AFRL-RI-RS-TR-2015-142

ATTACKING TIME

TRUSTEES OF DARTMOUTH COLLEGE

JUNE 2015

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ AIR FORCE MATERIEL COMMAND ■ UNITED STATES AIR FORCE ■ ROME, NY 13441

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2015-142 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
DANIELLE M. GAMBINO
Work Unit Manager

/ S /
WARREN H. DEBANY, JR
Technical Advisor, Information
Exploitation and Operations Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* JUN 2015	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* SEP 2009 – MAR 2015

4. TITLE AND SUBTITLE ATTACKING TIME	**5a. CONTRACT NUMBER** FA8750-09-1-0213
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 35885G
6. AUTHOR(S) Stephen Taylor	**5d. PROJECT NUMBER** DARC
	5e. TASK NUMBER OL
	5f. WORK UNIT NUMBER 01
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Trustees of Dartmouth College, 11 Rope Ferry Road, #6210, Hanover, NH 03755-1404	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RIGA, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2015-142

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The primary goal of this research project was to explore an alternative to conventional network defense based on attacking an adversary's timeliness. This involved devising methods and techniques to *increase attacker workload* and *operate through attacks* even if the attacks are never detected. The research has resulted in a new way to structure distributed systems based on a *non-deterministic defense-in-depth*. This defense combines a series of breakthrough technologies that collectively provide an insurmountable barrier to the tactical viability of Advanced Persistent Threats (APT's). The ideas have been incorporated into a clean-slate, proof-of-concept operating system -- ***Bear*** -- that operates on Dell workstations, ARM embedded processors, and large-scale multicore blade-servers.

15. SUBJECT TERMS

cyber defense, defense-in-depth, advanced persistent threat, operating system

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** DANIELLE GAMBINO
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** (*Include area code*)
U	U	U	UU	36	N/A

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table **Page**

ACKNOWLEDGEMENTS

The work described in this report represents the collective work of a talented group of graduate students at Dartmouth College. Within the strategic framework, much of the detail is taken directly from joint writings with these students. In any group effort there is naturally considerable collaboration and overlap, the following gives a rough breakdown of the primary activities each student is engaged in; in alphabetic order: Scott Brookes (signals, diversity and memory systems), Jason Dahlstrom (hardware hiding), Robert Denz (kernel, hypervisor, and multicore), Michael Henson (memory encryption), Morgon Kanter (hypervisor, camouflage, and diversity), Stephen Kuhn (hypervisor, networking, network hiding, and forensics), Kathleen McGill (rMP and resilience), Colin Nichols (kernel and hypervisor), Martin Osterloh (systems engineering and reverse engineering).

1 SUMMARY

The primary goal of this research project was to explore an alternative to conventional network defense based on attacking an adversary's timeliness. This involved devising methods and techniques to *increase attacker workload* and *operate through attacks* even if the attacks are never detected. The research has resulted in a new way to structure distributed systems based on a *non-deterministic defense-in-depth*. This defense combines a series of breakthrough technologies that collectively provide an insurmountable barrier to the tactical viability of Advanced Persistent Threats (APT's). The ideas have been incorporated into a clean-slate, proof-of-concept operating system -- ***Bear*** -- that operates on Dell workstations, ARM embedded processors, and large-scale multicore blade-servers. To operate through attacks on multi-processors, a system of fault-tolerance was devised that dynamically regenerates processes in response to failures or inconsistent behavior. This method relocates regenerated processes to areas of the network that are uncompromised and includes resource management algorithms that dynamically restructure concurrent computations to avoid potential threats.

2 INTRODUCTION

Our method for mitigating APT's is based on a *non-deterministic defense-in-depth* in which a collection of innovative technologies are applied, either in isolation or in combination, to successively increase attacker workload and operate through attacks. These techniques prevent an adversary from operating on timescales that lie within the tempo of US military operations. In outline, the *mitigation methods* realized by these technologies are:

1. Non-deterministic refresh to deny surveillance, privilege escalation, and persistence [1].
2. Code size and attack surface minimization to reduce vulnerabilities [1].
3. MULTICS-style protection based on 64-bit extended paging tables (EPT) [1].
4. Diversity to deny reverse engineering and throttle vulnerability amplification [2].
5. Full memory encryption to deny access to code and data in memory and shrink the protection boundary to the chip boundary [3-6].
6. Network hiding to dynamically change network properties [7,8].
7. Camouflage to deny system identification [9].
8. Hardware hiding to own and control the base of trust [10].
9. Course-grain forensics to allow discovery of zero-day exploits [11].
10. Resilience through dynamic process regeneration and remapping to operate through attacks [12-19].

This body of knowledge has been published in 3 Ph.D. theses [3,7,12], 1 M.Sc. thesis [1], 13 published papers, and 8 technical reports. All of these publications are available through password-protected access at:

http://engineering.dartmouth.edu/~d1266j2/styled/index.html

At the time of writing, the research component of one additional Ph.D. thesis on hardware hiding has been completed, together with an additional paper; thesis preparation is in progress. In addition, this work leverages contributions associated with a complementary ongoing project, “Resilient Diffusive Clouds” under the DARPA MRC program. These contributions are mentioned briefly in this report, solely for clarity and completeness in understanding the overall vision of the project.

It is not the goal of this report to repeat either the material or the extensive bibliographies provided in the project publications, but rather to provide a cohesive explanation of the body of work taken in its entirety. This entirety is embodied in a new, clean-slate, proof-of-concept concurrent operating system – ***Bear*** – that operates on Dell workstations (9010/9020), ARM embedded processors (M4/A8/A9), a system-on-a-chip device (Xilinx Zynq), and large-scale multi-core blade servers (Dell PowerEdge). It must be recognized however, that the findings are distributed over this collection of architectures ***not*** ported to each of them. The reason for this distinction is that, at the time of the research, all of the needed underlying hardware capabilities were not available on any single platform. For example, Intel processors provided virtualization and protection support for guest operating systems (VT-x) and devices (VT-d); this was not available on ARM processors. Similarly, on-chip encryption/decryption engines and FPGA technology were available on ARM-based devices but not Intel processors with virtualization. Only recently, within the few months, has there been a confluence of these technologies, with both Intel and Xilinx recently announcing *future* offerings that will combine *all* the needed capabilities into a single processor design – thereby opening the door to an eventual integration of the techniques within a operating single system available on multiple hardware platforms.

The Attacking Time project is delineated by a focus on multi-processor, *but single core*, implementations, with preliminary explorations of some areas -- in particular network hiding **(Method #6)** and resilience **(Method #10)** – conducted on Linux. The Linux work was necessitated by the fact that a mature version of the Bear system, that could accommodate the ideas, was under development concurrently and not of sufficient stability to support development until the latter part of the project. The MRC project is projecting the technologies into multi-core, multi-processor systems (cloud computing) while introducing native implementations of resilience.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 ASSUMPTIONS.

At the outset of the project, team members already had extensive experience with the combination of vulnerabilities, exploitation methods, and TTP’s that have, only recently, received the unclassified DARPA designation “Advanced Persistent Threat” (APT). The starting assumption was a threat-model that directly encapsulates the core notions of this designation as illustrated in Figure 1. An APT involves several steps that may include *surveillance* to determine if a vulnerability exists [20], use of an appropriate exploit or other access method [20], privilege escalation [21], removing exploit artifacts, and hiding behavior [22]. Surveillance may involve obtaining a copy of the binary code and using reverse engineering [23,24] or fuzzing [25] to facilitate a broad range of attack vectors including return oriented programming [26]. The implant then *persists* for a time sufficient enough to carry out some malicious effect, obtain useful information, or propagate intrusion to other systems. The ubiquitous use of a small number of operating system types and versions in distributed systems and clouds, has the effect of amplifying vulnerabilities: an exploit developed against one version may be used against any host using a similar version.

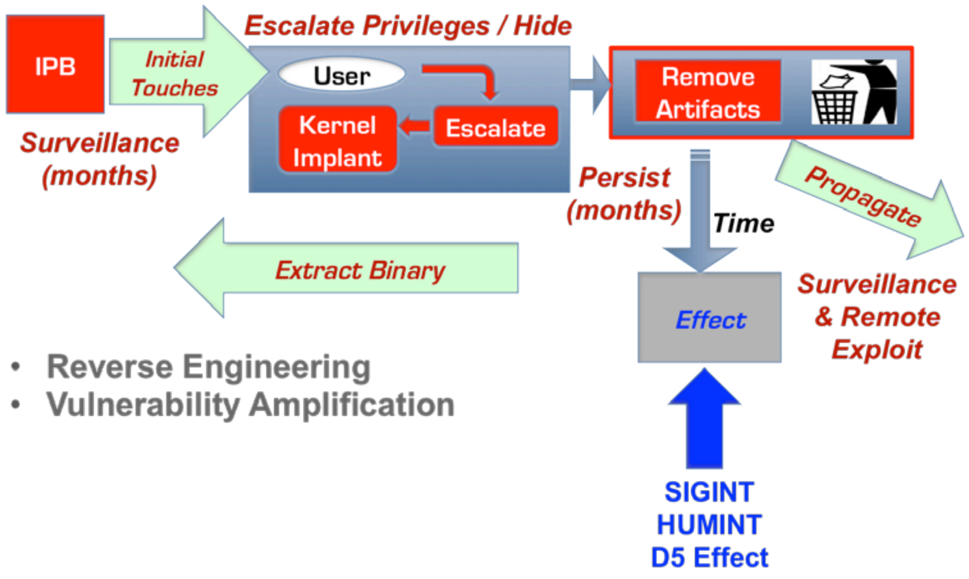

Figure 1: APT Threat Model

A central aspect of this process is **timeliness**: the value of information is always qualified by time. For example, a cyber attack, using a network of APT's, on the air tasking process associated with a military mission would have little utility if it is not perpetrated within the finite timescale (24 - 72 hrs) covered by the ATO planning and execution process. Figure 2 shows the attackers process from the perspective of timeliness.

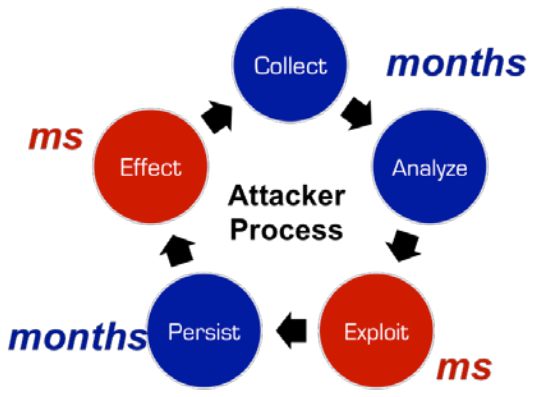

Figure 2: Timeliness in the Attack Process

Unlike the time to execute an exploit or effect, the time spent in surveillance and persistence may range from minutes to *months or even years* depending upon the intended effect. Moreover, the presence of an intrusion may *never* be detected by network defenses but instead may be recognized indirectly due to either a deviation from expected behavior, the adversary's execution of some D5 effect, or derived from other intelligence sources (SIGINT, HUMINT, etc). Unfortunately, it is precisely the short-timescale areas designated in Figure 2 that are the domain of anomaly and rule-based intrusion detection systems (IDS) and the associated correlation tools. For rule-based detectors, there is no defense against *zero-day attacks* – if an exploit has not been used before, there will be no rule or derivative rule that renders it detectable. Sadly anomaly detectors also fail due to a sad truism:

Not all malicious attacks are anomalous, and not all anomalies are malicious.

In other words, good APT's will hide their behavior and false alarms will obscure their activities.

Current operating system designs have sought to utilize a *static* base of trust and extend trust into software through deliberate layering to combat such threats [27]. Unfortunately, a wide variety of vulnerabilities have appeared that undermine kernel security allowing attackers to implant code, hide, and persist at the highest levels of privilege [28]. The number of vulnerabilities is directly correlated with the size of the code base [29], indicating that there is substantial value in the intellectual process of *reducing the attack surface*; most current operating system designs run into <u>millions</u> of lines of code. Moreover, they compound the opportunity for compromise by granting device drivers unnecessary levels of privilege in order to attain, what in recent years has become, diminishing returns in performance.

3.2 METHODS AND PROCEDURES.

3.2.1 Core Ideas. Recall that our approach assumes that adversaries will conduct *surveillance*, will be successful in gaining access, will obtain critical system code for reverse engineering, and will *persist undetected* to carry out effects at a later date. To mitigate the risks associated with APT's, we non-deterministically discard the current <u>kernel</u>, user processes, and device drivers **(Method #1)**. They are replaced by new instances, bootstrapped in the background from read-only gold standards. The cumulative effect of this change in design style is to *increase attacker workload* by continually invalidating surveillance data and denying persistence over time-scales consistent with tactical missions. Unlike other approaches to computer security, no attempt is made to detect intrusions: instead, we focus on continually validating, preserving, and re-establishing the ability of a mission to proceed.

These concepts have been incorporated into the x86, 64-bit version of the ***Bear*** operating system. The full system is depicted in Figure 3 and is composed of a minimalist *micro-kernel* with an associated *hypervisor* that share code extensively to reduce the attack surface **(Method #2)**. The core functions of scheduling user processes and protecting them from each other are handled by the micro-kernel. All processes and layers are hardened by strictly enforcing MULTICS-style read, write, and execute protections that only recently became available using 64-bit x86 address translation hardware **(Method #3)**. This calculated reduction in versatility is unlikely to impact military applications but explicitly removes vulnerabilities associated with code execution from the heap or stack.

All potentially contaminated user processes, device drivers and services are executed with user–level privileges and are strictly isolated from the micro-kernel via a message-passing interface. A notional system task mediates between processes and the kernel to implement the interface. Unlike a conventional rendezvous mechanism in which processes block until synchronization, this asynchronous buffered design provides a single uniform treatment of system calls, inter-process, and inter-processor communication. The interface also supports distributed computing through an MPI-like programming model that maps processes to processors using a user level demon, *rMP*.

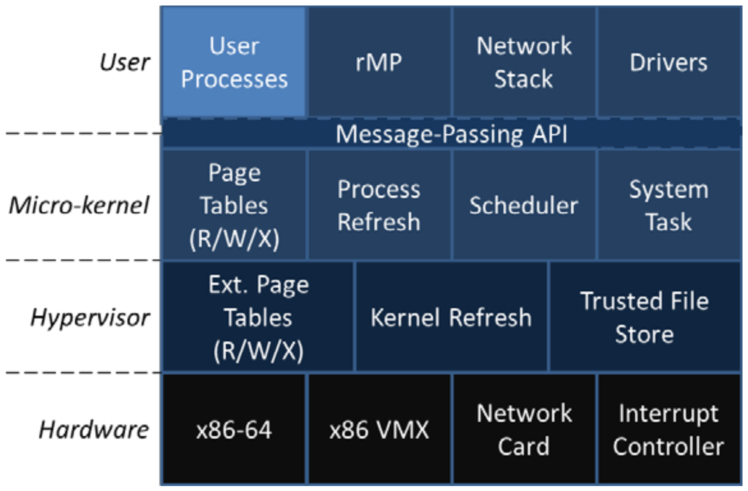

Figure 3. The Bear Operating System Layers

To deny persistence in compromised device drivers and services, the micro-kernel randomly and non-deterministically regenerates them from gold-standard images resident in a *trusted* <u>*read-only*</u> *file store*. This store is currently realized by loading all system code directly into a read-only RAM-disk using an iPXE NIC-assisted boot process. The file system is accessible only from the kernel and hypervisor; however, it could alternatively be realized via read-only memory (ROM) or via an out-of-band, write-enabled channel to flash on new hardware. Unlike the MINIX re-incarnation process, regeneration is carried out without regard to the perceived fault or infection status. User processes can also be refreshed through pre-arranged or designated schedules; for example, every few hours, at night, or just prior to a tactical mission.

To deny persistence in the micro-kernel, it is also non-deterministically refreshed from a gold-standard image in the trusted file store, but by the hypervisor. Unlike traditional hypervisors, which are intended to support a general virtual machine execution environment, this minimalist hypervisor is designed to support *only* the operations required to bootstrap a new micro-kernel and change its network properties (e.g. IP & MAC address) so as to invalidate an adversary's surveillance data. The current running and bootstrapping instances of the micro-kernel are isolated in hardware through extended page tables, implemented with Intel VT-x extensions. Similarly, the network card is isolated through a mapping scheme based on Intel VT-d extensions.

Protecting the Micro-kernel. The micro-kernel architecture leverages the latest x86-64 address translation hardware to provide isolation and MULTICS-style read, write, and execute (R/W/X) privileges for processes. Recent x86-64 processors no longer support segmentation, but they do feature control bits that enable the kernel to allow or deny reading, writing, and execution of a particular memory page. This is achieved using three protection bits in x86-64 page table entries: To isolate user processes from the kernel, the kernel clears the user/supervisor bit (U/S, bit 2) on its own pages. If any user process attempts to read, write, or execute code in these pages, the processor traps to the kernel. Bear enforces protections for process memory using the read/write (R/W) and execute disable (XD) bits. When a process is loaded, the bits are set so that the process text (code) is readable/executable. Conversely, process data and stack are designated readable/writeable. These decisions yield a protected address space where the permission bit configurations are also shown below in Table 1.

Table 1: Page Table Protection Configurations

Memory Type	**U/S Bit Value**	**R/W Bit Value**	**XD Bit Value**
User Process Text	1	0	0
User Process Data	1	1	1
User Process Stack	1	1	1
Kernel Text	0	0	0
Kernel Data	0	1	1
Kernel Stack	0	1	1

rMP – remote Message Passing API. The memory space of each process is strictly isolated from that of other processes and the micro-kernel by page protections. All processes interact via a simple MPI-like asynchronous message-passing interface. This allows the same isolation ideas to be used for inter-process communication within the same processor, across multiple processors, and between user processes and the kernel. The interface provides only two asynchronous, blocking, communication primitives:

- **msgsend(dest, &sendbuffer, size)** – send a message from *sendbuffer* of length *size* bytes to process *dest*.
- **msgrecv(src, &recvbuffer, size, &status)** – receive a message from process *src* (or ANY process) into *recvbuffer* of length *size*; *status* is a structure designating the *source* of the message and its *length*, messages that are larger than *size* are truncated.

Both primitives are realized using software interrupts that isolate user-processes from the micro-kernel. All messages are buffered in the kernel at the *receiver*. The *msgsend* operation causes a process to be blocked until a message is sent (i.e. injected into the kernel, if the receiver is at the same host, or the network if it is on a remote host). Return from this primitive allows the *sendbuffer* to be re-used. The *msgrecv* operation causes a process to be blocked until a message is transferred into the *recvbuffer* from the kernel. System calls, such as -- fork(), exec(), and exit() -- are implemented directly in the kernel which is capable of modifying kernel data structures (e.g. pages, scheduling queue's etc); distributed computing is achieved by forwarding messages to a remote host via a mapping process *rMP* (c.f. *Figure 1).* All of our work on resilience (**Method #10**) was explored using MPI implementations of these primitives on Linux.

The micro-kernel leverages *user-space separation of privilege* to minimize kernel size. In this approach, device drivers are given only the access rights needed to operate. Thus they require *no kernel intervention* other than startup in order to execute. This allows system calls serviced by user-space processes – the network stack, the file-system, and so on – to operate entirely in user-space. Borrowing terms used by MINIX, the system call *policies* remain in user-space, but are joined by the system call *mechanisms*, in the form of *entire* device drivers. Consequently, a significant amount of privileged code is excised from the kernel, creating a small attack surface with few entry points.

It is instructive to contrast this approach with that used in MINIX: a number of user-level tasks service system calls. These tasks – such as the process manager, file-system, info server, and so on – enforce system call *policies* and carry out bookkeeping, but they do not contain the actual *mechanisms* to carry out a system call. That is left up to either drivers or the kernel. However, even drivers are reliant on kernel code to perform their functions, and they have their *own* set of system calls that are directly serviced by the kernel. The result is a small reduction in kernel code and data, but a significant increase in complexity.

Attack Mitigation. Despite efforts to insulate the kernel from user processes, there are still methods to get code into the kernel memory space. For instance, while carrying out inter-process communication, the kernel may buffer user data in kernel memory-space. Furthermore, a hardware implant could potentially inject code directly into kernel memory. Once kernel memory is contaminated, an attacker need only find a method to divert kernel execution to this code.

Bear's treatment of kernel memory is designed to expressly deny this avenue of attack. At all levels, Bear enforces the policy that no memory region may be both *writeable and executable* simultaneously. In the Bear kernel, there are four classes of buffers: those created by the kernel's small-memory allocator, those created by the kernel's large-memory (page) allocator, static buffers in the kernel binary, and temporary buffers located on the kernel stack. The small-memory allocator is used to dynamically allocate space for data structures within the kernel (e.g., message buffers, process structures, hash tables, linked lists, etc.). All memory regions returned by the small-memory allocator are protected from execution using the XD bit in kernel page tables. The large-memory allocator provides free pages (or multiple pages) for process or kernel use. If used by the kernel, pages from this allocator are protected from execution via the XD bit in the kernel page tables. Static buffers in the binary and dynamic buffers on the kernel stack are similarly protected from execution via the XD bit in the kernel page tables. Thus, no buffers have both write and execute permissions enabled.

It is well known that robust memory protections are not enough to secure a system from return-oriented programming (ROP) even in the presence of non-executable buffers. These attacks leverage small sections of the code already resident in memory, known as gadgets. The payload of a ROP exploit is a series of specially crafted return addresses, which link together gadgets to perform whatever action the attacker desires. ROP exploit development is facilitated by a large codebase, such as GNU Libc (glibc).

To increase the difficulty of crafting these attacks, we emphasize the reuse of *common data structure* abstractions throughout kernel and hypervisor so as to reduce the attack surface and a collection of techniques that introduce diversity into binaries **(c.f. Methods 4 and 5 below)**. Generic implementations of common data structures, including a linked list and hash table, were created with flexibility in mind. Application-specific data is always stored in these structures through the use of opaque void pointers, and application-specific functionality is added through the use of function pointers in the API. The result is lean, robust, multi-purpose code; for example, the function for removing a process from the scheduler is also the function for removing an element from a hash table.

Mitigation of Corrupted Device Drivers. Unfortunately, device drivers are a frequent source of vulnerability; they are always resident and often developed by third-party vendors, whose priorities are fast turnaround, inter-operability and performance, rather than security. Recall that the Bear micro-kernel refreshes each device driver at nondeterministic intervals. This allows the kernel to *operate through* attacks, preserving trust while denying the attacker the ability to persist over tactically relevant timescales. The upper and lower bound on the duration of a device driver instance is configurable, and could be set higher or lower based on threat or mission deadlines. Driver refresh is achieved by interrupting the driver, freeing its memory, and re-allocating new resources for its replacement. The kernel then loads the driver's gold-standard image from a protected, read-only store. As a result, compromised drivers are not able to persist over long time-scales. Once driver regeneration is complete, the kernel schedules the driver, and normal operation is resumed. Although the hardware state is lost, this is not typically detrimental to a system functioning. In a server environment, it may involve a few dropped packets, but these will be re-transmitted by normal protocols. Down-time associated with refreshing the driver could be minimized by creating the new driver process in the background using underutilized computing cores, although this has not yet been necessary.

The main objectives for driver design in Bear are to protect the operating system from corruption, encapsulate the device driver using hardware mechanisms, and facilitate on-the-fly refresh of the drivers. Putting the driver in an isolated user-level process and utilizing process refresh techniques accomplishes most of these goals. Unfortunately, a compromised device driver has unique hardware resources at its disposal that open up avenues of attack not available to most user processes.

Traditionally, the x86 architecture provides four rings (or levels) of privilege, numbered 0 through 3. Processes on the outside ring are the least-privileged and have no access to critical functionality, while the innermost ring has full privileges. For obvious reasons, user processes usually reside in the outermost ring 3, and the operating system resides in ring 0. When considering where to put device drivers, rings 1 and 2 appear to be likely candidates. Unfortunately, upon close inspection of hardware support for rings 1 and 2, it was discovered that ring 0 is *not* truly protected from code running in the intermediate rings. Intel's memory management unit only supports two access levels – user (ring 3) and supervisor (rings 0, 1, and 2). Thus, code running in rings 1 and 2 has exactly the same memory access privileges as the kernel. This violates one of Bear's primary design principles – namely, complete isolation of device driver code from the kernel. We thus chose instead to place device drivers in ring 3. Rings 1 and 2 actually provide few meaningful benefits compared to ring 0. In contrast, ring 3 provides complete isolation from the kernel through hardware mechanisms.

Until recently, drivers were able to command a device to read/write to *any physical address* via DMA. In most modern PCs, only the number of address lines on the bus limits a peripherals access to memory. Thus, on most machines, devices can read or write to *any address*. Mechanisms to limit DMA access have recently become available in COTS hardware. The centerpiece of device protection is the input/output memory management unit (IOMMU), which provides a layer of address translation and access control between devices and physical memory. On Intel platforms, the IOMMU is part of a larger set of device virtualization technologies known as VT-d. For systems that do not support VT-d we provide a simple mapping scheme that is consistent with a full VT-d implementation. At the time of writing a full VT-d implementation for the E1000 NIC was close to completion.

Protecting the Hypervisor. Recall that the normal role of virtualization is to share the underlying hardware between multiple operating system instances. In contrast, the Bear hypervisor exists primarily to undermine network surveillance, deny persistence in the micro-kernel, and reestablish trust. Re-establishing trust is performed by periodically reloading the micro-kernel from gold-standard images located in the read-only store. This has the effect of expunging root-kits, bots, or other malware. Additionally, the hypervisor strives to utilize all available hardware mechanisms to provide protection for both itself and the kernel.

To mitigate the threat of well-timed attacks, the hypervisor refreshes the kernel at nondeterministic intervals. The upper and lower bound on the duration of a kernel instance is configurable, and could be set higher or lower based on the threat environment. To achieve kernel refresh, the hypervisor assumes control of the system, frees the memory associated with the previous kernel, and allocates resources for the next kernel. The hypervisor then loads the kernel binary from the trusted store and relinquishes control to the kernel, which boots and resumes normal operation. Due to its code size, the microkernel boots in less than 1 second; consequently, there was little reason to leverage multiple cores to perform booting in the background as originally expected [c.f. **Method #6** and Reference 8].

The hypervisor also provides *protection* for the kernel by leveraging extended page tables (EPT). EPT is a hardware address translation capability present in newer Intel CPUs (AMD has similar technology). EPT provides an extra layer of address translation that is transparent to the guest operating system. This allows a hypervisor to manage physical memory while giving the guest the illusion of physical memory access. EPT also allows the hypervisor to control what type of operations are allowed for a given memory region, opening the door to protection of the kernel.

Bear's hypervisor configures EPT to provide read/write/execute controls on both the kernel code and static data. Thus, any attempt to patch the kernel or execute code located in a static buffer will result in a trap to the hypervisor. At that point, the hypervisor can refresh the kernel or take an alternative action, such as invoke forensic tools. Hypervisor memory is inaccessible from the guest.

***3.2.2 Diversity (Method #4)*.** Our work on diversity is supported under the DARPA MRC program; it is mentioned here only for completeness and to allow a recognition of why non-deterministic refresh is central to the mitigation of APT's: The refresh process uses a radical new diversifying technology for binary images that introduces non-deterministic changes to every binary image used by the operating system including both the hypervisor and kernel. Source-to-source transformations add small random changes to every source-code block. Subsequently, at load time, every function is allocated to a random location in virtual memory as it is copied out of the read-only trusted file store. Diversity disrupts the entry point used to execute a malicious implant, the exit point used to return from an implant to normal operation, and all blocks within any function used for inline patching. The impact of this process is to ensure that every time a user process, device driver, service, <u>*kernel*</u> or <u>*hypervisor*</u> is loaded, it employs a completely unique binary image. This ensures that any reverse engineering conducted to craft exploits or embed code in the binary will only operate for a short window in time – *the window between refreshes*. Since the development cycle for zero-day exploits is often counted in *months or years.* This prevents remote exploitation on the timescales of tactical missions. Moreover, it eliminates vulnerability amplification by ensuring that every image on distributed system or cloud is unique.

3.2.3 Memory Encryption (Method #5). Michael Henson, who was supported under an Air Force scholarship, conducted the work associated with this method while thesis supervision was supported by the Attacking Time project. Experimental work was conducted on an ARM A8 processor. Since this processor does not support virtualization, the research was incorporated in a *kernel-only* version of the ***Bear*** system.

Recently, a new generation of commodity processors have appeared that include security technologies, such as encryption engines, on-chip within the trusted boundary provided by the processor. These processors include the Intel i7, AMD bulldozer, and multiple ARM variants, including the A8 and A9. The creation/use of such processors begs the question: Can these technologies be leveraged with sufficiently low overhead in order to improve operating system security? This work explored the idea of enhancing security through *memory encryption.* In particular, it introduced three new technologies:

- *Static Encrypted Processes:* This technology employs one-time decryption within the trusted boundary of the chip. Since the one-time cost of encryption is amortized over the life of a programs execution, its overhead is negligible. The technique can be used to protect industrial control systems employing microcontrollers and other real-time processors. These devices typically lack memory management and make little to no use of cache.
- *Dynamic Encrypted Processes:* This technology provides a general, full memory encryption mechanism for code and data. It is appropriate to any multi-tasking operating system that employs a memory management unit (MMU) and cache including smart phone and other mobile computing devices. Two micro-benchmark programs targeting the specific areas where overhead is introduced (context switching and cryptopaging of heap and code) showed reasonable performance impact of approximately .12% and 1.2% per minute respectively given a page size of 4 KB and typical mobile smartphone workloads.
- *Mutually Distrusting Processes:* This technology extends dynamic encrypted processes to protect processes from each other by uniquely keying each process. At its finest granular level, this technique induces a performance penalty of approximately 1920 cycles or 2.4 microseconds per context switch (~ 480 microseconds per minute) for the key search—an extremely small overhead for the additional protections afforded.

Collectively, these technologies *increase attacker workload* by ensuring that both code and data are always encrypted outside the trust boundary afforded by the processor. To overcome this barrier requires physical access to the device and exotic reverse engineering techniques, such as acid etching, that are generally the domain of only a few, highly skilled, internationally recognized, specialists in reverse engineering. A side-effect of the approach is that it introduces another form of non-determinism by introducing a *synthetic form of diversity* into code and data: every processor's image is completely different in RAM. This makes it significantly more difficult to determine the vulnerabilities present on a particular system, use the same attack vector against multiple hosts, or steal sensitive code and data, perform reverse engineering of code, modify data, and inject code. The core contributions of this work were:

- The first practical full-memory encryption system implemented on a general-purpose commodity processor.
- A survey and comparative analysis of memory encryption techniques covering three decades of research with proposed solutions; these employ widely varying assumptions and experimental conditions.

- A collection of novel memory encryption techniques providing *synthetic diversity* and *increasing attacker workload*. These techniques protect against software and hardware based confidentiality and integrity attacks; the techniques are portable to currently deployed general-purpose, security-enhanced processors.
- Analytical results that include performance benchmarks and analysis on the overhead of memory encryption down to *process segment granularity*.
- Empirical evidence and analytical analysis that demonstrate protection through memory encryption against confidentiality and integrity attacks.
- Techniques to employ self-modifying code within the memory hierarchy to achieve memory encryption.

The techniques and technologies have been demonstrated in proof-of-concept implementations and exemplars. Memory encryption has been implemented on the ARM Cortex A8 processor to provide *automatic* and *transparent* protection for applications. This is achieved through extensions to the Bear microkernel. These extensions involve modifications to linker scripts, initialization, process creation and context switching routines as well as new modules for interfacing with the A8's on-chip encryption decryption unit (EDU). The ideas have been demonstrated by encrypting processes while they reside in external RAM (eRAM) thereby adding synthetic diversity. The concepts cover application deployment regimes that range from unsophisticated microcontrollers, with no memory management unit (MMU) and cache, to full-functioned multi-processing operating systems utilizing a memory management unit (MMU) and L1/L2 cache. Various *granularities* of protection are considered from a complete code base to individual process. Finally, exception-handling routines have been developed and experiments executed to understand the protections afforded against code and data injection.

The work extends the base of technologies available for *trusted computing*. While definitions of trusted computing abound, in this work it was defined as the process by which a trusted subset (software and hardware) of a system, known as the *trusted computing base* (TCB) is *amplified* to provide security assurances about the operation of the larger application or system. Hardware components of the traditional TCB include encryption coprocessors, random number generators, and small amounts of protected space for operation on sensitive code and data. The main application of trusted computing in operating systems design is for "trusted boot" in which the TCB checks the integrity of each component in the boot process, perhaps halting the boot process when a problem is discovered. Additionally, the TCB has been used as a means for providing digital rights management. While the underlying security and integrity of hardware are often assumed to be axiomatic by those programming higher layers, this is not typically the case. The inclusion of security hardware within commodity processors means that these general purpose CPUs may now be treated as part of the TCB. While the processor boundary may not have been designed to meet stringent guidelines, such as the PCI, it does, however, provide natural barriers to penetration and observation. This work sought to expand upon current trusted computing capabilities such as trusted boot by continuing to protect applications dynamically as they execute. While memory encryption provides significant protection against multiple attack vectors, it should be used as part of the non-deterministic defense-in-depth strategy and include other trusted computing capabilities such as trusted boot as well as encryption of data-at-rest.

Any security can be circumvented given enough resources and motivation and memory encryption is no exception. The goal of the work, to *increase attacker workload*, can be applied under two alternative scenarios: In any time-sensitive operation, as occurs on the battlefield, an increase in attacker workload serves to force the adversary outside of the useful timeframe of any sensitive data collected. For a commercial example, the increased workload would influence the attacker to choose a weaker attack surface, on a different device (preferably at another business).

3.2.4 Network Hiding (Method #6). Stephen Kuhn, who was supported under an Air Force scholarship, conducted the work described in this method while thesis supervision was supported by the Attacking Time project. The work was conducted on Linux since the Bear system was in its early stages of development when the method was explored; the underlying mechanisms that enable this technology **(Methods #1-4)** were eventually incorporated into the Bear system, however the work was not retargeted since no new research insights could be gleaned from such an activity.

Offering assured and available Internet services is a key challenge faced by, service providers, the military, and commercial corporations. These services include web hosting, file storage, remote software access, and central database access. The concentration of information at servers makes them a primary target for adversaries to exploit, forcing providers to expend considerable resources in protecting them. Typical defense mechanisms use rule based methods: a combination of intrusion detection systems, firewalls, and virus scanners, at the network level. In corporate and military environments additional host base systems are commonly deployed such as virus scanners, root-kit detectors, file-system integrity checkers, anomaly detectors, and more recently website application firewall software. The shortcoming of these defenses, especially in the presence of zero-day exploits, has already been mentioned.

The challenge of internet addressing and administrating Domain Name Services or DNS, leads most systems administrators to assign static addresses to their servers. Static address assignment allows DNS servers to maintain long update cycles, preventing services from appearing offline and unnecessary traffic. Since a DNS server caches previous address lookups, there is a significant performance advantage from a server remaining in place. Unfortunately, this presents a static observable target for adversaries to analyze using network scanners such as NMAP and Nessus. This surveillance process provides a roadmap to the available vulnerabilities and allows appropriate exploits to be isolated or developed. After access is gained, persistence on the static target allows stable reentry point to carry out effects.

The combination of stationary targets, undetectable malicious software, and the small timescales over which a host is exploited allows the attacker to operate inside the defenders OODA loop. The network hiding method non-deterministically moves servers around the IP space of a network to deny surveillance and periodically reconstitutes them to deny persistence. As in our other methods, the goal is to increase attacker workload to the point where the timeliness of attacks is significantly longer than the timescale of day-to-day operations, making attacks irrelevant even if they are successful and never detected.

Our proof-of-concept network hiding provides web services through Apache and is based on server relocation and reconstitution operations. The server's location is periodically migrated within the local enclave IP space and into alternative enclaves, potentially behind differing boundary defenses, using multiple network interface cards (NIC's). This has the effect of presenting a moving target to adversaries, while maintaining connectivity to local clients.

Leveraging the KVM hypervisor technology, the server was repeatedly reconstituted to a fresh service state, using a different operating system image, akin to a full system reinstall **(c.f. Method #4)**. This has the effect of changing the attack surface while ensuring that malicious code is removed without attempting to detect its presence. This reconstitution denies persistence over long timescales while falsifying any existing surveillance information that the attacker may somehow have garnered. These operations can be carried out at random intervals and times **(c.f. Method #3)**. This presents a completely non-deterministic view of the network structure from outside an organizations local area network, while maintaining availability within it. The approach presented several technical challenges, in particular how to:

- Control server relocation and reconstitution,
- Preserve connectivity with existing clients, and
- Advertise service so that currently unconnected, but authorized, clients are able to locate and connect to the server.

Each server can be assigned one out of several network interface cards (NIC's), and each NIC is connected to a separate logical enclave assumed to be behind a unique firewall/proxy server. Each logical enclave has a DHCP server that serves IP addresses randomly from a large non-routable IP space, orders of magnitude larger than the number of hosts located in the enclave. A non-deterministic finite state machine (FSM) was used to control network hiding: it generated a new virtual machine in the background of the currently running instance, assigned it a new NIC card, and bootstrapped the operating system onto it. After a non-deterministic time, control was quickly switched to the new virtual machine. The old virtual machine was allowed to continue execution so as to service existing open connections; it terminated when they were all closed or a bounding timeout occurred. The process is implemented through KVM hypervisor commands to define, start, undefine (or remove), and destroy (or terminate execution) virtual machines. These operations were subsequently built into the Bear hypervisor.

A private DNS server was used by the FSM to allow clients to locate a server after it has migrated around the network. Beyond the simple lookup of IP-addresses, the latest DNS implementations, such as BIND provide enhanced functionality for dynamically configuring name references. This involves the update of two components of the DNS record: the CNAME, describing translations between names, and the 'A' records, translating names to IP addresses. Each virtual server must be uniquely named, however, from the viewpoint of external clients; all servers must be accessed through a single consistent name. The CNAME reference system allows this aliasing by creating a name that references another name. The result is a two-step process for resolving generic service names, such as www.myserver.org. The DNS server resolves the alias from www to the unique server name and then resolves the unique server name and returns the numeric IP address.

Dynamic DNS or DynDNS provides the control software to dynamically push updates to a DNS server when a new address is selected. DynDNS was originally invented to support home users on modems whose addresses frequently change. Previously the only modality for effecting updates required modifying the static configurations of the BIND/DNS server and fully restarting the system, causing a loss of service. Applying updates through the DynDNS mechanism allows the method to maintain a consistent server presence, while virtual machines and their associated IP addresses may change.

The network hiding method was successfully demonstrated in the presence of all three core types of web content under Apache: Static pages of plain html, Streaming pages containing video or file transfers, and stateful pages involving server side applications that maintain state on the server (e.g. shopping carts).

3.2.5 Camouflage (Method #7). Most traffic in modern networks is between clients and designated servers, rather than client-to-client. As a result, servers represent high-value targets for exploitation. Before a server vulnerability can be exploited, however, it must first be discovered and a clear picture of the operating environment on the server must be developed. The goal of this method is to increase attacker workload by camouflaging servers to appear to be running a different, potentially vulnerable operating system, encouraging the use of a known, detectable exploit.

Several different tools are available for operating system and service detection; two of the most popular of these are NMAP and Nessus mentioned previously. At their core, these scanners operate due to an inherent issue in protocol specification and implementation: protocol specifications typically leave many implementation details up to the developer. This allows for multiple implementation strategies and ideas to be used in alternative products. When these design details diverge between competing systems, the differences can be observed by merely using the protocol and system fingerprinting can be achieved. NMAP and Nessus both exploit the differences in implementation details for their discovery process. For example, for operating system detection they have databases specifying which details are expected on which system. By examining the response to normal TCP/IP traffic, they discern which operating system the responses must have originated from. Similar systems exist for application-layer protocols in Nessus.

Since detection is achieved by examining implementation differences, it is possible to create deception by *modifying* responses to normal traffic. Early uses of this idea, implemented in Morph, were able to transparently camouflage a system to appear as Windows 2000, OpenBSD, or Linux 2.4. The concept has also been used in FreeBSD, which scrubs its fingerprint so that it is not detectable by scanners. Linux 2.4 provides a program called IP Personality that allows it to take on alternative operating system characteristics (IP Personality). All of these packages focus on manipulating TCP/IP protocol details to prevent operating system detection.

The technology developed under this method provides a general camouflage capability that presents a false server fingerprint. The capability is implemented as a table-driven finite state machine that operates across the entire protocol stack, simultaneously falsifying both operating system and service properties. It follows the normal transitions of the protocol stack but emits responses associated with an alternative system so as to provide camouflage. The false fingerprint may be created to provide known vulnerabilities, that if exploited can trigger an alert or honeypot the attacker. The camouflage has been demonstrated by disguising a Microsoft Exchange 2008 server running on Windows Server 2008 RC2 to appear as a Sendmail 8.6.9 server running on Linux 2.6. Both the NMAP and Nessus network scanners were deceived into incorrectly identifying the Exchange server. It is important to recognize that camouflage need not be a perfect deception: it is sufficient to sow enough confusion that an attacker is unable to take timely actions.

3.2.6 Hardware Hiding (Method #8). In 2010 Xilinx, the leader by market share for FPGA products, announced its first-in-class Zynq “Extensible Processing Platform” (EPP). Zynq, shown in Figure 4, combines rich FPGA logic with a dual-core ARM Cortex-A9 processor contained within the trusted boundary of a common die. The ARM’s dual-core Cortex-A series is the same architecture used in Apple’s iPhones. In 2011 Altera, number two by market share, answered the Zynq with their Hard Processor System (HPS) product – a similarly equipped dual-core ARM Cortex-A9 processor with FPGA logic and a comparable peripheral set. Xilinx’s Zynq processor became commercially available in the third quarter of 2012. Altera’s product was released in the middle of 2013. These devices are now being rapidly adopted as a general building block for a broad range of embedded systems applications including automobile driver assistance, factory automation, consumer electronics, military radios, medical imaging, broadcast cameras, and both wired and wireless communications (including routers and switches). In 2014 Xilinx and Altera announced next-generation devices that include additional logic, higher clock speeds, and 64-bit ARM Cortex A-50 processors. In 2014 Intel announced next-generation XEON processors with on-chip FPGA logic.

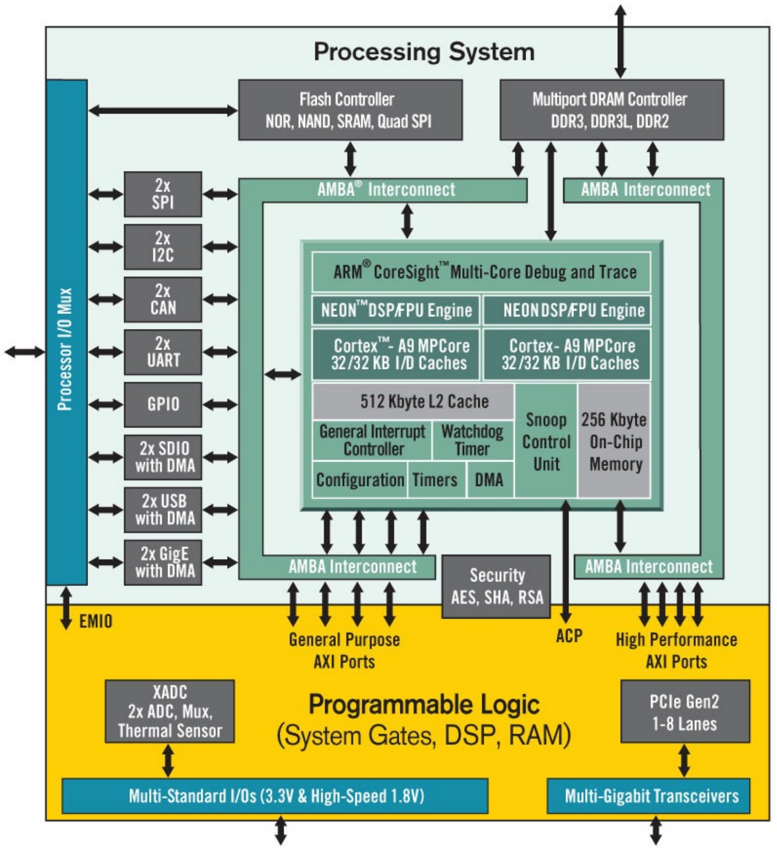

Figure 4: The Zynq System-on-a-chip Architecture

From a security perspective the impact of onboard FPGA logic is to make it possible to load *hardware designs* in addition to firmware and software onto the Zynq. As a result, security mechanisms can be embedded in a system in a way that is *not visible* to software running on the processor or embedded within flash (such as a malicious implants). In addition, the availability of on-chip encryption hardware opens the door to integration of full memory encryption **(Method #5)**. These devices are not simply an incremental design evolution to embedded processors, but the onset of a radical new way to structure systems: They open the door to a new generation of secure operating systems and embedded applications, where core system components are *completely hidden in hardware*.

The vulnerability of operating systems is often characterized as a “race to the bottom” of the software stack: if an APT is able to gain the lower most privileged levels of execution, it is able to hide. By hiding operating system structures read-only in hardware it is possible to *own* the base of the stack in hardware and thereby guarantee that APTs are observable. This method has been explored from several different perspectives in this project. Operating system internals, such as the Bear scheduler and all associated process structures, have been directly implemented in the FPGA providing not only higher levels of security, but also increased performance. In addition, a proof of concept hardware monitor has been implemented that continually hashes process code from the FPGA – thereby allowing detection of code patching. The maturity of the available co-design tools was also assessed as part of these efforts and the resulting performance tradeoffs were quantified.

3.2.7 Course-Grain Forensics (Method #9). Stephen Kuhn, who was supported under an Air Force scholarship, conducted the work described in this method while thesis supervision was supported by the Attacking Time project. The work was conducted on Linux since the Bear system was in its early stages of development when the method was developed; the underlying mechanisms that enable this technology **(Methods #1-4)** are integrated into the current ***Bear*** hypervisor.

The use of a minimalist hypervisor in the Bear system opened the door to discovery of zero-day exploits. The approach leverages the *hypervisors* ability to introspect into a virtual machine, running on top of it, and observe its state. Unfortunately, to maintain an account of malicious process actions, current hypervisors track every byte in memory transactions, significantly reducing system performance. To alleviate this burden, this work employs a novel course-grain, low overhead, tracking technology that can be incorporated within message-passing microkernels. The technology lowers the recording burden by storing <u>*only*</u> those actions designating the *trail of progress* that can potentially originate from an exploit. This trail provides a new course-grained forensics technique for exploit discovery that correlates message traffic with process actions. This approach uses the following core technologies:

- **Memory Introspection**: A memory translation technology that makes it possible to observe arbitrary memory locations within a running microkernel from the hypervisor and decode the resulting information. This represents an enabling technology for new tools that are able to decode the content of memory, monitor all running processes, and/or unwind the function call stack. All of these higher-level tools can operate from the safety of an outside observer, the hypervisor, to prevent tampering with the results by any malicious code present in the kernel or higher layers of the system.
- **Event Tracking**: A low-overhead tracking technology that provides detection and recording for events of interest for later analysis. This technology is not provided by hardware on Intel 64-bit architectures. Events that are of particular interest in exploit discovery are: process creation and inter-process communication.
- **Network Tracking**: A low-overhead tracking technology that allows course-grain recording of network events. The record of these events provides the ability to correlate process creation history with network traffic/messages.
- **Automated Exploit Discovery**: An automated, high-speed correlation technology that combines the history of process creation with the history of network traffic. This technology rests upon a unique database addressing scheme that facilitates identification of all packets that may have been associated with an intrusion, assuming an anomalous event has been identified. It also illuminates what actions were taken by the attacker on the system after the intrusion.

Collectively these technologies are used to provide:

- Operating system mechanisms to associate messages with *process genealogy*. This work involved novel techniques for observing virtual machine state, recording and correlating recorded network traffic.
- Exploit discovery techniques and algorithms based on indexed, course-grain *tracking* of processes and *elimination* of messages. The latter significantly reduces the search space associated with exploit discovery.
- An experimental study of forensic exploit discovery that locates test exploits against a known ground truth and quantifies the overheads associated with introspection, tracking, and traffic elimination.

3.2.8 Resilience (Method #10). This final method is concerned with resilience: the ability of a distributed application to operate through faults, errors, and computer network attacks. The applications are assumed to operate on multiple processors and share information through message-passing in a manner consistent with the industry standard -- MPI. Since computer systems are increasing in scale across the board—the number of processors per computer, the amount of memory, the speed and capacity of networks, etc. – a central goal is *scalability:* The ability to scale an application with the available technology. Two key questions were of central importance:

1. How do we design an application that will operate through failures and attacks even if they are never detected?
2. How to automate process management of distributed applications in a scalable manner?

The work described here was developed at a time when the Bear operating system was in its early stages of development. Thus to make progress, it was developed on top of Linux using the rMP message passing primitives described previously. These primitives were implemented using macros that expanded directly into OpenMPI communication primitives. Thus all of this work transfers directly to the Bear operating system which provides a native implementation of rMP.

A variety of approaches have emerged to provide reliability for distributed applications that rely on the static replication of resources. These include checkpoint/restart, process migration, and process replication. In contrast, *computational resiliency* provides reliability through *dynamic* replication of resources. The Scalable Concurrent Programming Library (SCPlib), developed at Syracuse University in the late 1990‘s, was an early proof-of-concept for distributed and dynamic process replication. SCPlib was a library that allowed application programmers to build resilient process groups and enable process failure detection with dynamic process regeneration. However, the main conclusion from this research was that resiliency is too complicated for application programmers. To be practical, the concepts need to be provided to distributed applications *automatically* and *transparently*.

Figure 5 shows the relative impact of resiliency and static replication on the reliability of applications under attack. Static replication alone provides graceful performance degradation to the point of failure because each attack reduces the number of replicas permanently. In contrast, resiliency dynamically reconstitutes the desired number of process replicas after each attack, allowing the application to continue operation with the same level of assurance. The malicious actions have no long-term effect.

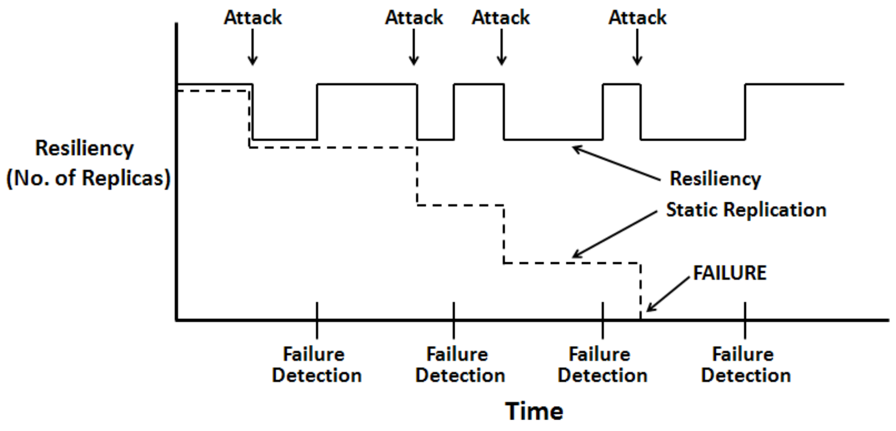

Figure 5. Computational Resiliency and Static Replication

The approach in this project built on the concept of resilience introduced in the SCPlib research but provides *automatic* and *transparent* resilience for large-scale concurrent applications. This approach is achieved by implementing resiliency mechanisms within the *operating system*. A collection of novel operating system technologies have been designed to dynamically replicate processes, automatically detect inconsistencies in their behavior, and transparently restore the level of resiliency as the computation proceeds. Figure 6 illustrates how this strategy is achieved. At the application level, three processes share information using message passing. The underlying operating system directly implements a resilient view that replicates each process and organizes communication between the resulting *resilient process groups*. Individual processes within each group are mapped to different computers to ensure that a single failure or attack cannot impact an entire group.

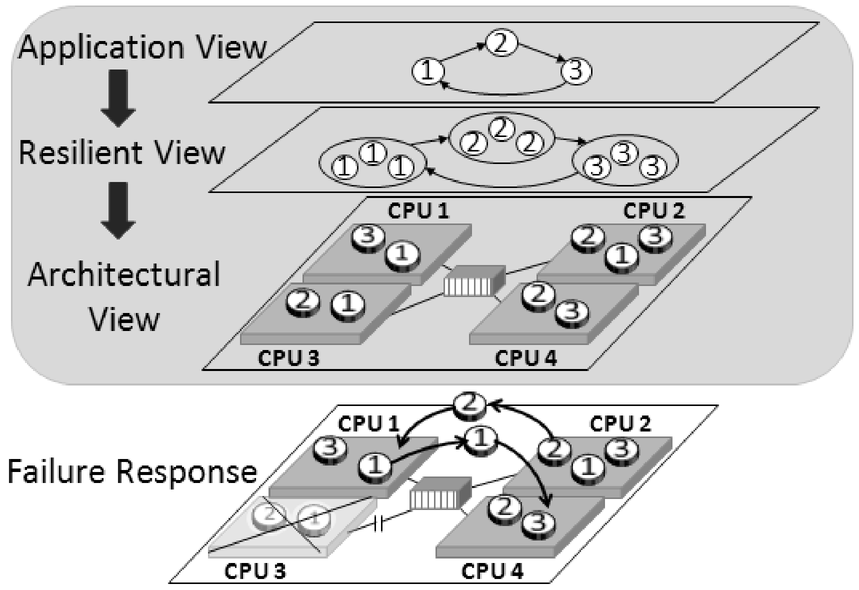

Figure 6. Dynamic process regeneration

The base of the figure shows how the process structure responds to failure or attack. The figure assumes that an attack is perpetrated against processor 3, causing processes 1 and 2 to fail or to portray communication inconsistencies with other replicas within their group. Failures are detected by communication timeouts and message comparison. These failures trigger automatic process regeneration; the remaining consistent copies of processes 1 and 2 dynamically regenerate a new replica and migrate it to processors 4 and 1, respectively. As a result, process resiliency is reconstituted, and the application continues operation with the same level of assurance.

In order to achieve this approach, several features are required that are not directly available in modern operating systems. Process *replication* is needed to transform single processes into process groups. Point-to-point communication between application processes must be replaced by group communication between process groups. Mechanisms to detect process failures and inconsistencies must be available to initiate process *regeneration*, and process *migration* is required to move a process from one processor to another. Finally, as processes move around the architecture, it is necessary to provide control over where processes are *mapped*. In order to prevent prohibitive communication costs, process management policies are used to *maintain locality* within process groups. This tactic enables *locality-based failure detection*, in which transit delays from replicated messages are used to predict an upper bound on the delay for communication timeouts.

These basic resiliency capabilities add complexity to process management. Scalable solutions require distributed and automated implementation. Process scheduling algorithms must meet both resiliency and performance requirements: (1) Maintain locality between processes in the same process group. (2) Map process replicas to different processors. (3) Distribute the load across the system.

The method developed involves novel operating system technologies that provide resilience for distributed applications. The significant contributions concern resiliency *mechanisms* and *policies* associated with a resilient message-passing technology, *rMP*. The resiliency mechanisms achieve process replication, adaptive failure detection, and dynamic process regeneration automatically and transparently within an operating system. Resiliency policies are explored through design and evaluation of alternative algorithms for distributed process scheduling. Finally, an analytical framework has been developed to enable concrete reliability analysis of the proposed approach to resilience. The contributions of this work include:

- A Linux prototype of the rMP technology that provides a resilient application programming interface (API) and constitutes a minimalist alternative to the Message Passing Interface (MPI). The API is implemented through a kernel-level communication module that provides applications with automated resiliency mechanisms.
- A collection of failure detection algorithms based on adaptive communication timeouts and message comparison. Adaptive failure detection uses process group locality as a basis to detect anomalies in message delay during group communication. Comparison of replicated messages allows detection of process inconsistencies through majority voting.
- Replication and migration mechanisms enable transparent regeneration of message-passing processes. Regeneration is accomplished by blending prior work in

Linux-based migration with the distributed communication support of the rMP technology.

- The problem of non-determinism in resilient applications is introduced and explored. Non-deterministic processes pose a significant challenge for replication technologies. A preliminary solution is proposed that preserves resiliency for rMP applications.
- An analytical framework is presented that includes a collection of performance benchmarks and analyses that allow the overhead of resilience to be evaluated.
- Resilient process management policies are explored through a comparative analysis of robotic swarming algorithms for distributed process scheduling. Swarming algorithms enable distributed processes to achieve a common goal while imposing *swarm cohesion* (i.e. locality between members of a swarm). The basis for the comparison is a novel set of benchmarks with an associated sensitivity analysis that capture the primary attributes of the process management problem.
- A novel DIFFUSE algorithm is presented, inspired by the notions of *heat diffusion* and *robotic swarming*. Heat diffusion is emulated to distribute processes across scalable computer architecture. Robotic swarming techniques are used to maintain locality between replicated processes. This work combines concepts from both perspectives to integrate the goals of resilience and performance in a single strategy.
- A second analytical framework is developed to perform reliability analysis of the rMP design with respect to common fault and threat models

4 RESULTS AND DISCUSSION

4.1 CORE RESULTS.

The core results from the project are constituted in proof-of-concept prototypes that realize the concepts and methods described in Section 3. However, some particularly noteworthy aspects of these implementations are described here.

The expected number of potential vulnerabilities in a code base is roughly proportional to the number of lines of code [29]: approximately 0.16 errors per thousand lines. The Bear kernel and hypervisor were designed to extensively share code in order to minimize the attack surface **(Method #2)**. As noted earlier, these have considerable overlap in functionality. For example, memory management, PCI device auto-detection, and interrupt configuration must be performed at both levels. Accordingly, the Bear source code consists of self-contained modules that can be compiled and used in either the hypervisor or the kernel to provide these services. Furthermore, these code modules share generic implementations of well-known data structures, including a linked list and a hash table. These flexible implementations eliminate code redundancy.

To demonstrate the relative size of the Bear attack surface, we compare the number of lines of code (LOC) in the kernel and hypervisor with those of other state of the art systems. The lines of code were counted using the open-source code analyzer *cloc* using only C sources and assembly code. As a result, the Bear results are accurate while the other results represent a lower-bound. Collectively, the Bear hypervisor and micro-kernel combined offered *three orders of magnitude* less code than monolithic solutions at the time of publication. This results into a correspondingly small attack surface that aggressively applies the latest hardware protection mechanisms, with a small number of predicted vulnerabilities.

Table 2: Kernel Comparison – Lines of Code

Kernel	Lines of Code
Bear Kernel	9,454 (7,399 shared)
Linux Kernel	10,639,311
FreeBSD	3,707,252
MINIX 3.2.0	16,109

Table 3: Hypervisor Comparison – Lines of Code

Hypervisor	Lines of Code
Bear Hypervisor	8,701 (7,399 shared)
Xen 4.1	262,191 (+ Dom0 kernel)
VMWare ESX	>150,000 (+ service terminal)

The number of lines of code in the 64-bit hypervisor and microkernel combined was 10,756 at the time of publication, two-thirds the size of the 32-bit MINIX kernel, with an expected defect incidence of less than two errors for a mature code base. The corresponding attack surface for the micro-kernel executable image was 62.02 Kbytes, the hypervisor was 54.78 Kbytes, bringing the combined attack surface size to 116.8 Kbytes. On ARM processors the microkernel is less than 3000 lines of code. Even with anticipated multicore expansions we would not expect the size of the system and attack surface to grow beyond double these numbers. This is easily capable of residing in a modern cache in its entirety.

Despite the desire for secure systems, the reality is that no system will see practical use without acceptable performance. To establish a baseline, the Bear system was benchmarked against Ubuntu Linux using the standard AIM9 benchmarking suite; the results are shown in Table 5. Bear was set to context switch every 10msec by default; therefore the Add benchmark, for example, involves approximately 15,000 context switches. Although this number is small compared to the number of addition operations, the benchmark would highlight adverse performance in memory management or interrupt handling. In contrast, Linux includes optimizations not present in Bear that allow processes to run for longer time slices based on the system state.

Obviously, our presumption is that Bear will be slower: the Linux kernel was released in 1991 and has been under continuous improvement and optimization ever since. In contrast, Bear is a research prototype developed primarily to explore resilience over the last two years. In addition, Bear used a simple file system and a simple, slow disk driver as a stopgap measure until a more suitable read-only file store could be integrated. Thus, benchmarks such, as fork and exec, involving file operations are dominated by the disk driver’s (lack of) performance.

Table 4: Performance Comparison

Routine	Ubuntu 12.04	Bear w/o Hypervisor	Bear with Hypervisor
Add	144 sec	150 sec	151 sec
Mul	250 sec	263 sec	261 sec
Div	1335 sec	1807 sec	1812 sec
Fork	0.3 sec	3.2 sec	3.2 sec
Exec	0.3 sec	3.2 sec	3.2 sec

Our primary conclusion is that the additional overhead created by the hypervisor, our source of kernel resilience, is negligible. All 5 tests show that enabling the hypervisor does not lead to a significant performance loss – either in time or CPU cycles. Furthermore, Ubuntu 12.04 running on a Linux kernel 2.6.38-15-generic is *only* 5% faster than Bear on the core Add and Mul benchmarks. As expected, the optimizations present in Ubuntu result in better performance on other benchmarks. However, there exist simple optimizations to e.g., Bear's admittedly naïve implementations of fork and exec, that could substantially increase performance with little effect on attack surface.

4.2 MEMORY ENCRYPTION (METHOD #5)

Our research associated with memory encryption solves the problem of increasing attacks on *data in use* in memory (e.g. cold boot attacks, memory scraper viruses, bus snooping and injecting, etc.) which has been exacerbated by the growing size of memory, changing usage models which invalidate old assumptions of volatility and increasing adoption of full disk encryption (FDE). In order to mitigate this problem, the idea of increasing the artificial diversity of RAM in order to *increase attacker workload* to the point where the costs of attack outweigh the benefits via memory encryption was explored. In this way, commercial systems can be protected since criminals typically target the most vulnerable systems. Further, time sensitive information such as that used in military operations would not be available to attackers until after its useful life (e.g. after a mission is complete).

While there have been three decades of research into memory encryption, that research has focused primarily on the design of the ideal monolithic processor with a hardware engine integrated into the fetch-decode-execute gateway. Other, more recent research has focused on software only approaches but these have proven too costly in overhead. However, recently there has been a commoditization of security hardware into processors such as the Intel AES-NI and various ARM architectures. The hypothesis in this work was that memory encryption could now be implemented with acceptable overheads using this nascent security hardware.

The hypothesis was explored through the various prototypes described in Section 3.2.3 from static encrypted processes (SEP) and through dynamic (DEP) and mutually distrusting processes (MDP). SEP sought to introduce synthetic diversity into memory to protect microcontrollers and other real-time processors commonly used in industrial control systems (e.g. lacking a memory management unit and little to no cache) via a one-time decryption into internally protected space. This technique produced very little overhead. DEP sought to introduce synthetic diversity into memory to protect smart phone and other mobile computing devices characterized by multitasking operating systems including memory management units and cache. For the first time in the memory encryption literature, implementation on commodity hardware enabled exploration of protection at *process segment* granularity. Protection of code and the PCB-stack data was *transparent* to processes (and developers) and the overhead was quite modest since that overhead was only experienced at context switch time (approximately 200 times per minute). Protection of heap objects is transparent for objects that fit into iRAM. However, for large structures that do not display temporal/spatial locality changes were required to application code and the protection was more expensive. Still, the results were better than those in the literature. Further, heap objects that fit into iRAM demonstrated *better* performance than unprotected versions since iRAM has shorter access times than eRAM. Finally, the DEP approach was extended to protect mutually distrusting processes (MDP) from each other via an increase in key granularity (i.e. a unique key per process) resulting in a very modest (~1900 cycles) increase in overhead.

These techniques protect against software and hardware based confidentiality and integrity attacks and are portable to currently deployed general-purpose, security-enhanced processors. An analytical framework was presented to include performance benchmarks and analyses on the overhead of memory encryption at *process segment granularity*. This work is the first in the genre to identify and explore the integrity protections afforded by memory encryption. The problem of *self-modifying code* associated with memory hierarchy interaction in a memory encryption system was introduced and explored. Finally, memory encryption techniques were explored through a comparative analysis of three decades of research and proposed solutions. Widely varying assumptions and experimental conditions were controlled to provide a basis for comparison of that research.

These systems and techniques were demonstrated in proof-of-concept implementations and exemplars. Memory encryption has been implemented to provide *automatic* and *transparent* protection for applications. This transparency is achieved through extension of a secure microkernel that was ported to an ARM Cortex A8 processor. The techniques have been implemented as modifications to linker scripts, initialization, process creation and context switching routines as well as new modules for interfacing with the encryption decryption unit (EDU). These techniques have been demonstrated by encrypting processes while they reside in external RAM (eRAM) thereby adding *synthetic diversity*. The implementations cover a range from an unsophisticated processor with no memory management unit (MMU) and cache to one with an MMU and 192 KB of L1/L2 cache. Additionally, various granularities of protection are explored. Finally, exception-handling routines have been developed and experiments executed to understand the protections afforded against code and data injection. The low overhead results for typical workloads (~1.3%) and ability to easily optimize even the worst-case examples to ~7% overhead indicate that memory encryption is viable today on *security-enhanced commodity processors*.

4.3 HARDWARE HIDING (METHOD #8).

Several results particularly notable from our work with hiding in hardware on the Zync processor:

- A hidden hardware scheduler significantly decreased the operating system attack surface hiding up to 90% of the core kernel logic and data structures in the FPGA.
- There was a simultaneous improvement in performance: up to 50% for hand-coded VHDL.

- When pushing data from the processor to the FPGA fabric, there were architectural limitations in the processor-FPGA interconnect that limited performance.
- FPGA logic utilization was extremely low, only 2-5% for the scheduler, allowing significant room for expansion in the use of the FPGA.
- The Xilinx Vivado design suite is advancing the state-of-the-art of integrated C/HDL co-design. Currently, however, the C, HDL, and high-level synthesis (HLS) tool chains are not integrated sufficiently to support fully automated transformation of C-code into FPGA bit-streams.
- There are several constraints in HLS based on incomplete support for the high-level languages that limit its utility.
- There are additional limitations to HLS interfaces between the processor and FPGA that inhibit performance of transformed code; the current generation of HLS tools resulted in a 50% reduction in performance over the hand-coded VHDL.
- A sha256 hardware monitor performed at 40 times the performance of an equivalent C code implementation and was hidden in the FPGA.

Our general conclusion is that the current co-design tools are inadequate for complex systems efforts in which hardware specification, in a high-level language such as C, is *automatically* translated into combination processor-FPGA implementations. Instead there are two separate functional tool chains: standard ANSI-C compilers for the dual-core A9, and Verilog/VHDL for the FPGA fabric. However, it is possible to accelerate application of the technology base for co-design through the exploration of new hidden security mechanisms. We believe that good exemplars will point the way for early adoption of system-on-a-chip devices in DoD applications.

5 CONCLUSIONS

Military systems have gained tremendously from the cost and flexibility benefits afforded by widespread adoption of commercial off the shelf (COTS) technology -- to the point where it is now difficult to imagine how we might operate, with similar levels of assurance and efficiency, using non-COTS methods. However, in times of tension, critical mission capabilities *must* continue to operate, even if major components of “the network” are unavailable and the systems upon which we rely are repeatedly compromised by error, fault, or malicious action. It therefore behooves us to apply Occam’s razor to pare back the layers of complexity that have been thrust upon us by commercial vendors, in light of the controlled environment in which DoD operates, to selectively improve *resilience* and *increase attacker workload*.

Our approach is to use COTS subsystems, accepting their imperfections, but augmenting them with ideas from the fault-tolerance, distributed computing, and encryption communities. The research described here has explored how we might pursue this goal using three basic non-deterministic precepts:

- Don’t trust what you have – *continually validate, replicate and regenerate*,
- Don’t advertise what you do – *continually hide and camouflage*, and
- Don’t be predictable – instead be *mobile* and *non-deterministic*.

The Bear system uses overlapping regenerative techniques, combined at every layer of the system, from the user to the hardware. These methods deny surveillance by continually invalidating surveillance data, hiding in the network, and using camouflage. Persistence is denied by non-deterministically replacing, refreshing, replicating, diversifying, and/or relocating components so as to continually re-establish trust. The methods can be incorporated individually, as independent modes, or collectively and continuously for critical missions.

6 REFERENCES

[1] C. Nichols, “Bear - a Resilient Core for Distributed Systems”, Masters Thesis, Thayer School of Engineering at Dartmouth College, 2013.

[2] M. Kanter, “Enhancing Non-determinism in Operating Systems”, Ph.D. Thesis, Thayer School of Engineering at Dartmouth, October 2013. (Funded by MRC)

[3] M. Henson, "Attack Mitigation through Memory Encryption", Ph.D. Thesis, Thayer School of Engineering, September 2014 (Co funded by USAF).

[4] M. Henson and S. Taylor, “Memory Encryption: A Survey of Existing Techniques”, To *ACM Computing Surveys*, **vol 46**, issue 4, Article 53, March 2013.

[5] M. Henson and S. Taylor, "Attack Mitigation through Memory Encryption of Security Enhanced Commodity Processors", *In the Proceedings of the 8th International Conference on Information Warfare and Security (ICIW '13),* Hart, D. (eds.) pp. 265-268. March 2013.

[6] M. Henson and S. Taylor. "Beyond Disk Encryption: Protection on Security Enhanced Commodity Processors" *Proceedings of the 11th International Conference on Applied Cryptography and Network Security (ACNS '13),* June 25-29, 2013.

[7] S. Kuhn, “Automated Forensic Techniques for Locating Zero-day Exploits”, Ph.D. Thesis, Thayer School of Engineering, December 2013. (Co funded by USAF)

[8] S.,Kuhn, and S.Taylor, "Increasing attacker workload with virtual machines," In Proceedings of MILCOM 2011 , pp.2176-2181, 7-10 Nov. 2011.

[9] M. Kanter and S. Taylor, “Camouflaging Servers to Avoid Exploits”, TR11-001, Thayer School of Engineering at Dartmouth College, Feb 2010.

[10] J. Dahlstrom and S. Taylor, “Migrating an OS Scheduler into Tightly Coupled FPGA Logic to Increase Attacker Workload”, *In proceedings of MILCOM 2013*, pp 986-991, November 2013.

[11] S. Kuhn and S. Taylor, "A forensic hypervisor for process tracking and exploit discovery," *In Proceedings of MILCOM 2012*, pp.1-5, Oct. 29 2012-Nov. 1 2012.

[12] K. McGill, “Operating System Support for Resilience”, Ph.D. Thesis, Thayer School of Engineering at Dartmouth College, 2011.

[13] K. McGill and S. Taylor, “Computational Resiliency for Distributed Applications,” In *Proceedings of MILCOM 2011*, pp 1472-1479, Nov 2011.

[14] K. McGill and S. Taylor, “Application Resilience with Process Failures,” *In Proceedings of the 2011 International Conference on Security and Management,* Las Vegas, Nevada, July 2011.

[15] K. McGill and S. Taylor, “Robot Algorithms for Localization of Multiple Emission Sources,” *ACM Computing Surveys (CSUR),* **vol 43**, no 3, April 2011.

[16] K. McGill and S. Taylor, “Diffuse algorithm for robotic multi-source localization,” In *Proceedings of IEEE 2011 International Conference on Technologies for Practical Robot Applications,* April 2011.

[17] K. McGill and S. Taylor, “Comparing Swarm Algorithms for Multi-source Localization,” *In Proceedings of the 2009 IEEE International Workshop on Safety, Security, and Rescue Robotics*, Denver, Colorado, November 2009.

[18] K. McGill and S. Taylor, “Comparing Swarm Algorithms for Large Scale Multi-source Localization,” *In Proceedings of the 2009 IEEE International Conference on Technologies for Practical Robot Applications*, Woburn, Massachusetts, November 2009.

[19] C. Nichols, S. Taylor, J. Keranen, and G. Schultz, “A Concurrent Algorithm for Real-Time Tactical LiDAR”, *In the proceedings of 2011 IEEE Aerospace Conference,* Big Sky, Montana, March 2011.

[20] D. Kennedy, J. O’Gorman, D. Kearns, and M Aharoni, “Metasploit: The Penetration Testers Guide”, No Starch Press, 2011.

[21] L Davi, A Dmitrienko, AR Sadeghi, M Winandy, “Privilage Escalation Attacks on Android”, Information Security, Springer 2011.

[22] Greg Hoglund and Jamie Butler, “Rootkits”, Addison-Wesley Professional Press, 2005.

[23] Chris Eagle, “The IDA Pro Book”, No Starch Press, 2011.

[24] Eldad Eilam, “Reversing”, Wiley, 2005

[25] J.E. Forrester and B.P. Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing", *4th USENIX Windows Systems Symposium,* Seattle, August 2000. Appears (in German translation) as "Empirische Studie zur Stabilität von NT-Anwendungen", iX, September 2000.

[26] Checkoway, Stephen; Halderman, J. Alex; Feldman, Ariel J.; Felten, Edward W.; Kantor, Brian; and Shacham, Hovav; (2009); “Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage”, *in Proceedings of the USENIX/ACCURATE/IAVoSS Electronic Voting Technology Workshop*, August 2009.

[27] W. A. Arbaugh, D. J. Farber, and J. M. Smith. “A secure and reliable bootstrap architecture.” In *Proceedings of the 1997 IEEE Symposium on Security and Privacy* (SP '97). IEEE Computer Society, Washington, DC, USA, 65-. 1997.

[28] B. Blunden. The Rootkit Arsenal: Escape and Evation in the Dark Corners of the System. USA: Jones and Bartlett Publishers, Inc. 2009.

[29] Pandey and Tiwari, “Reliability Issues in Open Source Software.” International Journal of Computer Applications, vol. 34 issue 1, pp. 34-38. 2011.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ANSI	American National Standards Institute
APT	Advanced Persistent Threat – cyber implant that persists and hides
ARM	Advanced RISC Machines – a type of computer processor
BIND	Domain Name Service software
CPU	Central Processing Unit
COTS	Commercial of the shelf
DEP	Dynamic Encrypted Processes – encryption technology
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name Service
EPP	Extensible Processing Platform – a type of processor
EPT	Extended Page Tables
eRAM	external Random Access Memory
FDE	Full Disk Encryption
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HLS	High Level Synthesis
HPS	Hard Processor System – a type of processor
HUMINT	Human Intelligence
IDS	Intrusion Detection System
IOMMU	input/output memory management unit
IP	Internet Protocol
KVM	Kernel-based Virtual Machine
MAC Address	Media Access Control Address – identifies a network interface
MDP	Mutually Distrusting Processes – encryption technology
MINIX	mini-Unix -- a micro-kernel based operating system
MMU	Memory Management Unit
MPI	Message Passing Interface software system
MULTICS	Multiplexed Information and Computing Service – an operating system
NMAP	Network Mapper software
OODA	Observer Orient Decide Act -- decision cycle
RAM	Random Access Memory
ROM	Read-only memory
ROP	Return Oriented Programming – a form of cyber attack
rMP	resilient Message Passing software system
SEP	Static Encrypted Processes – encryption technology
SIGINT	Signals Intelligence
TCB	Trusted Computing Base
TTP	Tools, Techniques, and Procedures -- operational aspects
VHDL	VHSIC Hardware Description Language
VT-d	Intel virtualization technology for Directed I/O (devices)
VT-x	Intel virtualization technology
XD-bit	Execute Disable bit (or NX– no execute) – a form of memory protection