the desired functionality from scratch. But good building blocks are often more general and contain more functionality than necessary for the specific usage scenario at hand.
- **Development Errors:** Developers have been known to produce software systems that contain errors. These errors can be the result of simple coding errors, specification misunderstandings, incorrect specifications, or misunderstandings of language features, library interfaces, or other aspects of the software development environment, to name a few possibilities.
- **Vulnerability Insertion:** Malicious attackers may insert vulnerabilities into widely used subsystems so that they can successfully attack systems that incorporate the subsystems. One can view the vulnerability as simply additional undesirable functionality.

A disadvantage of this kind of functionality oversupply is that (from the perspective of any given user) it produces systems with large attack surfaces (each additional piece of functionality typically increases the attack surface) in which most of the attack surface comes from functionality that the user does not need and may not even be aware of. So automatic techniques that remove superflous functionality can significantly reduce the size of the attack surface and eliminate the corresponding vulnerabilities all without substantially impairing the utility of the system for the current user.

We also consider techniques that may affect desired functionality. The observation here is that users may be willing to accept different variants of a given piece of desired functionality. If this functionality contains a vulnerability, it may be possible change the functionality to eliminate the vulnerability while still providing acceptable service to users.

We next discuss several techniques that we have used successfully to change desired functionality or eliminate undesirable functionality.

2 Input Rectification

Most errors are exposed only by a few inputs — errors that occur on most inputs are usually detected and eliminated during testing. The goal of input rectification is to automatically convert inputs that expose errors into inputs that the system can process without error [14, 19, 16].

One approach is to identify a set of constraints that characterize the *comfort zone* of the software system — a set of inputs that are similar to those the system has seen before and for which it is almost certain to deliver expected and acceptable behavior [16]. The rectifier then automatically converts each input into an input that is within the comfort zone, typically by discarding pieces of the input that violate the constraints. The goal is to enable the system to process a safe input that is close as possible to the original input (and therefore should deliver most of the benefit to the user) while ensuring that the input is within the comfort zone of the program.

We have demonstrated that this approach can successfully eliminate vulnerabilities in the Pine email client [16]. The presented results use handcrafted rectifiers. We anticipate that it should be possible to build rectifiers automatically using the following approach:

- **Fault Attribution:** Given an input that exposes an error, use taint tracing [8] to identify the input regions involved in the computation that contains the error.
- **Constraint Synthesis:** Synthesize a constraint that the error-exposing input regions fail to satisfy.
- **Constraint Enforcement:** Perhaps using techniques similar to data structure repair [2, 6, 5, 4, 3, 3], deploy a constraint enforcement technique to automatically convert the input to an input that does not contain the error.

If successful, this approach would make it possible for a system to automatically analyze an attack to produce a rectifier that eliminates the attack from all future inputs.

3 Functionality Excision

It is often possible to view a computation as a collection of tasks [20]. It is possible to empirically partition the tasks in a program into *critical* and *forgiving* tasks [17, 1, 15]. Eliminating a critical task usually causes the computation to fail. Eliminating a forgiving task may introduce some noise into the result that the computation produces, but typically does not cause the program to fail [17, 15]. It is possible to generate behavioral variation by eliminating forgiving tasks, ideally under the direction of a blame assignment mechanism that analyzes a successful attack to find the task that it exploited. It is possible to view this mechanism as eliminating the functionality of the eliminated task. Once again, this mechanism may make it possible to vary the behavior of the system (in a directed way) to automatically avoid vulnerabilities.

It is also possible to apply this mechanism (at a potentially finer granularity) to less structured programs by excising code at the granularity of statements, basic blocks, procedures, modules, or other program units [14]. The basic idea is to find and eliminate parts of the system that contain counterproductive or undesirable functionality. Examples of potentially dangerous functionality that may be suitable targets for this mechanism include interpreters for embedded scripting languages and vestigial pieces of functionality left over from early versions of the system.

4 Functionality Replacement

It is often possible to find multiple implementations of the same functionality. Switching in different implementations can deliver combinatorially generated system variation that may change the system enough to neutralize an attack. We also en-

vision the use of machine learning techniques to automatically synthesize alternate implementations of different pieces of functionality. Even in situations in which it is difficult to automatically synthesize a complete version of the desired functionality, these automatically synthesized alternate implementations may enable the system to deliver acceptable service to its users while eliminating the vulnerability present in the original implementation.

5 Loop Perforation

Many programs contain loops. For many of these loops, reducing the number of executed loop iterations reduces the amount of time required to execute the computation. This transformation typically changes the result that the system produces. But it is often possible to find time-consuming loops which still produce acceptable results after this transformation [11, 9]. This mechanism can produce, automatically, a range of computations with different implementations that all provide acceptable results. If the current implementation of the loop has a vulnerability, it may be possible to eliminate the vulnerability by changing the number of iterations the loop performs. Consider, for example, a loop that copies data from one buffer to another. If the second buffer is too small to hold the data, eliminating a block of the last loop iterations may eliminate a buffer overflow vulnerability.

6 Dynamic Reconfiguration via Dynamic Knobs

Many systems come with static configuration parameters. Changing the parameter settings can often either expose or eliminate vulnerabilities — for example, misconfigured systems often exhibit vulnerabilities.

We have recently developed a technique that can automatically convert static configuration parameters into dynamic configuration patterns that can be changed without requiring the system to restart [10]. This mechanism should make it possible to automatically eliminate misconfiguration vulnerabilities without disrupting the execution of the system. It can also make it possible to dynamically change the configuration so that the system continually presents a different configuration to potential attackers.

7 Observed Invariant Enforcement

It is possible to observe normal executions of the system to build a model (in the form of a set of invariants) that characterizes that normal execution [13]. Because normal executions do not usually exhibit vulnerability exploitations, such exploita-

tions may fall outside the model. It is often possible to force the system back within its normal operating mode by changing the state to satisfy any violated invariants [13]. This invariant enforcement can eliminate otherwise exploitable security vulnerabilities [13].

8 Cyclic Memory Allocation

Memory leaks can cause a system to fail by exhausting its address space. It is possible to eliminate memory leaks via the simple expedient of statically allocating a buffer, then cyclically allocating items out of that buffer (instead of allocating a new element each time) [12]. While this allocation strategy may wind up allocating multiple live elements in the same buffer slot, the experimental results indicate that it can enable systems to survive otherwise fatal memory leaks while degrading gracefully in the presence of overlaid live elements.

9 Failure-Oblivious Computing

Memory addressing errors such as null pointer dereferences or out of bounds memory accesses can cause programs to fail and open up vulnerabilities for attackers to exploit. Failure-oblivious computing dynamically checks for memory errors, discarding out of bounds or otherwise illegal writes and manufacturing values for illegal reads. For the tested set of benchmark programs, this technique closes memory vulnerabilities and enables programs to provide service to legitimate users [18].

10 Conclusion

Security vulnerabilities can be seen as undesirable functionality present in a system. One way to eliminate such vulnerabilities is to change the functionality in a way that eliminates the vulnerability. We have identified and experimentally evaluated several mechanisms that can change the functionality of the program in ways that may eliminate security vulnerabilities while still leaving the program able to provide acceptable functionality.

References

1. Michael Carbin and Martin C. Rinard. Automatically identifying critical input regions and code in applications. In Paolo Tonella and Alessandro Orso, editors, *ISSTA*, pages 37–48. ACM, 2010.

2. Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. Inference and enforcement of data structure consistency specifications. In Lori L. Pollock and Mauro Pezzè, editors, *ISSTA*, pages 233–244. ACM, 2006.
3. Brian Demsky and Martin C. Rinard. Automatic detection and repair of errors in data structures. In Ron Crocker and Guy L. Steele Jr., editors, *OOPSLA*, pages 78–95. ACM, 2003.
4. Brian Demsky and Martin C. Rinard. Static specification analysis for termination of specification-based data structure repair. In *ISSRE*, pages 71–84. IEEE Computer Society, 2003.
5. Brian Demsky and Martin C. Rinard. Data structure repair using goal-directed reasoning. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *ICSE*, pages 176–185. ACM, 2005.
6. Brian Demsky and Martin C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Trans. Software Eng.*, 32(12):931–951, 2006.
7. Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors. *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. ACM, 2007.
8. Vijay Ganesh, Tim Leek, and Martin C. Rinard. Taint-based directed whitebox fuzzing. In *ICSE*, pages 474–484. IEEE, 2009.
9. Henry Hoffman, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures. Technical Report TR-2009-042, Computer Science and Artificial Intelligence Laboratory, MIT, September 2009.
10. Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Power-Aware Computing with Dynamic Knobs. Technical Report TR-2010-027, Computer Science and Artificial Intelligence Laboratory, MIT, May 2010.
11. Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin C. Rinard. Quality of service profiling. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *ICSE (1)*, pages 25–34. ACM, 2010.
12. Huu Hai Nguyen and Martin C. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In Greg Morrisett and Mooly Sagiv, editors, *ISMM*, pages 15–30. ACM, 2007.
13. Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. Automatically patching errors in deployed software. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *SOSP*, pages 87–102. ACM, 2009.
14. Martin C. Rinard. Acceptability-oriented computing. In Ron Crocker and Guy L. Steele Jr., editors, *OOPSLA Companion*, pages 221–239. ACM, 2003.
15. Martin C. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In Gregory K. Egan and Yoichi Muraoka, editors, *ICS*, pages 324–334. ACM, 2006.
16. Martin C. Rinard. Living in the comfort zone. In Gabriel et al. [7], pages 611–622.
17. Martin C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In Gabriel et al. [7], pages 369–386.
18. Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.
19. Martin C. Rinard, Cristian Cadar, and Huu Hai Nguyen. Exploring the acceptability envelope. In Ralph E. Johnson and Richard P. Gabriel, editors, *OOPSLA Companion*, pages 21–30. ACM, 2005.
20. Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.