ROP Defense in the Cloud through LIve Text Page-level Re-ordering
The LITPR System

Angelo Sapello[1], C. Jason Chiang[1], Jesse Elwell[1], Abhrajit Ghosh[1], Ayumu Kubota[2] and Takashi Matsunaka[2]

[1]*Intelligent IA Systems Research, Vencore Labs, Inc., Basking Ridge, NJ, U.S.A.*

[2]*Network Security Laboratory, KDDI R&D Laboratories, Saitama, Japan*

Keywords: Return Oriented Programming, ROP Mitigation, Program Randomization.

Abstract: As cloud computing environments move towards securing against simplistic threats, adversaries are moving towards more sophisticated attacks such as ROP (Return Oriented Programming). In this paper we propose the LIve Text Page-level Re-ordering (LITPR) system for prevention of ROP style attacks and in particular the largely unaddressed Blind ROP attacks on applications running on cloud servers. ROP and BROP, respectively, bypass protections such as DEP (Data Execution Prevention) and ASLR (Address Space Layout Randomization) that are offered by the Linux operating system and can be used to perform arbitrary malicious actions against it. LITPR periodically randomizes the in-memory locations of application and kernel code, at run time, to ensure that both ROP and BROP style attacks are unable to succeed. This is a dramatic change relative to ASLR which is a load time randomization technique.

1 INTRODUCTION

Cloud computing environments currently implement several standard security protections such as network firewalls to protect against simplistic threats. With the incorporation of such protection and the increasing number of IT operations moving to the cloud, adversaries are exploring more sophisticated means to breach these environments. Typically such breaches occur via privilege escalation attacks. While privilege escalation is a concern on any system, it is even more so on cloud systems where an attacker gaining elevated privileges in one domain could potentially damage the hypervisor and other domains running on the system. Return-Oriented Programming(ROP) attacks are one way that an attacker can bypass defenses of a running system to gain elevated privileges. We propose a two tiered system called LIve Text Page-level Re-ordering (LITPR) to defend domain applications from within the domain's kernel and also defend the domain's kernel from within the hypervisor (see figure 1).

At a high level, the LIve Text Page-level Re-ordering (LITPR) system randomizes the location of the code segment (i) periodically (not just at load time) and (ii) at the fine grained page-level rather than at segment-level while the cloud based application is executing (see figure 2). ROP attacks rely on the identification of specific code blocks at specific memory locations. These code blocks are used to perform attack specific actions. LITPR relocates code blocks in a randomized fashion. Therefore, an attacker is unlikely to discover even a single address and even if they do, they learn nothing about the layout of the remainder of the application. They are restricted to a single code page. To ensure that the randomized code continues to run the LITPR system must ensure that all pointers from the code to other parts of the code, shared libraries and data are updated during the randomization process.

1.1 Return Oriented Programming(ROP) Attacks and Related Work

Return-Oriented Programming (ROP) was first discussed by Hovav Shacham in his seminal paper (Schacham, 2007) as a technique that can cause Intel x86 CPUs to interpret unmodified binary code in an unintended manner and expanded upon later in (Roemer et al., 2012). ROP controls the execution sequence of binary code through manipulating the content of the call stack in the memory. It starts with

Sapello, A., Chiang, C., Elwell, J., Ghosh, A., Kubota, A. and Matsunaka, T.
ROP Defense in the Cloud through LIve Text Page-level Re-ordering - The LITPR System.
DOI: 10.5220/0006305402190228
In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 191-200
ISBN: 978-989-758-243-1
Copyright © 2017 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

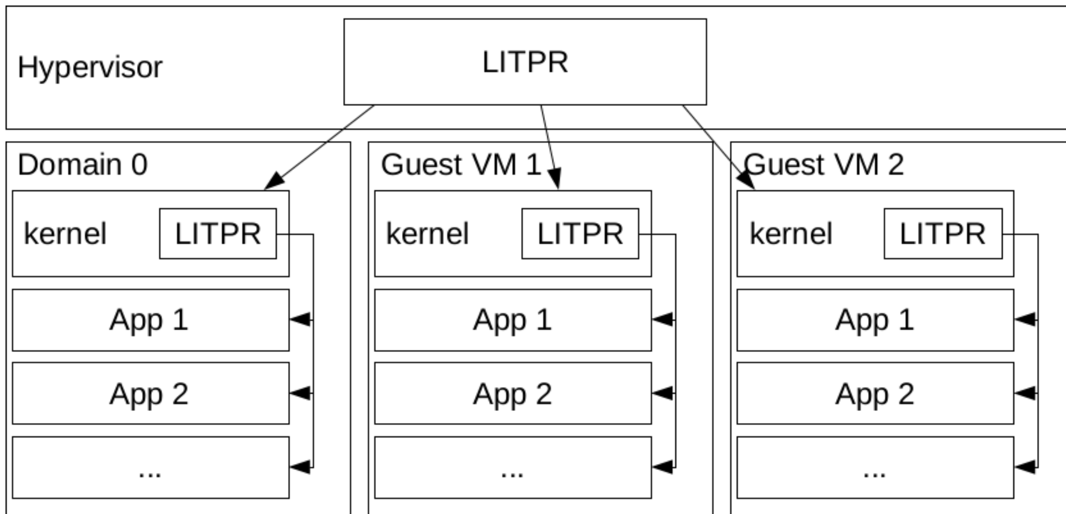


Figure 1: High level view of LITPR operating in a cloud machine.

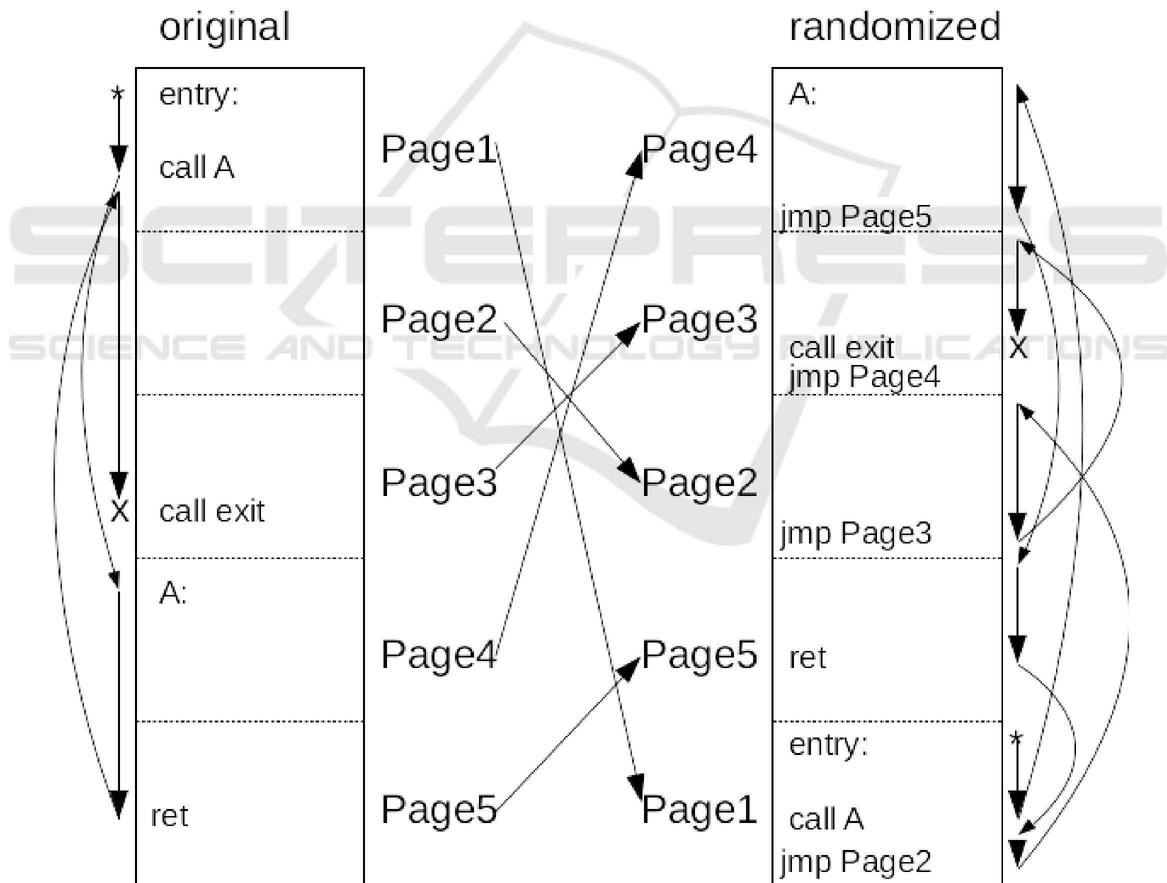


Figure 2: Visual representation of page-randomized program execution. The arrows on the left side of the figure show the normal execution path of a simple executable. On the right side of the figure is a randomized version of the same executable with arrows on the right showing the new exection path.

identifying code blocks (a.k.a. gadgets) ending in return instructions, and then uses the call stack to chain together a sequence of these gadgets to execute unintended logic. Each gadget starts with a byte with a

value (in the range of 0x00 - 0xff) that can be interpreted by the CPU as a legitimate opcode and ends with the value of the opcode of a return instruction (e.g., 0xc3 on the Intel platforms), of a call instruction (0xff on the Intel platforms), or of any instruction with the semantics of return in their execution context (Onarlioglu et al., 2010). These gadgets are like short subroutines, always ending in a return, so that when CPU hits the return instruction of a gadget it fetches the return address on the stack to execute the next gadget. If the stack is purposely filled with data of malicious intent as a result of system vulnerability such as buffer overflow being exploited, a sequence of gadgets can be chained up to execute unintended logic.

To thwart ROP attacks, it is critical to prevent attackers from exploiting gadgets. Previous work has focused on gadget removal (Onarlioglu et al., 2010) and analysis of program execution sequences (Wang and Jiang, 2010)(Abadi et al., 2009). Gadget removal has been shown to be an inadequate protection since gadget removal is not always possible (Checkoway et al., 2010). Program execution sequence analysis can be prone to inaccuracies as well as system efficiency issues. The high-level idea of the LITPR system is that by changing the memory locations of pages containing code (resulting in changed gadget locations) frequently enough, an attacker could be prevented from identifying the memory locations of all the gadgets needed for composing an attack. With this approach, there is a high likelihood of ROP attack failure since memory locations of at least some of the discovered gadgets would be obsolete by the time attacks are launched.

ROP attacks require some amount of time investment from an attacker attempting to exploit an arbitrary application. Address space layout randomization (ASLR) (Team, 2016) and gadget removal (Onarlioglu et al., 2010) represent means to protect against such attacks but recent research (Bittau et al., 2014) on BROP (Blind ROP) has shown that the former is not sufficient protection while the latter is not always feasible. Even more recent work (Gras et al., 2017) shows that ASLR side-channel attacks can bypass ASLR without detection (crashes, exceptions, etc.). However, searching for Page Table Entries (PTEs) as suggested still suffers two flaws against the LITPR system, namely the attack as demonstrated took approximately 25 seconds to obtain a single address and the attack assumes that knowledge of a single data buffer location is sufficient to defeat the address randomization defense. Our system deals with both issues by periodically re-randomizing the code and ensuring that knowledge of a data location does not provide adequate knowledge about specific code locations. Other attacks against ASLR include using branch predictors (Evtyushkin et al., 2016), memory de-duplication (Bosman et al., 2016), and other timing based attacks (Hund et al., 2013). However, these too assume knowledge of a single address at a moment in time is sufficient to bypass ASLR and while this is true for ASLR, it is not sufficient to defeat the proposed LITPR system.

Other systems have been proposed to perform fine-grained address randomization even including re-randomization such as (Giuffrida et al., 2012). However, these systems rely on the availability of source code, heavy integration into a specific compiler and have significant overhead associated with relinking code at runtime. Our proposed LITPR system can be run against binary code for which the source is unavailable, does not rely on the compiler used to perform the original compilation of the source and should perform significantly faster at runtime (including the ability to partially reorder code to minimize impact on the running system).

1.2 ELF and Position Independent Execution (PIE) Preliminaries

The LITPR system exploits and extends an existing concept in the operating system known as dynamic linking. Dynamically linked executables use a feature called Position Independent Execution (PIE) provided by the compiler to generate code that is not required to be loaded at a fixed memory address (either physical or virtual). On the Linux operating system, these executables are stored in the Executable and Linkable Format (ELF). Knowing the details of the information stored in this format allows us to make use of the dynamic linking information to change the ordering of the code pages of the executable.

Dynamic linking at a high level works as follows. At the time a compiler generates object code, it does not know where the subroutines will be loaded into the memory. As a result, the resulting object code generated by the compiler contains a number of symbols that need to be replaced by the starting addresses of the subroutines after the object code has been loaded into memory. This process is called dynamic linking. In fact, what we propose to do, in a sense, can be regarded as dynamic re-linking. This is because code pages in the physical memory will be pointed to by different virtual pages due to page table updates. Whenever a new virtual page replaces the current virtual page as the index for a physical page, it is necessary to update all the subroutine references pointing to the current virtual page with the new vir-

tual page. Symbol tables can be used to quickly identify the references affected by the change. The symbol tables and relocation tables are stored in the ELF binary.

2 LITPR DESIGN

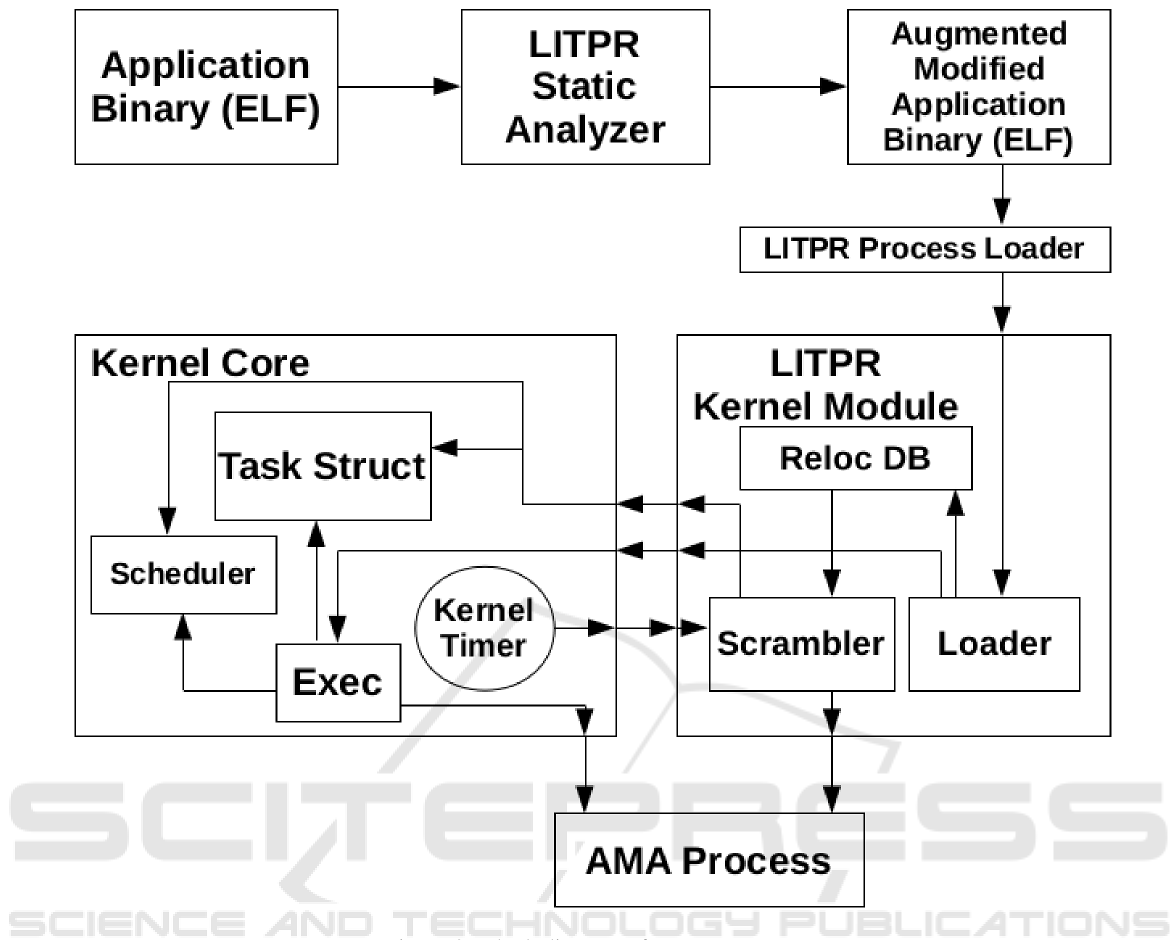
Figure 3 shows the overall system design for LITPR application randomization. The system can be split into two parts, static analysis preformed offline to prepare an application binary and live re-ordering in which an application is loaded, executed and periodically randomized.

This system implementation discussion deals with the application level randomization, but similar techniques can be applied to randomize virtual machine kernels. In this case, static analysis is essentially the same except it acts on the kernel image and randomization is carried out by the hypervisor. The two techniques (application and kernel randomization) can be combined to create an even more secure system.

2.1 Static Analysis

The static analysis stage of the LITPR system performs the offline task of preparing application binaries for live randomization (see figure 4). While some of the information needed for randomization is provided by the compiler in the ELF binary in the form of symbol and relocation tables, additional information must be discovered. The steps involved in static analysis are:

- **Binary Parsing**: the statically linked binary is loaded and parsed, collecting up text and data sections and interpreting special sections including relocations, symbols, exception handler frames and string tables.
- **Disassembly**: the libcapstone disassembler is used to interpret the machine code in the text sections of the binary. Since this process can sometimes be error prone (non-code in code sections, padding with zeros instead of nops), the static analyzer uses simple information to break the disassembly at function boundaries so each function in the program is disassembled independently.
- **Symbol Relocation Mapping**: special relocations are added to the relocation table to update the binaries symbol table. This is necessary to ensure the resulting binary provides valid information to a debugger such as gdb.
- **Exception Frame Relocation Mapping**: special relocations are added to the relocation table to update the exception handle frame (.eh_frame) section of the resulting binary. Again this ensures that the resulting binary is properly loadable by a debugger. In particular, the exception handler frame provides the debugger the information it needs to parse the stack and provide a back trace.
- **Relocation Translation**: the relocations stored in the binary are interpreted and translated into the format required by the randomizer. This involves finding the source and target of the relocation and linking them to the relocation so that any changes in their addresses can be reflected in the output relocation.
- **Code Relocation Discovery**: the disassembled text sections are searched for relocations that may not have been included in the relocation table. These typically include short jumps and hard-coded addresses in the startup code. The linker assumes that even if the code is loaded at a different virtual address the relative relationship between jumps and their targets will not change and therefore providing relocation information is unnecessary. Since the static analyzer and randomizer do change these relative relationships, we must know about these relocations.
- **Data Relocation Discovery**: some binaries contain hard-coded addresses that are not associated with any relocation information. This is somewhat rare and this stage is therefore optional.
- **Code Rewriting**: the text sections are translated in multiple passes, each time finding short jumps whose targets are now too far away due to a previous pass (short jump targets can be at most 127 bytes away) or cross a page boundary (since randomization will move these targets out of range). Also during each pass no-operation (nop) instructions are inserted at the end of each page as necessary to leave room for jumps to the next page and cleanup nops that have been pushed onto the top of the following page. After the code stabilizes (which is guaranteed to happen in a finite number of passes), a final pass replaces the nops at the end of each page with a long jump to the next page and relocations for these jumps are added to the relocation table.
- **Relocation Site Updating**: since the code has moved around relocations must by reapplied to the code and data to ensure that they point to the correct targets in the code. During this stage relocations that were only needed for code rewriting but not for randomization are deleted to minimize the time required to randomize the binary later. For example, short jumps within a page will re-



SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Figure 3: Block diagram of LITPR system.

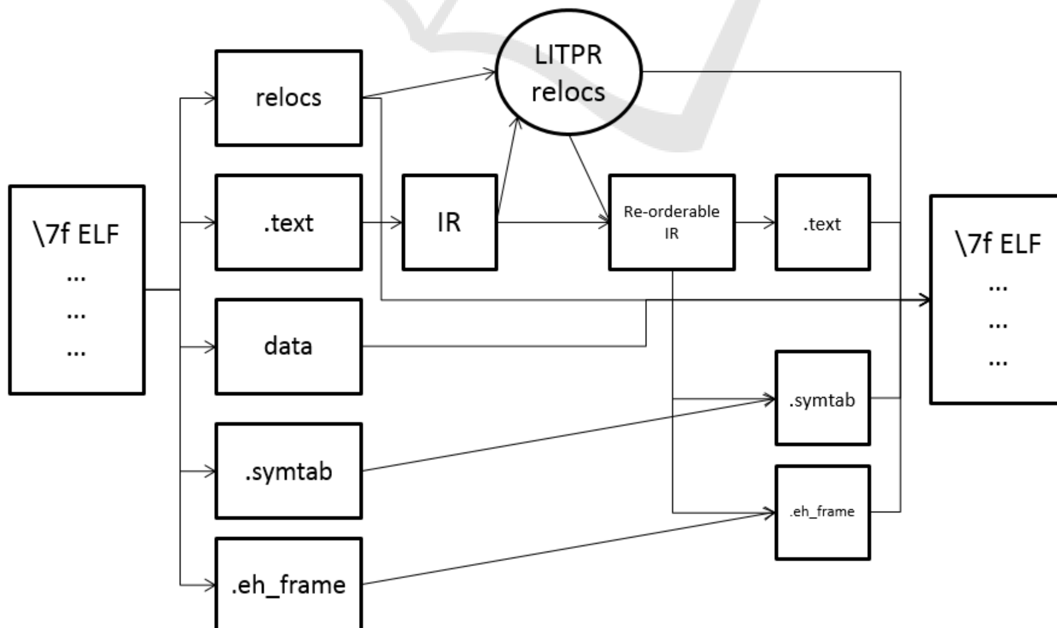


Figure 4: Transform diagram of static analysis.

main valid even if the page is relocated, so these relocations can be deleted.

- **Binary Writing**: the results of the previous stages are re-linked into a new binary. The relocation table is written to the section .pjtdata in a custom format so as not to be confused with standard ELF relocations and also to simplify the randomization process.

2.2 Live Re-ordering

The kernel module will obtain and store the relocation information from the binary at application load time and start a timer. (Alternatively, this timer could be dynamically driven by application behavior such as accepting a new connection.) Each time the timer fires the module will:

1. Pause the application.
2. Re-order the pages of the text segment (or a subset of the pages of the text segment): This randomization is done on the page table entries mapping virtual to physical addresses. Updating these mappings is significantly faster than copying/moving the pages in physical memory.
3. Update the relocation sites in memory: Utilizing the relocation data generated during static analysis, code references are updated to reflect the new page mapping.
4. Scan and update the stack: Return addresses and temporary variables referencing code are held in the stack and not part of the relocation data. These pointers need to be discovered and updated to prevent the program from crashing.
5. Update kernel structures referencing the applications code: The kernel keeps pointer to application code for things such as exception and signal handling and system callbacks for asynchronous I/O. These pointers need to be updated as well.
6. Resume the application.

Of course, depending on system requirements the time to complete this task could be significantly shortened by only doing a partial re-ordering on only a subset of the text pages. Doing this partial re-ordering frequently enough should provide the same protection as a full re-ordering, although further study of this claim would be needed.

During implementation we are likely to find other code references that need to be captured and updated during randomization. As a catch all, randomization can move the code to an entirely new location in virtual address space. Attempts to execute code at the old addresses will be caught by exception handlers and analyzed to determine if these attempts were legitimate or part of an attack.

2.3 Experiments

The current implementation of the LITPR system takes a statically linked binary with relocations and jump tables disabled (–static and –emit-relocs flags passed to the linker and -fno-jump-tables flag passed to the compiler), analyzes it and outputs a page-level randomized version of the program. By doing this any gadgets an attacker might find with static analysis of the same binary will be located not only in different locations (commonly thwarted by Address Space Layout Randomization (ASLR)) but also with different offsets relative to each other making it significantly harder to launch an attack. Ultimately, it is envisioned that this randomization can be done repeatedly at run-time to thwart blind ROP attacks as well.

2.3.1 User Space Experiment Setup

The testing environment was provided by an AMD FX(tm)-8350 Eight-Core Processor @4.0GHz, with 16MB RAM running Ubuntu Linux 12.04.5 LTS.

We were interested in the following metrics:

- Randomization time: In the final product, randomization will occur in real time on the running system. In the initial solution this will require pausing the system. We have taken three sub-measurements:
  - Page list randomization: the time required to select random numbers and generate the permuted list of pages. This is separated from the rest as it may improve with a better shuffle algorithm.
  - Virtual memory remapping: the time required to issue the full set of requests to remap the code pages of the target program to achieve the new layout. Currently this is done with three mremap syscalls per page swap (the extra call is due to the use of a temporary virtual address since the physical address is unknown in user-space). This may improve by offloading the entire task to the kernel rather than issuing it piecemeal. This would avoid repeated context switches and reduce the number of remap operations by 33%.
  - Relocation rewriting: the time required to iterate through the relocation table, compute and rewrite the relocation in the reordered code. This is straight forward and not likely to improve.

- Test program task completion time: This is a comparison of the time required for the unmodified test program to complete a deterministic task on a fixed input with the average time required for randomized variants of the test program to complete the same task with the same input. This will give the performance penalty for randomizing the test programs.

hash_page_static: This was the initial test program due to its simple yet predictable behavior. This program computes a SHA-1 hash of the standard input using an assembly optimized SHA-1 hash implementation. It is an entirely CPU-bound process which provides a worst-case performance indicator. The inputs used for testing the performance of hash_page_static were random chunks of memory generated by /dev/random, saved to disk and then reloaded and fed to hash_page_static and each of its variants with the "dd" application. The input sizes were powers of 2 starting at 1MB and going up to 64MB. 11 variants were generated (the initial recoded but not reordered output of the static analyzer and 10 randomized versions). Each variant was tested with 100 different random memory chunks of each input size.

ffmpeg: This test program was selected for two purposes. First, its complexity provided a good stress test of the static analyzer to help find bugs in user-space before attempting to analyze a Linux kernel. Secondly, it is again a very CPU intensive application with highly optimized code intended for high performance while at the same time behaving very predictably. The input to ffmpeg and its variants was an M2TS (MPEG-2 Transport Stream) formatted video file with a H.264 encoded video stream and AAC encoded audio stream with a combined encoding rate of 11Mbps VBR (variable bitrate). ffmpeg was tasked with converting the first 60 seconds of this input to a mp4 formatted file with AC3 encoded audio and MPEG-2 encoded video at the default encoding rate.

2.3.2 Kernel Experiment Setup

Kernel testing was performed on a Dell R620 server with two Intel Xeon E5-2600 6 core processors running CentOS 5 XEN virtual machines with kernel 3.13.9. Each virtual machine was assigned two cores to test symmetric multiprocessing (SMP) behavior. Kernel experimentation was split into two experiments. In the first experiment a ROP attack was launched against a kernel with a fabricated vulnerability. This attack was then performed again on a randomized version of the same kernel to determine whether this is an adequate defense against kernel ROP attacks. The second experiment was a series of performance evaluations to determine the performance impact of randomized execution on the kernel.

To test our kernel defense against ROP attacks we first constructed a ROP attack that works in kernel space. The goal of the attack was to clear the non-executable (NX) bit of the __supported_pte_mask variable in the kernel. Doing so disables non-executable page protection on all subsequently created virtual memory pages. This allows an attacker to launch standard buffer overflow attacks (non-ROP attacks) on the system by making all newly created data pages executable.

The goal of our experiment was to determine if we could prevent a ROP attack against the kernel assuming some buffer overflow vulnerability exists. To this end we created a new system call with such a buffer overflow vulnerability. We then ran the ROPgadget(Salwan, 2016) script on the newly created vmlinux kernel binary image. This tool locates and reports the locations of ROP gadgets. The output of this script is then searched for gadgets that could be used to build an attack that is equivalent to "__supported_pte_mask &= 0x7fffffffffffffff". From these gadgets a payload is constructed to attack the kernel.

The experiment then uses two user programs to launch the attack. The first program which we will call shellcode is a toy program that contains a buffer overflow vulnerability and launches a traditional (non-ROP) buffer overflow attack against itself to try to obtain a shell. The second program which we will call kernel_attack calls the vulnerable system call in the kernel and delivers the previously constructed attack payload.

Performance was evaluated by running the Trinity v1.6 system fuzzer (Jones, 2016) 150 times set to run 1000 random non-blocking system calls. Measurements were taken using three separate timing methods:

- Bash time function: measures the total wall clock time the process executed.
- rdtsc: measures the CPU ticks elapsed for each system call (from the user application's perspective)
- strace: measures the elapsed time of each system call from the kernel's perspective (using kernel profiling)

Table 1: file statistics of the hash_page_static program before and after static analysis.

	Binary size (bytes)	Text segment size (bytes)	".text" section size (bytes)	".data" section size (bytes)	Number of relocations
Input	1320691	828233	612616	39007	N/A
Output	2089531	828233	615374	39007	24026

Table 2: file statistics of the ffmpeg program before and after static analysis.

	Binary size (bytes)	Text segment size (bytes)	".text" section size (bytes)	".data" section size (bytes)	Number of relocations
Input	25408763	18296744	13131784	862660	N/A
Output	35185027	18296744	13324265	862660	305508

Table 3: breakdown of times involved in randomization of hash_page_static.

	Mean (μs)	Standard deviation (μs)
Page list randomization	7.3205	0.562
Virtual memory remapping	968.2	15.6
Relocation rewriting	500.7	7.67

Table 4: breakdown of times involved in randomization of ffmpeg.

	Mean (μs)	Standard deviation (μs)
Page list randomization	57.452	1.42
Virtual memory remapping	7599.1	68.3
Relocation rewriting	4047.2	32.0

2.4 Results

2.4.1 User Space

Tables 1 and 2 show the sizes of hash_page_static and ffmpeg before and after static analysis. Randomization performs reasonably well taking a total of 11.7ms for ffmpeg as can be seen in table 4 and 1.5ms for hash_page_static (table 3). The final application performance results were a little surprising. In table 5 we see that for hash_page_static it turned out that in many cases the randomized version performs better than the original program. We believe this is likely due to more efficient L1 instruction cache utilization. This did not happen with ffmpeg (table 6), and that makes sense since the code base is much larger and less likely to benefit from the randomization. What we see instead is the expected result, that by moving code around we de-optimize some compiler optimizations and this along with the added page jumps causes CPU intensive applications to incur an additional 2-3% performance drop. In particular the static analyzer replaces a large number of 8-bit jumps with 32-bit jumps to allow jump across page boundaries. Also, moving the code causes jump targets that had previously been aligned to the 64-byte cache boundary to no longer be so aligned. In the future we could improve this by realigning jump targets as part of static analysis. In any case we believe these performance results are reasonable.

2.4.2 Kernel

To test whether our system provides adequate protection against kernel ROP attacks, prior to launching kernel_attack we attempt to run shellcode. On both the undefended and defended kernel, the shellcode program causes a SIGSEG and fails. On the undefended kernel, launching kernel_attack causes a SIGBUS appearing to fail. However, subsequently launching shellcode succeeds demonstrating that the kernel attack was in fact successful. On the defended kernel, launching kernel_attack causes a SIGSEG. In this case however, subsequent attempts to launch shellcode continue to cause a SIGSEG indicating that the kernel attack has been thwarted.

Multiple methods of measurement were used for measuring the kernel's performance since the impact of our system was quite small and difficult to measure. Even so, the raw numbers made little sense. Instead

Table 5: runtime performance of hash_page_static.

Input size	1MB	2MB	4MB	8MB	16MB	32MB	64MB
Unmodified	0.185700 ± 0.017044	0.374100 ± 0.027932	0.753900 ± 0.033163	1.496900 ± 0.053023	3.014800 ± 0.111476	6.004500 ± 0.187154	11.986300 ± 0.309292
Analyzed	0.187500	0.381400	0.769200	1.513300	3.049800	6.095600	12.160400
Randomized	0.182109 ± 0.019783	0.365818 ± 0.028135	0.773082 ± 0.044338	1.461582 ± 0.073405	2.929191 ± 0.129238	5.845982 ± 0.237880	11.689209 ± 0.491306
Performance drop	-1.93%	-2.21%	-2.76%	-2.35%	-2.84%	-2.64%	-2.48%

Table 6: runtime performance of ffmpeg.

	Mean (seconds)	Standard deviation
Unmodified	123.958000	13.995810
Analyzed	127.042000	13.591639
Randomized	127.082000	14.098960
Performance drop	2.52%	Not computed

we chose to bin the performance differences into histograms for each measurement method. Figures 5, 6 and 7 show the percent performance degradation as measured by bash time, strace and rdtsc respectively.

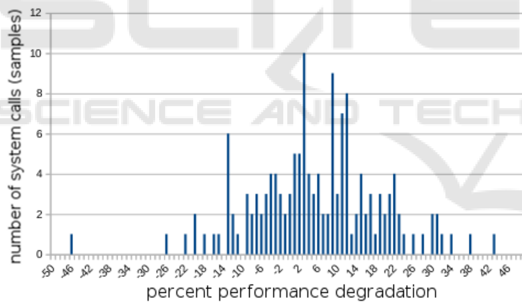


Figure 5: Histogram of kernel percent performance degradation as measured by bash time.

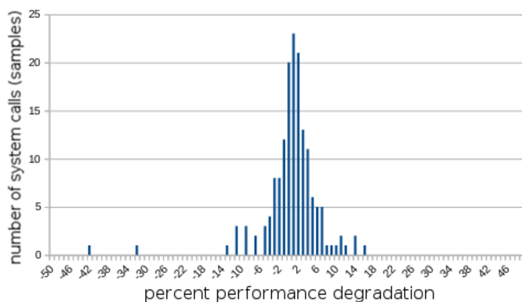


Figure 6: Histogram of kernel percent performance degradation as measured by strace.

Each measurement method had its own issues of either precision (spread) or outliers. While precision is a natural issue related to the outputs of bash time and strace, the outliers, particularly those present in rdtsc, were a little more concerning. One reason for the outliers is that while we modified the trinity program to be as deterministic as possible, at times it could still execute the same system call with the same parameters and have the call be valid in one run, but not valid on the next (such as accessing a particular node in the procfs tree and having the the process exit between runs). Another possibility is that the process or VM was migrated to a different CPU and therefore the tick start and end times were not from the same source. In any case the three figures clearly show an approximate 2% average performance degradation which was consistent with the previous user-space ffmpeg results.

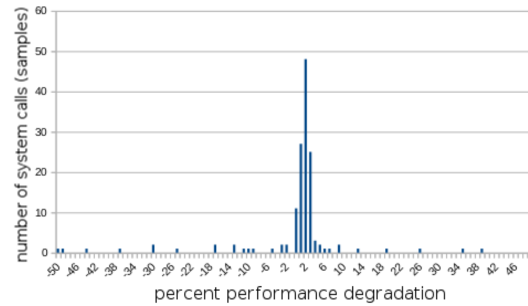


Figure 7: Histogram of kernel percent performance degradation as measured by rdtsc.

3 FUTURE WORK

The most important piece of future work to be done is implementing the kernel and hypervisor modules to perform the application and kernel randomization at runtime. The results described in this paper give us confidence that this randomization can be done quickly and efficiently without damaging the running system.

While it was originally thought that such attacks only apply to x86 or variable length encoded ISAs

(Instruction Set Architectures), a generalization to fixed-width ISAs and RISC architectures is possible (Buchanan et al., 2008). As such, we would like to extend our work to non-x86 ISAs. This should be possible with little change to the overall design of the LITPR system.

REFERENCES

Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. (2009). Control-flow integrity - principles, implementations, and applications. In *ACM Transactions on Information and System Security*, volume 13.

Bittau, A., Belay, A., Mashtizadeh, A., Mazieres, D., and Boneh, D. (2014). Hacking blind. In *Proceedings of the IEEE S&P conference*, Oakland, CA, USA.

Bosman, E., Razavi, K., Bos, H., and Giuffrida, C. (2016). Dedup est machina: Memory deduplication as an advanced exploitation vector. In *Proceedings of IEEE Symposium on Security and Privacy*, San Jose, CA, USA.

Buchanan, E., Roemer, R., Schacham, H., and Savage, S. (2008). When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38, New York, NY, USA.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Schacham, H., and Winandy, M. (2010). Return-oriented programming without returns. In *ACM Conference on Computer and Communication Security 2010*, pages 559 – 572.

Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. (2016). Jump over aslr: Attacking branch predictors to bypass aslr. In *Proceedings of IEEE Symposium on Microarchitecture*, Taipei, Taiwan.

Giuffrida, C., Kuijsten, A., and Tanenbaum, A. S. (2012). Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of USENIX Security Symposium*, Bellevue, WA, USA.

Gras, B., Razavi, K., Bosman, E., Bos, H., and Giuffrida, C. (2017). Aslr on the line: Practical cache attacks on the mmu. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA.

Hund, R., Willems, C., and Holz, T. (2013). Practical timing side channel attacks against kernel space aslr. In *Proceedings of IEEE Symposium on Security and Privacy*, San Francisco, CA, USA.

Jones, D. (2016 (accessed Nov. 18, 2016)). *Trinity System Call Fuzzer*. `https://github.com/kernelslacker/trinity`.

Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., and Kirda, E. (2010). G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, Austin, Texas, USA.

Roemer, R., Buchanan, E., Schacham, H., and Savage, S. (2012). Return-oriented programming: systems, languages, and applications. In *ACM Transactions on Information and System Security*, volume 15.

Salwan, J. (2016 (accessed Dec. 12, 2016)). *ROPgadget*. `https://github.com/JonathanSalwan/ROPgadget`.

Schacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security 2007*, pages 552 – 561.

Team, P. (2016 (accessed Nov. 18, 2016)). *PaX address space layout randomization (ASLR)*. `http://pax.grsecurity.net/docs/aslr.txt`.

Wang, Z. and Jiang, X. (2010). Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*.