Program Differentiation

DANIEL CHANG, STEPHEN HINES, PAUL WEST, GARY TYSON, DAVID WHALLEY
Computer Science Department, Florida State University
Tallahassee, Florida 32306-4530, USA
{dchang, hines, west, tyson, whalley}@cs.fsu.edu

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Mobile electronics are undergoing a convergence of what were formerly muliple single application devices into a single programmable device – generally a smart phone. The programmability of these devices increases their vulnerability to malicious attack. In this paper, we propose a new malware management system that seeks to use program differentiation to reduce the propagation of malware when a software vulnerability exists. By modifying aspects of the control flow of the application, we allow various portions of an application executable to be permuted into unique versions for each distributed instance. Differentiation is achieved using hardware and systems software modifications which are amenable to and scalable in embedded systems. Our initial areas for modification include function call/return and system call semantics, as well as a hardware-supported Instruction Register File. Differentiation of executables hinders analysis for vulnerabilities as well as prevents the exploitation of a vulnerability in a single distributed version from propagating to other instances of that application. Computational demands on any instance of the application are minimized, while the resources required to attack multiple systems grows with the number of systems attacked. By focusing on prevention of malware propagation in addition to traditional absolute defenses, we target the economics of malware in order to make attacks prohibitively expensive and infeasible.

Keywords: Program differentiation; Malware prevention and mitigation; Return address indirection; System call indirection.

1. Introduction

Like general purpose computing systems, mobile devices and the software loaded on these devices are subject to a host of security threats and malicious software (malware) attacks due to vulnerabilities in their coding. Solutions to preventing malware become more challenging as the complexity and interconnectivity of these systems increase [36]. The increasingly complex software systems used in modern smart phones contain more sites for potential vulnerabilities, a problem exacerbated as application developers continue to integrate third party software with *plugins* for such user applications as web browsers and search engines. Recent exploitations of Google Desktop, Microsoft Internet Explorer, and MobileSafari on the Apple iPhone are examples [25, 24, 12, 21, 20, 14].

Rootkits are a grave concern due to their tenacity, detrimental effect on systems, and difficult detection. Typically they target kernel vulnerabilities to infect system-level programs and to conceal their existence. The rootkit applications themselves include key loggers, network sniffers, and a staging system to launch other attacks like Denial-of-Service and more. The primary use of Rootkits is to inject malware and to collect sensitive user information. This is especially problematic for mobile devices that are increasingly used to store private data.

Traditional approaches have sought to provide an absolute defense to specific malware attacks by patching software vulnerabilities or detecting and blocking malware [28, 27, 6, 5, 16]. However, the current situation represents a programmatic arms race between the patching of existing vulnerabilities and the exploitation of new ones. Despite increased awareness, vulnerabilities continue to be produced, as observed in McAffee's position paper citing Windows Vista as being less secure than its predecessors [18, 23, 34]. Most recently, the modified Mac OS X system on the Apple iPhone fails to even implement widely accepted best practices such as a non-executable heap or address randomization of memory area starting locations [20, 14]. Ultimately vulnerabilities will be found and malware will go undetected long enough to exploit such. We propose a different approach to managing malware based on limiting the ability of viruses to propagate even in the presence of undiscovered software vulnerabilities. When used in conjunction with traditional malware defenses, this approach greatly increases the difficulty and cost for malware developers to exploit vulnerabilities across a wide range of vulnerable systems.

Mitigation through the use of program differentiation has an analogue in biological systems, which not only presume attack will occur but in fact have well-known, openly visible vulnerabilities [31, 9]. Beyond protective walls, biological entities also rely on a system that mitigates subsequent proliferation of biological attacks. Biological systems defend both at the individual level and the population level. While anti-virus software can convey individual system protection, they do nothing to limit the rapid propagation of new viruses across a large set of homogeneous application code [40, 22]. The correlation between malware propagation and resulting damage leads us to explore mitigating attacks by thwarting the propagation.

Program differentiation seeks to make each executable instance of an application unique. There are various ways of achieving this in software. One simple method would be to invoke different compiler transformations, or a different transformation ordering, to obtain different versions of the same application. However, this approach has two problems that make it infeasible. First, the vulnerable portions of the application must be the ones affected and there is no way to guarantee those unknown vulnerabilities are modified by this differentiation approach. The second problem is a more severe software engineering one. Multiple distinct versions of the same program code can be difficult to produce and highly impractical to maintain and update; subtle software errors in an application will likely change behavior in only a subset of differentiated versions. Performance may also differ widely between instances of the program. A solution is needed that differentiates program executables while preserving program semantics and control flow, and maintaining a single code profile for maintenance. We propose changes in hardware support for control flow instructions to achieve differentiation that changes the binary representation of applications without changing the execution order of instructions in the processor pipeline. The hardware modifications change how functions are called (and returned), how system calls are specified and how instructions fetched from memory are interpreted. This prevents the propagation of viruses by making each instance of a vulnerable application sufficiently different to require a nontrivial mutation of malware code for each infection. Figure 1 depicts the general workflow of our proposed methodology. Differentiation can occur before distribution, during initial configuration or even at load time for each invocation of the application. Whenever performed, differentiation modifies the original application using an undifferentiated instance and a configuration database, generating a potentially unique executable for each application instance.

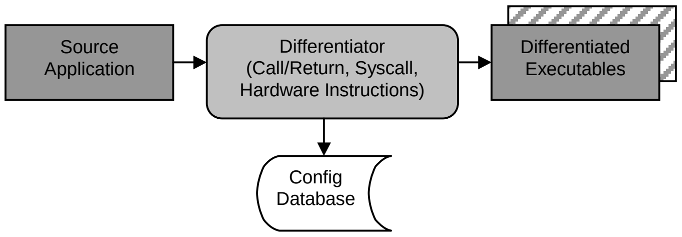
Source Application
Differentiator (Call/Return, Syscall, Hardware Instructions)
Differentiated Executables
Config Database

Figure 1. Software Differentiation Workflow

Google's Android operating system for mobile devices is a prime example of the new breed of embedded systems we seek to reinforce, and thus is also an ideal candidate for our differentiation approaches. Android is a Linux-based operating system targeted at cell phones, tablets and other mobile systems. The basic system includes a Linux version 2.6 Kernel, basic libraries, and an application framework that provides core Java programming language functionality. Java applications are then built on a layer above the core system, with each running in its own process within an instance of the Dalvik virtual machine [1]. Nevertheless, given Android's open architecture and consumer product platform, the libraries and framework provide broad access to system functionality including phone calling, GPS systems, and process control. As of the first quarter of 2010, sales of Android-base smartphones exceeded iPhone OS units, commanding a 28 percent market share and making it the second best-selling smartphone system in the quarter behind RIM's Blackberry OS [19].

Android-based devices, like mobile devices in general, are a growing target for attack due to their large and ever-growing installed base as well as their inherent network access. Given that an Android device includes the open source operating system as well as applications, it provides us with an ideal platform for our approaches, which integrate operating system level enhancements with application level differentiation.

2. Software Exploits

Vulnerabilities in software have not only proven costly, but are continually increasing in number [36]. They can be broadly separated into processor architecture exploits and higher-level software exploits that are independent of the target architecture. High-level techniques are generally attacks on input strings that are interpreted within a source level interpreter in the application, such as overwriting of an SQL command string with an unauthorized set of commands to compromise the database system. Mitigation of such attacks must be handled by the application and further coverage is outside the scope of this paper's processor-level security systems. By far the most common security exploits related to processor architecture are buffer overflow attacks, which overwrite memory locations in vulnerable applications [6, 32]. These attacks exploit vulnerabilities in the control flow conventions of the target architecture to gain control of system resources. The vulnerabilities exploited by the highly prolific Code Red worm and recently discovered in the Apple iPhone browser are examples of buffer overflow vulnerabilities [40, 22, 20, 2]

These attacks require an intimate understanding of the program code, including data and program object locations, branching, and address usage. The attack requires an unbounded input buffer, used to insert a payload of malicious code, and a vulnerability that allows the control flow to execute the payload code. Unfortunately, distribution of identical versions of a software executable facilitates propagation of a successful attack; once a vulnerability is found and exploited the attack is applicable to every other distributed instance. It is this commonality that allowed the 2001 Code Red worm (CRv2), which exploited a vulnerability in Windows NT and Windows 2000 operating systems, to infect more than 359,000 Internet Information Servers on the Internet in less than 14 hours [40, 22]. Mitigation techniques, such as hardware restrictions that disable execution of code from within the stack region of memory, seek to eliminate the vulnerability. Unfortunately, there are buffer overflow variants that do not require the insertion of payload code, but instead jump to existing routines in the application to compromise the system. Commonly referred to as *return-to-libc* attacks, the target address in the application code is often a library routine to manipulate systems components (such as to invoke a shell or delete files).

Moreover, despite advances in security methodologies in software development the prevalence of software vulnerabilities continues to grow. This trend will likely continue as software systems become more numerous and complex. Greater opportunities for new vulnerabilities arise from the tremendous growth in high-demand, third-party software applications that require administrative privileges or trusted access. The Google Desktop application demonstrated several vulnerabilities in user-level software that had system-level implications. A flaw discovered in December 2004 allowed malicious websites to illicitly read local search results from a target machine [25, 24]. The following year, a flaw in the Internet Explorer web browser combined with Google Desktop allowed an attacker to retrieve private user data or even execute operations on remote domains while impersonating the user [12, 21]. The extreme integration of third-party applications is another growing threat, as demonstrated by the recent discovery of a buffer overflow vulnerability in the MobileSafari browser on the new Apple iPhone [20]. Through the exploit, a malicious web site could deliver a payload that allowed access to and transmission of any phone data [14]. The threat of malware on cell phones has loomed large in the past few years as they increase in computational power. Now that modern cell phones are simply full-fledged computer systems they are subject to computer system threats, only at much larger distribution scale [29].

Unfortunately, current anti-virus support is limited to identifying existing vulnerabilities or a few restricted patterns of attack. This approach means that systems security is always lagging behind the discovery of new vulnerabilities and fast propagation can defeat even the most active malware defense. Differentiation offers the promise of either eliminating or significantly slowing propagation independent of the type of software vulnerability exploited. Used in conjunction with existing mitigation techniques, program differentiation provides the strongest deterrent to the spread of future malware.

3. Differentiation Sources

The goal of differentiation is to restructure each instance of an application in a manner that makes the exploitation of inherent vulnerabilities in either the application or execution environment more difficult. At the same time, the implementation of a differentiation technique should not hinder application maintenance, change functional behavior, or result in dramatic performance differences between instances. Particularly in embedded systems, the ideal differentiation technique should have minimal to no impact on performance and be scalable to the available resource budget. Finally, the overhead required to support differentiation should be minimal in both space and execution time. We propose three independent mechanisms to provide differentiation using a combination of hardware and software techniques. Each of these techniques utilizes indirection and by permuting indices supports differentiation. The first two schemes manipulate function call and system call semantics using both hardware and software modifications. The third scheme modifies how instructions are interpreted when fetched from memory and provides additional restrictions on execution of the most vulnerable instructions. These mechanisms are not only orthogonal to each other, but can be used in conjunction with all other available protection schemes.

3.2 *Return address differentiation*

Our first approach is to introduce a level of indirection into the function call return address stored on the stack. The return address is the typical target of buffer overflow attacks, which attempt to overwrite the address to point to a payload placed in the buffer. By replacing the return address with an index to a table of return addresses, we prevent the injection of a direct address and instead force the attacker to analyze the behavior of the new Return Address Table (RAT). This requires modification of function call and return semantics to access the return address through the RAT. Function calls must utilize a register to pass the index in much the same manner as the return address is currently passed. The return instruction must be modified to use the index to load the return address from the RAT before jumping to the instruction following the function call. Any buffer overflow would now override the RAT index. Without knowledge of the ordering of return addresses in the read-only RAT, the attacker can only jump to a random return address location in the existing code. With some modification to memory management access, the table itself can be marked unreadable by all instructions except function return. This removes code inspection from the arsenal of the malware attack. Statistical attacks using random indices can be thwarted by increasing the size of the RAT. The cost paid is a fixed increase in storage requirements for the return address table and a slight performance penalty on each function return due to the required table lookup. Instruction set modifications include the *call* or *jalr* instructions and *ret*. Calling conventions replace the automatic movement of the program counter for the next instruction (generally *PC+4*) with an index specifying the RAT entry containing the address of the next instruction. This is performed by an additional instruction, though in many cases this additional instruction is loop invariant and can thus be performed much less often than the function call. The call instruction could be totally eliminated; however, this would have implications on micro-architectural resources like the return address stack. Remaining calling conventions remain unchanged. The *ret* instruction is modified to first read the index off the stack, and then use the index to load the return address from the RAT. In a *load-store* architecture, this would be performed by multiple instructions. In either case, the performance impact is less than expected since the branch prediction will continue to utilize the return address stack, which generally contains the correct address.

Modifications to the function return code sequence are outlined in Figure 2. The return behavior is updated to: 1) retrieve the index from the program stack; 2) access the RAT to obtain the return address; and then 3) jump to the return address. The RAT itself is made read-only, and in retrieving the return address bounds checking can be imposed by using logical instructions to mask the index value and prevent the use of any out-of-range indices. Even a small number of RAT entries results in a combinatorial number of permutations. This directly attacks the economics of a malware attack, making a random attack extremely unlikely to produce any useful predictable behavior and grossly thwarting the ability to have a wide-ranging impact with a single system attack. For the attacker who wishes to analyze software or a system to improve malware success and propagation, the permutation complexity elicits a signature analysis behavior that is easily detectable by traditional intrusion detection systems. Still, some consideration must be given to the use of shared library routines. Since these functions are shared by different applications, care must be taken when using any process resources, particularly the RAT. If all system processes use the differentiation calling conventions then libraries pose no difficulties; RAT indices are evaluated in the context of a process and each process contains a unique RAT. Returns from library

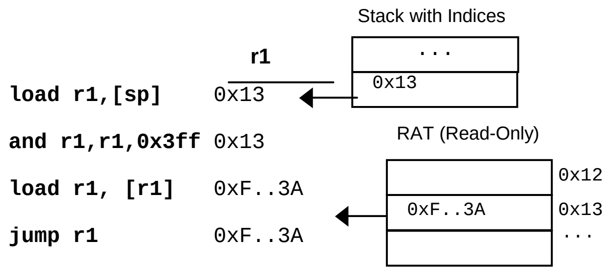

Figure 2. Secure Return Code Sequence

functions would still use the index into the process RAT and jump to the correct location. In a system where not all applications use differentiation call semantics, the return from a library function would then be undifferentiated, using the return address on the stack and jumping to the correct location. This mixed environment does not offer the same level of protection as a fully differentiated executable; however, only library routines are left vulnerable and ideally they would tend to be a more stable code base not as easily targeted by malware.

3.2 *System call differentiation*

Another candidate for indirection are system call conventions, which specify a system call identifier generally passed using a specific data register. System calls can be used to manipulate files, memory or process permissions, and can be compromised by malware that changes the system call identifier prior to execution. System calls are implemented by jumping to a function in the operating system (*entSys()* in Linux), which then uses the system call identifier to index into a jump table to the correct handler function. We thus have the same basic approach as with the RAT. Differentiation of the table (the *sys_call_table* in Linux) will provide a different mapping of system call identifiers to handler functions for each system (not each application) at no additional overhead. The only requirement to perform system call differentiation is to permute the entries in the *sys_call_table* and update any system calls in the applications. It is quite rare for an application to directly reference a system call since almost all calls are performed in the standard systems libraries (*libC* and others). This simplifies the differentiation process. Of course, this approach means that all applications running on the system share the same system call identifiers. This is likely not a problem since viruses tend to propagate by infecting the same application on different systems, not by attacking different applications on the same system. However, by duplicating the *sys_call_table* for each process, differentiation can be performed for each process on the same system. The only additional requirement is an increase in the stored state of the process and de-referencing of the process table pointer in *entSys()*. This approach again provides a level of indirection that requires an attacker to now gain access to a particular executable's custom system call table in order to identify targets for control flow redirection. Furthermore, the custom *sys_call_table* can be pruned to only contain those used by the particular program, reducing malware ability to initiate unexpected system calls. Figure 3 depicts the general process of accessing the lookup table containing the system call specifier, which ultimately results in one additional function call and one additional load from a table in

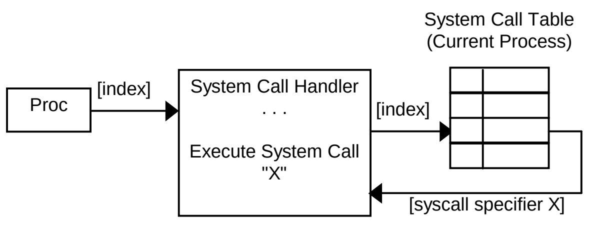

Figure 3. System Call Table Lookup

memory. Normally a register contains the identifier for the system call to perform. We would differentiate modules by replacing the identifier with an index into the system call lookup table.

3.3 *ISA differentiation*

The final modification to support differentiation provides the strongest level of protection. Instructions can be obfuscated by using a level of indirection in the decoding of instructions fetched from memory. This enables a portion of the Instruction Set Architecture (ISA) encoding to be changed for each program instance while leaving instruction execution unchanged. Keeping the decoded/executed instruction stream the same allows software engineers to more easily maintain an application, since any version can theoretically be transformed into a different version by applying the appropriate mapping of indirection specifiers. This facilitates the debugging and patching of differentiated executables, a task easily achievable using previous techniques. There have been several possible approaches for indirectly accessing instructions that could support differentiation. Computational accelerators fuse multiple operations into single operations by providing a programmable set of functional units [4]. An accelerator could be configured to use only simple instructions with a specific new opcode/operand encoding that could vary amongst differentiated executables. The FITS system allows for mapping of an ISA, customized for a particular executable, to a configurable processor [3]. The programmability of opcode and operand decoders in FITS allows for their permutation in the instructions supported by the ISA. Both, however, have drawbacks in either potential performance penalties or increased implementation time.

Ultimately, a most attractive option is instruction packing, a technique that can be readily adapted to provide differentiation at the hardware instruction level [15]. This technique promotes frequently occurring static and/or dynamic instructions into instruction registers, which can then be indexed for execution by using just a few bits. The small size of these indices allows multiple such references to be "packed" together into a single 32-bit instruction. Parameterization of register numbers and immediate

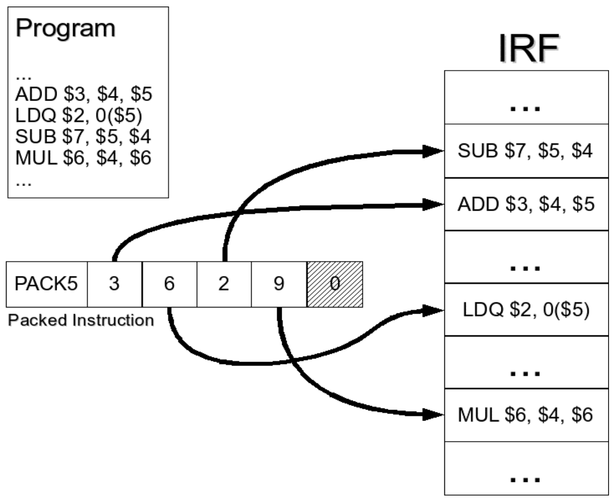

Figure 4. Indirection with an IRF

values increases the number of instructions that can be promoted. This reduces code size and improves energy efficiency, as the Instruction Register File (IRF) requires less power to access than the instruction cache. Using instruction packing, the indices of the packed instructions can be permuted to generate new executables. With a 32-entry IRF (and one instruction register reserved for a *nop*), there are 31! possible permutations, leading to quite a large space for differentiation of a single application. Figure 4 shows an example program being permuted within the IRF. The four instructions are mapped into the IRF and the appropriate identifiers are specified for the packed instruction. Since we only have four instructions to pack together, the fifth slot is mapped to the *nop*, which need not necessarily be at entry 0. Packing instructions with an IRF is also the least intrusive solution, as it requires the fewest changes to the baseline ISA. The tightly packed instruction format can be supported using just a few spare opcodes in almost any existing ISA. An IRF needs to be added to the pipeline, and the fetch and decode stages need to be modified to be able to fetch instructions from it. Instructions are placed in the IRF at load-time for an application, and must be restored on context switches. This allows separate applications to have completely different IRF entries. Instruction packing clearly satisfies all of the necessary requirements for providing an easily permuted instruction indirection scheme. Furthermore, the additional energy and code size savings (with no performance overhead) make this technique even more attractive for implementing hardware instruction level differentiation, especially on restrictive embedded systems. The IRF structure also allows for scalability, balancing the size of the IRF with the amount of differentiation desired. [15].

Using an IRF to support differentiation also provides an additional benefit for protecting code from malware. Since the IRF provides a totally independent way to specify the instructions that reside within it, it is possible to disallow those instructions from being fetched directly from the memory system. So if we always promote certain critical instruction into the IRF, then we can execute the processor in a safe mode that would not decode those actual instructions when being fetched from the memory system. By targeting syscall, call, return, adds to the stack pointer and short conditional branches, it becomes difficult for any payload malware to perform critical, or even common, instructions without identifying the IRF permutation. Additionally, empty IRF operands can be used to verify proper control flow with the inclusion of a simple validating state machine in the beginning of the processor pipeline. This could be as simple as requiring some parity calculation for the instructions. This has little or no impact on application performance, and while the virus can replicate the calculation, each instance of the application can use the free IRF encodings for different validation checks. For the malware to propagate, the virus must correctly handle an arbitrarily large number of validation checks. Again, this technique offers scalability in an embedded system up to the level of desired or allowable protection.

3.4 *Intrusion detection with Micro State Machine differentiation*

We also propose developing a variety of small state machines that monitor some identifiable pattern of behavior of the application, such as sequences of system calls or numbers of arguments in function calls within a program. These Micro State Machines (MSMs) can be defined at compile-time or load-time, and ultimately can be easily implemented in hardware without any effect on the pipelined instructions of a software program. A wide variety of micro state machines can be made available, executed by referencing an index from within a software executable. These include any state monitoring patterns proposed in the previous literature. By requiring a program to execute any or all of these micro state machines, the integrity of program behavior can be checked outside of the regular program execution. Given the wide variety of small state transitions that can be monitored, differentiation is introduced into applications by varying the particular MSM called for a given executable version and/or varying the order of calling multiple micro state machines. Indeed, each application instance may only need to verify one aspect of the entire state monitoring problem, with variation of which MSM is implemented in any given instance. This comes with no direct performance penalty and forces an attacker to have to understand and thwart the monitoring of a large number of state transition behaviors. The possible combinations of state monitoring are ideally too large for any malware to comprehensively determine, and the act of attempting to do so is a detectable behavior pattern that can be further used to identify the presence of malware.

The choice of MSMs is limitless given all the system processes and properties that can be tracked and checked for inconsistency. Note that we are not requiring that a given verification be perfect or guaranteed - that is, the particular section need not be invariant nor must any attack be guaranteed to modify it. Just as with intrusion detection systems, in this method we seek to increase the probability of detecting threats and thereby responding to them. Subject to resource restrictions we increase the strength of hardening by increasing the number and variety of MSMs and the depth to which any single MSM is implemented. At this application level the saving of data and restarting of verification for context switching must be considered. By parameterizing an MSM to support specification of a starting state, the MSM can be properly continued after context switches.

An example addressing code driven by direct control, such as application code, is the tracking of a portion of an application's call graph. With compiler support analysis can be performed on an application with the aim of identifying portions that are attractive targets with low variability. An MSM can then periodically verify the actual call behavior of the running process.

Internal events driven by code such as system operations represent another different pool of data for MSM analysis. For example, translation look-aside buffer (TLB) behavior can be monitored and verified by an MSM for generally anomalous behavior across all user applications. A TLB miss that occurs long into the execution of an application would represent code that had never been previously executed suddenly being called, which suggests injected code. Of course, this can sometimes be correct behavior as in the case of just-in-time (JIT) compiled code, so careful analysis for such an MSM must be made, for instance to only be sensitive to TLB misses from particular ranges of memory.

Analyzing system wide operational semantics represents another higher level where we can achieve monitoring across all user applications. Common SQL injection attacks can be detected by searching database-destined values for special characters or embedded logic statements. But another attack behavior is the systematic delivery of varied inputs to evince changes in output, typically when direct viewing of SQL query results is not visible. Repeated executions of the same SQL query code can be a semantic checked by an MSM when not expected in the normal operation of a particular application. The behavior and state of virtual machines executing processes also represent system semantics that can be validated by an MSM. The Android operating system provides a prime opportunity for such monitoring, as it uses a virtual machine for each running process. Validation of erroneous behaviors can be monitored, as well as execution of code from memory regions not belonging to the specific application within the virtual machine. Since each running process is in a virtual machine, operations invalid for the specific application can be documented and tracked, such as phone call access from applications that have no business accessing the phone. Other behaviors such as the previously mentioned TLB misses can also be monitored specifically relative to the application within a given virtual machine.

The variable selection of micro state machine types and quantity again aids embedded system designers by providing a scalable defense method. Increasing the number of micro state machines results in a smooth increase in the level of defense. The number can be increased up to the desired defense level, or as is more likely the case up to the limit of available resources for defense. Ideally, if an IRF is employed the unused slots in non-fully packed instructions are perfect locations for placing the triggers or counters for implementing such micro state machines. These triggers can be sought during the pipeline fetch phase with no direct penalty on the application processing performance.

4. Evaluation

We have thus proposed several methods for implementing differentiation of software executables using hardware support. To perform an initial test of the worst-case effect on performance of our approaches we developed software versions of two of the approaches, implementing both a Return Address Table (RAT) and a Linux kernel modification implementing system call indirection, targeting binaries to the Alpha architecture. Both were evaluated using the M5 Simulator, a modular platform for computer system architecture research, encompassing system-level architecture as well as processor microarchitecture [35]. M5 supports the Alpha architecture and has a system-call emulation mode that can simulate Alpha binaries. Moreover, M5 provides a full system simulator that models a DEC Tsunami system in sufficient detail to boot a Linux kernel. We specifically used the M5 full system simulator to evaluate the results of our Linux kernel modifications to support system call indirection.

4.1 *Function call return address*

Our actual implementation of the RAT involved modification of a GCC version 4.0.2 cross-compiler installation using glib version 2.3.6., configured to produce Alpha executables. We inserted a program in between the compilation and assembly stages (just before the execution of the *as* assembler program), which post-processed all return (*ret*) instructions in user code to rewrite the program assembly. In addition, an array to hold return addresses was linked in with every executable program. The address retrieved from the stack as the "return address" is actually treated as an index into the RAT. Logical shift instructions are inserted in order to isolate the portion of the address representing the index. The resulting index is then combined with the base pointer and offset of the starting point of the RAT. A single load instruction is then inserted to retrieve the actual return address to be used. The resulting post-processed Alpha assembly file can then be assembled and linked by the remaining GCC compile chain.

In Alpha assembly *ret* instructions use a return address stored by convention in register $26. Figure 5 shows the assembly added by our post-processing application to replace the normal *ret* instruction. The value being passed through register $26 is now an index into our RAT, which we implemented with 1024 entries. Since the maximum literal size in Alpha assembly is 255, we use logical shifts to isolate ten bits in position 12-3, zeroing the three least-significant bits since the index must be quad-word aligned. Once the index is identified, it is added to the base address of our RAT as well as the base pointer to obtain the location in

```
# $26 holds value originally passed as return address
# isolate bits 12-3 to get a 1k address to a quadword
# maximum literal size is 255, so shift left then right
# must be quad word aligned so zero right 3 bits
srl $26,3,$26    # >> 3 to get on quad boundary
sll $26,57,$26   # << 3 then 54 to isolate ten bits
srl $26,54,$26   # >> to get 10 bits pos [12-3]

# load from return address table
# base of table+index must also be offset from bp $29

addq $26,$29,$26 # add base pointer to index

# add table offset to base+index
ldq $26,ratable($26) !literal

# perform return using retrieved address
ret $31,($26),1
```

Figure 5. Return Address Post-Processing

memory from which to retrieve the actual return address. A simple return is now performed, with register $26 now containing the actual return address as retrieved from the RAT. Differentiation of multiple software executable versions is achieved by permuting the order of return addresses in the table.

We compiled several benchmarks from the MiBench benchmark suite [13] using our modified GCC compiler and executed them, verifying their essential correctness. The resulting increases in instruction count and committed loads are presented in Figure 6. Instruction count increased by only 1.04% on average, with this including a single outlying increase of 4.32% for the Telecomm benchmarks. Among the other benchmarks the highest increase was only 0.31%. The increase in loads showed somewhat similar behavior, with an average increase of 3.38%, which includes two extreme cases of a 7.75% and 8.13% increase for the Office and Telecomm benchmarks respectively. Among the remaining benchmarks the highest increase was just under 0.6%.

4.2 *System calls*

Our actual implementation for system call differentiation involved modification of the Linux 2.6.13 kernel distribution provided with the M5 simulator. Each running process is associated with a *task_struct*, which is defined in the scheduler. We modified the scheduler to include an array that would hold a unique copy of the system call table for each process, which is populated when a new process is created. Since the actual handling of system calls occurs in assembly routines, we added a

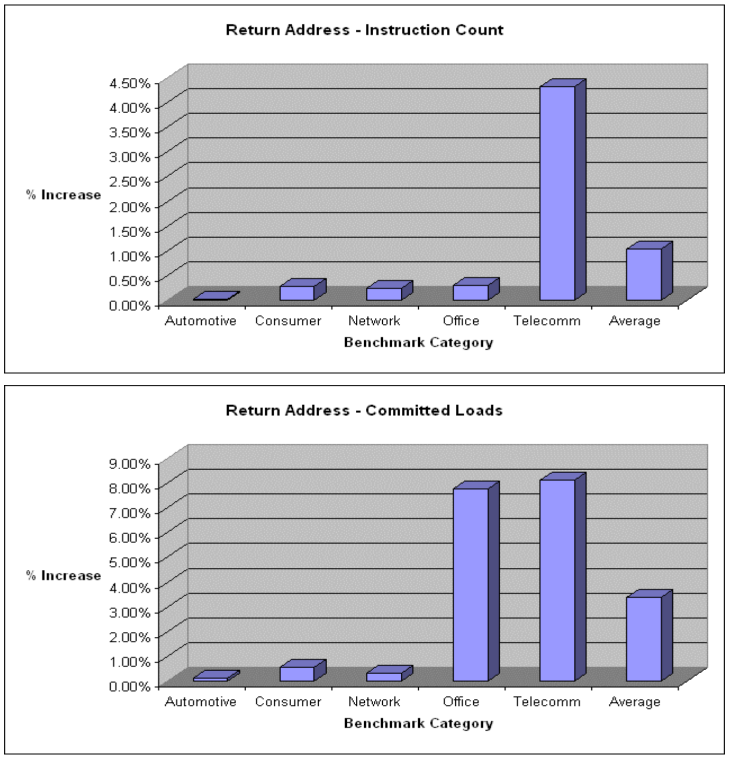

Figure 6. Return Address Table Simulation

function to the scheduler that is visible from the assembly language routines and that returns entries from the system call table. We modified the system call handler to use the current system call specifier (normally an index into the default system call table) and pass it to our function. The function uses the value as an index into the system call table for the currently running process and returns the corresponding actual system call specifier, which is then used to make the system call. The most significant changes needed for implementation of system call tables involve modification of the system call handler assembly routine. Figure 7 shows the modifications to the applicable assembly source file in the Linux kernel (new code italicized). Originally the system call identifier (an index into the original system call table) is passed through register $31, but now the register contains an index into the system call table for the currently running process. We set this index as an argument and call the *get_cur_sys_tbl* function we created in the scheduler to obtain the system call table from the currently executing process and retrieve the correct system call identifier corresponding to the index argument. The retrieved system call identifier is then used to dispatch a system call in the normal fashion. Differentiation can be achieved by permuting the system call table contents for each process, thus changing all the system call specifiers used within the actual machine language of each executable version. The functional behavior of each executable is unaffected, since any two permutations of system call tables will ultimately result in the same actual function call being executed at the same points in the control flow.

We ran unmodified, Alpha-

```
entSys:
    SAVE_ALL
    lda     $8, 0x3fff
    bic     $sp, $8, $8
    lda     $4, NR_SYSCALLS($31)
    stq     $16, SP_OFF+24($sp)
    /* remove (lda $5, sys_call_table) */
    /* we obtain system call elsewhere */
    lda     $27, sys_ni_syscall
    cmpult  $0, $4, $4
    ldl     $3, TI_FLAGS($8)
    stq     $17, SP_OFF+32($sp)
    /* remove (s8addq  $0, $5, $5)
    /* since no offset */
    stq     $18, SP_OFF+40($sp)
    blbs    $3, strace
    beq     $4, 1f

    /* set first argument to the offset */
    /* (register saved by SAVE_ALL) */
    addq    $31, $0, $16
    /* load retrieval function and call it */
    /* register $0 will then have actual index */
    lda     $27, get_cur_sys_tbl
1:  jsr     $26, ($27), alpha_ni_syscall
    /* restore first argument */
    ldq     $16, 160($sp);

    /* use register $27 to make system call.*/
    addq    $0, $31, $27
    jsr     $26, ($27), alpha_ni_syscall
    ldgp    $gp, 0($26)
    blt     $0, $syscall_error  /* call failed */
    stq     $0, 0($sp)
    stq     $31, 72($sp)   /* a3=0 => no error */
       . . .
```

Figure 7. System Call Handler Code

compiled versions of several MiBench benchmarks in the M5 simulator using the modified Linux kernel with system call implementation. Using a separate system call table per process entails some increase in loads due to the work required to retrieve the system call table from the running process. The average increases in committed loads are provided in Figure 8, which is nominal at 0.15%, with Office benchmarks having the highest average increase of 0.51%. This can be attributed to the relative infrequency of system calls in typical applications. For this reason we did not include any figures for the negligible to undetectable change in execution time. This is to be expected given this infrequency as well as the large amount of work performed during a system call compared to the small amount of work from our few additional loads.

4.3 *Instruction level indirection*

Implementing instruction level indirection will require the addition of an IRF and its associated instruction extensions to the processor along with modification of the compiler to support instruction packing. The actual permuting of the contents of the IRF randomly at compile/link-time is a simple operation. Since the IRF is relatively performance-neutral, the resulting processor design will feature reduced application code size, improved energy efficiency, and an increased resistance to malware propagation. Previous application of instruction packing on these MiBench benchmarks has shown an energy savings of 15.8% with a corresponding code size reduction of 26.8%. This is for a 4-window 32-entry IRF that seeks to maximize code density both statically and dynamically. Reserving a few unused instruction registers to trigger micro state machines minimally impacts the overall improvements provided by instruction packing. Reserving 5 additional slots (beyond the one for *nop*) results in a code size reduction of 24.2% and a corresponding energy reduction of 14.9%. Average execution time is within 0.3% of the original case. The vast majority of tightly packed instructions do not utilize all 5 slots, and thus there is ample room to extend these instructions with references to micro state machine changing operations. Simultaneously, not all instructions that

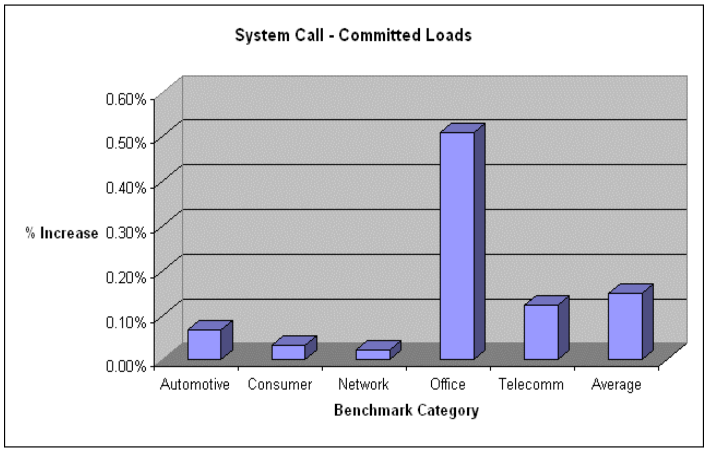

Figure 8. System Call Table Simulation

feature a loosely packed instruction field can actually make use of that available storage area. Having micro state operations fill these slots prevents any additional code size increase or execution time increase by providing a simply-decoded mechanism for modifying intrusion detection state machines. In extremely rare stretches of code that are very densely packed with few free slots, additional tightly or loosely packed instructions can be inserted to trigger the appropriate state changes, thus keeping our mechanism applicable to all software applications.

5. Related Work

There has recently been growing activity in the cell phone industry regarding security in cell phone applications and hardening of cell phone systems against viruses and other malware. One example is the Symbian Signed initiative to digitally "sign" applications for use on Symbian OS devices [33]. Although this can help to verify the integrity of participating software it entails increased development costs and complexity. Also, as with any certification-based system it is not scalable. Any successful application development will promote increasing code quantities that inevitably outstrip available code review resources. Signing may limit the applications a user may voluntarily install, but ultimately malware may infiltrate a system through means other than a user's active installation. Again the real problem is limiting and even stopping the malicious behavior of viruses and malware, given the assumption that malware will eventually have a chance to execute.

Many techniques have been proposed to defend against buffer overflow attacks, including implementation of non-executable stack areas [27], placement of *canary* marker values on the stack [6], encryption of pointer values in memory [5]. The approaches entail various levels of effectiveness and performance impact, and have met with various proposed shortcomings or defeats, including the failure to actually implement such well-accepted defenses. A stunning example is the most recent Mac OS X system on the Apple iPhone, which simply failed to implement a non-executable heap [20, 14]. Attempts to audit code to identify common vulnerabilities, either by hand [10, 26], or by automated methods [8, 39], such as searching for the use of unsafe library functions, have proven costly, sometimes prohibitively so [17, 11]. Indeed, writing correct code seems the most difficult defense to implement, with United States Computer Emergency Readiness Team (US-CERT) statistics showing an increase from 1,090 vulnerabilities reported in 2000 to over 4,200 in the first three quarters of 2005 alone [36]. Given a successful attack, Intrusion Detection Systems (IDS) have focused on detecting violations of security policy by monitoring and analyzing some characteristic of system behavior, with the goal of identifying, reporting on, and ultimately terminating anomalous behavior that may be indicative of an attack [16].

Actual randomization of the otherwise predictable code footprint of software has been considered in techniques such as Address Space Layout Randomization (ASLR), which inserts random memory gaps before the stack base and heap base (and optionally other memory areas such as the mmap() base). Attacks become less likely to succeed in executing a malicious payload and more detectable due to the behavior pattern of failed attacks (typically program crashes) [28]. PaX is a patch for Linux kernels that, among other security measures, employs ASLR.

However, Several methods for defeating ASLR protection have been proposed. For stack-based attacks, adding a nop slide to the beginning of the payload increases the chance of having the effective code land past the beginning of the targeted memory area. Also, the ASLR mmap() base randomization ignores the 12 least significant bits representing the page offset. Given a buffer entry function and a printf() call within the same page, the printf() can be repeatedly returned to using a format string bug to report the stack frame contents and determine the actual stack base offset [7]. The number of randomized bits used in an ASLR scheme can be effectively reduced by only requiring discovery of the 16-bit mmap() base randomization, thus allowing any buffer-overflow exploit to be converted to a tractable attack [30].

SHARK provides architectural support for protection from Rootkits [37]. By generating Process Identifiers (PID) in hardware and then encrypting page tables, SHARK attempts to prevent compromised operating systems from running malicious code. Program differentiation differs in that SHARK attempts to provide a single wall of protection, while Program Differentiation provides a Defense-in-Depth approach. Defense-in-Depth adds to overall defense since, if an attacker manages to break the security on one machine, the attack will not be spread to others. Furthermore, SHARK requires a significant architectural addition, while a significant portion of Program Differentiation can be implemented in software. In fact, program differentiation works well in combination with other protection schemes. Overall security is enhanced with any new hardware or software protection mechanism since it must be exploited for virus propagation to occur. Not only can other protections exist independently of our proposed differentiation schemes, the *methodology* of differentiation may be readily applied to other protection mechanism to make propagation even more challenging.

Other related research has attempted to analogize software defense to biological immune defense systems. Natural immune systems are designed to operate in an imperfect, uncontrolled, open environment that is analogous to current computer system environments. It has been proposed that principles of immune systems, including distributability of response, multi-layering of defense responsibilities, and diversity, be applied to computer systems to increase security [31]. Given that diversity is an important source of robustness in biological systems, computing machines are notable in that they suffer from their extreme lack of such diversity. Proposed diversification methods have included random addition of nonfunctional code, reordering of code, and various software and system transformations [9].

6. Conclusions

While vulnerabilities in software systems will continue to invite new malware attacks, we believe that proper mitigation techniques can reduce the impact of these attacks. Towards this goal, we propose expanding our current malware defense focus from the traditional approaches of absolute attack prevention to include efforts toward preventing malware *propagation*. By differentiating software program executables, we seek to thwart malware propagation when a vulnerability exists in a given software application. This is accomplished through virtualizing the control flow of the application, enabling function call/return and system call semantics to be permuted into unique versions for each application instance. We also show how an existing processor design utilizing an Instruction Register File (IRF) can achieve further security though the use of differentiation of instructions in the IRF as well as by restricting vital instructions from being fetched directly from memory. We show that these techniques require minimal overhead with respect to increased memory footprint and execution time. In worst-case software simulations of our control flow differentiation method using function calls and returns we found only a 1.04% average increase in instruction count and 3.38% average increase in loads. In software simulation of a Linux kernel implementing a per process system call table the increase in committed loads is nominal at 0.15%. Automation by the compiler and implementation of actual simple hardware support structures can ultimately result in little to no performance degradation for such differentiation.

The benefits of the minimal performance impact of our techniques, as well as their scalability, are ideal for embedded systems. These techniques can be implemented in whatever quantities are desired or allowable in a system without draining precious performance resources in the pursuit of defense. Current trends show that software programs will continue to possess vulnerabilities that are discoverable by attackers. However, inherent in differentiation is the targeting of the economics of malware, making *profitable* high-impact attacks prohibitively expensive and infeasible. While we accept that there may be successful attacks on any single distributed executable version, by dramatically increasing the ratio of effort to damage scope, attackers incur a substantial disincentive to developing malware *even in the presence of an exploitable software vulnerability*. This research was supported in part by NSF grants CNS-0615085 and CNS-0915926.

References

1. Android Dev Guide. Android Basics. http://developer.android.com/guide/basics/what-is-android.html. July 2010.
2. S. Bahtkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. *12th USENIX Security Symposium*, Washington, D.C. Aug. 2003.
3. A. C. Cheng, G. S. Tyson. An Energy Efficient Instruction Set Synthesis Framework for Low Power Embedded System Designs. *IEEE Transactions on Computers*, vol. 54, no. 6, pp. 698-713. June, 2005.

4. N. Clark, H. Zhong, and S. Mahlke. Processor Acceleration Through Automated Instruction Set Customization. In *Proceedings of the 36th Annual IEEE/ACM international Symposium on Microarchitecture*, pp. 129-140. 2003.
5. C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard(TM): Protecting Pointers from Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pp. 91-104. Aug. 2003.
6. C. Cowan, P. Wagle, C. Pu, S. Beattie, J. Walpole. Buffer overflows: Attacks and Defenses for the Vulnerability of the Decade. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, Hilton Head, SC, pp. 119-129. January 2000.
7. T. Durden. Bypassing PaX ASLR protection. *Phrack Magazine*, Volume 59, http://www.phrack.org /issues.html?issue=59
8. HalVar Flake. Auditing Closed-Source Applications. *The Black Hat Briefings 2000*, Oct. 2000.
9. S. Forrest, A. Somayaji, D. Ackley. Building Diverse Computer Systems. In *Sixth Workshop on Hot Topics in Operating Systems*, pp. 67-72, 1997.
10. Gentoo Linux Security Project. http://www.gentoo.org/proj/en/security/ audit.xml.
11. GCC Extensions. Bounds Checking Patches for GCC Releases and GCC Snapshots. http://gcc.gnu.org/extensions.html.
12. M. Gillon. Google Desktop Exposed: Exploiting an Internet Explorer Vulnerability to Phish User Information. http://www.hacker.co.il/security/ie/css_import.html. Nov. 30, 2005.
13. M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. *IEEE Workshop on Workload Characterization*. Dec. 2001.
14. S. Hansell. Stealing Data From an iPhone Is Easy, but Don't You Dare Use a Ringtone You Didn't Pay For. *The New York Times Online*. http://bits.blogs.nytimes.com/2007/07/23/stealing-data-from-an-iphone-is-easy-but-dont-you-dare-use-a-ringtone-you-didnt-pay-for/. July 23, 2007.
15. S. Hines, J. Green, G. Tyson and D. Whalley. Improving Program Efficiency by Packing Instructions into Registers. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pp. 260-271. IEEE Computer Society June 2005.
16. S. A. Hofmeyr, A. Somayaji, and S. Forrest. Intrusion Detection using Sequences of System Calls. *Journal of Computer Security* Vol. 6, pp. 151-180. 1998.
17. R. Jones and P. Kelly, Backwards-compatible Bounds Checking for Arrays and Pointers in C Programs. In *Proceedings of the Third International Workshop on Automatic Debugging*, pp. 13-26, May 1997.
18. McAfee. McAfee's Position on Vista. http://www.mcafee.com/us/local_content/misc/vista_position.pdf.
19. D. Melanson. NPD: Android ousts iPhone OS for second place in the US smartphone market. *Engadget*, , http://www.engadget.com/2010/05/10/npd-android-ousts-iphone-os-for-second-place-in-us-smartphone-m/. May 10, 2010.
20. C. Miller, J. Honoroff, J. Mason. Security Evaluation of Apple's iPhone. Independent Security Advisors, http://www.securityevaluators.com/iphone/ exploitingiphone.pdf. July 19, 2007.
21. N. Mook. IE Flaw Puts Google Desktop at Risk. http://www.betanews.com/article /IE_Flaw_Puts_Google_Desktop_at_Risk/1133545790. Dec. 2, 2005.
22. D. Moore. The Spread of the Code-Red Worm (CRv2). http://www.caida.org/analysis/security/code-red/coderedv2_analysis.xml.
23. E. Montalbano. McAfee Cries Foul Over Vista Security. *InfoWorld (via IDG News Service)*. http://www.infoworld.com/article/06/10/03/ HNmcafeefoul_1.html. Oct. 3, 2006.
24. R. Naraine. Google Patches Desktop Search Flaw. *eWeek.com*. http://www.eweek.com/article2 /0,1895,1744115,00.asp. Dec. 20, 2004

25. S. Nielson, S. Fogarty, D. Wallach. Google Desktop Security Issue (Technical Report TR04-445). *Computer Security Lab: Rice University*. http://seclab.cs.rice.edu/2004/12/20/google-desktop/. Dec. 20, 2004.
26. OpenBSD Project. Security. http://openbsd.org/security.html.
27. Openwall Project. Linux Kernel Patch from the Openwall Project. http://www.openwall.com/linux/.
28. PaX Team, Documentation for the PaX Project, http://pax.grsecurity.net/docs/aslr.txt, 2003
29. J. Schwartz. IPhone Flaw Lets Hackers Take Over, Security Firm Says. *The New York Times Online*. http://www.nytimes.com/2007/07/23/technology/23iphone.html?ex=1186286400&en=24cdfcebb35507dd&ei=5070. July 23, 2007.
30. H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address Space Randomization. *ACM Conference on Computer Security*, 2004.
31. A. Somayaji, S. Hofmeyr, and S. Forrest. Principles of a Computer Immune System. In *Proceedings of the Second New Security Paradigms Workshop*, pp. 75-82. 1997
32. G. Suh, J. Lee, D. Zhang and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking**.** In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 85-96, 2004.
33. Symbian Signed. The Complete Guide to Symbian Signed. http://www.symbiansigned.com/. Oct. 2, 2006.
34. TechWeb. McAfee Slams Microsoft Over Vista Security. *Software Technology News by Techweb*. http://www.techweb.com/wire/software/193101281. Oct. 2, 2006.
35. University of Michigan, Department of Electrical Engineering and Computer Science. The M5 Simulator System. http://m5.eecs.umich.edu/wiki/index.php/Main_Page.
36. U.S Cert Coordination Center. CERT/CC Statistics 1998-2005. http://www.cert.org/stats/#vulnerabilities.
37. V. R. Vasisht, H. S. Lee. SHARK: Architectural Support for Autonomic Protection Against Stealth by Rootkit Exploits. *Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pp. 106-116, Nov. 2008.
38. D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pp. 156-169, May 2001.
39. D. Wagner, J. Foster, E. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the 2000 Network and Distributed System Security Symposium*, Feb. 2000.
40. C. C. Zou, W. Gong, D. Towsley. Code Red Worm Propagation Modeling and Analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 138-147. 2002.