A Framework for Testing Security Mechanisms for Program-Based Attacks

Ben Breech and Lori Pollock
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716
{breech,pollock}@cis.udel.edu

ABSTRACT

Program vulnerabilities leave organizations open to malicious attacks that can result in severe damage to company finances, resources, consumer privacy, and data. Engineering applications and systems so that vulnerabilities do not exist would be the best solution, but this strategy may be impractical due to fiscal constraints or inadequate knowledge. Therefore, a variety of program and system-based solutions have been proposed to deal with vulnerabilities in a manageable way. Unfortunately, proposed strategies are often poorly tested, because current testing techniques focus on the common case whereas vulnerabilities are often exploited by uncommon inputs.

In this paper, we present the *design* of a testing framework that enables the efficient, automatic and systematic testing of security mechanisms designed to prevent program-based attacks. The key insight of the framework is that dynamic compilation technology allows us to insert and simulate attacks during program execution. Thus, a security mechanism can be tested using *any* program, not only those with known vulnerabilities.

1. INTRODUCTION

Program vulnerabilities leave organizations open to attacks that can result in severe damage to company finances, resources, consumer privacy, data, etc. Exposing and identifying security vulnerabilities is notoriously difficult; research efforts in software testing focus almost exclusively on common case; i.e., the program behavior that users are likely to encounter when they use the program correctly. This approach is not conducive to exposing security flaws as vulnerabilities are typically found using inputs that users would not normally enter. Consider the typical stack smashing attack [1] which seeks to overflow a program buffer and trick the program into running arbitrary code. Such an attack would require the user to enter the binary code for particular instructions which is improbable at best.

The lack of testing strategies targeted towards security concerns results in the software community being more reactive than proactive with respect to security vulnerabilities. Software engineers currently have no easy way of testing for security problems, thus problems are typically found after the software has been released, unfortunately when they can result in large amounts of damage. Once a program's vulnerability has been discovered, programmers typically modify the code to add a security mechanism tailored to the known vulnerability and that program.

The best solution would be to engineer programs so that vulnerabilities are not present, but this is not entirely feasible, primarily because attackers continue to find new vulnerabilities. A variety of strategies for preventing vulnerabilities have been proposed (see, for instance, [14, 13, 23, 21]) involving all aspects of the program and its execution environment. These techniques can be broadly termed *program-based* as they focus upon the program or its execution environment.

Testing of these techniques is often poor. A program with a known vulnerability is found and recompiled with a particular protection scheme. The particular input that exploited the vulnerability is then provided to the program to determine whether the protection mechanism succeeded. This testing strategy does not inspire great confidence in the security mechanism since it could only be tried on a few programs with one particular input triggering one particular vulnerability.

In this paper, we present the *design* of a framework which enables the automatic and systematic testing of various security mechanisms. The key insight is that such mechanisms can be tested without resorting to specially designed test cases by utilizing dynamic compiler technology. Dynamic compilers are particularly well suited for this problem because they can enable a user to *modify program state and instructions* during the execution of the program.

The broad impact of this framework will be increased confidence in security mechanisms developed for program-based vulnerabilities as well as a framework for experimental investigation into new security mechanisms.

This paper is organized as follows: section 2 presents necessary background on program-based attacks. Section 3 gives the design of our framework, with section 4 showing an example of using our framework. Section 5 describes our prototype and preliminary results, while section 6 gives an overview of various issues we are currently addressing. Finally, section 7 presents some concluding remarks and work yet to be finished.

2. PROGRAM-BASED ATTACKS

Loosely defined, *program-based attacks* are attacks that are maliciously initiated on a program as input. These attacks are usually possible because of a vulnerability introduced by a programming flaw. This distinguishes the types of attacks, and associated preventative measures that our framework tests from other attacks, such as network attacks.

The most common program-based attack is the *buffer overflow* attack wherein an attacker attempts to write past the end of an array ("overflow"). One of the earliest examples is the Morris Worm [22], although there is anecdotal evidence of earlier attacks [11]. Part of the worm attempted to spawn a shell by overflowing a buffer in the `fingerd` Unix program.

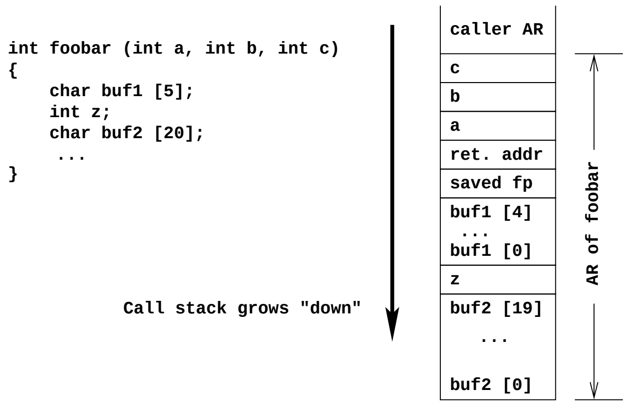
Languages that provide runtime array bounds checking will typically throw an exception when an overflow occurs. The exception can either be caught or will terminate the program. In other languages, especially C, writing past the end of the array will not necessarily cause program termination. Instead, other data is overwritten. This enables an attacker to write arbitrary values into variables.

A favorite target for buffer overflow attacks is stack allocated variables. These are variables defined locally in a function. Space for them is allocated automatically by the compiler as part of the *activation record* (AR) of the called function. Also included in the activation record are the parameters passed to the function and the *return address*, i.e., where program control should go when the function ends execution.

Figure 1 shows the beginning of a function code along with the associated activation record. By convention, call stacks "grow down" (i.e., the next AR pushed is at a lower address than the previous AR). When a call is made to function `foobar`, the caller pushes the parameters, from right to left, onto the stack,[1] followed by the return address. The next pushed item is the saved frame pointer (fp). The frame pointer is used to determine offsets to function parameters and local variables. The saved fp is the caller's frame pointer. The callee will create its own fp as part of the function initialization. Finally, the function allocates space for the locally defined variables, in this case, a 5 character array, `buf1`, an integer, `z`, and a 20 character array `buf2`.

An access to `buf2 [0]` through `buf2 [19]` would be legitimate as there is space allocated on the stack. However, accessing `buf [20]` would not be legitimate. This is a bounds error. C does not check array bounds, however, so the access is permitted. The actual used value will be 20 bytes (20 `* sizeof (char)`) away from `buf2 [0]`. This address references the first byte of `z`. A value written into `buf2 [20]` will overwrite the first byte of `z` causing mysterious errors. This is an example of a buffer overflow.

[1]This is the standard C calling convention. Other languages, such as Pascal and Fortran push parameters left to right.



```
int foobar (int a, int b, int c)
{
    char buf1 [5];
    int z;
    char buf2 [20];
     ...
}
```

Figure 1: Example activation record

We can continue this overwriting. We can overwrite the second byte of `z` by writing to `buf2 [21]`. By writing to `buf2 [24]`, we can overwrite `buf1 [0]`. Clearly we can place arbitrary values into `z` and also `buf1`. However, there is no reason to stop with overwriting `buf1`. Data can still be written past `buf1` and into the saved fp and, finally, the return address. By modifying the return address, an attacker can make program control jump to an arbitrary point and execute instructions. This is the classic *stack smashing* attack. Many tutorials [1, 15, 19] exist providing canned implementations of this kind of attack. Typically, the input used to overflow the buffer includes code to spawn a shell process ("shell code"). The return address is modified to point to the shell code, which is stored on the stack and gets executed when the function returns.

Stack allocated variables are not the only target. Attacks exist against variables stored on the heap [10], the frame pointer [18], allocated buffers upon the call to `free` and even format strings passed to the `printf` family of functions.

The most obvious technique for handling these vulnerabilities is to change programming practices so that common vulnerabilities are not possible. This can be accomplished by using a language, such as Java, that provides array bounds checking, adding bounds checking to the language [17, 3], or by auditing the code to find potential vulnerabilities and fixing them by using facilities, such as `string` in C++, that allocate more space as needed, or by carefully using "safe" library functions, such as `strncpy` in C. While this is the preferred approach, it suffers from several drawbacks: the performance penalty may be significant; the knowledge, experience and effort needed may be financially expensive, and, most importantly, this approach does nothing for legacy code or for software components for which source code is not available.

Mechanisms have been proposed to address these concerns [13, 16, 9, 12, 23, 2], but they are usually aimed at stopping one particular attack. The usual method for testing these protection mechanisms is to find programs with known exploits, add the mechanism to the program, and then run the exploit against the program to determine if the mechanism successfully prevented, or detected the attack.

3. FRAMEWORK DESIGN

Our research is focusing on building a security testing framework for applications that will provide automated, general support for security testing by allowing engineers to simulate particular attacks and examine how a proposed strategy protects their software. Furthermore, such a framework could allow an engineer to rapidly prototype a possible strategy to handle newly discovered vulnerabilities for which no strategy exists as well as attempting new attacks that have not yet been addressed.

Consider the typical stack smashing attack described in section 2. In the RAD security mechanism [9], copies of the return address are maintained in a special area of memory, a sort of 'ghost' stack. When a function is called, the return address is put onto both the call stack and the ghost stack. When the function returns, the return address is read from the ghost stack instead of the call stack, which overrides damage done to the return address by any overflow.

In order to test such a scheme, attacks against the return address must be performed. One method of attack generation would be to attempt to find buffer overflow vulnerabilities in a program, construct an exploit, and try it. This is a very time consuming task that, if done successfully, would eliminate the need for the security mechanisms in the first place. Another approach would be to modify the compiler to insert attacks. This approach requires the source to the compiler and also excellent knowledge of the compiler's source to insert the proper code. This can be too costly as different attacks require different modifications to the compiler.

The key insight for the framework is that it exploits the technology of dynamic compilation which has typically been used to optimize a compiled program as it executes. With the capability to monitor and compile on-the-fly, we foresee other uses for this technology, including the framework for security testing. By using a dynamic compiler, attacks could be inserted dynamically, without modifying the source code or binary stored on disk. Binary rewriting tools, such as DynInst [8], might also be used, although we feel that the dynamic compiler approach affords an easier implementation. Furthermore, this approach requires no modification to the static compiler, and engineers need not audit code bases looking for possible vulnerabilities.

Figure 2 shows the phases in a *dynamic compiler*. We stress that these phases are performed entirely at runtime. The input to the compiler is usually an intermediate program created by a static compiler, although compiling the source code on the fly is possible (usually in interpreters). If source code is being compiled, it is done on an on-demand basis, i.e., function `foo` will be compiled when `foo` is first called.

When execution first begins, the dynamic compiler performs any necessary source compilation. Sequences of instructions, called *basic blocks*, are constructed. A basic block is a contiguous sequence of instructions that would be executed sequentially by the CPU. A basic block can be optimized or analyzed *online* (during execution) and then given to the CPU for execution. The process of dynamic translation, basic block construction, online optimization and analysis and then execution keeps repeating until the program terminates.

Within the testing framework, the RAD scheme would be tested by compiling the ghost stack code into the original application and then running the modified code with the standard test cases through a dynamic compiler. The engineer can instruct the framework to modify return addresses of any or all functions; this allows the ghost stack idea to be tested in the general case rather than in specific instances. As the program executes, the framework would monitor the execution to determine how many attacks were made and how many were defeated. When the program is finished, the engineer would be provided with a coverage report detailing the methods executed by the test case, success rates of the attack and a vulnerability report detailing the parts of the program that are still vulnerable.

Thus far, we have discussed only the stack smashing attack. We will continue to use stack smashing as an example throughout the rest of the paper. However, we note that the framework is not limited to simulating only stack smashing attacks. The framework should also be able to simulate other attacks, such as those against function pointers. In general, we expect any attack that can be simulated by adding or modifying executable instructions to be handled. Attacks involving subtle data manipulations, such as the format string attacks against `printf` [20], may pose a problem for our framework.

The requirements we set for such a framework include

1. *Generality* – The framework must support a variety of source languages, different kinds of vulnerabilities and must be able to test a wide variety of program-based security mechanisms.
2. *Systematic testing* – The framework must be able to insert attacks at any appropriate point in the program. This allows for more thorough and systematic testing of program-based security mechanisms.
3. *Automatic* – A user should only need to specify the kinds of attacks that should be attempted and have the option of specifying the program points to be attacked. After that, the framework should perform all its tasks automatically without need for user intervention.
4. *Robustness* – The framework should report the places where the mechanism failed to protect the program while reporting few false positives. In our case, a false positive would occur if the framework reported the mechanism under test successfully protected the program where it actually did not.
5. *Low overhead* – The framework must have reasonable overhead, in terms of both space and time to be considered practical.

Framework Architecture. Figure 3 shows a diagram of

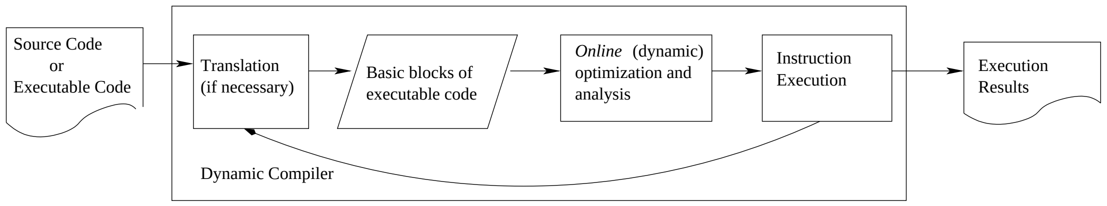


Figure 2: Dynamic Compilation Phases

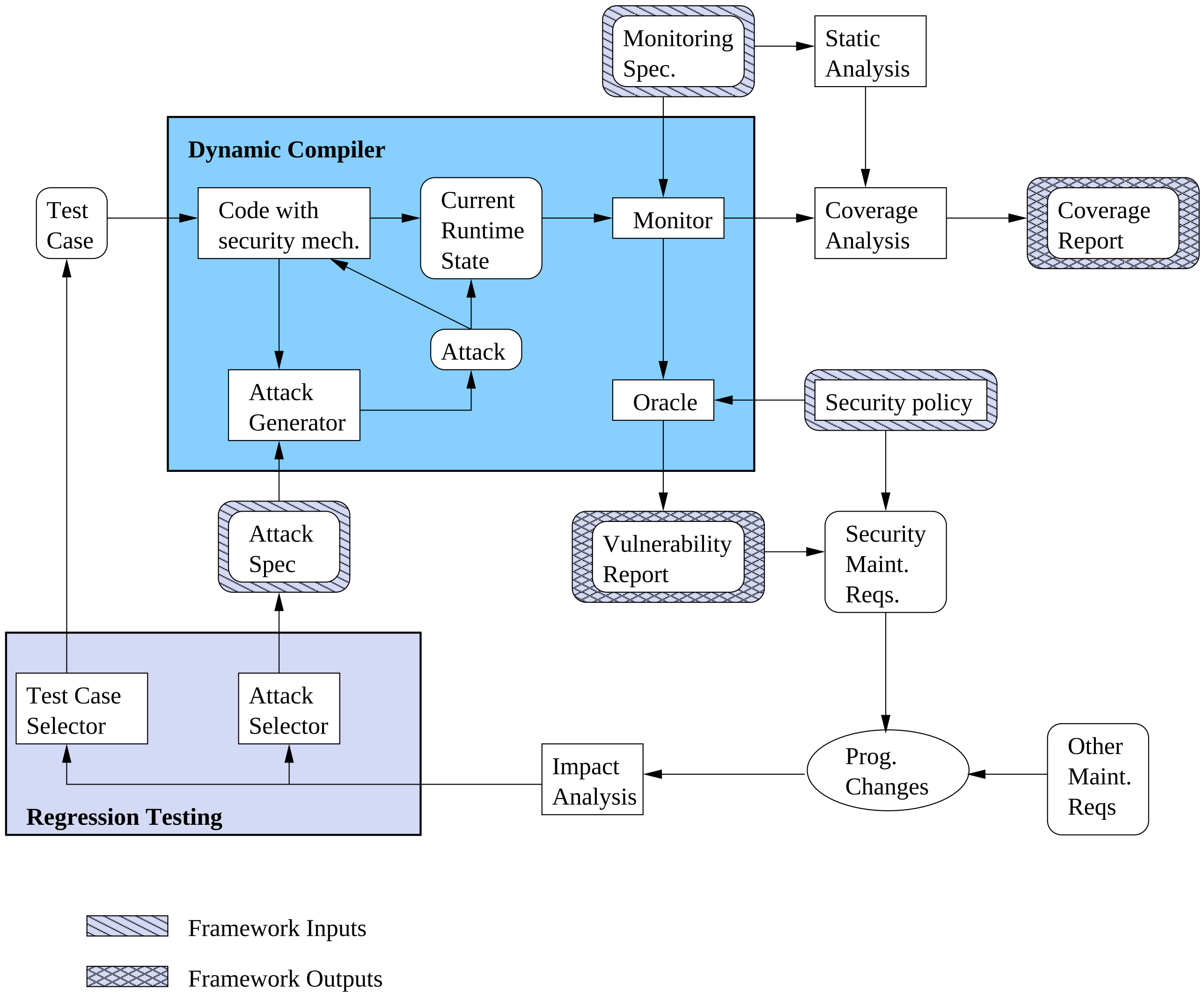


Figure 3: Proposed security framework

the proposed framework. The key components of the framework are modules within a dynamic compiler. During execution, the dynamic compiler gives the instructions that are about to be executed to each module for modification or analysis. The components are:

- *Attack Generator* – Responsible for producing and inserting the attacks. The attacks may modify program instructions or program state.
- *Execution Monitor* – Gathers information about the program's execution as instructions are executed. These instructions could have been modified by the attack generator.
- *Oracle* – Determines if the actual program behavior deviates from the expected behavior under the specified security policy.

Each of those components has inputs that are read upon the initialization of the framework. The inputs describe the specific actions that each component should perform. The inputs are:

- *Attack Specification* – Details the kind of attack that should be attempted, perhaps a change to return addresses or return values.
- *Security Policy* – Describes the proper behavior of the program e.g., calls that the program is or is not allowed to make.
- *Monitoring Specification* – Describes behavior that should be checked, such as a particular function being called that indicates the attack was successful. The monitoring specification is dependent on the attack specification and security policy.

There are two outputs produced by the components of the testing framework:

- *Vulnerability Report* – Describes the attacks that were successful, or the tests that failed.
- *Coverage Report* – Details the possible attack locations that were checked by the test suite. The coverage report is derived from coverage analysis and static analysis tools in conjunction with the execution monitor's output.

The other components of the framework deal with aspects of the software maintenance cycle. The vulnerability report along with the security policy may dictate that changes be made to either the program or the selected security mechanism. Other maintenance requirements may also dictate that program changes be made. These changes also must be tested. The test cases and attacks that should be re-run are selected during the regression testing phase.

4. THE TESTING PROCESS

As an example of testing a security mechanism within the framework, we consider the RAD example discussed earlier. We can test RAD by compiling it into any program that is convenient. That program is then run through a dynamic compiler with the framework modules loaded to test the mechanism. The program is run with a typical test case selected either by hand or by the test case selector. We note that the test case may be one that would normally be used in the testing phase. It does not need to be a case that attempts to exploit any vulnerability.

Upon starting up, the framework modules would read their particular inputs. Since RAD is designed to protect against the stack smashing technique, the attack specification would describe attacking the return addresses of a set of functions (perhaps the functions defined in the program, or all functions including those in libraries). The monitoring specification describes simply examining function returns.

During execution, the dynamic compiler starts creating basic blocks and hands them to the attack generator. The attack generator can examine each block to determine if it is the start of a function that is protected by the RAD mechanism, and thus, should be attacked. If so, code can be added to the start of the function to change the return address, thus simulating the effects of a buffer overflow attack. The monitor then examines the code. In this case, the monitor would record how many times return addresses were adjusted, the return instructions that were executed and the different return addresses that were used. The information on how many return addresses were modified and how many return instructions were executed will be passed to the coverage analyzer after the program has terminated. The information on what return addresses were used can be passed to the oracle, which then decides if an attack succeeded.

The entire process continues until the program terminates. Upon termination, the coverage and vulnerability reports are written. Based on these reports, the effectiveness of the RAD technique in preventing stack smashing attacks can be determined.

5. PROTOTYPE IMPLEMENTATION

We have started building a prototype of the framework. The DynamoRIO [6] dynamic compiler, which we have used for other work [4, 5], is the keystone of the framework. Figure 4 shows a simplified overview of DynamoRIO (readers interested in more details of the construction and underlying architecture of DynamoRIO should see [7]).

DynamoRIO is a native dynamic compiler that accepts statically compiled binaries as input. Thus, using DynamoRIO enables any compiled language to be used with the prototype testing framework. Additionally, DynamoRIO allows a user to write a client module to analyze or change instructions online. This capability provides a layer of abstraction to a user, enabling one to perform online analysis without modification to the DynamoRIO compiler itself. Thus, knowledge of the inner workings of the compiler is not required.

Upon startup, DynamoRIO initializes itself and then makes

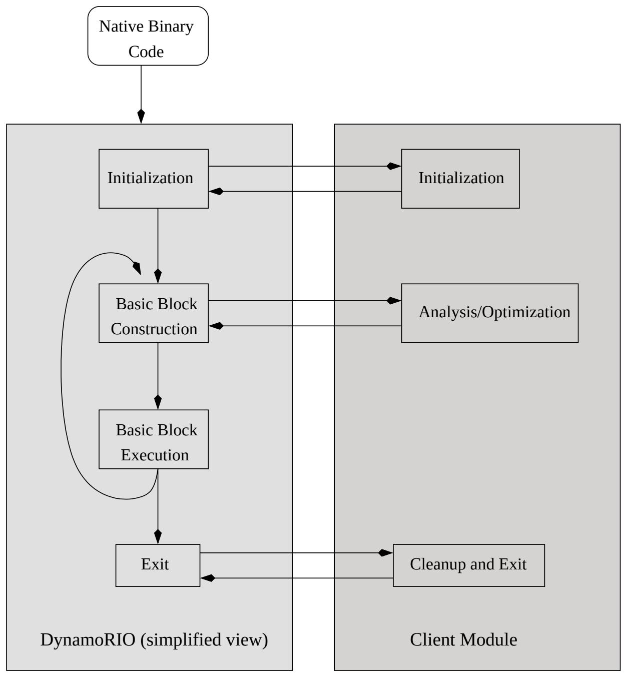


Figure 4: Simplified Overview of DynamoRIO

a call to the client module telling the client to initialize itself. DynamoRIO then begins executing code by building basic blocks from the program instructions. A *basic block* is simply a contiguous sequence of instructions that will be executed by the CPU. Each basic block is then passed to the client module for analysis or other adjustments. The block is returned to DynamoRIO, which passes the block to the CPU for native execution. This process continues until the program stops executing, at which point DynamoRIO calls the client module's cleanup routine and then exits.

The testing framework's key components can be implemented as a DynamoRIO module. Upon initialization, the framework inputs are read. During execution, the basic blocks are examined, and attacks can be inserted as desired. Additionally, the monitor can examine the blocks looking for the appropriate information. Finally, at exit, the framework writes the desired output reports.

The attack generator is currently in the early stages of implementation. Using DynamoRIO, we have successfully been able to simulate attacks against the return addresses in several test programs. This was done by writing a DynamoRIO module that determined when a particular function was called. Code was then added to modify the return address of a test function, `foo`, simulating the results of a stack smashing technique. The return address of the function `foo` was changed to be another function, `bar`, already in the test program. Upon return, `foo` jumped to `bar`. Thus a successful attack was carried out.

6. ISSUES

We acknowledge that there are several outstanding issues that we are currently addressing.

1. *Simulation of attacks.* We are using a dynamic compiler to *simulate* attacks by modifying existing code or by adding new code into the running program. A drawback to this approach may be that the generated attacks may not perfectly mimic existing attacks. For example, a dynamic compiler can insert code into a function to change the return address, which simulates the end result of a stack smashing attack. However, an actual buffer overflow will overwrite everything between the start of the buffer and the return address. Mechanisms, such as StackGuard [13], that rely upon this behavior may fail in the simulation. The solution to this issue remains to be investigated. It can be argued that the simulated attack is more subtle and mechanisms should be able to guard against the simulation (in the case of StackGuard, this would represent a case where the canary value was correctly determined by an attacker). Alternatively, the injected code could overwrite values between ranges, though this may be expensive to do.

2. *Efficiency.* In order to be practical, the framework must have minimal overhead. The attack generator and monitor examine instructions during their execution. If the overhead is too large, the performance penalty will make the framework almost useless.

3. *Monitoring Information.* The monitor is responsible for extracting information about the execution of the program during runtime. Its goal is to determine coverage information and to give the oracle enough information to determine if an attack was successful.

   In the case of stack smashing attacks, one possibility is for the framework to attack the program in such a way that a function is called completely out of order (e.g. for function `Q` to call function `Z` even though no code exists in `Q` to cause that call). The monitor could be looking for potentially out of order calls, which the oracle would then verify.

4. *Goals for the Attack Generator, Monitor and Oracle.* At this preliminary stage, there are two goals for the design of the attack generator, monitor and oracle: (1) no extra code should have to be compiled into the binary and (2) the program should not be terminated after the first successful attack. The first goal follows from the desire to have the framework perform its duties at any stage of the development cycle. This implies that a program binary potentially could be shipped to customers once it has passed the final tests performed by the framework. Furthermore, the framework could be applied to binaries that customers are actually using rather than trying to simulate the same conditions on a development workstation. Both of these ideas are compromised if the framework requires code to be compiled into the binary.

   The second goal is for the framework to keep performing even after a successful attack. The motivation is that to get a sense of how well a particular mechanism is protecting a program, all possible attack points must be examined. Killing the program early does not allow for this examination. The framework cannot simply jump to the `exit()` system call as part of a successful attack.

5. *Automatically specifying attacks.* Finally, issues regarding automatically specifying the attacks will be explored. It is simply unreasonable to expect a software engineer to devise code for the attack generator. Instead, a method allowing an engineer to provide some meta data about an attack, while the attack generator handles the details will be developed.

7. CONCLUSIONS AND FUTURE WORK

We have presented the *design* of a framework allowing the testing of security mechanisms for program-based attacks. Such a framework would allow for the more efficient testing of these mechanisms, without resorting to complex methodologies. The key insight of the framework is that dynamic compilation technology allows us to insert and simulate attacks during program execution.

Implementation work has begun on the framework itself, starting with the attack generator. Several simple attacks have been implemented on small test programs. These attacks need to be expanded in a general fashion to work with all programs. Research into the monitor and oracle will also performed, in particular, investing what information is needed to determine successful attacks and coverage information.

8. REFERENCES

[1] AlephOne. Smashing the stack for fun and profit. http://www.insecure.org/stf/smashstack.txt.

[2] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. *USENIX Annual Technical Conference*, 2000.

[3] R. Bodík, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. *Programming Language Design and Implementation*, 2000.

[4] B. Breech, A. Danalis, S. Shindo, and L. Pollock. Online impact analysis via dynamic compilation technology. *International Conference on Software Maintanence*, 2004.

[5] B. Breech, M. Tegtmeyer, and L. Pollock. A comparison of online and dynamic impact analysis algorithms. *European Conference on Software Maintenance and Reengineering*, 2005.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization*, 2003.

[7] D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, M.I.T., 2004.

[8] B. Buck and J. K. Hollingsworth. An API fro runtime code patching. *Journal of Supercomputing Applications and High Performance Computing*, 2000.

[9] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. *International Conference on Distributed Computing Systems*, 2001.

[10] M. Conover. w00w00 on heap overflows. http://www.w00w00.org/files/articles/heaptut.txt.

[11] C. Cowan. Re: Buffer overflow and the OS/390. http://cert.uni-stuttgart.de/archive/bugtraq/1999/02/msg00081.html, 1999.

[12] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. *USENIX Security Symposium*, 2003.

[13] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. *USENIX Security Symposium*, 1998.

[14] S. Designer. NonExecutable user stack. http://www.openwall.com/linux.

[15] DilDog. The tao of windows buffer overflow. http://www.cultdeadcow.com/c{D}c_files/c{D}c-351.

[16] H. Etoh and K. Yoda. GCC extension for protecting applications from stack-smashing attacks. http://www.research.ibm.com/trl/projects/security/ssp/, 2000.

[17] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. *Automatic and Algorithm Debugging*, 1997.

[18] Klog. Frame pointer overwrite. http://www.phrack.org/show.php?p=55&a=8.

[19] Mudge. How to write buffer overflows. http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html, 1995.

[20] T. Newsham. Format string attacks. http://www.lava.net/~newsham/format-string-attacks.pdf.

[21] R. Sekar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Model-carrying code (MCC): A new paradigm for mobile-code security. *New Security Paradigms Workshop*, 2001.

[22] E. H. Spafford. The Internet worm program: An analysis. *Computer Communication Review*, 1988.

[23] G. Zhu and A. Tyagi. Protection against indirect overflow attacks on pointers. *International Information Assurance Workshop*, 2004.