Bear – A Resilient Operating System for Scalable Multi-processors[1]

Stephen Taylor, Michael Henson, Morgon Kanter,
Stephen Kuhn, Kathleen McGill, and Colin Nichols
tr11-005

Abstract

This paper describes a minimalist operating system design aimed at scalable multi-processor systems whose primary goal is *resilience*. The design is expressly targeted toward critical military applications for the purpose of operating through failures, errors, and malicious attacks. Lessons learned from several key proof-of-concept components, implemented as Linux kernel modules, are currently being incorporated into a new from-scratch system.

Current operating system designs have sought to utilize a base of trust in hardware and extend trust to software through deliberate layering. Our approach assumes instead that adversaries will conduct *surveillance*, will be successful in gaining access, and will *persist undetected*. We propose multiple, overlapping, non-deterministic techniques that continually re-establish trust by dynamically regenerating core components of distributed computations and their underlying execution environment. The cumulative effect of these changes in design style is to *increase attacker workload* by denying surveillance and persistence over time-scales consistent with tactical military operations. 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 military mission to proceed – *living with insecurity.*

Introduction

Today's commercial off the shelf (COTS) hardware is inherently insecure: It has been shown that malicious circuitry can be incorporated into an IC with relatively few gates making detection extremely difficult [1]. This circuitry can provide a wide range of effects rendering it impossible to establish a hardware base of trust and reliably extend trust into other operating system layers. One method to mitigate this vulnerability is to utilize trusted foundries for microelectronics fabrication, where every stage of the manufacturing pipeline is controlled [2]. However, irrespective of the hardware mechanism involved, there are two basic use cases for a hardware implant: a time-bomb that performs a triggered effect *without command and control* (C2) or a mechanism to enable effects under remote C2. The first option is of limited use and is analogous to any other general failure or error; it can, and routinely is, combated by skilled practitioners within the armed services through diversity and/or spare equipment. The second more interesting case can be mitigated by denying or degrading C2: increasing attacker

[1] This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-09-1-0213.

workload to the point where there can be no significant impact on the time-scale of tactical missions.

Intrusions with remote C2 may involve *surveillance* to determine if vulnerability is present, use of an appropriate exploit or other access method, and *persistence* for a time sufficient enough to carry out some malicious effect. The time spent in surveillance and persistence may range from seconds to months, depending upon the intended effect.

This paper describes mechanisms that we have developed to *deny surveillance and persistence*. The overall system design is shown in Figure 1 with existing proof-of-concept implementations, based on Linux kernel modules, signified by an asterisk (*). The design separates core functions into four layers typical of modern micro-kernel designs such as MINIX [3]. The number of vulnerabilities existing within a system is directly correlated with the *size of the code base*, indicating that there is substantial value in the intellectual process associated with applying Occam's razor to reduce the attack surface. Multiple, overlapping regenerative techniques are 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, and/or relocating components so as to continually re-establish trust. The methods can be incorporated individually, as independent "war-mode", through loadable modules or collectively and continuously for critical missions.

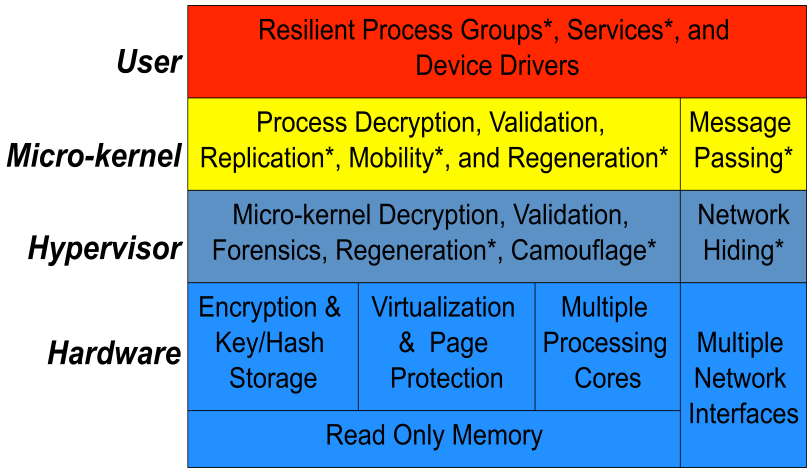

Figure 1. Software Layers

The hardware layer shows key existing COTS technologies that we seek to project up into every layer of the operating system design. *Multiple-cores* are used to carry out a wide variety of background operations involving decryption, validation, and non-deterministic copying and moving. All executable code is stored *encrypted* and *hashed*; it is validated before decryption through *hashing* and *page protection* (similar to segment protection in MULTICS), and decrypted piecemeal just-in-time for *virtualized* execution.

Unfortunately, the desire for field-upgradable hardware has opened a new dimension to malicious code in firmware and/or flash [4]. In consequence, there are some simple additions to COTS systems that are particularly valuable for improving resilience. These include *read-only memory* from which to draw *encrypted gold-standard images* that represent the code of final recourse, *removable links* on the primary write-lines to core flash components, and/or an *out-of-band network channel with associated micro-controller* to control flash updates and/or provide a forensic interface. The latter facility can be used to repeatedly re-flash devices when they are not in use or at designated system refresh times.

The Hypervisor Layer

The hypervisor layer exists to provide operations on micro-kernels. The normal role of virtualization is to share the underlying hardware between multiple operating system instances. In contrast, the minimalist hypervisor in Figure 1 exists primarily to *regenerate* and *hide* the micro-kernel. Regeneration involves a complete replacement of the micro-kernel for the purpose of expunging root-kits, bots and other malware. SafeFox is a pioneering virtual browsing environment that also uses a clean slate approach to prevent persistence in application code [5].

Regeneration may occur whenever users are momentarily inactive, at fixed intervals by arrangement, or non-deterministically with appropriate warnings. The hypervisor regenerates a new micro-kernel in the background from a gold-standard copy stored in the underlying hardware; the active OS is then torn down while instantaneously switching to this new system. No attempt is made to detect intrusions or malware, and the operation occurs even if there is no malware present. Although this technique could be applied rapidly, we perceive that its most useful applications would be immediately prior to a critical event and successively at convenient intervals, perhaps hours apart.

Each new micro-kernel instance need not be exactly the same: a different version, selected non-deterministically, invalidates existing surveillance data and offers the opportunity to project known vulnerabilities with associated detection software. In addition, the presence of multiple NIC cards in the underlying hardware allows each new instance to non-deterministically choose an alternative network connection. These may be physically connected to completely different network segments, potentially behind different external proxies. From a surveillance perspective, the host appears to be a completely different machine available for only a short period at different parts of the network. This invalidates surveillance data with every move, in the style of pioneering work conducted at BBN. Finally, the hypervisor may also camouflage the micro-kernel, through alterations to its traffic, to project a completely different micro-kernel from that which is actually executing. Camouflage may also project known vulnerabilities and be associated with detection software.

Our proof-of-concept implementation of *regeneration* and *network hiding* is implemented using KVM and has explored the difficult end case associated with web servers which

offer static pages in addition to streaming and stateful content [6]. Since much of the traffic in modern networks is between clients and servers, rather than client-to-client, servers represent high-value, statically situated, concentration points for surveillance data. A variety of alternative Linux distributions, running the Apache web server, were chosen as gold-standard images. Figure 2 outlines the non-deterministic process executed by the hypervisor at a single host.

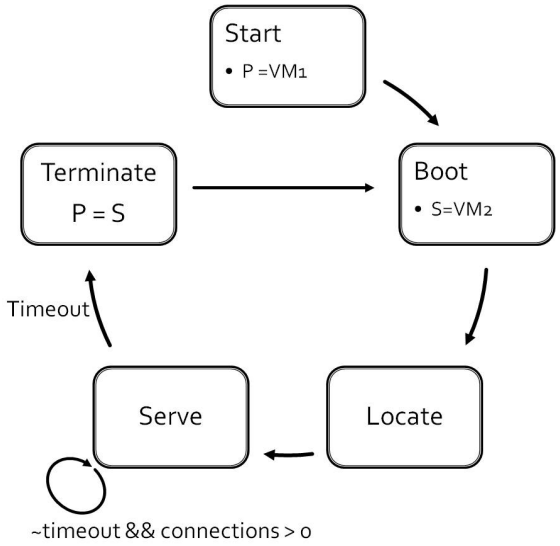

Figure 2: Regeneration and Network Hiding

The process begins when the hardware is restarted. The hypervisor enters the ***Start*** state and a *primary* virtual machine ***P*** is created by selecting a random operating system variant, chosen from the set of pre-configured images. A random network is chosen by selecting an associated NIC card from the available pool provided by the hardware; it is assigned a random IP address inside the chosen network segment. The network services for the primary virtual machine are then initialized with these properties, and an Apache web server is bootstrapped on top of the operating system. The state machine subsequently cycles through four primary states.

In the ***boot*** state a *secondary* virtual machine ***S*** is created in the background with a different operating system, web server, IP-address, and MAC address (NIC card). On multi-core systems this operation has negligible impact on the performance of the primary virtual machine. The ***locate*** state is used to contact a private DNS server to inform it of the network address of the primary server. Authenticated clients can use the DNS server to locate the primary server. In the ***serve*** state the primary server responds to incoming connection requests and serves web content. During this activity a random timeout is set, and a running count is kept of the number of active connections. The active server continues to serve as long as the timeout has not expired and there are active connections. If the timeout expires, the ***terminate*** state is used to switch virtual machines: the primary virtual machine terminates but only when all existing connections to it close. The secondary virtual machine becomes the primary virtual machine serving all new connections. The ***boot*** state is then re-entered where a new secondary is created.

This basic process limits the extent of a single connection for streaming and stateful content to the period of the timeout, which may be on the order of hours. If only static pages are being served they can be automatically handled by the retry mechanism of TCP/IP allowing the server to move at a higher rate. For clients, the timeout could be much smaller presenting a constantly moving target.

Current hypervisors do not provide convenient support for dynamically switching network cards and introspection into connection information. Their role is to provide a general sharing mechanism for the underlying network hardware in much the same way as a bridge. The more simple multiplexing operations described here offer the opportunity not only to inspect traffic but also change its characteristics for the purpose of *deception*. We have explored this concept in a proof of concept *camouflage* module that presents a false server fingerprint [7]. 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. It was able to reliably deceive Nessus OS detection, Nmap OS detection and service detection, and RING OS detection into incorrectly identifying the Exchange server.

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 within IP, TCP, and SMTP. The state-machine transitions between states by following normal protocol specifications on incoming traffic, but it responds with false outgoing traffic. The false fingerprint included a known vulnerability whose exploitation was detected automatically when used. 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.

The Micro-Kernel Layer

The role of the micro-kernel is to provide operations on processes. Unfortunately, operating system developers have been slow to embrace innovations such as read/write/execute protections available in MULTICS and trust concepts such as the Trusted Platform Module (TPM). In addition, there has not been an aggressive movement to project encryption techniques throughout the fundamental structures of commercial operating systems. Two processor technology trends are now setting the stage for a change in direction: protection capabilities integrated into paging mechanisms and the emergence of encryption as a first class citizen in processor design through both memory encryption and specialized instructions. When combined with multiple processing cores, to hide the overhead of housekeeping operations, and a small *key* store protecting a larger *hash* store, these innovations offer a unique opportunity: to change the basic *fetch-execute* cycle into one that universally applies a *fetch-validate-decrypt-execute cycle.* This transition would apply protection, hashing, and decryption on-the-fly, at different levels of granularity, to continuously re-establish code trust at run-time.

Our system design seeks to explore these ideas by leveraging the mature body of knowledge that has evolved in building modular micro-kernels. A particularly interesting aspect of MINIX 3 is that it places device drivers outside kernel-space and provides a re-incarnation server to restart them in the event of failure [3]. Our goal is to extend this concept to non-deterministically restart potentially compromised drivers from established gold-standards when they are idle, even if they do not fail, for the purpose of re-establishing trust. Carrying this process further, we seek to expand the regeneration of processes to include *replication* and *mobility* to provide a distributed form of resilience. This approach gives rise to a view of concurrent applications that is best described by a biologically inspired analogy:

> A concurrent application is composed of a large number of message-passing processes, each analogous to a new strain of ***roach***. Roaches have survived for millennia -- you can stamp on them, spray them, strike them with a broom but you'll never kill them all or prevent them from their goal of finding food (computing resources). To foil eradication efforts, they have evolved a few simple strategies: they are highly mobile (process migration), constantly pregnant (replicate), and operate in family groups for mutual support (collaborative). Our new strain also uses camouflage to conceal its location.

A wide diversity of DoD systems have sought to use *replication* in one form or another as a mechanism to provide fault-tolerance. These approaches provide graceful degradation of system performance to the point of failure, but are not sufficient to *guarantee* progress in the presence of repetitious malicious effects. We seek technologies that *dynamically regenerate* replication in response to inconsistencies and invariants sampled over time. The net impact of this process is to allow the level of system assurance to be maintained, ensuring that a military mission may continue unabated.

These concepts have been demonstrated in a recent proof-of-concept implementation based on Linux kernel modules that is illustrated in Figure 3. At the user level, a concurrent application is expressed through processes that cooperate through pair-wise message-passing primitives. The underlying operating system implements a resilient view that replicates each process and organizes communication between the resulting *process groups*. Point-to-point communication among user processes is implemented by multi-cast communication between process groups. Individual processes within each group are mapped to different computers to ensure that a single attack or failure cannot impact an entire group.

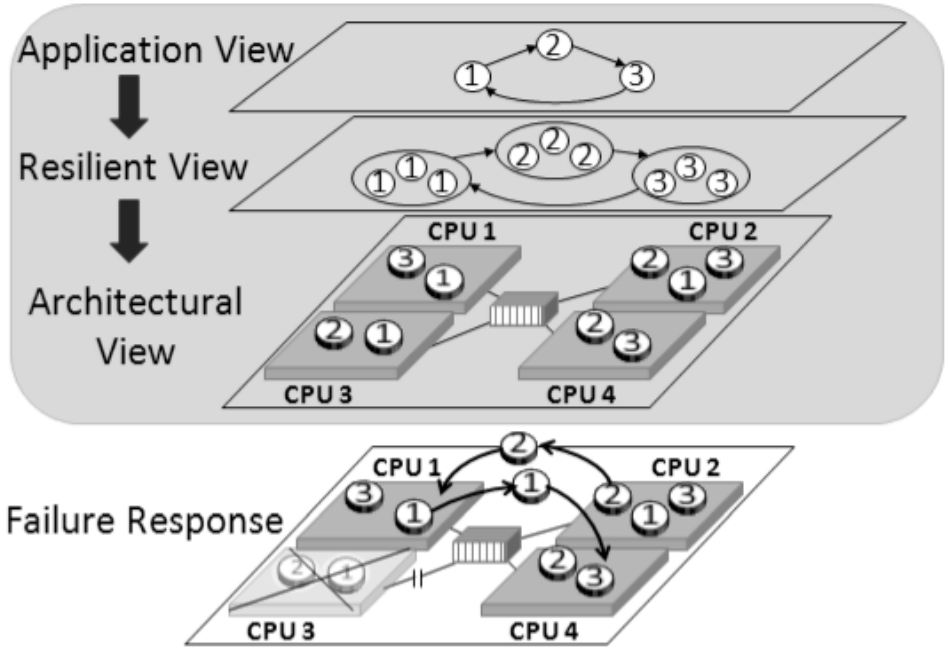

Figure 3. Dynamic process regeneration

The base of the Figure 3 shows how the process structure responds to an attack or failure: An attack perpetrated against processor 3, causes processes 1 and 2 to fail or to portray inconsistencies in behavior or communication when compared to other replicas within their respective groups. These inconsistencies are detected either by behavioral alerts, communication timeouts, and/or message comparison. Inconsistencies trigger an automatic process regeneration: the consistent copies of processes 1 and 2 are used to dynamically regenerate a new replica and migrate it to alternate processors 4 and 1 respectively. As a result, the process structure is reconstituted, and the application continues operation with the same level of assurance.

Our early research in this area culminated in a concurrent programming library, SCPlib, in which several non-trivial applications were implemented [8]. Unfortunately, the level of added complexity involved in directly programming resilience compounded the already complex task of concurrent programming. Lessons learned from this activity indicated that operating system mechanisms to *transparently* support resilience would not be unreasonably complex provided that a *simple* message-passing interface was employed. Support for the wide array of communication primitives in the industry standard Message Passing Interface (MPI) appears impractical.

Our current proof of concept uses a simple MPI-like application programming interface (API): A concurrent computation comprises a set of ***n*** communicating processes, numbered ***0*** to ***n-1***. A computation is initiated using a system call of the form: **msgrun <*program*> <*args*>**. These processes communicate using two blocking communication primitives:

- **msgsend(dest, &buff, size)** – send a message from *buff* of length *size* bytes to *dest*.

- **msgrecv(src, &buff, size, &status)** – receive a message from *src* (or ANY) into *buff* of length *size*; *status* is a structure designating the *source* of the message and its *length*, messages that are larger than *size* are truncated.

Each process may determine the number of processes in the computation from command line arguments and may use a system call of the form **msgpid(&mypid)** to determine its position in the numeric ordering. This simple interface explicitly excludes the ability to busy-wait on messages, a consistently problematic issue when running multiple applications concurrently. It also serves to provide *rendezvous* style inter-process communication on a single processor [3].

The API is sufficient to express all three of the prevalent concurrent problem solving strategies that utilize *functional*, *domain*, and *irregular* decompositions. For the purpose of experimental evaluations, we have developed a suite of message-passing applications that exemplify these strategies involving numerical integration, iterative solution of partial differential equations, and a non-trivial LiDAR processing algorithm respectively. The additional cost of resilience is insignificant when compared to the inherent cost of process replication required to achieve fault tolerance [9]. The application exemplars have been executed with triple redundancy; induced failures are detected through communication timeouts with recovery as shown in Figure 3. The kernel performs process mobility by copying the entire process state (including registers, heap, etc) to an alternative processor using kernel level TCP connections. This operation involves scanning the page tables, mapping in and copying the pages associated with the process. A forwarding protocol is used to handle messages that are currently in transit when the destination process is being migrated. Unlike other approaches to process mobility, this approach uses no global operations or check-pointing, is completely asynchronous, and leaves no residual dependencies [10, 11]. This allows it to be used non-deterministically at any host in the network with no network-wide observables. Currently, simple round-robin process mapping techniques, that place members of the same process groups in different processors, have been used in our experiments.

The transparent realization of resilience on large-scale concurrent architectures necessitates an automatic approach to process scheduling. Our early work in this area resulted in a general algorithm for load balancing based on the *heat diffusion equation* [12]. This approach has several attractive properties: It uses a simple, fast, scalable algorithm involving only nearest neighbor communication; global progress and convergence are guaranteed through well-established mathematical analysis. The algorithm has been shown, through simulation, to simultaneously balance multiple independent load distributions over large-scale architectures, even with huge random load injections. Vector based extensions to the algorithm allow multiple resources (including communication, memory, and CPU load) to be balanced simultaneously [13].

In extending this work to resilience we seek to introduce two basic scheduling constraints: (1) process replicas are *mapped* to different processors, and (2) replicas maintain *locality* within their groups in order to bound error detection timeouts. To achieve these constraints, we have developed a diffusive algorithm that incorporates

elements of swarming algorithms prevalent in the robotic literature. This algorithm has similar global progress and convergence properties to heat diffusion. It implicitly achieves the constraints required for resiliency through emergent properties associated with swarm dynamics: the mapping constraint through *obstacle avoidance* and the locality constraint through *swarm cohesion* [14].

Conclusion

Military systems have gained tremendously from the cost and flexibility benefits afforded by widespread adoption of commercial technology -- to the point where it is now difficult to image how we might operate, with similar levels of efficiency, using manual 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 actions. 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 improve resilience.

One approach is to use COTS subsystems, accepting their imperfections, but augment them with ideas from the fault-tolerance, distributed computing, and encryption communities. The research described in this paper explores how we might pursue this goal using three basic rules:

1) Don't trust what you have -- *validate*, *replicate and regenerate*,
2) Don't advertise what you do – *hide and camouflage*, and
3) Don't be predictable – instead be *mobile* and *non-deterministic*.

Notice

Reference

[1] S. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware," In Proceedings of the First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET), April 2008.

[2] S. Pope, "Trusted Integrated Circuit Strategy," IEEE Transactions on Components and Packaging Technologies, Vol. 31, No 1, pp. 230-234, 2008.

[3] Tanenbaum and Woodhull, "Operating Systems: Design and Implementation," Prentice Hall, 2006.

[4] PowerEdge R410 replacement motherboard contains malware, http://en.community.dell.com/support-forums/servers/f/956/t/19339458.aspx, Jul 10 2010.

[5] W. Jiang, H. Yih, and A. Ghosh, "SafeFox: A safe lightweight virtual browsing environment," Proceedings of the 43rd Hawaii International Conference on System Sciences (HICSS), Jan 2010.

[6] S. Kuhn and S. Taylor, "Increasing Attacker Workload with Virtual Machines", submitted to MILCOM 2011. (Available as Thayer Technical Report TR11-002 at http://thayer.dartmouth.edu/tr/reports).

[7] M. Kanter and S. Taylor, Camouflaging Servers to Avoid Exploits, submitted to MILCOM 2011. (Available as Thayer Technical Report TR11-001 at http://thayer.dartmouth.edu/tr/reports)

[8] J. Lee, S. J. Chapin, and S. Taylor, "Reliable Heterogeneous Applications," IEEE Transactions on Reliability, special issue on Quality/Reliability Engineering of Information Systems, Vol. 52, No 3, pp. 330-339, 2003.

[9] K. McGill and S. Taylor, "Operating System Support for Resilience," Submitted to IEEE Transactions on Reliability, (Available as Thayer Technical Report TR11-003 at http://thayer.dartmouth.edu/tr/reports).

[10] G. Valle, C. Morin, J. Berthou, I. Dutka Malen, and R. Lottiaux. Process migration based on gobelins distributed shared memory. In Proceedings of the workshop on Distributed Shared Memory, pages 325-330, May 2002.

[11] C. Wang, F. Mueller, C. Engelmann, and S. Scott. Proactive Process-Level Live Migration in HPC Environments. In Proceedings of the 2008 ACM/IEE Conference on Supercomputing, 2008.

[12] A. Heirich and S. Taylor "Load Balancing by Diffusion", Proceedings of 24th International Conference on Parallel Programming, vol 3 CRC Press pp 192-202, 1995. Outstanding Paper Award.

[13] J. Watts, and S. Taylor, "A Vector-based Strategy for Dynamic Resource Allocation", Journal of Concurrency: Practice and Experiences, 1998.

[14] K. McGill and S. Taylor, "Robot Algorithms for Localization of Multiple Emission Sources," ACM *Computing Surveys (CSUR)*, March 2011.