Smokestack: Thwarting DOP Attacks with Runtime Stack Layout Randomization

Misiker Tadesse Aga
University of Michigan
Ann Arbor, MI, USA
misiker@umich.edu

Todd Austin
University of Michigan
Ann Arbor, MI, USA
austin@umich.edu

***Abstract*—Memory corruption vulnerabilities in type-unsafe languages are often exploited to perform a control-flow hijacking attack, in which an attacker uses vulnerabilities to corrupt control data in the program to eventually gain control over the execution of the program. However, widespread adoption of control-flow attack defenses such as Control-flow Integrity (CFI) has led attackers to exploit memory errors to corrupt non-control data that can not be detected by these defenses. Non-control data attacks can be used to corrupt security critical data or leak sensitive information. Moreover, recent attacks such as data-oriented programming (DOP) have generalized non-control data attacks to achieve Turing-complete computation capabilities within the programmer-specified control-flow graph, leaving previously proposed control-flow protections unable to stop these attacks.**

In this paper, we present a stack-layout randomization scheme that can effectively thwart DOP attacks. Our approach, called Smokestack, provides each function invocation with a randomly permuted ordering of the local stack organization. In addition, we utilize true-random value sources combined with disclosure-resistant pseudo-random number generation to ensure that an adversary cannot anticipate a function's invocation permutation of automatic variables. Our evaluation on SPEC benchmarks and various real-world applications shows that Smokestack can stop DOP attacks with minimal overhead.

I. Introduction

Despites decades of security research, memory corruption still poses a great threat to software systems. This is due to the fact that most high performance applications are written with memory unsafe languages like C and C++, which are inherently prone to memory corruption. Memory corruption is typically exploited to deploy control-flow attacks in which the execution flow of a program is manipulated to execute code sequences not anticipated by the programmer, with the ultimate goal of circumventing system security measures.

Due to the prevalence and power of control-flow attacks, various mitigations have been proposed, such as Control flow integrity (CFI) [6], which enforces the runtime execution path of a program to adhere to the statically determined Control-Flow Graph (CFG), and Code Pointer Integrity (CPI) [25], which provides memory safety for code pointers. These techniques have been shown to be effective at confining programs to the programmer-specified control flow graph. However, widespread adoption of control-flow attack protections has led to attacks that corrupt non-control data to perform malicious operations. Non-control data attacks do not violate the constraints imposed by these defenses as they do not violate control-flow of the program, rather they reuse existing control-flow to manipulate program data. Moreover, recent works have shown that Turing-complete computation capabilities can be achieved without leaving the statically determined CFG [10] or without modifying code pointers [22]. Control-flow bending [10] bypasses CFI protections by swapping target addresses of a indirect branches with another valid address from the same branch. Data-Oriented Programming (DOP) [22] enables an attacker to execute sequence of instructions within the legitimate control flow of the program by repeatedly corrupting non-control data. In this paper, we broadly term any attack that provides a programming capability without leaving the programmer-specified CFG as a *data-oriented programming (DOP)* attack.

Address randomization defenses, *e.g.*, address space layout randomization (ASLR), can be used as a first line of defense against DOP attacks. However, information leaks are increasingly being utilized to bypass these defenses, including fine-grained and runtime re-randomization based techniques [33]. In addition, information leaks coupled with attackers' knowledge of the program semantics can successfully bypass state-of-the-art randomization techniques and allow an attacker to launch a successful runtime attack.

Most DOP attacks take advantage of the deterministic nature of the stack layout of programs to grant an attacker the ability to control operands used by sequences of existing code to synthesize an attack payload. Stack layout randomization could be a powerful tool to stop DOP attacks. However, prior stack layout randomization techniques fall short in the presence of memory disclosure by relying on one-time static randomization or coarse-grained random padding. Additionally, it is essential to have a disclosure resistant source of entropy at runtime for randomizing the stack, as it has been shown that a powerful attacker can gain access to the memory variables used to drive pseudo-random number generation [23].

In this paper, we evaluate the effectiveness of previously proposed stack layout randomization techniques at stopping real-world DOP exploits. We show that previous stack-layout protections can be easily overcome by DOP attacks. To address this deficiency, we present Smokestack, a runtime stack-layout randomization technique that randomizes function stack layout at each invocation, using a true random permutation selection that is protected against memory disclosure attacks. Using these

defenses, Smokestack is able to thwart proposed and real-world DOP attacks.

Objectives and Contributions. Our objective is to develop a runtime solution resilient to DOP attacks, *i.e.*, attacks that manipulate program execution but do not leave the programmer-specified CFG. The main contributions of this paper are as follows:

Evaluation of prior stack randomization techniques: We evaluate the effectiveness of prior stack layout randomization schemes at stopping data-oriented attacks. To this end, we developed a real-world DOP attack based on a recently disclosed vulnerability [4] that is able to achieve a Turing-complete computation capability despite the constraints imposed by previous stack-layout randomization schemes.

Smokestack: We present a novel runtime stack layout randomization solution, dubbed Smokestack, which is capable of stopping data-oriented attacks. Smokestack randomizes the stack layout of functions for every function invocation, thereby thwarting attacker attempts to discover stack frame layouts. Smokestack implements true-random selection of stack layout permutations that cannot be anticipated, even by attackers with full control over data memory.

Implementation and detailed evaluation: We implemented Smokestack in the LLVM compiler framework. Our implementation provides a secure random permutation, at function invocation, using an intrusion-resistant pseudo-random number generator (based on the Intel AES-NI instruction set extensions), which is seeded from a true random number source. We present a comprehensive performance evaluation of Smokestack on the SPEC 2006 benchmarks and additional DOP vulnerable versions of real-world applications. In addition, we assessed the effectiveness of Smokestack at stopping data-oriented attacks using synthetic as well as real-world DOP attacks.

The remainder of the paper is organized as follows. Section II presents background on data-oriented attacks and assesses the strength of previous stack layout randomization techniques. Section III presents details of our proposed runtime stack layout randomization scheme. Sections IV details our LLVM-based implementation. Section V presents a performance and security evaluation of our prototype system. Finally, Section VII concludes the paper.

II. Background

Runtime attacks exploit memory corruption vulnerabilities in type-unsafe languages like C and C++ to control vulnerable programs. These vulnerabilities are commonly exploited in a control-flow hijacking attack, where an attacker uses memory errors to corrupt control data, such as a function pointer, return address, or C++ virtual function table, to eventually hijack the execution flow of the program. To mitigate this problem a wide range of control-flow protections have been proposed. This includes enforcement based techniques like CFI [6], CPI [25] as well as randomization techniques like address space layout randomization [2]. Enforcement based mitigations either prevent corruption of control data [25] or stop indirect jumps from leaving the programmer-specified CFG. CFI instruments the program to enforce that all indirect branches target only valid addresses within the enforced control-flow graph.

With wide spread adoption of powerful control-flow protections like CFI [6], the next avenue for an attacker is using a memory error to overwrite non-control data to cause non-control data attacks [13]. These attacks have been shown to have detrimental effects such as the leaking of secret keys (HeartBleed) [16]. In non-control-data attacks, a memory corruption vulnerability is exploited to corrupt non-control data, data that is not directly used for indirect control transfer. These attacks do not leave the valid edges of the CFG, however, the data used for certain sequence of instructions and how it is utilized is controlled by the attacker. Chen et al. [13] demonstrated that non-control-data attacks can be used to overwrite sensitive data used for decision-making and can cause leakage of sensitive data or cause privilege escalation by overwriting variables used in authorization decisions. Recent works such as control-flow bending [10] have relaxed non-control data attacks to include attacks that hijack the execution of the program and yet still adhere to a valid control-flow path in the CFG of the program enforced by the CFI policy. Moreover, Hu et al. [22] demonstrated a more generalized form of non-control data attack, called data-oriented programming (DOP), that achieves Turing-complete computations with rich expressiveness by manipulating only non-control data. These attacks work without creating any new edges in the static control-flow graph of the program and hence are not detected by CFI solutions.

A. Data-Oriented Programming

Data-oriented programming attacks work by corrupting non-control data to execute sequences of instructions within the program with attacker-controlled operands. Each sequence of instructions whose operands are controlled by the attacker, called a DOP gadget, performs a particular operation which contributes towards the overall attack payload. DOP attacks can achieve Turing completeness by chaining DOP gadgets together through controlling a vulnerable loop, called a DOP gadget dispatcher, that is enclosing DOP gadgets and its loop counter is controllable by the attacker.

```
1  func() {
2   int *ctr, *size = 0, *step = 1;
3   char *buff[LEN]; int *req;
4   for(;ctr < MAX; ctr++){
5     get_input(buff, req); //vulnerable function
6     if(*req == 0)
7       *size += *step;
8     else if(*req == 1)
9       *size -= *step;
10    else
11      *step = *req;
12  }
13 }
```

Listing 1: Example DOP attack.

Hu *et al.* [22] demonstrated that DOP attacks can bypass wide spread defenses like ASLR to perform malicious operations on real-world applications. Listing 1 shows a simple

program vulnerable to DOP attacks. If there is a buffer overflow vulnerability in the input accepting function (`get_input()`), where an attacker can overflow the fixed-sized buffer `buff`, it could lead to an attacker able to control local variables `size`, `step`, `ctr` and `req`. This grants an attacker the ability to perform addition, subtraction and copy operations on any memory value, in any order desired by the attacker.

Robust enforcement techniques such as Softbound [27] and DFI [11] can protect against non-control data attacks. But their high overhead makes them unsuitable for production systems. On the other hand, relatively light weight enforcement techniques have been proposed to mitigate non-control data attacks, which typically rely on protecting only sensitive data, *e.g.*, kernel data [12] and programmer-annotated critical data [31]. However, these mitigations are inherently ineffective against generalized DOP attacks.

Randomization based protections on the other hand typically do not stop the vulnerability rather they make the vulnerability difficult to exploit by making certain assets used for the attack unpredictable. The degree with which they make the vulnerability difficult to exploit depends on the amount of entropy introduced by the protection scheme. Most proposed randomization based defenses randomize the location of code at varying levels of granularity including at the function level [7] [24], basic block level [35], and instruction level [21]. These defenses are ineffective in protecting against DOP attacks, as they do not depend on the location of code.

B. Prior Stack Randomization Efforts

Given the strong reliance that DOP attacks have on manipulating stack variables, it suggests that previously proposed stack layout randomization efforts may provide a basis to stop DOP attacks. Prior works in stack randomization perform one or more of the following transformations:

Stack base address randomization. This transformation randomizes the base address of the stack by allocating random-sized padding on the stack at the beginning of the program to make the absolute address of stack allocations unpredictable during the runtime of the program [19][2][18].

Random padding at function entry. This transformation adds a random padding at the beginning of functions to randomize the relative alignment between stack frames. Forest et al. [18] proposed adding a random padding before stack frames of functions with buffer variables at compile time. They use the size of the stack frame (greater than 16 bytes) to identify functions containing buffer variables. For every stack frame allocation greater than 16 bytes, it adds one of the 8 possible paddings (8, 16, ..., 64 bytes) randomly.

Static stack layout randomization. This technique permutes stack allocations in a function at compile time to randomize the relative distances between objects in a stack frame [19].

The main weakness of prior stack layout randomization schemes is that they focus only on protecting corruption of code pointers in the stack, which requires knowledge of the absolute distance between the vulnerable buffer and code pointer of interest. However, DOP attacks only require relative distance between the vulnerable buffer and the variable of interest, a local variable used in a DOP gadget and a DOP gadget dispatcher, which is sufficient for a successful attack. Consequently to assess the effectiveness of prior stack randomization efforts in stopping DOP attacks, we developed a proof-of-concept DOP exploit for a recently disclosed vulnerability in *librelp* logging library (CVE-2018-1000140) [4], which was fixed in subsequent versions of *librelp*. In the following section we present the details of the attack.

C. Bypassing Previous Stack Randomization Efforts

The vulnerability in *librelp* is caused by improper use of *snprintf()*. The C library function *snprintf()* writes a null terminated series of characters and values to a non-zero sized buffered and returns the number of bytes that would be written assuming there was sufficient space, excluding the terminating null byte. If *snprintf()* is used to process untrusted input in a loop assuming it returns the number of bytes actually written, it might lead to a buffer overflow vulnerability. For example, if an attacker manages to control the size of the string to be written on the boundary of the buffer, successive iterations of the loop will grant the attacker a non-linear overflow of the buffer which is able to bypass protections such as stack canaries. Listing 2 shows the vulnerable code in *librelp*.

relpTcpChkPeerName() checks valid Subject alternative names (SANs) within a X.509 certificate for a peer name until it finds a match. While doing so, it copies all SANs checked so far to a buffer for error reporting. Our proof-of-concept attack exploits the stack-based buffer overflow in the *relpTcpChkPeerName()* function, shown in Listing 2, to construct a DOP gadget dispatcher and series of DOP gadgets by repeatedly corrupting local variables of functions up in the call hierarchy used for controlling a loop and performing operations in the socket initializing function —*relpTcpLstnInit()*. Using static analysis, we discovered gadgets for *MOV, DEREFERENCE* and *STORE* operations. Moreover, we were able to de-randomize statically randomized stack layout and random stack padding by taking advantage of the semantics of the underlying program.

To exploit the vulnerability we supply a maliciously crafted X.509 certificate containing more than 32KB of "subject alt names" to a GnuTLS enabled RELP logging service. By manipulating the size of the string for "subject alt name" on the 32KB boundary, we were able to vary the gap precisely enough to control which part of the stack to overwrite. This control is essential to the attack as it enables the attack to avoid unintended corruption of adjacent stack resident data which might lead to a crash.

Bypassing static stack layout randomizations: To bypass static stack layout randomization schemes, an attacker can use a memory disclosure to perform a read attack and infer the layout of the stack by analyzing its contents. In our proof-of-concept exploit we were able to de-randomize static permutation of allocation within a stack frame using either information leak and semantics of the program or brute-force attacks, as the permutation is done at compile time and is the same for every run of the program.

```
relpTcpChkPeerName(..., gnutls_x509_crt_t cert){
  ...
  char szAltName[1024];
  char allNames[32*1024]; /* for error reporting*/
  bFoundPositiveMatch = 0;
  iAllNames = 0;

  iAltName = 0;
  while(!bFoundPositiveMatch) {
    szAltNameLen = sizeof(szAltName);
    gnuRet = gnutls_x509_crt_get_subject_alt_name(
          cert, iAltName, szAltName,
          &szAltNameLen, NULL);
    if(gnuRet < 0)
      break;
    else if(gnuRet == GNUTLS_SAN_DNSNAME) {
    ...
      /* stack based buffer-overflow */
      iAllNames += snprintf(
            allNames+iAllNames, sizeof(allNames)-
            iAllNames, "DNSname: %s; ", szAltName);
      relpTcpChkOnePeerName(pThis, szAltName,
                  &bFoundPositiveMatch);
    }
    ++iAltName;
  }
  ...

done:
  return r;
}
```

Listing 2: Vulnerable function in *librelp* logging library.

Then we use the *snprintf* vulnerability in the vulnerable function to repeatedly overwrite the local variables used for the DOP gadgets and gadget dispatchers to perform our DOP attack. Using this approach, we were able to bypass compile time permutations as well as padding-based stack-layout randomizations.

III. Smokestack Runtime Stack Layout Randomization

Memory errors are typically utilized for control-flow attacks that require corrupting control data to hijack the control-flow of the program, when the corrupted control-data is used for control-flow transfer. Randomizing the data layout can mitigate these attacks as they require the absolute address of the data to corrupt. Prior works achieve this goal by adding a random-sized padding in the beginning of the stack frame which adds uncertainty to the address of stack resident objects. These randomization schemes only consider control-flow attacks and hence rely on the assumption that obfuscating the absolute address of stack resident data is sufficient to stop runtime attacks. Even though they are shown to mitigate control-flow attacks, purely data-oriented attacks are not stopped by these protections. Non-control data attacks such as DOP utilize the relative distance between variables in order to use memory corruption vulnerabilities to control local variables, which is kept intact with these approaches. These attacks require a precise control over data, such as stack resident local variables, used as operand in DOP gadgets and loop counters used to stitch DOP gadgets in a particular order to realize a malicious computation. Section II-C shows that an attacker can use a DOP attack to undermine static stack layout randomization and random padding schemes.

A. Design Objectives

The main objective of this paper is to provide a practical mitigation technique to stop stack-based non-control data attacks. To achieve this goal, our solution has to meet the following requirements:

- Provide a runtime stack randomization scheme resilient to memory disclosure that effectively mitigates DOP attacks. This is a key requirement as our threat model assumes a powerful attacker who can perform active probing to reverse engineer randomized allocations.
- Have low performance and memory overhead on both CPU-bound and I/O-bound applications.
- Be compatible with legacy code. This requirement includes source and binary as well as modular support to enable gradual migration of code.

B. Threat Model

In this paper, we assume a powerful attacker who is able to gain read/write access to all writable data memory. However, we assume the attacker is unable to write to non-writable data/code sections and registers used by our instrumentation code. In all, we consider a strong adversary capable of:

- Bypassing deployed protections such as ASLR using memory disclosure vulnerabilities in the program or microarchitectural features such as branch prediction [17] and get full read access to all code pages mapped in the address space of the program.
- Exploiting memory vulnerabilities and using the semantics of the underlying program to reverse engineer a randomized stack layout of a function based on a disclosed stack frame data that allows the adversary to instantiate a runtime attack on future calls of the same function. This could be a program where an attacker can get a hold of the source code or the binary to perform static analysis.
- Performing a brute-force attack with a finite number of attempts before being detected by the system. This assumes a service that restarts after a crash.

C. Overview of Smokestack

The primary way to perform non-control-data attacks is to identify local stack variables or register variables spilled from the caller that are saved on the stack at the entry of a function and restored before it returns through static analysis of the binary or runtime memory disclosure. Then, successive steps of the attack exploit memory corruption vulnerabilities to control the identified local variables to execute the attack payload using the instructions in the vulnerable program. Thus, our protection needs to ensure the absolute address of the variables as well as the relative distance between them changes with every function invocation, making the information gathered with prior probings of the program futile. Smokestack achieves these requirements by dynamically deciding, at the function

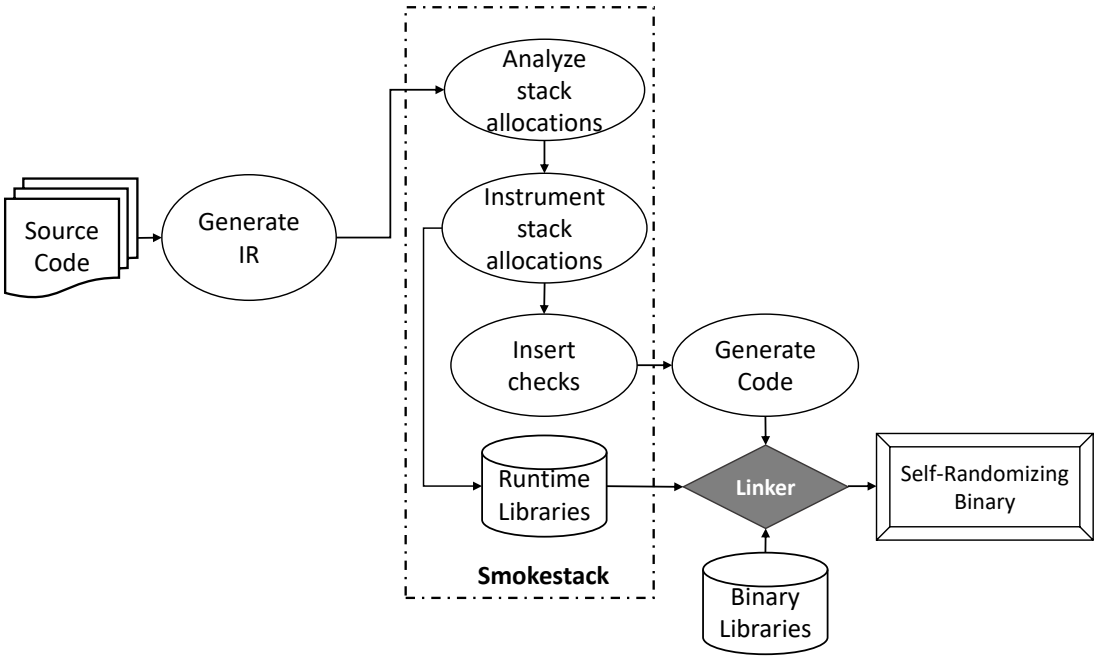

Fig. 1: System Overview: Overview of our runtime stack layout randomization scheme with the components of Smokestack highlighted.

prologue, the ordering, relative distance and alignment between all stack resident objects.

Smokestack performs allocation of stack frames with live re-randomization while retaining all the desirable features of stack allocation such as automatic deallocation of stack objects for all possible control-flow paths. It achieves this by replacing each stack allocation in a function with a slice into the total stack frame allocation, where its index within the total allocation is decided dynamically at the function entry based on true-random perturbation of the local variables.

Figure 1 shows the overview of Smokestack infrastructure. To avoid the performance overhead associated with computing a permutation at runtime Smokestack embeds a read-only permutation box, P-BOX, that contains all the possible permutations of all unique stack frames in the program in a shared library that gets dynamically linked with Smokestack-hardened programs. Section III-E presents optimizations we employed to reduce the associated performance and memory overheads. The details of the Smokestack compilation and runtime follow.

D. Discovering Stack Allocations

In this phase, we identify the stack frame allocations for all functions in the program. This includes for each function gathering the type and alignment requirement of its resident objects. We then use this meta data to generate possible permutation of its allocations and the total allocation considering the alignment requirements of all objects for all possible permutations. This step requires adding padding to fulfill the alignment requirements of allocations for every possible permutation, which also adds extra source of entropy to our randomization scheme.

Generating Random Permutation: In this stage, we generate all the possible permutations of stack allocations for each function in the program. To do this, we represent each stack allocation as an index from the start of the stack frame, allocation size and alignment requirement. We then generate all possible permutations, in lexical order, for all the allocations

Algorithm 1 Permutation Generator: This algorithm generates all the possible permutations of stack allocations within a function

```
1:  procedure ALIGN(ind, alignment)
2:      if ind % Alloca.alignment == 0 then
3:          return ind
4:      else
5:          return (ind / alignment + 1) * alignment
6:  procedure PERMUTE(F)
7:      P_Table ← ∅
8:      N ← Count(F.Alloctions)
9:      for p_index in 0 to N! do
10:         temp ← p_index
11:         ind ← 0
12:         Alloca ←F.Allocations
13:         Indexes[N] = {0}
14:         for a_index in 0 to N do
15:             curr_fact ← (N − a_index)!
16:             e ← temp/curr_fact
17:             temp ← temp % curr_fact
18:             ind ← ALIGN(ind, Alloca[e].alignment)
19:             Indexes[e] ← ind
20:             ind ← ind + sizeof(Alloca[e])
21:             Alloca.pop(e)
22:         P_Table.append(Indexes)
23:     return P_Table
```

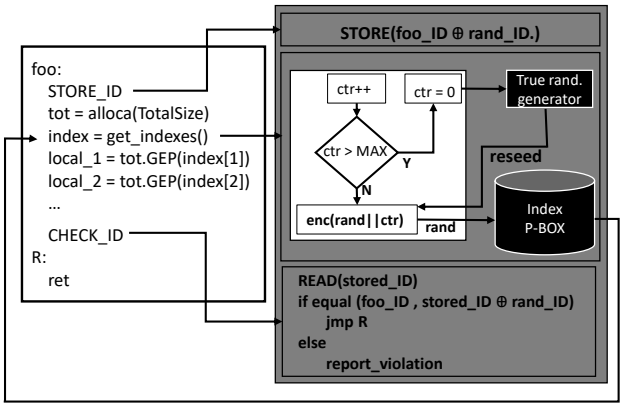

Fig. 2: Function call and return in a Smokestack: This figure shows an overview of the stack layout randomization steps involved in Smokestack function calls and returns, for the AES based random number generation schemes.

within the function. Algorithm 1 shows the pseudo-code of our permutation engine. The permutation engine takes all the stack allocation in a function as an input. Each iteration of the outer loop of PERMUTE procedure generates the p^{th}_{index} lexical order permutation of all the allocations within the stack frame of the function. After computing all the iterations, we will have a table, where the i^{th} row in the table is a set of indexes for the i^{th} lexical order permutation of allocations in the function. We then permute the rows in a table to avoid the lexical correlation between any two consecutive rows in a table. Finally, we store the permutation table of the function, which holds the indexes of each allocation for every possible order of allocations, in the P-BOX. A P-BOX is shared among all functions with the same stack format. In section III-E, we show how we reduce the associated memory overhead by sharing permutation tables between different functions. Tables in the P-BOX are indexed by a random number generated at function invocation, to get the indexes of the local variables from the base of the stack frame for that particular permutation.

1) Runtime Allocation of Randomly Permuted Stack Frames: This phase instruments the program in order to randomize the stack layout of each function call by randomizing the order and alignment of all of local variables. This is achieved by having a single stack frame allocation with a size equal to the total allocation and replacing all stack variable allocations in the stack frame with a slice in to the total stack frame allocation of the function. Fig 2 shows the instrumentation introduced by Smokestack. Upon a function invocation, a random permutation of the local variables is chosen using a random number to index the table associated with the function in the P-BOX to get a row of indexes. Then, allocations in the stack frame are assigned to their respective slices within the total allocation based on their respective index in the randomly chosen row of indexes. This is represented by LLVM's GetElementPtr (GEP) in the figure. This will ensure that the absolute address and the relative distance to a stack resident object, which can be used in a DOP gadget, is unpredictable for each invocation of a function.

Random Number Generation: We considered various random number generation schemes at the beginning of each function to choose a random permutation of local variables. We considered any form of pseudo-random number generation where the algorithm's state is in memory as unsafe, since a powerful attacker assumed by our threat model could certainly read and manipulate the state of a memory-based pseudo-random generator.

- *Generating a true random number at the entry of each function.* For this scheme, we considered the cryptographic random number generator on UNIX-like systems(*i.e.*, `/dev/random`) and `rdrand`, the on-chip hardware random number generator on Intel processors. As `/dev/random` stalls when the system's internal entropy pool is exhausted, we tested only `rdrand` on our prototype implementation.
- *Generating a cryptographically secure pseudo-random number.* For this scheme, we use AES counter mode encryption to generate a secure random number. We use a true random number generator to generate an encryption key and a nonce that are updated when a counter reaches a certain maximum value to guarantee the randomness. This is done by using a universal call counter to count the number of function calls before generating a new random number. At the entry point of a function we generate a pseudo random number using AES counter mode encryption using the last generated random number as an initial value and the call counter as a counter. We used the Intel's AES-NI extensions [20] to accelerate our random number generation. On our prototype implementation, we tested the approach by varying the number of rounds of the AES encryption to see the trade-off between security and performance.

Our instrumentation defers randomization of allocations whose size cannot be determined at compile to runtime by adding a random sized padding on top of the static total allocation. Variable length arrays (VLA), which are supported in the C99 standard, are one such example. We randomize the layout of stack frames with VLAs at runtime by adding a random sized dummy `alloca` before each VLA in a stack frame. This guarantees that both the absolute address of the VLA and its relative distance from other stack resident objects is randomized.

2) Protecting Smokestack Defenses: The final pass adds checks to detect attacks that bypass our instrumentations, for example, by using control-flow attacks to jump in to the middle of a function. To achieve this, the instrumentation phase adds a unique load-time identifier for each function which is XOR'ed with a random key at the prologue of the function. And, at the epilogue it is XOR'ed with the same random key and checked against the function identifier. These checks, together with the stack runtime stack layout randomization, can be a second line of defense for control-flow attacks.

E. Smokestack Performance and Memory Optimizations

To reduce the performance overhead and the memory footprint of our instrumentations, we applied the following optimizations to Smokestack:

- *P-BOX size of power of 2.* This optimization rounds up the size of P-BOX from *n!* to the nearest power of 2. This is achieved by wrapping around indexes *n!* to the nearest *power-of-2*. This optimization allows the replacement of a modulo operation in our instrumentation with a left shift operation. This has a significant impact in reducing the performance overhead of the instrumentation added to get random permutation indexes at the prologue of the program.
- *Rearranging Stack Allocations* This optimization rearranges stack allocations of functions to ensure that all tables in the P-BOX are associated with a unique combination of allocations. And, functions with the same combination of allocations use the same P-BOX entry table. For example, function `f1` with local variables `int, double` can share a P-BOX table entry with function `f2` with local variables `double, int`. This optimization reduces the memory footprint of the P-BOX and doesn't have any negative impact on correctness of the program as the actual order of the variables in the resulting binary is determined by subsequent phases of the compilation.
- *Rounding up Allocations.* This optimization reduces the memory usage of P-BOXs by sharing a table for functions having stack frames that differ only by one primitive allocation. For example, functions `f1(double, double, int)` and `f2(double, double)` share a P-BOX table at the expense of extra padding in the stack frame of `f2`.This optimization takes advantage of the fact that the least significant indexes within a permutation entry of a smaller table is same as the permutation of a bigger size in lexical order of permutation. This optimization also improves performance for frequently called functions.

IV. Implementation

We implemented Smokestack on top of the LLVM 3.9 compilation framework [26], modifying the LLVM libraries and the compiler-rt runtime. Our analysis and instrumentation passes operate on LLVM intermediate representation (IR), which is generated from source files using the LLVM `clang` front-end.

A. Analysis Passes

We implemented P-BOX generation in several LLVM passes. The first function pass gathers all stack allocation for all functions that have an on-stack memory object. Then, a module pass uses the meta-data generated by the function pass to generate a P-BOX table for each function, considering the alignment requirements and the optimization discussed in Section III-E. The final analysis pass generates the P-BOX for all the unique P-BOX entry tables.

Alignment requirements. For primitive types, the alignment requirement can be found as part of the IR instruction. For aggregate and user defined types, we have to consider both element alignment requirements and aggregate alignment requirements. An element could be a primitive type whose alignment requirement can be found easily or an aggregate type, which makes the process recursive. The alignment requirements of aggregate types, on the other hand, depends on the alignment requirement of the largest element in the aggregate type.

B. Instrumentation Passes

The instrumentation pass inserts an allocation with the size of the total allocation at the beginning of the function and inserts a call to a random number generator `inline` library function for all functions that have one or more than automatic variables. Then, it replaces all the `alloca` instructions in the function with `getelementptr` instructions whose indexes are decided by the set of indexes selected from the P-BOX entry table of the function using the generated random number. The final instrumentation pass inserts checks to detect attacks that bypass the stack allocation instrumentation. P-BOX tables are implemented as a read-only data in a runtime library and linked to the program, to enable a cache friendly approach.

V. Evaluation

This section presents the detailed performance and security evaluation of Smokestack. We ran our experiments on an Intel Xeon D-1541 processor running Ubuntu 16.04 Linux with 32GB of memory.

A. Performance Evaluation

We evaluated the performance overhead of Smokestack using SPEC 2006 benchmarks and other I/O bound real-world applications, *e.g.*, ProFTPD, Wireshark. The baseline was compiled with Clang with the default SPEC settings (-O2). The Smokestack hardened version was compiled the same way, except for the additional instrumentation passes and replacing the default stack smashing protection with our function identifier instrumentation, which has better security guarantees. We ran four experiments, which varied how random number generation was implemented. *pseudo* utilizes a memory-based pseudo-random number generator. This experiment is only included as a performance baseline, as it is considered completely unsafe by our threat model (since the attacker can anticipate the state of the generator at any time). The *AES-1* and *AES-10* experiments use the Intel AES-NI instructions to encrypt a true-random seed, with the former experiment only running one AES round and the latter running 10 rounds which conforms to the AES standard [30]. Finally, *RDRAND* uses the Intel RDRAND instruction to get a true-random number for use by the stack layout permutation code.

Figure 3 shows the performance overhead of Smokestack for the SPEC 2006 benchmarks and I/O bound applications. Our measurements show that the performance is dependent on the way we generate a random number. For unsafe pseudo-random number generation, the normalized performance varies

from a speedup of 2.6% to a slowdown of 7.2%, averaging to 0.9% slowdown over the SPEC2006 benchmarks. Using a cryptographically secure pseudo random generation (AES-128 10 rounds), the overhead spans from 0.6% up to 29%, and averaging 10.3%. To assess the overhead vs. security trade-off, we also examined the performance of a less secure pseudo-random number generation (AES-128 1 round), which has an average slowdown of 3.3%. For RDRAND-based true-random number generation, there was greater slowdown due to the bandwidth limitations of the true random number generator. This experiment experienced an overall average slowdown of nearly 22%.

To examine the source of performance gain on some benchmarks, we ran the Oprofile [5] tool with our SPEC 2006 experiments which clearly showed the variation on the *RESOURCE_STALLS* parameter depending on the benchmarks. Our analysis shows that the speedups are due to instruction scheduling and register pressure caused by the Smokestack instrumentations. On some benchmarks, Smokestack increases register pressure and consumes load delay slots during the CPU scheduling. The register pressure improved performance on benchmarks where registers are underutilized, and degrades performance if registers were already fully utilized. Our evaluation also shows that, call depth has moderate impact on the overall performance with maximum depth value of 394 (for perlbench). The stackframe size, however, showed a significant impact on performance. This is reflected in the relatively large performance hits reported on certain benchmarks, for example 445.gobmk has a maximum stack frame size of 85 KB.

On I/O bound applications we used for our performance evaluation, ProFTPD and Wireshark, Smokestack incurs negligible overhead, with the worst case performance overhead of 6%.

B. Memory Overhead

We evaluated the memory overhead of Smokestack by measuring the maximum resident set size (`ru_maxrss`) while running SPEC 2006 benchmarks. Figure 4 shows the results of these experiments. It's interesting to note that benchmarks with higher memory overhead, like *perlbench* and *h264ref*, have relatively lesser performance overheads. This is due to the fact that the source of the memory overhead is the addition of the index P_BOX in the read-only data section which doesn't strongly impact the I-cache miss rate.

C. Security Analysis

In this section we assess effectiveness of smokestack in protecting against DOP attacks. To this end, we first analyze the security vs. throughput of the sources of randomness we used for our prototype implementation, we then evaluate Smokestack's effectiveness in protecting DOP attacks in both synthetic benchmarks and real-world applications.

Source of randomness: We performed tests to examine the rate at which we can generate random numbers. Table I shows the rate at which we can generate random numbers, using random generation schemes with varying security guarantees, back-to-back on our test machine. While *pseudo* is fastest, it also offers no protection against a powerful attacker assumed by our threat model. *RDRAND*, in contrast, provides true random values for each invocation, but comes with a great delay. The *AES* based pseudo random number generators, on the other hand, provide a convenient trade-off between security and performance, with overall quite good performance.

TABLE I: SOURCE OF RANDOMNESS: SHOWS THE RATE AT WHICH RANDOM VALUES CAN BE GENERATED BY THE RANDOM GENERATOR SCHEMES WE TESTED FOR OUR PROTOTYPE IMPLEMENTATION.

source	Security	Rate (cycles/Invocation)
`pseudo`	None	3.4
`AES-1`	Low	19.2
`AES-10`	High	92.8
`RDRAND`	High	265.6

Penetration testing with synthetic benchmarks: We developed two types of DOP attacks that exploit buffer overflow vulnerabilities to control local variables used as DOP gadgets and a loop counter used as gadget dispatchers. The first set of attacks use a stack based buffer overflow vulnerability to corrupt variables in the stack to perform the attack. And, the second set of attacks overflow a buffer in the data segment or heap to overwrite local variables in the stack. We also considered two types of overflows, direct and indirect (overflows a buffer until a pointer is corrupted, and then use an assignment through the corrupted pointer to overwrite the target pointer) —an approach followed by the RIPE [36] control-flow attack benchmark suite.

Smokestack is able to prevent all the attacks by breaking the DOP gadgets and gadget dispatchers. All of the direct overflow attacks based on any buffer were stopped. Also, any indirect overflow attacks based on buffers in the data segment or heap corrupted unintended locations in the stack. All of the indirect overflows attacks failed on the first step, as they overwrote a different address than the intended pointer used to write to the target pointer.

Real Vulnerabilities: In our final set of security analyses, we tested Smokestack's ability to protect against attacks that exploit real vulnerabilities including our own proof-of-concept DOP attack on *librelp* logging library. The following reported DOP attacks were considered for our analysis:

The **Wireshark** network protocol analyzer prior to version 1.8 had a stack-based buffer overflow vulnerability (CVE-2014-2299 [3]) in mpeg reading function *cf_read_frame_r()*. This vulnerable function is called from *packet_list_dissect_and_cache _record ()* to copy user specified mpeg frame data to a fixed sized buffer *pd*. Hu et'al. [22] exploited this vulnerability by sending a maliciously crafted trace file that contains a frame larger than the buffer size (0xffff). Their DOP exploit overflows the buffer to overwrite variables *col, cinfo*, and parameter *packet_list* in the same function, *i.e. packet_list_dissect_and_cache_record ()*, and the loop condition *cell_list* in its caller function, *gtk_tree_view_column_cell_set _cell_data()*, with malicious input. *col, cinfo and packet_list* are used as DOP gadget operands and *packet_list* used for stitching together gadgets in

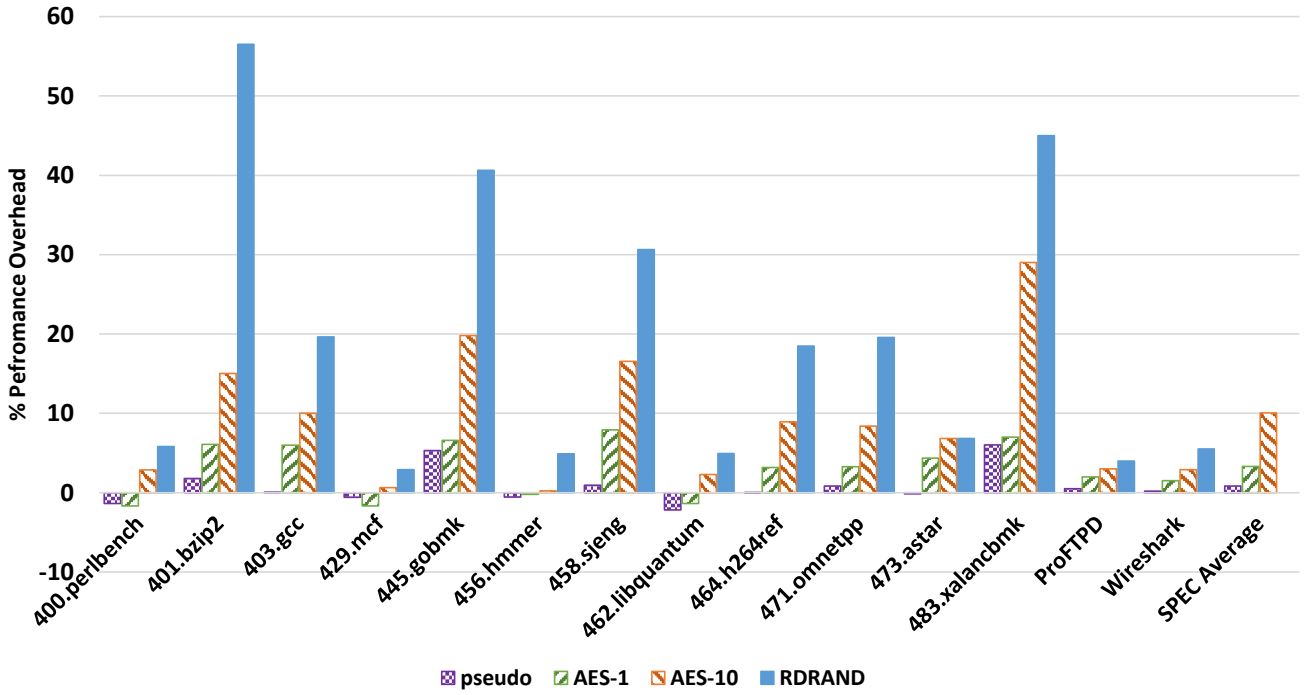

Fig. 3: Percentage performance overhead of Smokestack: This figure shows the percentage runtime overhead of Smokestack on SPEC2006 benchmarks and other I/O bound applications. The experiments varied the way random numbers are generated. `RDARAND` shows the use of `rdrand`, the on-chip random number generator on Intel processors. The others show use of cryptographically secure pseudo-random number generation schemes, AES-128 counter mode with 10 rounds (`AES-10`) and a less secure variant with 1 round (`AES-1`). Finally, `pseudo` shows the overhead of using an insecure memory-based pseudo-random number generator.

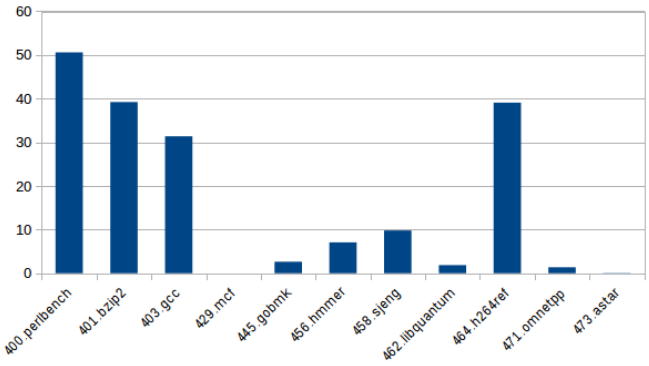

Fig. 4: Percentage memory overhead of Smokestack: This figure shows the percentage increase in maximum resident set size of Smokestack on SPEC2006 benchmarks.

the subsequent calls to the function *packet_list_change_record()* which contains all the DOP gadgets. We run this attack on a Smokestack-hardened version of the vulnerable Wireshark program. Smokestack stopped this attack by detecting the violations when the overflow corrupted unintended data like *Smokestack function identifier* as Smokestack changes the index of *pd* in the stack frame for every call of the function *packet_list_dissect_and_cache _record ()*.

ProFTPD has a stack-based buffer overflow vulnerability (CVE-2006-5815) due to the use of *sstrncpy(dst,src,negative argument)* in the *sreplace()* function [1]. Hu et al. has demonstrated DOP attacks[1] that extract private keys bypassing ASLR, simulate remotely controlled network bot and alter memory permissions exploiting this vulnerability.

Extracting private keys bypassing ASLR: ProFTPD stores its OpenSSL private key in a buffer which has a chain of 8 pointers pointing to it, with only the base pointer not randomized. A successful attack to extract the private key requires using memory disclosure to de-randomize these 7 global pointers. Hu *et al.* used a DOP attack composed of 24 DOP gadget chains (which requires corrupting operands of virtual operations consisting *MOV, ADD* and *LOAD* for 24 iterations) to successfully extract the OpenSSL private key bypassing the underlying ASLR. This attack was also demonstrated to bypass TASR [8], which does fine-grained re-randomization of code on every output system call.

The other two DOP attacks on ProtFTPD use *sreplace()* to corrupt relocation metadata in the *link_map* structure. This is then used by *dlopen()*, which is invoked in ProFTPD's PAM module to dynamically load libraries, to process the corrupted relocation metadata. The two end-to-end exploits used this to simulates a bot that repeatedly responds to network commands

[1]https://huhong-nus.github.io/advanced-DOP/

and alter memory permissions to bypass defenses including *w∧x*, *.rodata* and CFI defenses.

All these attacks repeatedly use a memory corruption vulnerability in *sreplace()* to chain together virtual instructions used in the DOP attack by repeatedly overwriting a loop counter, which is used as a DOP gadget dispatcher. We were able to detect all the attacks on the Smokestack-hardened version of the affected version of ProFTPD. Smokestack was able to stop this attack by randomizing the relative distance of the overflowed buffer with the loop counter used to stitch the DOP gadgets together and the operands used in the DOP gadgets, hence breaking the gadgets as well as the gadget chain.

VI. Related Work

Memory corruption attacks are the leading threat to system security. Several mitigation techniques have been proposed to protect against this threat. These protections can generally be categorized in to two major classes. The first class is enforcement-based protections that perform explicit checks based on predefined policies. These techniques vary from protections which guarantee full memory safety such as Softbound [27] + CETS [28] to protections that target particular type of exploit such as CFI [6], that protects against control flow hijacking exploits.

The other class is randomization-based protections, where critical assets used by an attacker for a runtime attack are randomized after they are acquired and before they are used in an exploit. Randomization-based solutions are more practical as they incur relatively lower overhead than enforcement-based solutions. Address space layout randomization (ASLR) [2] is a typical example, which is a widely deployed randomization-based techniques. It randomizes the base of sections of a program such as code, stack, heap and shared libraries in its address space at load time of the program. However, ASLR has been shown to be ineffective in the presence a single memory leak [34] or brute-force attacks [32]. Successive improvements to randomization-based techniques were proposed to increase entropy by decreasing the granularity of the randomization of the code section to function level [24], basic-block level [35], and instruction level [21]. However, subsequent works [33][15] have shown that compile-time and load-time fine-grained randomizations can be bypassed by runtime attacks that dynamically generate their payloads. Recently, periodic re-randomization [37] has been shown to be effective in stopping these attacks. But it has only been validated for code pointer protection as it protects against control-flow attacks that are resilient to static randomizations.

With the ultimate goal of taking control of the program, control-flow hijacking is usually the easiest and the primary way of exploiting memory corruption vulnerabilities. To address these attacks, a wave of mitigation techniques has been proposed. The leading approach being control flow integrity (CFI) [6], which is based on constructing the program's CFG prior to its execution and validating at runtime whether the execution path follows a valid edge in the CFG. With the advent of low overhead CFI techniques to protect corruption of control data, non-control data attacks has received significant attention by attackers.

Several works have been proposed to mitigate attacks based on non-control-data including enforcement, randomization and language-based approaches.

Data-Flow Integrity (DFI) [11] statically performs reaching definition analysis of instructions. DFI then instruments read access instructions to ensure that the last instruction that wrote to the location is within the reaching definition set of the instruction. Even though DFI is capable of mitigating DOP attacks, a complete DFI protection incurs a very high overhead (50 - 100% on SPEC 2000 benchmarks).

PointGuard [14] proposed encrypting all pointers when they are stored in memory and decrypting them just before they are loaded into registers. However, it uses a single key to XOR all pointers, hence a single leak of a known encrypted pointer from memory can be used to recover the key. Data Space Randomization (DSR) [9] tries to solve the shortcomings of PointGuard by using a different key for all variables. However, even that has been shown to be ineffective in face of memory leaks [34].

Giuffrida et al.[19] presented an ASLR technique that performs live re-randomization of program modules periodically. Unfortunately, their re-randomization technique induces significant runtime overhead when the re-randomization period is small. In addition, their proposed technique is tailored to microkernels, relying on hardware-isolation and runtime error recovery. Moreover, their stack randomization a static randomization and randomized padding which is not effective in protecting against DOP attacks.

YARRA [31] provides a C language extension for validating sensitive pointers pointing to critical data, such as secret keys, in the program as annotated by the programmer. It does this by using page protection to lock its protected data when running unsafe procedures. YARRA offers a security guarantee for non-control data attacks against only the annotated data. However, it incurs a significant overhead for protecting the whole program (*e.g.*, 6x overhead on gzip).

HardScope [29] ensures an intra-program memory compartmentalization by enforcing compile-time discovered variable scope constraints at run-time. HardScope instruments memory accesses at compile-time to check that the memory address requested is within the allowed memory areas. HardScope was demonstrated by extending the RISC-V instruction set with six new instructions. Its hardware support gives Hardscope low overheads. Even though Hardscope reduces the number of available DOP gadgets, it is still susceptible to DOP attacks that share access to the same global data structures or have a data flow reaching a global data structure.

VII. Conclusion

With widespread adoption of control-flow hijacking attack mitigations, non-control data attacks are becoming an increasingly popular source of attacks against systems. While randomization techniques in the code section of a program are gaining popularity due their efficiency, they are ineffective

in stopping DOP attacks. We also found previously proposed stack layout randomization efforts are not robust enough to stop DOP attacks. Smokestack gains additional power at stopping DOP attacks by randomizing stack frames for functions each time they are called. In addition, we leverage true-random stack layout permutation that is resistant to memory disclosure attacks, forcing the attacker to reverse engineer a function frame and deliver a payload in the same invocation. Our proof-of-concept implementation in the LLVM framework demonstrates that the approach can effectively stop DOP attacks with only minimal slowdown in program execution.

ACKNOWLEDGEMENT

This work was supported by DARPA under Contract HR0011-18-C-0019. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA

REFERENCES

[1] Stack-based buffer overflow in the sreplace function in ProFTPD 1.3.0, 2006. Accessed: 2018-03-14.

[2] Windows ISV software security defenses, 2010. Accessed: 2018-02-29.

[3] Stack-based buffer overflow in the MPEG parser in wireshark, 2015. Accessed: 2018-03-14.

[4] CVE-2018-1000140, 2018. Accessed: 2018-05-14.

[5] OPROFILE, 2018. Accessed: 2018-05-10.

[6] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[7] Sandeep Bhatkar, Daniel C DuVarney, and R Sekar. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, pages 17–17, 2005.

[8] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 268–279. ACM, 2015.

[9] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro. Data randomization. Technical report, Technical Report MSR-TR-2008-120, Microsoft Research, 2008.

[10] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, pages 161–176, 2015.

[11] Miguel Castro, Manuel Costa, and Timothy L. Harris. Securing software by enforcing data-flow integrity. In *OSDI*, 2006.

[12] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58. ACM, 2009.

[13] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 5, 2005.

[14] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard TM: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.

[15] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (Just-In-Time) return-oriented programming. In *NDSS*, 2015.

[16] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of HeartBleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.

[17] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 40. IEEE Press, 2016.

[18] Stephanie Forrest, Anil Somayaji, and David H Ackley. Building diverse computer systems. In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pages 67–72. IEEE, 1997.

[19] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, pages 475–490, 2012.

[20] Shay Gueron. Intel advanced encryption standard (AES) instruction set white paper, rev, 2010.

[21] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. ILR: Where'd my gadgets go? In *2012 IEEE Symposium on Security and Privacy*, pages 571–585. IEEE, 2012.

[22] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.

[23] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators. In *International Workshop on Fast Software Encryption*, pages 168–188. Springer, 1998.

[24] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 339–348. IEEE, 2006.

[25] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *OSDI*, volume 14, page 00000, 2014.

[26] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[27] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Notices*, 44(6):245–258, 2009.

[28] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: compiler enforced temporal safety for c. In *ACM Sigplan Notices*, volume 45, pages 31–40. ACM, 2010.

[29] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N Asokan, and Ahmad-Reza Sadeghi. Hardscope: Thwarting dop with hardware-assisted run-time scope enforcement. *arXiv preprint arXiv:1705.10295*, 2017.

[30] Vincent Rijmen and Joan Daemen. Advanced encryption standard. *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology*, pages 19–22, 2001.

[31] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Benjamin Zorn. Modular protections against non-control data attacks. *Journal of Computer Security*, 22(5):699–742, 2014.

[32] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[33] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[34] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, pages 1–8. ACM, 2009.

[35] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 157–168. ACM, 2012.

[36] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 41–50. ACM, 2011.

[37] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *OSDI*, pages 367–382, 2016.