Improved Kernel Security Through Memory Layout Randomization

Dannie M. Stanley*, Dongyan Xu*, Eugene H. Spafford*

*Department of Computer Science, Purdue University, West Lafayette, IN 47907
email: {ds,dxu,spaf}@cs.purdue.edu

***Abstract*—The vast majority of hosts on the Internet, including mobile clients, are running on one of three major operating system families. Malicious operating system kernel software, such as the code introduced by a kernel rootkit, is strongly dependent on the organization of the victim operating system. Due to the lack of diversity of operating systems, attackers can craft a single kernel exploit that has the potential to infect millions of hosts.**

If the underlying structure of vulnerable operating system components has been changed, in an unpredictable manner, then attackers must create many unique variations of their exploit to attack vulnerable systems en masse. If enough variants of the vulnerable software exist, then mass exploitation is much more difficult to achieve.

Many forms of automatic software diversification have been explored and found to be useful for preventing malware infection. Forrest et. al. make a strong case for software diversity and describe a few possible techniques including: adding or removing nonfunctional code, reordering code, and reordering memory layouts. Our techniques build on the latter.

We describe two different ways to mutate an operating system kernel using memory layout randomization to resist kernel-based attacks. We introduce a new method for randomizing the stack layout of function arguments. Additionally, we refine a previous technique for record layout randomization by introducing a static analysis technique for determining the randomizability of a record.

We developed prototypes of our techniques using the plugin architecture offered by GCC. To test the security benefits our techniques, we randomized multiple Linux kernels using our compiler plugins. We attacked the randomized kernels using multiple kernel rootkits. We show that by strategically selecting just a few components for randomization, our techniques prevent kernel rootkit infection.

I. Introduction

Organizations have strong economic incentive to standardize the software that they use across their enterprise [1]. Furthermore, organizations have strong economic incentive to choose market-leading software [2]. Such economic incentives have lead to a homogeneity of operating system software among network-connected hosts. Software homogeneity is antithetical to computing security[3], [4].

The vast majority of hosts on the Internet, including mobile clients, are running on one of three major operating system families[1]. Within each operating system family there exists many versions of the operating system code. However, relative to the total number of hosts, the number of unique versions of operating systems is minuscule.

[1]http://www.netmarketshare.com/

Malicious operating system kernel software, such as the code introduced by a kernel rootkit, is strongly dependent on the organization of the victim operating system. Due to the lack of diversity of operating systems, attackers can craft a single kernel exploit for a single unique operating system version that has the potential to infect millions of hosts.

One approach to strengthen computer security is *software diversity*. Organizations can choose to add diversity to their software ecosystem by purchasing less popular software that performs the same tasks. However, such a choice is often not economically feasible even if viable alternatives exists [1].

Another approach to software diversification is for the software itself to mutate; where multiple variants of the same version of a piece of software are used. The variants are compatible with each other and *function* the same from the end-user's point of view. However, some underlying component of their *structure* has been changed.

Computer virus authors have long used automatic software diversification to avoid detection. Each time the virus infects a new victim the virus mutates. This mutation prevents signature-based anti-virus software from detecting the virus. This class of malware is often referred to as *polymorphic* computer viruses [5], [6]. In much the same way, but inverted, polymorphic software can be used offensively to withstand malicious software. If the vulnerable components of a piece of software have been changed in an unpredictable manner, then attackers must create many unique variations of their exploit to attack vulnerable systems en masse. If enough variants exists, then mass exploitation is much more difficult to achieve if not impossible.

Some attacks, such as the return-oriented techniques introduced by [7], [8] and [9] require the memory addresses of key system libraries. An attacker must be able to predict the correct addresses or the attack will fail and the exploited program will likely encounter a fault. Depending on the exploited vulnerability, the attacker may be able to guess the addresses at run-time. However, often these attacks are crafted for predetermined memory layouts [10]. If the memory layout can be obfuscated in some way, then the exploitation can be thwarted.

Forrest et. al. make a strong case for software diversity and describe a few possible techniques including: adding or removing nonfunctional code, reordering code, and reordering memory layouts [11]. Our technique builds on the latter. We

describe two different ways to mutate an operating system kernel using memory layout reordering to resist kernel-based attacks.

Our randomization techniques occur at compile-time. Some software diversification techniques occur at run-time or load-time. Some, such as instruction set randomization, incur significant run-time overhead [12]. In contrast, our techniques incur no run-time overhead and are therefore suitable for any system including low-powered devices such as mobile phones and embedded devices.

Our randomization techniques are tailored specifically for operating system kernels. The techniques themselves have wider application, however they are practically well suited for an operating system kernel. Because our technique occurs at compile-time all dependent software must be recompiled to be compatible. In the user-space, this could be a significant undertaking; if for example, one wanted to randomize a system library all dependent software would have to be recompiled. However, the kernel compilation is self contained, loadable kernel modules not withstanding, and its code is executed only through well-defined system calls. Our system does allow for loadable kernel modules to be compiled with compatible randomization during kernel compilation or separately.

Practically, we would not expect our techniques to be adopted by every end-user. A small fraction of computer users would have the technical prowess to compile their own kernel and for many operating systems the kernel source code is not available. Rather, our techniques are better suited for organization-wide application. For example, military organizations often distribute their own version of the Linux kernel. If they employed our technique they could withstand all versions of kernel malware that wasn't specifically targeted for their organization. Similarly a mobile device manufacturer could randomize their kernels so that it would be invulnerable to the same attacks leveraged against other similar devices.

Our contributions are as follows: We introduce a new method for randomizing the stack layout of function arguments. We refine a previous technique for record layout randomization by introducing a static analysis technique for determining the randomizability of a record. We provide an implementation of our techniques using the plugin architecture offered by GCC. Finally, to evaluate the security benefits of our techniques we randomize multiple Linux kernels using our plugins and attack them using kernel rootkits. We show that by strategically selecting just a few components for randomization, our techniques prevent all tested kernel rootkits.

II. Related Work

A. Address Space Layout Randomization

Many forms of automatic software diversification have been explored and found to be useful for preventing malware infection [13], [11], [14], [12]. One prominent example of automatic software diversification is *address space layout randmization* (ASLR) [11], [14].

When a commodity operating system loader prepares a program for execution it lays out various sections in memory including the program text, user stack, user heap, and memory mapped files including dynamic system libraries. If ASLR is not employed, the location of these key sections is predictable. If ASLR is employed, then the location of these key sections is unpredictable because the starting address of each section is different upon every new execution of the program[2].

B. Polymorphing Software

Our work is partially inspired by previous work in *polymorphing software* by Lin et. al [15]. Our record field order randomization technique is similar to the data structure layout randomization method described therein. However we build upon their work with two separate but related efforts. First, we introduce a technique for automatically determining the suitability of a record for randomization. This is listed as a limitation of the previous work. Second, we introduce a novel polymorphing software technique.

III. Outline

The remainder of this paper is organized as follows: we begin in Section IV by laying out the design of our randomization techniques followed by a description of our GCC implementation in Section V. In Section VI, we evaluate the security merits of our approaches and the performance of our implementations. Finally, in Section VII, we summarize our contributions and findings.

IV. Design

Our design has three distinct but related parts: *record field order randomization* (RFOR), RFOR *suitability analysis*, and *subroutine argument order randomization* (SAOR). Each part is described in the following three sections.

A. Record Field Order Randomization

We have designed our field order randomization technique to occur at compile-time. During compilation we randomize the field order of the record *definition*. As a result, all instances of that record type are defined with the new field order in the resulting binary. Each compilation unit may have its own definition for a given record. Therefore, we use a seeded pseudo-random algorithm so that the same field order can be replicated across multiple compilation units. This also allows for modular software, such as loadable kernel modules, to be compiled with compatible record layouts.

Our field order randomization technique takes as input the source code of the software to be randomized, a set of one or more record names, a set of one or more randomization seeds, and a set of one or more padding flags. Each record name may correspond to a unique randomization seed or one seed may apply to all record names. Each record name corresponds to a unique padding flag that indicates whether or not the record receives padding. Our field order randomization technique produces as output a compiled binary. Given different randomization seeds and padding flags, our system may produce

[2]some systems, such as PaX on Linux, do not rerandomize the layout of child processes

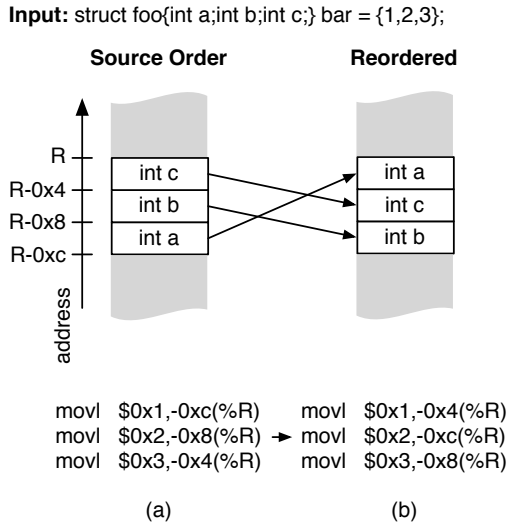

Fig. 1. Record field order randomization. "R" represents a general-purpose processor register

distinct binaries. However, our field reordering algorithm is not collision resistant; multiple seeds will produce the same field order. Naturally, records with more fields will have more layout permutations.

To increase the possible number of layout permutations, our system takes as input a padding flag for each record name. When this boolean flag is set to true for a given record name a pseudorandom number of bogus fields are inserted pseudorandomly into the record. The randomization algorithm used to add padding is also seeded with the same seed that determines the layout. We have bound the amount of padding automatically generated so that a record will have no more than two times the number of fields of its original. This upper bound was selected arbitrarily. In practice, some bound will be desired to keep the records from becoming space-inefficient.

Field order randomization takes place after the source code has been parsed into an abstract syntax tree but before any compiler optimization passes. Figure 3, located later in the implementation section (V-A), illustrates the abstract syntax tree for both a randomized (b) and unrandomized (c) record. Figure 1 illustrates the stack memory layout and machine instructions for assignment to an unrandomized (a) and randomized (b) record.

Given a record type τ, Algorithm 1 depicts field order randomization subroutine. We use the Knuth shuffling algorithm for reordering the fields [16].

When a kernel data structure is reordered using RFOR, attack code compiled, without randomization, against the kernel source code will be incompatible. For example, regarding the record randomization illustrated in Figure1, suppose the attack code was trying to assign a malicious value to field "foo.b." The malicious code will assume that "foo.b" is at offset `0x8` and assign the malicious value to offset `0xc` instead. A similar problem occurs when attack code tries to read from a predetermined offset. The results of the malicious code execution are unpredictable in the presence of RFOR. For our evaluation, detailed in Section VI, we found that malicious code often resulted in a non-destructive kernel "oops" (not panic) when a victim data structure was randomized. For all tested kernel rootkits, the malicious software failed to run as intended on systems compiled with RFOR. In some cases, infection was prevented entirely by RFOR.

Algorithm 1 Field order randomization with padding

$a[] \leftarrow \tau_{fields}$
$n \leftarrow$ count of items in $a[]$
if option to add padding is **true then**
  $x \leftarrow$ seeded pseudorandom integer $< max$ and $> min$
  for 1 **to** x **do**
    $a[] \leftarrow$ new field declaration
    $n \leftarrow n + 1$
  end for
end if
for $i = n - 1$ **down to** 1 **do**
  $j \leftarrow$ seeded pseudorandom integer ≥ 0 and $\leq i$
  swap $a[j]$ and $a[i]$
end for
$\tau_{fields} \leftarrow a[]$

B. Suitability of a Record for Field Reordering

Not all records are suitable for field reordering. If a record is used or defined, in a raw memory form, by an external system that expects a predetermined format, then the record may be an unfit candidate for randomization. For example, for our evaluation we randomized the Linux TCP header record (tcphdr). When TCP packets were formed using the randomized layout, the system could not communicate with other systems using TCP. If both end-points had their TCP headers reordered in the same way, then it may be possible for the two to communicate; assuming that deep packet inspection or similar transport-layer logic is not present on intermediary nodes and total header size doesn't change due to field realignment (see discussion of record resizing in Section V-A and Figure 4).

If the compiler supported it, the programmer could annotate the source code to indicate whether or not a record's fields may be safely reordered. For example, some compilers provide a way for the programmer to suggest how to align a variable type. In GCC, the programmer can specify type attributes using the keyword __attribute__ at the end of a type definition. The programmer can specify the "packed" attribute of a record to instruct the compiler not to realign its fields. We can leverage this paradigm for our purposes; we can add a custom attribute to the compiler that indicates that the record is unfit for field reordering. If the attribute is not present, the compiler could freely reorder fields as it sees fit. We describe our implementation of this approach in Section V-A. This approach may have benefits beyond randomization for security purposes. For example, if the compiler were free to reorder

fields it may be able to automatically improve record cache performance as described by Chilimbi et al [17].

In the event that such a compiler mechanism were widely adopted today, there exists much source code that is not already annotated. In the absence of compiler support and source code annotations, we have designed a static analysis technique for testing the suitability of a record for field reordering. Our initial design was informed by previous research at Hewlett-Packard in compiler optimization techniques [18]. However, our technique has a different purpose and differs significantly from this prior work.

First we define a few terms: x, y and z are free variables. τ represents the candidate type and $\neg\tau$ represents all other types. A type followed by an $*$ represents a pointer to a variable of that type. τ_n represents field n in τ. The function A returns the memory address of its argument.

There are four conditions that may disqualify a record for field reordering[3]:

1) $x_{\tau*} = A(y_{\neg\tau})$*:* If a variable of type τ pointer is positioned on the lefthand-side of an assignment[4] and the righthand-side expression does not result in a τ-pointer, then τ may not be suitable for field reordering.

An example of how this operation is potentially unsafe for field reordering is as follows: suppose that the righthand-side expression is the address of a buffer that is filled by a network receive function and the format of the receive buffer is consistent with the headers of a published network protocol. The protocol specification is external to our system and not subject to compile-time reordering. Therefore, if τ is reordered, then τ can no longer be used to reliably access parts of the network buffer.

2) $x_{\neg\tau*} = A(y_{\tau})$*:* If the righthand-side expression of an assignment[4] results in a τ pointer and the lefthand-side variable is not of type τ-pointer, then τ may not be suitable for field reordering.

An example of how this operation is potentially unsafe for field reordering is as follows: suppose the inverse of the previous example listed in Section IV-B1. Suppose that the righthand-side expression results in a memory address that is formatted as a reordered version of type τ and the lefthand-side is expecting a pointer to a region of memory that specifies network headers to be sent directly on the network, then the send buffer will be formatted in a way that is inconsistent with the network protocol and network communication will fail.

3) $x = A(y_{\tau_z})$*:* If the righthand-side expression of an assignment[4] results in the address of a field inside of a variable of type τ, then τ may not be suitable for field reordering.

An example of how this operation is potentially unsafe for field reordering is as follows: suppose that the x is used in the source code to calculate the address of a sibling field. If the sibling offset address calculation were not aware of the potential for field reordering, then the calculation would likely be incorrect. It is worth noting however, that code containing this kind of sibling calculation would be difficult to maintain considering alignment issues alone. However uncommon, it remains a possibility.

[3]Our system is designed to protect commodity operating system kernels. As a result, our design is pertinent to the C programming language specifically.

[4]In the source code, an assignment may be manifested in various forms. For example: passing a variable into a function is an assignment.

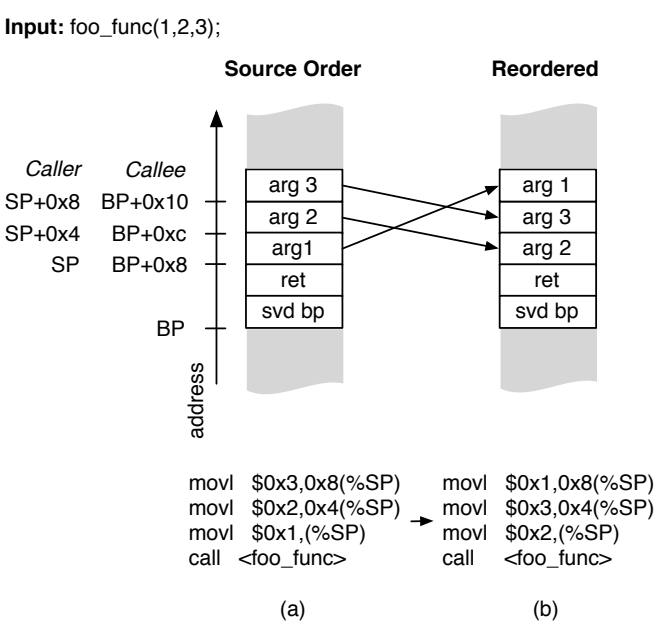

Fig. 2. Subroutine argument order randomization. "SP" represents the stack pointer register. "BP" represents the stack base pointer register.

4) τ *is a member of a union or record:* If τ is the type of a member included in a union definition, then τ may not be suitable for field reordering.

An example of how this use is potentially unsafe for field reordering is as follows: suppose that a union was constructed with two members τ and $\neg\tau$. Supposed that τ_x and $\neg\tau_y$ were the same offset in the union. If τ_x were relocated at compile time, then $\neg\tau_y$ would no longer point to the same offset. Additionally, suitability analysis would have to be performed on all unions that include τ because unsafe casting of the union may occur.

Similarly, if τ is the type of a field included in anther record, then τ many not be suitable for field reordering. Suitability analysis would have to be performed on all records that include τ because unsafe casting may occur.

C. Subroutine Argument Order Randomization

Like RFOR, we have designed our subroutine argument order randomization technique to occur at compile-time. During compilation we randomize the argument order for each *definition* of, *type* of, and *call* to a given subroutine. Also like RFOR, we use a seeded pseudorandomization algorithm so that the same argument order can be replicated across multiple compilation units.

Our argument order randomization technique takes as input the source code of the software to be randomized, a set of one or more subroutine names, a set of one or more randomization seeds, and a set of one or more padding flags. The meaning

Algorithm 2 Subroutine argument order randomization with padding

```
for all (definitions, types, calls) of S do
  a[] ← S_arguments
  n ← count of items in a[]
  if option to add padding is true then
    x ← seeded pseudorandom integer < max and > min
    for 1 to x do
      if S is definition then
        a[] ← new parameter declaration
      end if
      if S is type then
        a[] ← new argument type
      end if
      if S is call then
        a[] ← new call argument
      end if
      n ← n + 1
    end for
  end if
  for i = n − 1 down to 1 do
    j ← seeded pseudorandom integer ≥ 0 and ≤ i
    swap a[j] and a[i]
  end for
  S_arguments ← a[]
end for
```

of each input is analogous to the RFOR example outlined in Section IV-A. Our argument order randomization technique produces as output a compiled binary.

When a kernel subroutine is reordered using SAOR, attack code compiled, without randomization, against the kernel source code will be incompatible. For example, regarding the subroutine argument order randomization illustrated in Figure2, suppose the attack code made a call to a randomized subroutine. The callee variable that holds the first argument will get the value of the malicious callers third parameter. Like RFOR, the results of the malicious code execution are unpredictable.

Not all subroutines are suitable for SAOR. Subroutines that are called using a function pointer are not randomizable because the caller would not be identified at compile-time and the call stack would not be correctly reordered. Additionally, functions with a variable length argument list are likely not randomizable without modifications to the subroutine logic.

For our evaluation, detailed in Section VI, we found that like a system using RFOR, malicious software failed to run as intended on systems compiled with SAOR.

V. Implementation

To test our design, we implemented three plugins: RFOR, RFOR Fitness, and SAOR using the GNU Compiler Collection (GCC). GCC versions 4.5 and newer have the ability to load user-supplied plug-ins during compilation [19]. We leverage this plug-in architecture to realize our design. Following are the details for each plug-in.

A. Record Field Order Randomization

The RFOR GCC plugin provides compile-time record offsets randomization. The GCC plug-in architecture provides an event callback named "PLUGIN_FINISH_TYPE" that gives us a hook into the compilation process. The PLUGIN_FINISH_TYPE event occurs after a record or union type specifier has been parsed. The event data passed to our plugin's callback function is a pointer to the AST tree node for the most recently parsed record or union as illustrated in Figure 3(b) and (c). We are only interested in record types. As a result, we ignore events associated with unions.

We implemented two versions of this plug-in. Both versions perform the randomization step in the same way; using the algorithm described in Algorithm 1. To save the new layout, the pointer fields of each field declaration are updated in the AST as shown in Figure 3(c). The two versions of this plug-in differ only in how the records are selected for randomization. The first version, variant "A" randomizes all records encountered except those with the GCC attribute "noreorder" set in the source code, e.g. "__attribute__ ((noreorder))".

The second version, variant "B" randomizes only the records provided by name on the command line. The plug-in and its argument are provided by the user as GCC command line flags. The plug-in accepts multiple arguments, one argument for each target record name. All records of the same name will be randomized because the name is not necessarily unique across all compilation units. The randomization happens at type definition time and therefore applies to all instances of the record.

In addition to standard randomization, our plug-in allows the user to specify a boolean flag for each target record that indicates whether or not the record should be padded with bogus members. If yes, our plug-in adds a random number of bogus fields to the target record or records.

The total record size may change as a side effect of field reordering as illustrated in Figure 4. Notice that Figure 4(c) is a reordering of fields found in (b). If a record is not "packed," then the compiler may align fields for efficiency. One common optimization is for the compiler to align field offsets on word boundaries as shown in (b) and (c). If a record is packed, as in Figure 4(a) the record will always take up the minimal amount of memory and the total record size will not change as a result of randomization.

Our initial implementation performed randomization after the entire compilation unit had been parsed but prior to optimization passes (PLUGIN_PRE_GENERICIZE). We observed that if the size of the record changed as a result of randomization, then some compile-time calculations such as "sizeof" were incorrect. Also affected was offsetof (__builtin_offsetof) calculations which we later discovered was an essential calculation for common Linux kernel data structures like "list." GCC, folds the result of sizeof into a constant during parsing and leaves no indication that the constant was a result of the

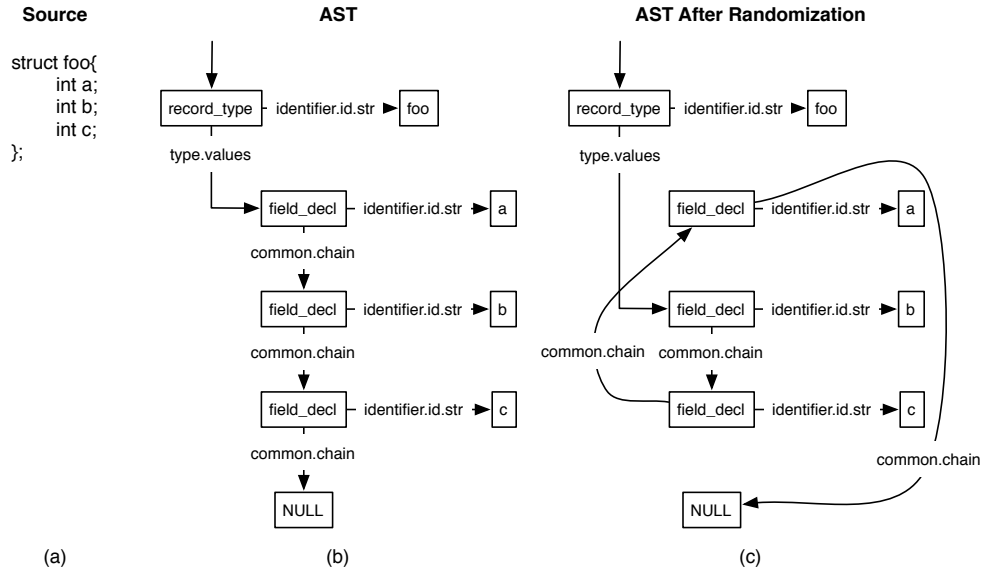

Fig. 3. AST-based randomization

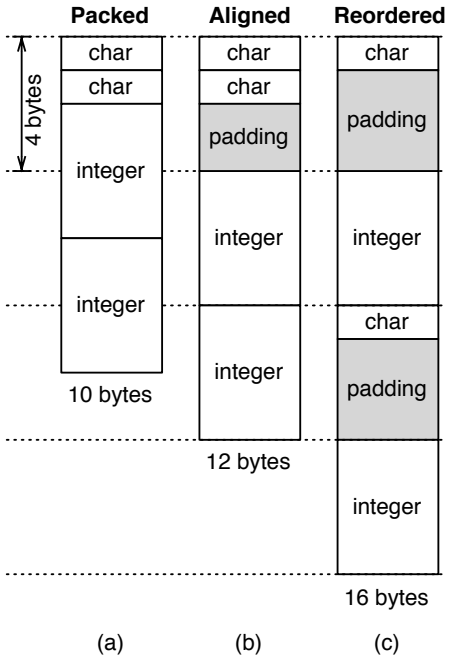

Fig. 4. Record size variation

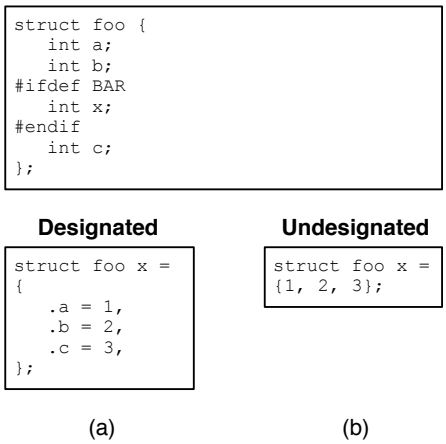

Fig. 5. Variable initializers

sizeof calculation. As a result, there was no reliable way for us to find the constant in the AST to update it. To remedy this problem, we moved the randomization procedure to the PLUGIN_FINISH_TYPE event as previously described. This event happens after the record definition is parsed but before it is an argument to sizeof and other similar compiler functions.

We discovered that this early randomization approach fixed size-related calculations but broke non-designated variable initializers. Figure 5 illustrates the difference between initializers. Currently, this is the only known limitation of our plug-in. The problem is that when a non-designated initializer is parsed, the compiler assumes that the source-code-order of values matches the source-code-order of the record members. This problem could be solved by modifying the way GCC stores initializers in the AST. Simply adding a node attribute that indicats how the initializer was formed would solve the problem. However, we discovered that, for the tested kernel records, non-designated initializers were not used for our evaluation. The reason for this is shown in the Figure 5. If the macro BAR is defined, then (b) would be an invalid initializer assuming that the programmer was intending to assign the value 3 to field c and not x. However, with or without defining BAR, initializer (a) would assign values correctly. Our static analysis tool, described in Section V-B, reports when and if the target record is declared with an initializer.

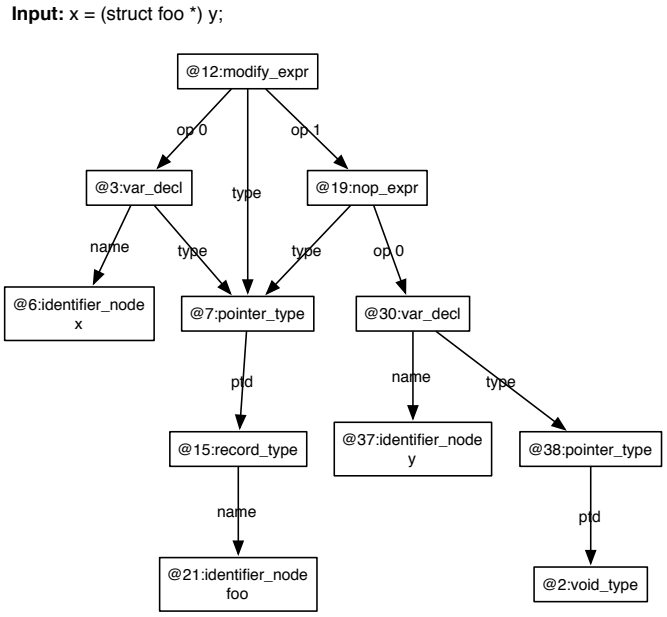

Fig. 6. AST of a cast from one pointer type to another

GCC permits an incomplete variable type, such as a flexible array, to be positioned as the last field of a record. Therefore, if the last element is an incomplete type, it cannot be moved during randomization. Our plug-in handles this situation by pinning the last field to the last position when the field type is incomplete.

B. RFOR Fitness Check

The RFOR Fitness Check GCC plug-in is a static analysis tool for determining the fitness of a candidate record for randomization by inspecting the abstract syntax tree during compilation. The inspection occurs after parsing has been completed but prior to optimization passes.

Again we leverage the GCC plug-in architecture callback event "PLUGIN_PRE_GENERICIZE" to do most of the work. The callback associated with this event received the function definition (FUNCTION_DECL) of the most recently parsed function definition. From this root node we traverse the tree using the API functioned named "walk_tree()." One of the parameters to walk_tree is a callback function name. Our callback function checks for four kinds of nodes: NOP_EXPR, CALL_EXPR, ADDR_EXPR, and CONVERT_EXPR.

A NOP_EXPR node represents, among other things, a cast of one type to another. Figure 6 shows the case where a pointer to a record is cast from another type. This case would be reported by our tool. We can use the NOP_EXPR to find the cases where memory is cast to or from a pointer to the type of the target record. If the target record for analysis was of type foo as shown in the figure, then we check to see if the left-hand side of the cast is a pointer to a variable of type foo by checking the corresponding nodes of the AST. If the right-hand side of the modify expression is not also a pointer to a variable of type foo, then our plug-in reports this case as a potentially unsafe operation.

GCC does not allow for direct assignment from one record type to another even if both records have the same layout[5]. Similarly, GCC does not allow for coerced assignment from one record type to another using a cast[6]. Therefore, all unsafe assignments to/from a record type will be pointer casts using the NOP_EXPR. Our first two cases: $x_{\neg\tau*} = A(y_\tau)$ and $x = A(y_{\tau_z})$ are therefore tested when a NOP_EXPR is encountered in the AST.

The third case: $x = A(y_{\tau_z})$ is tested also during the PLUGIN_PRE_GENERICIZE event. We have developed two detection mechanisms for this case. The first is the more straightforward and conservative approach. When an ADDR_EXPR is encountered we check the operand. If the operand of the address expression is the field of a target record, then we report the instance. The second implementation of this test is similar, however we only report the instance if the ADDR_EXPR is part of a CONVERT_EXPR and the address is immediately converted to an integer. In this case the the ADDR_EXPR is a child node to the CONVERT_EXPR so we ignore ADDR_EXPR and check the operand of all instances of CONVERT_EXPR. If the type of the convert expression is INTEGER, then we report the instance. More information about the motivation for this optional refinement is described in the evaluation Section VI-B3.

Often, a record field is passed by address to a function. As described in Section IV-B3, taking the address of a field and then using it to calculate the address offset of a sibling field is an unlikely occurrence. An even more unlikely occurrence is for such a calculation to be performed on the address of an argument to a function. Therefore, our evaluation ignores cases when a field address is taken in the context of a call parameter. To detect such occurrences, we collect all call parameters during the PLUGIN_PRE_GENERICIZE event that are pointers to fields of the target record type. Then later during AST tree traversal, when the address is taken of the field, the ADDR_EXPR corresponding to call parameters are not reported.

The fourth and final case, the case when the target record is nested inside of a union or record, is tested when the compiler event PLUGIN_FINISH_TYPE is encountered. The event data received is a finished type node. For our purpose we ignore all types except RECORD_TYPE and UNION_TYPE. Each type contains a set of FIELD_DECL nodes. We iterate over each field declaration and check the type. If the target type is found in a FIELD_DECL, the instance is reported as a potentially unsafe condition.

As mentioned in Section V-A, non-designated variable initializers are not compatible with field reordering. This is an implementation limitation. Therefore, as part of our fitness analysis, we report when initializers are used for the target record type. We use the PLUGIN_FINISH_DECL event to check for such cases. Unfortunately, by the time

[5]error: incompatible types when assigning to type
[6]error: conversion to non-scalar type requested

the PLUGIN_FINISH_DECL event occurs the initializers has already been parsed into its final form in the AST. At this point, from the AST perspective, a non-designated initializer is structured the same as a designated initializer. As a result, we report all instances where the target record is initialized. As previously described, a modification to GCC proper could remedy this situation by classifying the initializer nodes as either designated or non-designated in the AST.

C. Subroutine Argument Order Randomization

The SAOR GCC plugin provides compile-time subroutine argument order randomization. The GCC plug-in architecture provides two callback events that we leverage for this plugin named "PLUGIN_PRE_GENERICIZE" and "PLUGIN_OVERRIDE_GATE." We introduced the former in Section V-A. The event data passed to our plug-in's callback function is a pointer to the AST tree node for the most recently parsed function definition. The latter event, PLUGIN_OVERRIDE_GATE, occurs many times during optimization passes. However, we are only interested in the first occurrence. The first occurrence happens before optimization but after all PLUGIN_PRE_GENERICIZE events.

1) Randomize Call Expression Arguments: When a PLUGIN_PRE_GENERICIZE event occurs, we traverse the function definition's AST and inspect all call expressions. If the target of the call expression is a function targeted by randomization, then we reorder the call arguments. Because the call expression is allocated with a static number of arguments and to support adding bogus call parameters, we must build a new call expression with the new arguments using the API function named "build_call_array_loc." Once we build the new call expression we replace the original with the new one in the AST.

Also during the PLUGIN_PRE_GENERICIZE event, we capture the unique identifier for the completed function definition if the function is a target for SAOR. This identifier is used later for definition and type randomization. We discovered that we must randomize all of the call expressions prior to randomizing the function definition and type. Otherwise, if bogus fields were added to the definition, then the parser would complain when a corresponding call expression was parsed because the source code would not contain enough parameters; we cannot fix the call expression until after it has been parsed into an AST.

2) Randomize Function Definition and Type: The PLUGIN_OVERRIDE_GATE occurs after all functions have been parsed into the AST. The difference between this event and PLUGIN_PRE_GENERICIZE is that the latter occurs after each function is parsed. Therefore at the time that the PLUGIN_OVERRIDE_GATE event occurs our AST is in an inconsistent state. All calls to target functions have been randomized however, the function definition and type has not yet been randomized. Recall from the previous section that we had to delay randomizing these components until after all calls have been parsed to avoid compilation errors.

We collected the AST unique identifier for each of the completed target function definitions previously during the PLUGIN_PRE_GENERICIZE event. Now we iterate over that list of targeted function definitions and randomize the definition as well as the associated type for each.

Reordering the arguments for both the definition and the type was similar to RFOR randomization illustrated in Figure 3. We obtained parameter declaration head list node and the argument type head list node from the GCC macros DECL_ARGUMENTS and TYPE_ARG_TYPES. Using the algorithm described in Algorithm 2, we shuffled each list by updating the next pointer of each element. For the type, we used the function GCC build_function_type to reinitialize the type attributes.

VI. Evaluation

We performed our evaluation on a 2.30GHz four core Intel®Core™i5-2410M CPU with 8 GB of RAM running 32 bit Fedora (15), Linux (2.6.38.6-26), and GCC (Red Hat 4.6.3-2). Our experiments were performed in KVM/QEMU (0.14) virtual machines with 1024 M of memory allocated for each.

A. Security Benefits

To evaluate the security benefits of record field order randomization, we employed our RFOR plug-in to compile two different Linux kernel versions, 2.6.26 and 3.3[7], and attempted to run four kernel rootkits against the protected system.

Kernel rootkits in the wild are kernel version-sensitive. As a result, two of the tested rookits were rewritten to be compatible with our test systems using the principles outlined in the original work. The hidefile rootkit is based on [20]. The hideproc rootkit is based on [21]. The other two rookits Adore-NG and hp were used in their original form with few compatibility modifications.

One cannot compile Linux kernel versions less than 2.6 with GCC version 4 or greater. The plug-in architecture was not introduced in GCC until version 4.5. As a result, the oldest kernel that we were able to randomize was 2.6.26 using GCC 4.7.3 and our RFOR plugin. We were able to test the adore-ng rootkit against this version of the Linux kernel.

We identified five security sensitive records in the Linux kernel based, in-part, on previous work: task_struct, module, file, proc_dir_entry and inode_operations [22]. We compiled our test kernels using variant B of our RFOR plug-in and specifying, depending on the experiment, some subset of the five security-sensitive records as inputs (with no padding added).

All of the tested rootkits were introduced into the kernel as kernel modules. As a result, randomizing a single record type named "module" defeated all tested rootkits before they were loaded into the kernel. The specific failure was that the module loading procedure could not find the module's name in the attacker-provided module struct.

[7]We used Fedora's custom version 2.6.43.8-1 which is a patched version of vanilla kernel 3.3

It is possible for malicious kernel code to be introduced through means other loadable kernel modules [23]. We performed some further experiments without randomizing the "module" record type to test our protection mechanism against rootkit behaviors. We found that if we randomize the record type named "task_struct," we prevent hp, adore-ng, hideproc from hiding system processes. If we randomize the record type named "module," we prevent hidefile from hiding system files.

Similar to our evaluation of RFOR, to evaluate the security benefits of subroutine argument order randomization, we employed our SAOR plug-in to compile Linux kernel version 3.2.46 with all calls-to, declarations-of, and types named "pid_task" randomized. This kernel subroutine returns the corresponding task_struct for a given process identifier (PID). We then attempted to use the hp rootkit against this kernel. The hp rootkit was not able to hide a process as a result of SAOR randomization. Though the rootkit loaded successfully into kernel space, its payload was neutralized.

B. Randomizability Analysis

We used our RFOR Fitness plugin to test the suitability of the record type "task_struct" for randomization. For each potentially unsafe condition found in the source code, our plugin emits a warning message complete with file name and line number[8].

For our experiments we compiled the Linux kernel version linux-2.6.38.8 using our RFOR Fitness plug-in with "task_struct" as an argument and the linuxconf configuration template named "allnoconfig."

The plug-in reported that there were 312 conditions found in the kernel source code that may disqualify task_struct from randomization. After careful analysis we found that many of the cases weren't actually a problem for randomization and could be automatically detected. We discovered that we could employ whitelists to reduce the total number of false-positive. Following is a list of heuristics that we used to create the whitelists. These heuristics are provided as an aid to analysis. The specifics, like function names and types, will vary from system to system. The RFOR Fitness plug-in results for task_struct both with and without whitelists are provided in Table I.

1) Generics: Often large code bases provide a reusable set of generic types, macros, and functions that implement common data structures like lists and queues. These data structures can be reused for a variety of types. For example, the Linux process list is a list of task_structs. The same list implementation used for processes, can be used to create other kinds lists. To allow the list to hold multiple types, at some point the type will be untyped and cast as a generic list member. This casting trips the assignment to/from $\tau*$ condition.

We found that we could whitelist specific Linux types and drastically reduce the number of conditions reported by the RFOR Fitness plug-in. Specifically we whitelisted the following pointer types for assignment: list_head, hlist_node, plist_node, rcu_head, sched_entity, sched_rt_entity, __wait_queue.private, raw_spinlock.owner, mid_q_entry.callback_data. Additionally, we whitelisted the function named heap_insert[9]

In addition to generic data structures, there were three generic error-related functions we determined to be safe for field reordering. Specifically we whitelisted the following functions: IS_ERR, PTR_ERR and ERR_PTR.

2) Memory Allocation Functions: When memory is dynamically allocated the raw memory begins untyped and is cast as a type. The assignment of this untyped memory to a $\tau*$ is reported by our RFOR Fitness plug-in. We found that we could whitelist specific Linux memory allocation function and reduce the number of conditions reported by the RFOR Fitness plug-in. Specifically we whitelisted all of the malloc functions detected automatically by GCC using the DECL_IS_MALLOC macro in addition to the kmem_cache_alloc and kmem_cache_free functions.

3) Address of Field Conversion: Our plugin reports when the address of a field of τ is taken. This is a frequent occurrence. As mentioned in Section IV-B3, this is likely not problematic for field reordering. However, our plug-in detects the condition. To aid in analysis, we whitelist the condition where the address of the field is taken but not immediately converted into an integer.

The problematic case for field reordering is when the address of a field is used to calculate the relative position of a sibling field. To do this math, the address would have to be converted into an integer eventually. Again we emphasize that this kind of calculation has not been observed in source code, is difficult to get correct due to compile-time field realignment, and would make the source code difficult to maintain; yet, it remains a possibility. Our plug-in can be configured to report all instances if desired.

TABLE I
RFOR FITNESS REPORT FOR TASK_STRUCT

Test	W/O Whitelists	W/ Whitelists
Assignment To task_struct *	52	1
Assignment From task_struct *	46	2
Address Taken of task_struct Field	214	0
	312	3

C. Performance

We used the Phoronix test suite to measure the system performance both with and without the record randomization. The results are shown in Table II. With a single record randomized, task_struct, randomization has no observable performance impact.

To test the performance impact of our compiler plug-in's on compilation time, we used the "time" command to

[8]In some cases an approximate location is provided.

[9]Many data structure operations were macros. This function was an exception.

TABLE II
PERFORMANCE IMPACT OF RFOR

Phoronix Benchmark	**w/o RFOR**	**w/ RFOR**
Gzip Compression	36.12s	35.97s
Timed ImageMagick Compilation	231.14s	232.46s

TABLE III
TIME TO COMPILE KERNEL

Options	**Avg Time**
w/o Plug-Ins	86.730s
w/ RFOR Plugin	87.9087s
w/ Fitness Plugin	87.2170s
w/ SAOR Plugin	87.8303s

measure compilation time with each plug-in. For each test we compiled a minimal Linux kernel version 2.6.38.8 using the configuration template named "allnoconfig." We ran each compilation test three times and calculated the average. The percentage increase of compilation time was between 0.5% and 1.35%. The average results are shown in Table III[10].

VII. SUMMARY

In conclusion, we have demonstrated that memory layout randomization is an effective defense against kernel rootkits and that these compile-time techniques incur no run-time overhead making them suitable for even low-powered devices. Further, we have demonstrated that our static analysis technique is an effective way to automatically determine the suitability of a record for field order randomization.

REFERENCES

[1] P. Klemperer, "Markets with consumer switching costs," *The Quarterly Journal of Economics*, vol. 102, no. 2, pp. 375–394, 1987.

[2] R. Anderson, "Why information security is hard-an economic perspective," in *Proceedings of the 17th Annual Computer Security Applications Conference*, ser. ACSAC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 358–.

[3] P.-Y. Chen, G. Kataria, and R. Krishnan, "Software diversity for information security," in *Workshop on the Economics of Information Security (WEIS), Harvard University, Cambridge, MA*, 2005.

[4] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. P. Pfleeger, J. S. Quarterman, and B. Schneier, "Cyberinsecurity: The cost of monopoly," *Computer and Communications Industry Association (CCIA)*, 2003.

[5] E. H. Spafford, "Computer viruses as artificial life," *Artif. Life*, vol. 1, no. 3, pp. 249–265, Jan. 1994.

[6] C. Nachenberg, "Computer virus-antivirus coevolution," *Commun. ACM*, vol. 40, no. 1, pp. 46–51, Jan. 1997.

[7] Solar Designer, "Getting around non-executable stack (and fix)," Bugtraq mailing list, August 1997.

[8] S. Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique," 2005.

[9] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561.

[10] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 298–307.

[10]For our tests, task_struct was not nested in a union therefore the related test results are excluded

[11] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, ser. HOTOS '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 67–.

[12] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM conference on Computer and communications security*, ser. CCS '03. New York, NY, USA: ACM, 2003, pp. 272–280.

[13] Y. Zhang, H. Vin, L. Alvisi, W. Lee, and S. K. Dao, "Heterogeneous networking: a new survivability paradigm," in *Proceedings of the 2001 workshop on New security paradigms*, ser. NSPW '01. New York, NY, USA: ACM, 2001, pp. 33–39.

[14] PaX Team, "Address space layout randomization," Open Source Security, Inc., Tech. Rep., 2003.

[15] Z. Lin, R. D. Riley, and D. Xu, "Polymorphing software by randomizing data structure layout," in *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 107–126.

[16] D. E. Knuth, *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[17] T. M. Chilimbi, B. Davidson, and J. R. Larus, "Cache-conscious structure definition," *SIGPLAN Not.*, vol. 34, no. 5, pp. 13–24, May 1999.

[18] R. Hundt, S. Mannarswamy, and D. Chakrabarti, "Practical structure layout optimization and advice," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 233–244.

[19] *GNU Compiler Collection (GCC) Internals*, 4th ed., Free Software Foundation, Inc., 2013.

[20] S. Boyko, "Driver to hide files in linux os," August 2012.

[21] Unknown, "How to: Hijacking the syscall table on latest 2.6.x kernel systems," June 2010.

[22] J. Rhee, Z. Lin, and D. Xu, "Characterizing kernel malware behavior with kernel data access patterns," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: ACM, 2011, pp. 207–216.

[23] E. Levy, "Smashing the stack for fun and profit," *Phrack Magazine*, vol. 49, pp. 14–16, August 1996.