Diversification of Stack Layout in Binary Programs Using Dynamic Binary Translation

Benjamin Rodes, Anh Nguyen-Tuong, John Knight, James Shepherd, Jason Hiser, Michele Co, Jack W. Davidson
Department of Computer Science, University of Virginia
Charlottesville, VA 22904
Email: {bdr7fv, an7s, jck, jws2yp, jdh8d, mc2zk, jwd}@virginia.edu

***Abstract*—Despite protracted efforts by researchers and practitioners, security vulnerabilities remain in modern software. Artificial diversity is an effective defense against many types of attack, and one form, address-space randomization, has been widely applied. Present implementations of address-space randomization are either coarse grained or require source code. We present an approach to fine-grained randomization of the stack layout that operates on x86 binary programs. Randomization is applied on a function-by-function basis. Variable ordering on the stack is randomized and random-length padding inserted between variables. Optionally, canaries can be placed in the padding regions. Transform determination is speculative: the stack layout for a function is inferred from the binary, and then assessed by executing the transformed program. If a transform changes a program's semantics, progressively less aggressive transforms are applied in sequence. We present results of applying the technique to various open-source programs including details of example exploits that the technique defeated.**

***Keywords*-security, artificial diversity, stack frame layout, address randomization, binary program**

I. Introduction

We present an approach to stack-layout randomization that does not require access to the source program or other development artifacts. Randomization of the layout of local variables in the stack frame is implemented using dynamic binary translation. The goal of our approach to stack-layout randomization is to enhance the security of a software system using only the binary form of the software. The ability to randomize based on limited information is especially important, because having software of unknown quality and for which only the binary form is available is a common circumstance.

Stack layout randomization is a form of artificial diversity. Artificial diversity is an approach to protecting a vulnerable program from defined classes of attack by artificially varying one or more aspects of the program. Each instance of the program provides identical normal functionality but differs from the other instances in the details of some characteristic such as an element of addressing that has been randomized. Exploiting a vulnerability requires determination of these details, and that determination usually requires a state-space search. Artificial diversity can (but does not necessarily) generate entropy that makes the cost of this search prohibitively expensive to an adversary.

Many attacks on vulnerable software succeed because the perpetrators have knowledge of the layout of program elements in memory. Thus, one important form of artificial diversity is address space layout randomization (ASLR) in which the layout of elements in memory is randomized. Implementations that are currently deployed for binary programs [16] are coarse gained; they randomize details such as the base addresses of the stack, the heap, and the code. More aggressive fine-grained randomizations have been developed that randomize details such as the order of functions and the stack layout. These approaches rely, however, on the availability of the source program [6, 7, 9, 17].

The security benefits of randomizing the stack layout are many; exploits might be completely thwarted or severely limited. The benefits realized depend on the nature of the exploit, the vulnerability, and the randomization used.

There is a need, then, for a stack-layout randomization technique that can provide fine-grained randomization without using information from the program source. We sought to develop a technique that would meet the following goals:

- Operate on binary programs designed to execute on a common target platform (x86) using only data gathered from the binaries themselves.
- Require only relatively simple analysis of the binary program.
- Provide a fine-grained transformation; that is, randomize stack frames of functions so that the order of variables is changed.
- Apply to real programs and scale to large programs.
- Incur a small overhead during execution.

In this paper we describe our technique (which we refer to as SLR) and show that it achieves these goals. We also present a performance assessment.

SLR uses static analysis to develop a hypothesis about the layout of local variables in the stack frame for each function. It transforms each function to reorder the stack based on this hypothesis, then evaluates the hypothesis by executing the transformed program. In this sense, SLR is a speculative technique; static analysis of binary code rarely determines the stack layout perfectly, so static analyses must be verified

empirically. If an analysis prescribes a transformation that changes the semantics of a function, different and less fine-grained randomizations are tested until a behavior- and semantics-preserving randomization is found.

The key contributions of this paper are: (a) the use of a speculative and dynamically verified approach to the determination of the stack-frame layout, and (b) the use of dynamic binary translation to implement stack-frame randomization.

In the next section, we present the assumed threat model. The way in which SLR develops the hypothesis about the stack layout is presented in section III. In section V we describe the overall process by which the hypothesis is evaluated and modified if needs be. We describe the implementation of stack randomization in section IV, and present our evaluation of SLR's efficacy and performance in section VI. Related work is discussed in section VII, and we conclude in section VIII.

II. Threat Model

In this work, we assume the following threat model:

- The subject program is created and distributed in binary form to an end user and possibly to a malicious third party.
- The subject program is released with no guarantees of the absence of software faults that leave it vulnerable to exploitation by an attacker.
- The subject program is assumed to be free from self-modifying code and from intentionally planted back-doors, trojans, etc.
- No development information is available about the subject program.
- The environment in which the program runs is trusted and we assume the attacker does not have direct access to the hardware on which it runs.
- The attacker has access to an unprotected version of the program on which to test malicious inputs that he or she can submit to the protected program.
- Attacks of interest are those which rely on data being located predictably, such as buffer overflow and format string attacks.

In summary, these characteristics are typical of programs that are usually described as *Software Of Unknown Provenance* (SOUP).

III. Determining the Stack Layout Hypothesis

Randomization of the stack frame layout for a function requires determination of:

- The current layout of the stack frame, i.e., the addresses of the function's local variables as set by the compiler.
- The instructions that manipulate the different variables in the stack frame.

In principle, if this information were available, the layout of the stack frame could be changed and the instructions that access the stack frame modified to reflect the new layout.

The new layout of the stack frame could be based on any security-relevant criteria. For example, memory objects could be placed in random order, padding introduced before, after or within the stack, canaries included, encryption applied, or scalars placed at lower addresses than arrays. Items on the stack could also be removed and promoted to the heap. In our current approach, we limit transformations to placement of memory objects in random order and the introduction of random length padding. This choice was made so as to gain information about the potential of simple stack-layout randomization.

Starting with a binary program, precise determination of the stack layout and the instructions that reference the stack is problematic. Modern compilers employ a wide range of techniques to minimize both the use of storage and program execution time. The result is binary programs with unpredictable structures. For example, when generating code for the x86 architecture, compilers frequently inline functions, fold constants, pack stack frames, unroll loops, and include hand-written assembly for common functions like memcpy.

Our approach to SLR is based on two assumptions:

- Knowledge of the size of variables is all that is needed to determine boundaries. Type information of stack variables is useful for SLR, because type information helps to determine the size of variables. This benefit from type information is especially important for determining fields in records (structs in C). However, our goal with this work was to assess the possibility of using simple heuristics for boundary determination, and so we do not determine nor use variable type information.
- The predominant mechanism by which instructions access stack variables is through scaled or direct addressing based on an offset indicating the variable starting location. Where indirect addressing is used, that use is for access to variables whose locations can be inferred from previous direct or scaled addressing based on an offset indicating the variable starting location.

The second assumption does not necessarily hold, and so we use a speculative approach. The assumption is used to create an initial layout inference and an inference of which instructions access variables in this layout. These inferences are then evolved and refined.

Layout inferences are produced using (a) a set of simple heuristics, (b) assumptions about the manner in which the stack is allocated and deallocated, and (c) assumptions about the general stack frame layout. These assumptions hold for binaries produced by C/C++ compilers that use the cdecl x86 calling convention in which stack frames are in the form seen in Figure 1.

A key requirement for the heuristics is the use of an out arguments region for outgoing parameters for called functions. In principle, any calling convention where this general structure is maintained could be supported. Dynamically allocated stacks are out of scope for this work, as are compiler conventions where the out arguments region is not used and the stack expands and contracts before and after each function call.

To support the modification of instructions that access the stack, in the implementation of stack-layout randomization, all of the instructions in the binary program are inserted individually into a database. During static analysis, the binary program is processed by a static analysis system. We use a recursive descent disassembler (IDAPro) [10] and a linear scan disassembler (objdump). To ensure we have all instructions in the database, we added a disassembly validator module. The disassembly validator iterates over every instruction found by both IDA Pro and objdump, and verifies that both the fallthrough and (direct) target instructions are inserted into the instruction database. This information in the database allows the control-flow graph to be constructed.

Since exact instruction start locations in the executable segment are not known, some of the instructions in the instruction database may not represent instructions that were intended by the program's original assembly code. We make no attempt to determine which are the intended instructions and which are not. Instead, SLR modifies all instructions matching patterns of local variable access (see below). Any data address that is mis-identified as a code address will not be executed and will result in corrupted data if transformed by SLR. We rely on testing to catch those cases. We believe the probability of data being interpreted as a stack accessing instruction is low.

Finally, static analysis determines the functions and the out-arguments region size that IDA Pro detects. Once static analysis is complete, the next part of the approach is to find out if the subject function has a stack frame. The presence of a local-variable region in the stack is detected by scanning each instruction in the entry block of the function's control-flow graph for a stack allocation instruction:

`sub esp,<constant>`

Scanning only the entry block avoids cases where a stack may be allocated differently depending on some condition. If the entry block does not contain this pattern, the function is determined to be non-transformable and is skipped, and the transformation process is restarted for the next function. Otherwise the stack size, the number of saved registers, and the out argument region size are recorded. The size of the out-argument region is determined during static analysis.

Once the existence of the local-variable region is established, each instruction of the function is analyzed and a set of local variable boundary inferences are determined based on instruction patterns accessing the stack. Memory access patterns accessing the stack can take one of three forms:

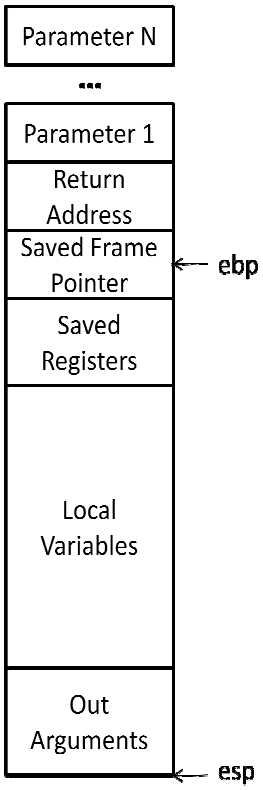

Figure 1. General form of the stack layout.

- Direct: A constant value added to **`esp`** or subtracted from **`ebp`** (e.g., **`[esp+0x10]`**).
- Scaled: A constant value added to an indexed (variable) offset from **`esp`** or **`ebp`** (e.g., **`[esp+eax+0x10]`**).
- Indirect: The stack is accessed by means other than through offsets to esp or **`ebp`** (e.g., **`[ebx]`**).

Inferring a boundary interface for every constant offset found in both direct and scaled memory accesses is a nave approach but surprisingly high levels of accuracy have been reported [2].

The possibility of inaccuracy is the reason we describe SLR as speculative. SLR assumes these inferred boundaries are correct and determines their validity by testing. If testing reveals that the program's semantics have been changed, three additional heuristics are used: (1) variable boundaries are assumed for direct memory access offsets only, (2) variable boundaries are assumed for scaled memory accesses only, and (3) the entire stack frame is treated as one large variable as a "catch all" case.

In summary, the four inferences are:

- All Offset Inference (AOI): Any constant offsets of **`ebp`** and **`esp`** for direct and scaled accesses that access the

local variables region of the stack are considered local variable boundaries.

- Scaled Offset Inference (SOI): Any constant offset used in scaled stack access instructions is considered a local variable boundary.
- Direct Offset Inference (DOI): Any constant offset used in direct stack access instructions is considered a local variable boundary.
- Entire Stack (ESI): The entire stack frame is considered one local variable.

A memory access relative to **`esp`** or **`ebp`** may access incoming parameters to the function, since the size of the frame is known, if a boundary exceeds the stack frame these offsets are ignored. Additionally, the out arguments region contains reusable space, and so any offset accessing this region is omitted from the layout inference.

For each function, all four inferences are applied to form four individual hypotheses. These four hypotheses are sorted by the number of variables detected and applied in order, highest to lowest. In practice, as might be expected, the order is almost always AOI, DOI, SOI, and ESI.

IV. Stack Layout Randomization

The overall SLR process is shown in Figure 2. This process is applied to each function in the program independently. Thus, the whole process shown in Figure 2 is repeated a number of times equal to the number of functions in the program.

The four hypotheses of the stack layout for the subject function are created for each function in the binary program using the four inferences described above, and test data is created or acquired for the program. Recall that the layout hypotheses are limited to the boundary address between individual variables in the stack frame, and that no attempt is made to determine the types of the variables.

The dependence of this approach on testing raises the question of where test cases will come from. The approach has no specific requirements that constrain the test data. Existing test cases can be used or new tests developed using any test-case development technology. Novel test-coverage metrics are suggested by the SLR concept. When assessing the effects of a transformation by testing, measurement of the extent to which stack variables have been referenced or set would provide insight into the degree to which the randomization has been assessed. We do not presently capture this coverage information.

Using the hypothesized stack layout determined by AOI (AOI is the heuristic that detects the largest number of variables because it is the most aggressive), a new layout is created by randomizing the hypothesized layout for the subject function (randomization 1 in Figure 2). In this randomization, space for variables is contiguous, i.e., there is no padding between variables.

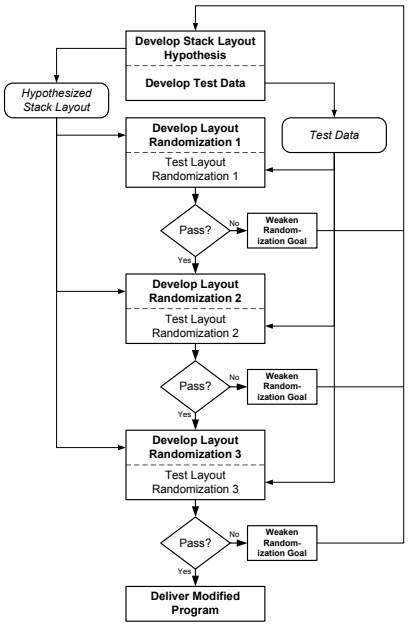

Figure 2. Overall approach to stack randomization.

Once the randomization has been created, the necessary binary rewriting rules for dynamic binary translation are defined to transform the binary function from its original stack layout to the new, randomized layout, and then the program is tested.

If testing is successful, then the entire process is repeated for the subject function with the same hypothesis about the stack layout but a different randomization (randomization 2 in the Figure 2), again without padding. The program is then tested again. This second transformation is carried out to check the results of the first transformation. The claim is that, if two randomizations leave the program in a form that passes the available test cases, the chances of the randomizations having altered the program's semantics are small.

If testing with the second randomization is successful, a third transformation is applied to the subject function, again starting with the original binary program and the original stack-layout hypothesis. In this third transformation, random padding whose length lies between 4096 and 8192 is added between local variables, before the stack frame, and after the out arguments region (if one exists). This third randomization is then tested as the other two were. If testing is successful, this is the randomization of the stack that is used in the transformed program.

If testing fails for any of the three randomizations of the subject function, then the semantics of the program have been changed by the stack randomization, and so the hypothesized stack layout must have been wrong. In that case, the hypothesized stack layout that detected the next largest number of variables in the function is selected, and

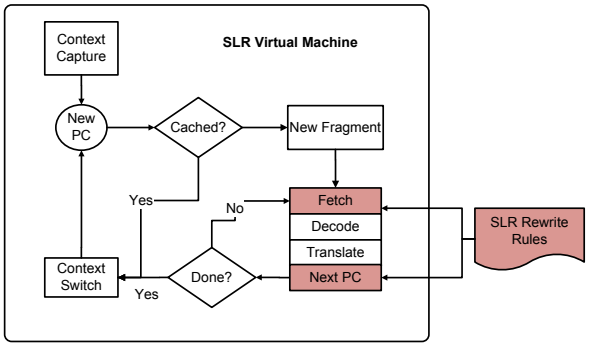

Figure 3. Basic approach used in PVM binary translation system.

the three-phase test process is repeated. Again, any failing tests lead to abandoning of the hypothesized layout, and the process resumes with the next one.

In the worst case, all four hypothesized stack layouts from the four different inferences will affect the program's semantics and be rejected. In that case, the subject function is left unmodified. We present data on the application success with these four approaches to stack layout analysis in section VI.

This speculative transformation and assessment process is repeated for each function in the program. Needless to say, for a large program with many functions, this process requires a lot of resources. Fortunately, the process can be conducted in parallel for all the functions in a program, and so the time taken for the analysis can be reduced by using additional equipment. Also, optimizations are possible in the transformation mechanism. For example, since transforms have a high success rate (see section VI), processing more than one function at a time might be possible.

V. Randomization by Dynamic Binary Translation

In order to implement the randomization in SLR, all instructions that reference the stack have to be modified so that they access the stack using locations that reflect the results of randomization. Modification of the program's instructions so that they can use the randomized stack is effected by dynamic binary translation using a per-process virtual machine (PVM) system (see Figure 3), in our case the PVM is Strata [14] .

PVMs load an application dynamically and mediate application execution by examining and translating an application's instructions before they execute on the host processor. Most PVMs operate as co-routines with the application that they are protecting. Translated application instructions are held in a PVM-managed cache called a fragment cache. The PVM is first entered by capturing and saving the application context (e.g., program counter (PC), condition codes, registers, etc.) Following context capture, the PVM processes the next application instruction. If a translation for this instruction has been previously cached, the PVM transfers control to the cached translated instructions.

If there is no cached translation for the next application instruction, the PVM allocates storage in the fragment cache for a new fragment of translated instructions. The PVM then populates the fragment by fetching, decoding, and translating application instructions one-by-one until an end-of-fragment condition is met (e.g., an indirect branch). As the application executes under the PVM's control, more and more of the application's working set of instructions materialize in the fragment cache thereby allowing the overhead imposed by dynamic binary translation to be small; in the case of the PVM used for SLR the average overhead measured on standard benchmarks is around 8% [19] .

The translations in SLR that the PVM applies are the modifications to instructions that are needed to implement the randomization of the stack layout. These modifications are documented as a list of rewrite rules. The rewrite rules define the instructions that have to be modified, the associated modifications, and the fallthrough map, i.e., the address of the next instruction. The rewrite rules are written by the stack randomization system based on a randomization of a stack-layout hypothesis.

In SLR, the PVM loads the rewrite rules and the PVM's instruction-fetching mechanism checks and then reads the SLR rewrite rules as appropriate. After modifying an instruction, the PVM modifies the PC from the fallthrough map that SLR provides in the rewrite rules. (see Hiser et. al [11] for further PVM details).

VI. Performance Assessment

To evaluate SLR we conducted three assessment experiments:

- We measured the effectiveness of the transformation mechanism for a set of open-source programs for which accepted regression tests are available.
- We measured the performance overhead imposed by the transform using SPEC 2006 benchmarks [15].
- We applied SLR to a set of test programs designed to measure protection against buffer-overflow attacks.

We present our results in the following subsections.

A. Transformation Effectiveness

To evaluate the effectiveness of SLR on real programs, we transformed 18 Unix core utilities version 7.4. Stack layout randomization depends on regression tests to validate layout inferences, and so we selected eighteen based on the availability of regression tests provided by the vendor.

We compiled all 18 utilities using gcc version 4.4.3 with O3 optimizations and dynamic linking on Linux kernel 2.6.32-35-generic as part of Ubuntu 10.04.03 LTS.

Program	Binary Size (in bytes)	Total Functions	Transformable Functions	Transformed Functions	All Offsets Inference	Scaled Offsets Inference	Direct Offsets Inference	Entire Stack
chgrp	178417	194	102	101	95	3	2	1
chmod	175762	183	98	97	94	2	0	1
chown	182694	198	104	103	95	5	2	1
cp	277293	281	155	154	143	10	0	1
dd	158884	169	86	85	78	6	0	1
df	224101	200	104	103	99	3	0	1
du	303089	267	150	149	140	7	1	1
install	249529	279	150	149	146	2	0	1
ln	166473	174	95	94	90	3	0	1
ls	265823	297	177	176	166	8	1	1
mkdir	133611	140	71	70	64	5	0	1
mv	237304	279	154	153	136	14	1	2
pr	166462	155	82	81	77	3	0	1
readlink	148021	155	81	80	74	4	1	1
rmdir	166931	113	57	56	54	1	0	1
rm	104519	192	98	97	88	8	0	1
tail	152468	156	86	85	79	3	2	1
touch	155188	145	73	72	66	5	0	1
Average	191476	199	107	106	99	5	1	1
(rounded to the nearest integer)								

Figure 4. Statistics for the binary programs used for assessment.

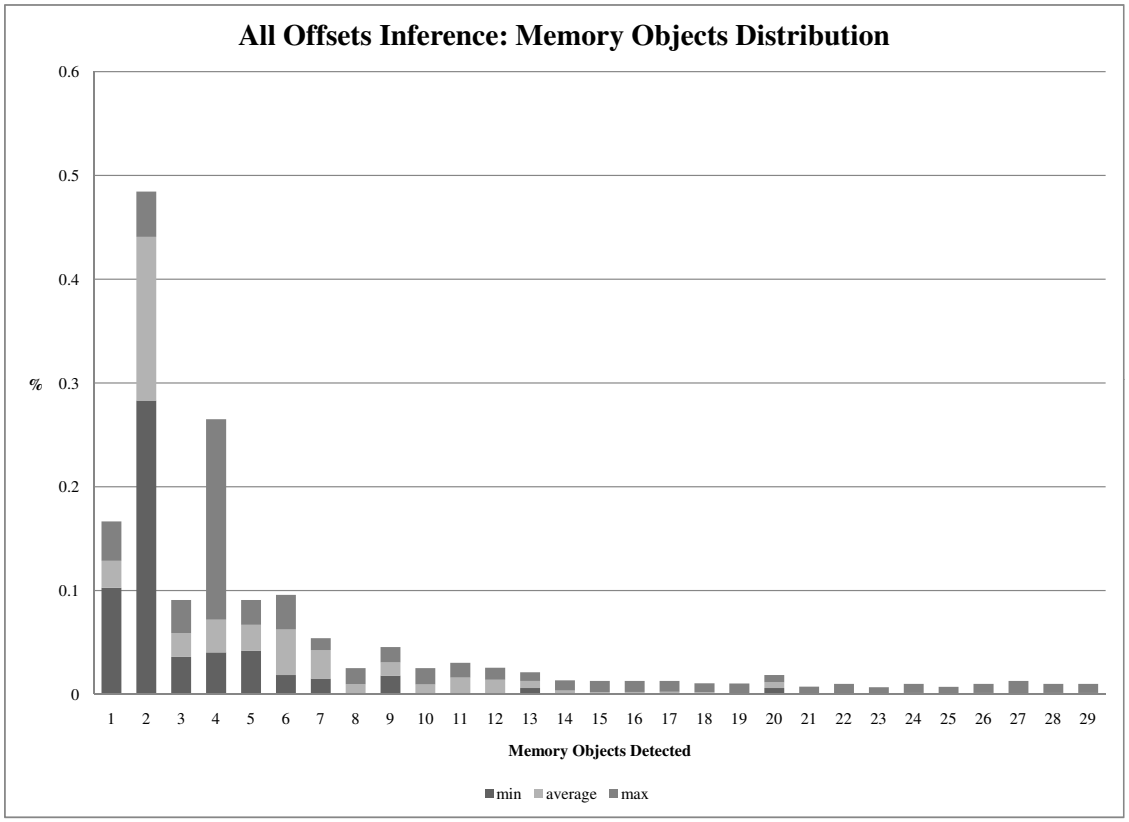

Figure 5. Performance of the All Offsets Inference heuristic

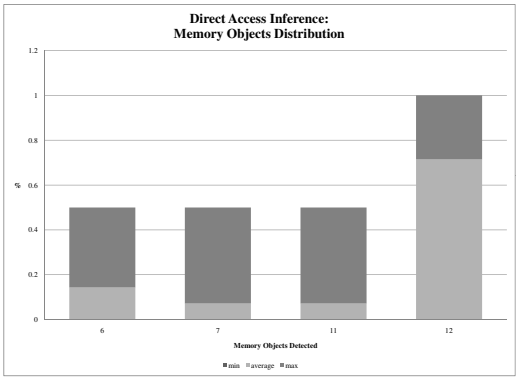

Figure 6. Performance of the Direct Access Inference heuristic

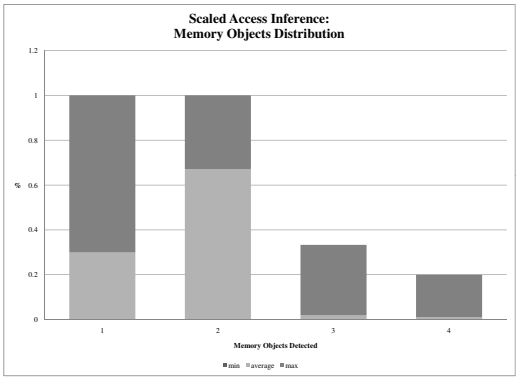

Figure 7. Performance of the Scaled Access Inference heuristic

Only those functions that were not dynamically linked were transformed, i.e., only those functions whose definitions can be found statically in the binary. Libraries should be transformed and evaluated separately as separate regression tests are needed to properly test all library functions. Additionally, libraries should only need to be transformed once and reused by any number of binaries.

Figure 4 shows the statistics of the binaries used and the way in which functions were transformed. For each program, all but one of the transformable functions were transformed using one of the layout inference heuristics. The exception was present in all programs and removed from transformation through regression testing.

Also present in every program was a function that was always reverted to using ESI, the most conservative layout heuristic. The vast majority of transformations, 93.6%, were based on layouts produced by our most aggressive heuristic, AOI. Thus 6.4% of all functions initially transformed with AOI resulted in semantic changes detected by regression testing that triggered a transformation rollback.

Figure 5, Figure 6, and Figure 7 show the distribution of the number of memory objects detected on the stack by each layout heuristic used for each function randomization, where a memory object may be a local variable or the out arguments region. In Figure 5, the distribution for AOI shows that 56.9% of all AOI inferences used found only one or two stack memory objects per function. Since AOI is the first inference used by default, this result suggests most functions use few or no local variables, perhaps leaving the stack with only an out-arguments region.

The distributions for both DOI and SOI, Figure 6 and Figure 7, show that DOI is generally more aggressive than SOI. Bhatkar et al. [7] report dynamic analysis performed on a suite of eight programs for which they find an average of 87% of the stack accesses made are to non-buffer stack variables. If non-buffer stack variables are most prevalent, then prevalence of direct accesses is also expected since this is the typical access mechanism for non-buffer variables. DOI would also be expected to infer the presence of larger numbers of variables than SOI because direct references are more common.

If a layout inference is only able to find one stack memory object, then the layout is reducible to the most conservative layout inference, ESI. In many cases SOI is either reducible to ESI, see Figure 7, or produces a very course grained layout consisting of only two memory objects usually indicating the separation between the out arguments region and the remaining stack. This might explain why few functions (at most two unique functions) are transformed with ESI. Course-grained inferences are less likely to break expected behavior, and by default if ESI and SOI produce the same number of memory objects, the scaled-offset inference is attempted first.

B. Execution Time Overhead

To evaluate run-time overhead, SLR was applied to a set of seven SPEC CPU2006 C benchmarks. Execution-time overhead (wall clock time) was measured on a system with a dedicated 4-core, AMD Phenom II B55 processor, running at 3.2 GHz. The machine has a 512KB L1 cache, a 2MB L2 cache, a 6MB L3 cache, and 4GB of main memory. Performance numbers were generated by running the benchmark 3 times. For these measurements, the test programs were compiled using gcc version 4.4.3 with O2 optimizations and dynamic linking on Linux kernel 2.6.32-34-generic as part of Ubuntu 10.04.03 LTS. The regression tests used for each benchmark consisted of a subset of the SPEC train and test input suites for each program.

Figure 8 shows the statistics for each program and how functions were transformed. This data is consistent with the results from the coreutils programs (see Figure 4). However, the regression tests used for SLR were not extensive as the goal of this experiment was to provide execution-time performance results. We provide these statistics to show that actual transformations were made to these benchmarks, and most transformations used our most aggressive layout heuristics. However, without extensive regression testing many of these transformations might actually be based on incorrect layout assumptions. The only functionality of any

Benchmark	Binary Size (in bytes)	Total Functions	Transformable Functions	Transformed Functions	All Offsets Inference	Scaled Offsets Inference	Direct Offsets Inference	Entire Stack
bzip	67339	101	65	64	60	2	0	2
lbm	16560	44	22	21	19	2	0	0
libquantum	44839	136	86	85	79	3	3	0
mcf	17047	49	25	24	24	0	0	0
milc	127066	281	169	168	156	8	0	1
sjeng	147758	176	103	101	93	3	2	3
sphinx3	187182	391	277	276	270	5	0	1
Average	86827	168	107	106	100	3	1	1
(rounded to the nearest integer)								

Figure 8. Statistics of the SPEC2006 benchmarks used for timing assessment

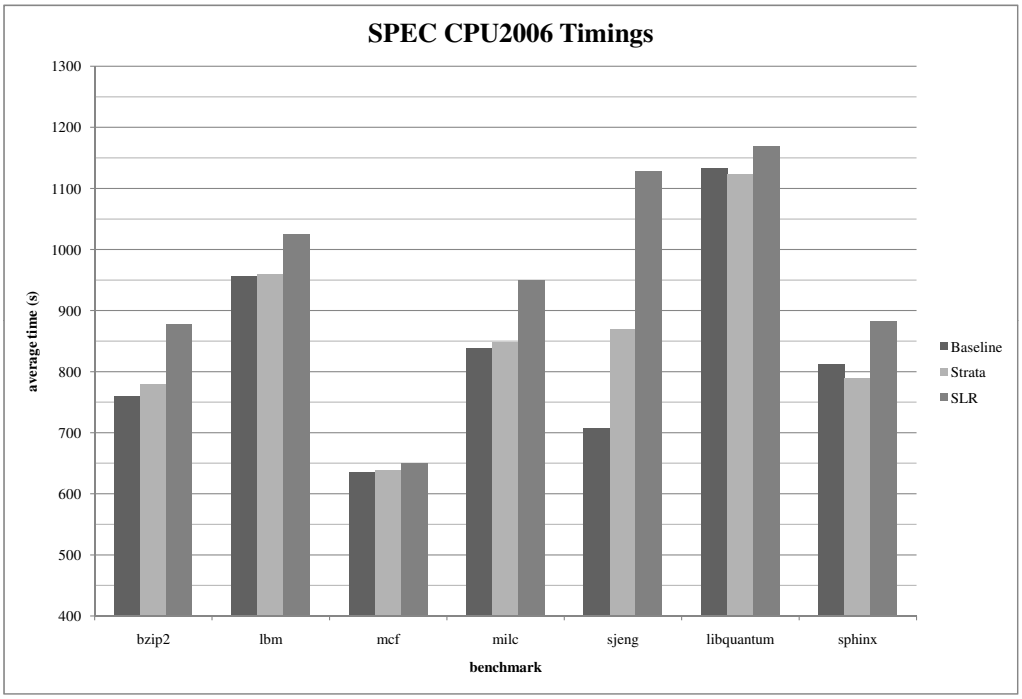

Figure 9. Timing assessment of SPEC 2006 benchmarks.

consequence for these benchmarks is that which is executed for performance testing, so as long as this expected output did not break and SPEC did not report any errors or output discrepancies, as compared with the baseline data, we considered the transformation successful.

Figure 9 and Figure 11 show the runtimes and run-time overheads of seven programs with and without SLR applied. The execution-time overhead of the PVM, the software dynamic translator, averaged 3.4% over the execution times of the compiler-produced binary (native) code. The average execution-time overhead of the PVM running the SLR binary rewriting rules over the native run was 15.6% and ranged between 2.4% and 59.5%. SLR, then, only incurs a 9% overhead over the PVM alone, i.e., dynamic binary translation but with no transformations being applied.

The current implementation of SLR makes no effort to keep stack variable alignment, and this can cause cache performance losses that can greatly increase the overhead in certain programs. The worst overhead we found was 59.5% for sjeng. Retransforming this program with stack

Unmodified Source			
Wilander Exploit #:	Run 1	2	3
-4	f	f	f
-3	f	f	f
-2	p	p	f
-1	f	f	f
1	f	f	f
2	f	f	f
3	p	p	p
4	f	p	f
7	p	f	p
8	s	s	s
9	p	p	f
10	p	p	p

(a)

Modfied Source			
Wilander Exploit #:	Run 1	2	3
-4	f	f	f
-3	s	f	f
-2	p	p	f
-1	f	p	p
1	f	f	s
2	f	f	f
3	f	f	p
4	f	f	f
7	p	f	p
8	s	s	s
9	p	s	s
10	p	p	p

(b)

Figure 10. Results of running the Wilander buffer-overflow exploit tests. In these tables, "f" indicates a segfault result, "p" indicates that the exploit was prevented and output was generated, and "s" means that the exploit was successful.

alignment, we were able to improve the overhead to 36.8%. The results of security analysis (see section C) suggest misalignment of the stack may have security benefits; we therefore leave it to the user to decide if the performance penalty of misalignment is worth the security benefits. The remaining programs had an overhead of less than 16%, with many falling under 10%. If we consider the overhead for unaligned sjeng as intolerable and accept the overhead for all other programs, then replacing the overhead results for sjeng unaligned with the alignment enforced randomization, the average overhead from SLR is 12.4%.

The more functions that are transformed by SLR, the more instruction rewriting rules must be generated. Reading and processing this file is part of the transformation's overhead. For the SPEC benchmarks we analyzed, less than 2 MB of rewriting rules were generated. However, for very large programs, it is conceivable that SLR could require large instruction-rewriting rule files. The rewriting rules are currently stored in ASCII plaintext, and we believe that a binary encoding of the rewrite rules and an efficient storage technique, such as gzip, could easily reduce the processing time and reduce the memory footprint.

C. Security Effectiveness

We applied SLR to the Wilander buffer overflow test suite [18], specifically to the stack-based buffer overflow subset consisting of twelve exploits. The Wilander suite provides buffer overflows that are able to determine the appropriate amount by which to overflow the vulnerable buffer at run-time. This functionality makes these overflows some of the most difficult cases to thwart.

For SLR, the Wilander suite is problematic because all valid inputs, except for no input, trigger a buffer overflow that results in execution of arbitrary code. Developing a set of regression tests for this suite is problematic, because the expected behavior (in some sense the correct behavior) is a buffer overflow that opens a shell. But this is precisely what SLR intends to defeat.

In order to apply SLR, we compared the binary with the original source code manually to determine which of the four layout heuristics could be used for each function.

The stack-based subset of the Wilander suite contains six functions each containing two exploits for a total of twelve stack-based exploits. From the manual analysis of the binary, we determined that five functions could be transformed with AOI, which correctly identified all variable boundaries.

For the sixth function, a buffer is randomly accessed using constant indices in the source code. As a result, the binary contains constant offsets that refer to part of the buffer, which AOI incorrectly identifies as variable boundaries. The most aggressive layout inference we were able to use that was not affected by the random access of array elements was SOI. The inference produced by SOI was very course grained, only identifying a boundary between the out-arguments region and the local-variables region.

The Wilander test suite was randomized three times using SLR, and all stack-based exploits were attempted for all three randomized versions. Any exploit which failed to achieve the goal of opening a shell was considered thwarted. In all three randomizations, SLR thwarted all but one of the twelve exploits; see Figure 10 (a). The only exploit that managed to succeed was implemented in the function which was transformed using SOI. Analysis of the exploit indicated that the exploit could be thwarted by layout randomization alone, because the overflow targets another variable on the stack. However, SOI was not sufficiently fine-grained to find the boundaries necessary for successful randomization.

For the other eleven exploits, the run-time behavior was either a segmentation fault or a test suite exploit failure report. Since the layout inferences for each functions were validated manually to identify variables correctly, a segmentation fault was considered evidence that SLR thwarted the attack.

The exploits that produced segmentation faults and those that produced exploit failure reports from the test suite were not consistent across the three randomizations. Some exploits require a certain layout of variables to succeed. If this layout is not found, such as if the target data is below the buffer, the attack is considered not possible and the exploit is aborted after printing a message indicating the attack was not possible. Since the layout of variables is random, in some variations the layout was susceptible to an attack while in others the attack was thwarted.

The cause of the segmentation fault for exploits not resulting in an exploit failure report was the random padding added by SLR. The padding caused the exploit to overflow a buffer not intended to be overflowed. The Wilander overflow exploits are very robust. At run-time, they calculate the distance from the buffer being overflowed to the target data. After calculating this distance, a block of characters

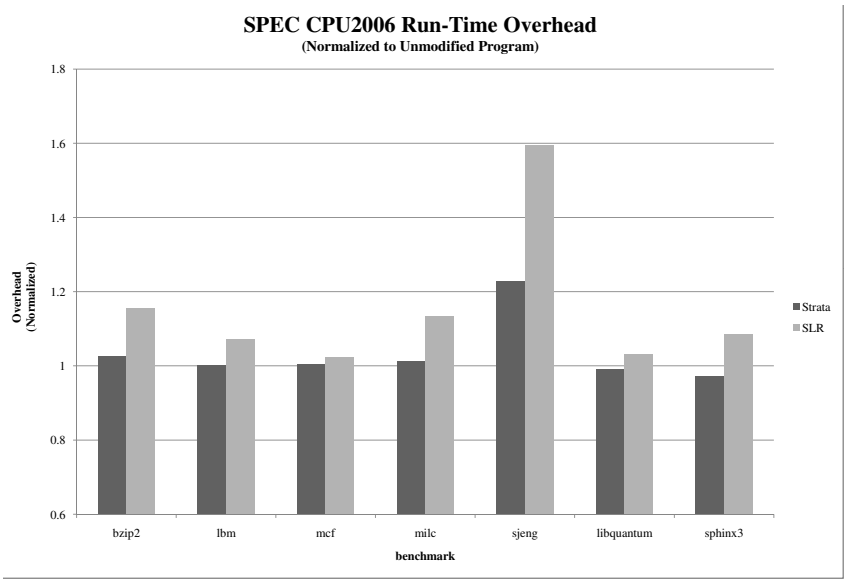

Figure 11. SPEC CPU2006 run-time overhead.

equal in length to the distance are placed in a temporary buffer of static size. However, SLR adds a random amount of padding between variables, thus increasing the distance. The minimum padding amount between any two variables exceeds the statically allocated space for the temporary buffer, and, as a result, an overflow occurs triggering a segmentation fault. In most situations this overflow occurs in the glibc function memset.

While these results show both padding and randomization were effective in thwarting all but one exploit, had SLR been applied to memset, the added padding might have obfuscated the overflow that generated the segmentation fault, thus allowing the exploit to continue and possibly succeed. We altered the Wilander test suite so that the temporary buffer had as much space as necessary to prevent the segmentation fault overflow, and reran the experiment. The results are shown in Figure 10 (b).

Because the test suite calculates distances necessary for overflow at runtime, and the previous cause of exploitation failure had been removed, the expected result was attack success for randomizations where the target data had not been moved so as to prevent the attack. Instead, SLR was successful in thwarting most attacks. All exploits rely on the assumption that local variables are placed on boundaries with addresses that are divisible by four. Stack-layout randomization adds a random padding between variables, and therefore four-byte alignment is not guaranteed. The buffer is still overflowed, but target data was not corrupted in the necessary way to execute the payload.

The Wilander suite was engineered to provide sophisticated exploits that are highly likely to succeed in overflowing a buffer. Nevertheless, SLR was able to thwart attacks, even when the code was altered to improve the probability of attack. Unexpectedly, the fact that the padding added between variables left boundaries on addresses that were not divisible by four defeated some attacks. For more common and likely less well engineered attacks, we would expect SLR to be equally successful.

VII. Related Work

Although a variety of different stack diversity mechanisms are available, the techniques are effected by either source-to-source transformation, recompilation of the original source using a modified compiler, or work at course granularity of memory.

One of the earliest approaches to stack protection was StackGuard [9]. StackGuard offers protection against overwriting of return address using canaries. ValueGuard supplements data items with canaries to permit detection of corruption of data [17]. All data items are replaced with a pointer to a combination of an equivalent data item and a canary.

A different approach to protection is the use of artificial diversity. Bhatkar and Sekar present an approach to data protection in which data representation is randomized using an exclusive-OR with a random key [6]. The encoding is reversed prior to the use of the data. PointGuard applies randomization to pointers, reversing the randomization prior to dereferencing the pointer [8].

Address Space Layout Randomization (ASLR) is a randomization technique in which the major runtime entities (stack, heap, code) are placed at random locations by the system software (see for example [12]). Bhatkar et al. have developed a comprehensive implementation of address

randomization as a source-to-source transformation that can continuously randomize local variable layouts, padding between the variables, and padding between stack frames [7]. An earlier work by Bhatkar et al. added random amounts of padding between the stack frame and local variable section of the stack frame provided only the binary, but limited the functions that could be transformed, especially because of the use of static binary rewriting tools [5].

Determination of stack-frame contents from program binaries has been studied by several researchers. Balakrishnan and Reps have developed a range of techniques using Value Set Analyses [1–4, 13], however reported techniques for variable recovery require extensive resources, and results have only been reported for small device drivers. Their results indicate that, while the approach provides a high percentage of accuracy for stack variables, the effort improves only marginally upon naïve techniques.

VIII. Conclusion

We have presented an approach to diversifying stack contents that does not require access to details of a program other than the binary form.

The core of the technique is a set of naïve inferences from which we derive hypotheses about the boundaries of memory objects on the stack. The approach is speculative in that the hypotheses are evaluated by testing the transformed program.

The approach has been implemented using dynamic binary translation and evaluated using a suite of test programs. The ability to randomize the stack frame for each function in relatively large programs was demonstrated, the execution-time overhead was shown to be reasonable, and the security efficacy of the technique was evaluated using a set of test programs containing carefully constructed vulnerabilities.

IX. Acknowledgements

This work is sponsored by the Air Force Research Laboratory (AFRL) under contract FA8650-10-C-7025. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the AFRL or the U.S. Government.

References

[1] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Compiler Construction*, ser. Lecture Notes in Computer Science, E. Duesterwald, Ed. Springer Berlin / Heidelberg, 2004, vol. 2985, pp. 2732–2733, 10.1007/978-3-540-24723-4_2.

[2] ——, "DIVINE: Discovering variables in executables," in *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 1–28.

[3] ——, "Analyzing stripped device-driver executables," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 124–140.

[4] ——, "WYSINWYX: What you see is not what you eXecute," *ACM Trans. Program. Lang. Syst.*, vol. 32, pp. 23:1–23:84, August 2010.

[5] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. Berkeley, CA, USA: USENIX Association, 2003, pp. 8–8.

[6] S. Bhatkar and R. Sekar, "Data space randomization," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 1–22.

[7] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. Berkeley, CA, USA: USENIX Association, 2005, pp. 17–17.

[8] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard™: Protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. Berkeley, CA, USA: USENIX Association, 2003, pp. 7–7.

[9] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*. Berkeley, CA, USA: USENIX Association, 1998, pp. 5–5.

[10] Hex-Rays, "IDA Pro," http://www.hex-rays.com/products/ida/index.shtml.

[11] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson, "ILR: Where'd my gadgets go?" 2011, *in submission*.

[12] M. Payer and T. R. Gross, "Fine-grained user-space security through virtualization," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '11. New York, NY, USA: ACM, 2011, pp. 157–168.

[13] T. Reps and G. Balakrishnan, "Improved memory-access analysis for x86 executables," in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, ser. CC'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 16–35.

[14] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 36–47.

[15] Standard Performance Evaluation Corporation, "SPEC CPU2006 benchmarks," http://www.spec.org/osg/cpu2006, 2006.

[16] The PAX Team, http://pax.grsecurity.net.

[17] S. Van Acker, N. Nikiforakis, P. Philippaerts, Y. Younan, and F. Piessens, "ValueGuard: Protection of native applications against data-only buffer overflows," in *Proceedings of the 6th International Conference on Information Systems Security*, ser. ICISS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 156–170.

[18] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS. The Internet Society, 2003.

[19] D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, and A. Nguyen-Tuong, "Security through diversity: Leveraging virtual machine technology," *IEEE Security and Privacy*, vol. 7, pp. 26–33, January 2009.