IET Information Security

Research Article

Dynamic reencryption of return addresses

ISSN 1751-8709
Received on 3rd August 2017
Revised 29th May 2018
Accepted on 24th July 2018
E-First on 25th September 2018
doi: 10.1049/iet-ifs.2018.5142
www.ietdl.org

Hyungyu Lee[1], Changwoo Pyo[1] ✉, Gyungho Lee[2]
[1]Department of Computer Engineering, Hongik University, Seoul, Republic of Korea
[2]Department of Computer Science & Engineering, Korea University, Seoul, Republic of Korea
✉ *E-mail: pyo@hongik.ac.kr*

Abstract: The authors present dynamic reencryption of return addresses to mitigate their leakage. The authors' method enforces programs to save return addresses as encrypted and renew the encryption states with fresh keys before or after vulnerable operations. When a function returns, it should restore the return address from its encryption using the most recent key not to cause a crash. Under the protection of their method, return addresses and keys may leak, but the disclosed bits become garbage because keys govern all return addresses through encryption, while changing before control-flow proceeds into a vulnerable region. As a result, it becomes probabilistically infeasible to build exploits for intercepting control-flow by using leaked return addresses or keys. They implemented the proposed method as an extension of the LLVM compiler that inserts reencryption code where necessary. They also have confirmed its effectiveness against information leak attacks carried out in the early stage of blind return-oriented programming (BROP). The performance overhead ranges below 11.6% for processor-intensive programs and 4.12% or less for web servers.

1 Introduction

Attacks on a program commonly start with overwriting a code pointer. When a branch instruction dereferences the compromised pointer, control-flow diverts to the overwritten address. Encryption of code pointers [1–4] has developed into effective and low-cost mitigation. Since decryption should precede dereferencing the pointers, attackers cannot intercept control-flow by overwriting the encrypted pointers blindly without knowing how to decrypt the pointers.

However, the effectiveness of code pointer encryption diminishes significantly if the state of encryption stays fixed during program execution. Code pointers in writable memory are not free from information leaks whether they are encrypted or not as demonstrated in new offensive arts, for example, incremental brute-force memory reading [5], crash-resistance [6], and side-channel analysis [7]. The disclosed pointers are useful for crafting exploits of control-flow interception. Though a disclosed pointer is encrypted, an attacker may still be able to break the encryption using various runtime data. Even worse, an adversary can launch a code reuse attack by overwriting another encrypted code pointer with the disclosed pointer if the two encrypted pointers share the same key [8]. On the other hand, if a disclosed code pointer is in plaintext, it can immediately serve as a clue to the whereabouts of executable pages whose addresses are indispensable for code reuse attacks [9, 10]. A more obvious case is overwriting the plain pointer with an address that an attacker desires to deviate from normal control-flow.

```
1  /*  uninteresting parts are omitted */
2  void worker_thread_cycle(void *data) {
3    rc = http_read_request_body(r);
4  }
5
6  int http_read_request_body(
       http_request_t *r) {
7    /* 'size' is an attacker-controlled
         value */
8    int size = r->content_length_n;
9    char buffer[BUFFER_SIZE];
10
11   reencrypt();
12   /* SIGSEGV if crashes */
13   r->connection->recv(size, buffer);
14 }
15
16 void signal_handler(int signo) {
17   if(signo == SIGSEGV)
18     worker_thread_cycle(data);
19 }
```

Fig. 1 *Example of leaking encrypted return addresses using crash-resistant threads and its mitigation by reencryption*

As an instance of code pointer leaks, consider Nginx web server sketched in Fig. 1, where the functions are supposed to encrypt their return addresses. The server launches multiple worker threads by calling function `worker_thread_cycle` defined at line 2. Each worker gets an HTTP request by calling function http_read_request_body at line 3, which we supposed to have the weakness of buffer overflow. If an attacker assigns variable `content_length_n` a value large enough to reach the least significant byte of the return address, variable size gets the value at line 8, and the call `recv` overflows array `buffer` at line 13. The buffer overflow leads to a crash when it corrupts the byte of the return address in the stack frame for function http_read_request_body, but worker threads can resist the crash by exploiting the signal handler defined at line 16. The handler catches a signal `SIGSEGV` generated by the crash, decides to continue executing the thread, and calls function `worker_thread_cycle` again at line 18. If workers are crash-resistant, an attacker can overflow `buffer` one byte at a time into the return address until none of the workers generate `SIGSEGV`. Then the last overwritten byte is possibly correct. The incremental probing continues until reading all bytes of the encrypted return address, which can be used for cryptanalysis, overwriting another encrypted return address, and other attackers' purposes. If dynamic reencryption of return addresses (DRORA) is in effect, the call to function reencrypt at line 11 `reencrypts` the target return address with a fresh key before the buffer overflow into the target at line 13. As a result, the attackers may disclose the first byte of the target, but the disclosed bits have changed when they try the next byte.

Periodic or occasional renewal of encryption states of code pointers at runtime promotes the pointers to moving targets and mitigates the vulnerability caused by encryption staleness. Assuming that adversaries do not know the locations of executable pages, it would suffice to hide code pointers rather than entire process images to hide executable pages because the code pointers are the clues from which adversaries begin searching for exploitable instructions called *gadgets* in the return-oriented programming [10]. In addition, code pointer reencryption provides another merit that nullifies the typical beginning of code reuse attacks even when the location of a code pointer is exposed. A code reuse attack starts with overwriting a return address to transfer control to a gadget, but the encrypted return address blocks the attempt by enforcing decryption before executing the return instruction.

A contributing approach to moving target defence is rerandomisation of address space layouts. This approach randomly relocates program components at various granularity before forking a child process or between critical system calls [11, 12]. It aims to preserve confidentiality but cannot when crash-resistance is available at the thread level. Besides, the rerandomisation may have high overheads because the complexity and workload of relocations are not trivial when the grain size is small. Rerandomisation interval also needs scrutiny. If the interval is fixed, a window of no protection may sneak between two consecutive randomisations [13]. Another moving target defence against stack smashing rerandomises stack canaries. DynaGuard [14] refreshes stack canaries of a child process when a process forks the child, but may suffer from non-linear overwriting of return addresses.

As the first step towards a complete dynamic reencryption of code pointers, we have developed *DRORA*. Our method enforces programs to save return addresses as encrypted and renew the encryption states with fresh keys before or after vulnerable operations. When a function returns, it should restore the return address from its encryption using the most recent key not to cause a crash. The proposed method inserts code fragments for reencryption with new keys before or after vulnerable operations at compile time. Each return address has a unique encryption key whose value is randomised before every reencryption. Our approach carefully selects proper occasions of reencryption to keep performance overhead low. Our work also includes the protection of encryption keys and relevant data structures by diversification. Under the protection of our method, return addresses and keys may leak, but the disclosed bits become garbage because keys govern all return addresses through encryption while changing before control-flow proceeds into a vulnerable region. Considering the high population of return instructions in SPEC CPU2006, which amounts to 85% of all indirect branches, dynamic reencryption of return addresses is worthwhile for preventing control-flow hijacking, and the history affirms its necessity [15].

We present the proposed threat model in Section 2, the probabilities of code pointer leakages under runtime reencryption in Section 3, and the framework of encrypting return addresses with multiple keys in Section 4. We extend the framework to a dynamic version in Section 5. Implementation details and experimental evaluation of the proposed scheme regarding effectiveness and performance overhead are presented in Sections 6 and 7. A brief overview of related work, discussion, and conclusion follow in Sections 8–10, respectively.

2 Threat model and assumptions on targets

The purpose of attacks is to disclose a return address or a key to craft exploits diverting control-flow. It does not matter whether the key and address are encrypted or not. Adversaries can overwrite the return address in a stack frame by overflowing a buffer or by dereferencing a data pointer compromised to point to the return address stored in the stack frame. Also, they can drive the target in a crash-resistant way at the thread-level so that they can continue brute-force memory reading indefinitely. While brute-forcing, they can observe a response from the target program to an injected attack payload. The responses include legitimate output, error messages, and useful phenomena for side-channel analysis.

On the target side, we assume several modes and security measures are in effect. The width of the address path is greater than or equal to 32. The static address space layout randomisation [16] is turned on for user space. Also, memory pages are protected by data execution prevention (DEP) [17] that allows execution of instructions only from the pages permitting execution. We simply assume hardware support by NX (No Execution) bit. No details of operating systems' policies for executable space protection are assumed. The system initialises memory pages with zeros. The target program has exception handlers that catch signals and launch new threads. Finally, we suppose that none can read from or write into a register directly except the instructions taking the register as an operand. As a result, if we reserve a register for an encryption key and use the register only for encryption, the key does not leak as long as the register does not spill into the stack. The vulnerabilities in processor architecture [18, 19] are beyond the scope of this paper.

3 Probabilistic effectiveness of dynamic reencryption

Let a *yield point* of a sample space be the number of trials in the worst case that gives us the probability 1 of guessing a fixed target belonging to the sample space. A yield point would be equal to the size of the sample space if we guess a value and compare it with an element one at a time. However, the yield point can be reached earlier than the foregoing way by incrementally probing the sample space. For example, given a 64-bit target, guessing the 64 bits as a whole requires 2^{64} trials in the worst case. However, if we probe the target one byte after another, we can lower the yield point to $8 \times 2^8 = 2048$ where each byte has $2^8 = 256$ bit patterns at most. A program compiled with StackGuard [20] leaks its 32-bit canary far before the yield point [14].

Dynamic rerandomisation has been underrated as the mitigation of information leaks in the area of address space layout randomisation (ASLR) because rerandomisation contributes to increasing entropy by a single bit [21]. Nevertheless, the computed probabilities of an information leak under dynamic reencryption show effectiveness against incremental byte reading. In particular, reencryption is effective when the size of a target is large, for example, more than 32 bits, and the period of reencryption is not too long. Table 1 shows the probabilities of disclosure by incremental byte reading at yield points when a target is periodically reencrypted. Each column represents a period of renewing a key and encryption state. For a target in memory, the unit of the period is the number of memory accesses between reencryptions. For example, consider incrementally probing a 32-bit target. If the state of encryption does not change, the yield point to incremental byte reading is 1024. However, the probability of disclosure under runtime reencryption with a period of 6 memory accesses is 5.940×10^{-7} at the yield point.

Table 1 Success probabilities of incremental attacks under dynamic reencryption when the search spaces of static randomisation are exhausted. We computed the probabilities following the model proposed in [5]

N. bytes (N. probe)	Periods of reencryption						
	1	4	6	128	256	1024	1536
1 (256)	6.328×10^{-1}	6.350×10^{-1}	6.365×10^{-1}	7.500×10^{-1}	1.0	1.0	1.0
4 (1024)	N/A	5.960×10^{-8}	5.940×10^{-7}	1.970×10^{-2}	1.531×10^{-1}	1.0	1.0
6 (1536)	N/A	N/A	9.095×10^{-13}	2.312×10^{-4}	7.830×10^{-3}	9.238×10^{-1}	1.0

Table 2 Instrumentation locations of DRORA. The first two entries accept the code for static encryption, and the rest of the entries accommodates the code to dynamise the encrypted return addresses

	Instrumentation locations	Responsibilities	Context
static encryption	function prologues	initial encryption	calls
	function epilogues	final decryption	returns
dynamic reencryption	before risky function calls	reencryption	memory copy functions
	after spawning a child	reencryption	process and thread spawning
	before throw	decryption	exception, stack unwinding begins
	beginning of catch clause	reencryption	exception, stack unwinding ends
	before longjmp	decryption	stack unwinding begins
	after setjmp	reencryption	stack unwinding ends

The effectiveness of runtime reencryption improves as the size of a target increases. For an 8-bit target, the probability at the yield point is more than 0.6 even though the target is reencrypted before every access. In contrast, for a 48-bit target, the probabilities of disclosure at yield points are still between $O(10^{-3})$ and $O(10^{-13})$ with periods of 256 or less. Therefore, it is safe to increase the interval of reencryption as the target size increases.

So far, the probabilistic model assumes periodic reencryption to make the model simple. Periodic reencryption requires a monitoring process that can interrupt the monitored process and initiate reencryption when a timer or memory access counter reaches a limit [13]. This approach makes a runtime system complex and entails the running of another process for protection. Moreover, it is hard to balance between performance and effectiveness concerning frequency. Frequent reencryption is beneficial to program security but damaging to performance. The opposite that is, 'good performance and loose security' holds for a long interval.

Alternatively, aperiodic reencryptions before risky operations can avoid the shortcomings of the periodic approach. Aperiodic reencryptions can be easily embedded in a program rather than triggered by a separate process because of execution cost. Though the reencryption is activated irregularly, it would appear to adversaries to continuously change if it precedes *every* risky operation that may open a hole for leakage.

4 Static encryption of return addresses

DRORA instruments code for cryptographic work at the locations listed in Table 2, whose first two entries accommodate code for static encryption. The static part encrypts a return address at a function prologue and decrypts the encrypted address at the corresponding epilogue. Though the cryptographic work starts at the beginning of execution, it is static because the encryption state of a return address does not change during the lifetime of the return address. The lifetime begins when a call instruction pushes the return address onto the stack and ends when a return instruction pops out the address from the stack. This section discusses the static part of DRORA. In particular, we focus on measures against encryption key and return address leaks. Dynamising the static part is discussed in the following section.

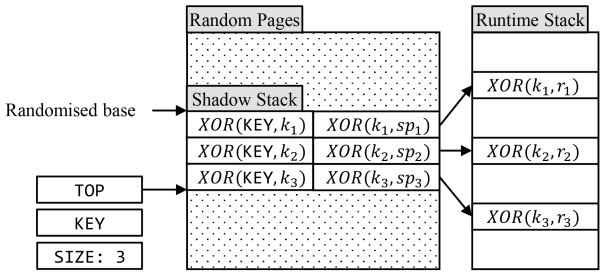


Fig. 2 *Structure of the shadow stack and its relationship with other data*

```
// extended function prologue
k = rnd();
*sp = XOR(k, *sp);
push_s(⟨XOR(KEY, k), XOR(k, sp)⟩);
⋮
// instructions of function body
⋮
// extended function epilogue
⋮
k = XOR(KEY, TOP → k);
*sp = XOR(k, *sp);
pop_s();
return
```

Fig. 3 *Structure of a function extended with code snippets for return address encryption and decryption*

4.1 Basic framework of return address encryption

Encrypting a return address with an individual non-shared key minimises damage when the key leaks. Relying on a single key for encrypting all return addresses is not safe because key leakage and control interceptions can occur at different times or locations for a program forking multiple processes. Though a process may crash disclosing the shared key due to an attack, other siblings and the parent can still be attacked using the disclosed key [22]. DRORA differentiates keys among return addresses as well as processes and threads.

To store a dedicated key for each return address, DRORA provides each process and thread with a shadow stack in thread-local storage (TLS), as shown in Fig. 2. TLS is global memory exclusive to each thread without being shared among threads. The stack keeps pairs of an encryption key k_i and pointer sp_i to the location storing a return address in the runtime stack, where the return address at the location $*sp_i$ is encrypted using the key k_i. Fig. 3 shows the instruction sequences for encryption and decryption of a return address. The instrumented prologue first produces a new key `k` and encrypts the return address at `*sp` with the key, where `sp` is the runtime stack pointer. Next, the snippet encrypts the pair ⟨`k, sp`⟩ to ⟨`XOR(KEY, k), XOR(k, sp)`⟩ and pushes the pair onto the shadow stack where KEY is a global key for randomising all other keys. Function `push_s()` also adjusts the shadow stack pointer `TOP` and its size `SIZE` accordingly. Note that reserved registers exempt from spills store `KEY`, `TOP`, and `SIZE`. If a register spills into memory, a window opens through which adversaries can read from or overwrite to the locations where registers spill [23].

Similarly, a code inserted at the end of a function epilogue preceding a return instruction decrypts the return address encrypted in the prologue. When control transfers back to the caller, the inserted code also removes the pair of encrypted key and stack pointer ⟨`XOR(KEY, k), XOR(k, sp)`⟩ from the shadow stack by adjusting `TOP` and `SIZE`.

DRORA uses the XOR instruction for crypto work because of its low performance overhead. If a cryptographically secure algorithm is employed instead of XOR, the program will slow down to an unbearable level. However, `XOR` cannot resist information leak attacks because `XOR` is the inverse of itself.

```
1 void foo(char *src)
2 {
3  char dest[10];
4  reencrypt();
5  strcpy(dest, src);
6 }
```

a

```
1 pid = fork();
2 if(!pid) { // child
3  reencrypt();
4 }
5 else {      // parent
6 }
```

b

Fig. 4 *Code fragments only conceptualise reencryption timing. Wrappers in Section 6.2 implement insertion of the calls to function* `reencrypt` *before or after calling risky functions*
(a) Call to copy function, ***(b)*** Process forking

Suppose `r` and `e` are a return address and its ciphertext, respectively, i.e. `e = XOR(k, r)`, where `k` is an encryption key. If an attack leaks `r` and `e`, key `k` is inferable by `k = XOR(e, r)`. Attackers can read the return address `r` and its encrypted text `e` through stack reading [22] or a side channel attack such as timing analysis [7].

Disclosing target system information such as return addresses does not require directly probing the target. Since deployed system software and microprocessors are not diverse enough to hinder guessing what they are, an attacker can build an identical system and gather various properties of the target such as memory layout and its security measures. With that information, they can draw a layout of virtual memory and the stack frame.

4.2 Hiding shadow stacks

DRORA takes some additional measures to hide the location and contents of the shadow stack. First, DRORA initialises the TLS pages for shadow stacks with random bits, in contrast to the default that initialises the pages with zeros. Memory pages initialised with zeros have a weakness of information disclosure [24] because the preceding and following zeros before and after shadow stacks draw the line along the boundaries of the stack. This observation is valid even when the location of the shadow stack is randomised. Starting from the base address, attackers traverse the shadow stack reaping bytes that make up encrypted addresses of the locations that have return addresses. Though a complex sequence of deciphering steps should follow the leaks, the disclosed bytes would serve as a strong clue to locations of return addresses. If a return address leaks, the overall text segment can be probed starting from the address.

Second, DRORA also randomises the location of the shadow stack within the segment as in ASLR. DRORA does not dynamically randomise the location for performance, but it does not easily locate an entry of the shadow stack because the bytes preceding and following the shadow stacks have random bits.

Finally, as explained in Section 4.1, DRORA also encrypts the key–pointer pairs of the shadow stack. As a result, it becomes harder than before to distinguish the shadow stack from the initial random bits. Although the whole shadow stack segment leaks, attackers cannot readily analyse the bits to discover the contents of the shadow stack. Note that the 16 high order address bits of ×86-64 processors are zeros. These high order zero bits appear in both return addresses in the runtime stacks and the pointers in the shadow stacks. DRORA fills the positions with random bits and takes care of them when randomised. Otherwise, the zeros become a clue about the layout of a randomised memory space.

5 Dynamising the static return address encryption

DRORA inserts code for dynamic reencryption when the hosting compiler generates an executable binary. An ideal method would reencrypt return addresses before every access to stack memory, but this would incur significant overhead. Also, extremely frequent reencryption is not necessary because not all memory accesses are exploitable. To maintain effectiveness with minimal performance overhead, DRORA limits runtime reencryption to the occasions immediately before or after risky operations, which are calls to copy functions, process forking, and exception handling.

5.1 Copy functions

Many functions of the standard C library, `strcpy()`, `memcpy()`, and `strdupa()`, e.g. copy bytes from one memory location to another without checking the size of the destination. When these functions copy data, attackers can attempt memory disclosure or a control-flow interception by forcing the functions to overwrite return addresses in the stack. Runtime reencryption precludes the possibility of successful attacks, even though the attack payload is crafted using a disclosed key. This is because the disclosed key is valid only in a limited interval ending before reencryption. DRORA currently inserts code fragments for reencryption before calling 60 risky copy functions of the glibc [25].

Fig. 4*a* shows an instance that reencrypts return addresses in the stack at line 4 before calling function `strcpy`. For each entry of the shadow stack, function `reencrypt` decrypts the entry, updates the key, locates the return address on the runtime stack, and finally reencrypts the return address at the location. Function `reencrypt` traverses the shadow stack, reencrypting all return addresses in the runtime stack.

5.2 Concurrent program execution

A process inherits its memory layout from its parent. As a result, both the process and its parent share the same set of encryption keys and encrypted return addresses. If a key leaks from one of the processes, it can be exploited for compromising other processes. To prevent key sharing among processes from a common ancestor, DRORA generates new keys and reencrypts return addresses of the forked child. As a result, all processes have encryption keys different from one another. Even when any of the processes leaks its keys, the keys become obsolete for other processes. This mitigation is similar to DynaGuard, which promotes StackGuard [20] to a dynamic version [14]. Fig. 4*b* shows the reencryption after forking a child.

As for threads, DRORA installs the shadow stack also in TLS, and thus DRORA can reencrypt return addresses in the thread stacks using the shadow stack. To hide the shadow stack in TLS, DRORA takes the same measure as the one for processes that initialises the shadow stack segment with random bits and randomly places the shadow stack in the segment. TLSs in ×86-64 systems have an additional feature beneficial for information hiding: Instructions should reference data in a TLS relatively from the segment register %fs. The register makes the leakage of the shadow stack difficult because the register does not spill into memory.

5.3 Stack unwinding

Throwing an exception or activating `longjmp/siglongjmp` initiates stack unwinding that deallocates stack frames and destroys any data object linked to the frames. Unwinding continues until the control-flow reaches an exception handler or a location marked by `setjmp/sigsetjmp`. As stack unwinding requires return addresses in the stack, DRORA maintains compatibility with stack unwinding by decrypting all encrypted return addresses before stack unwinding and encrypting the remaining return addresses in the stack again at the end of stack unwinding.

Fig. 5 illustrates the instrumentation that makes dynamic reencryption compatible with stack unwinding. The function `foo` calls bar, which raises an exception at line 3, but neither can catch the exception. Fig. 6*a* shows the state of the runtime stack before function call `decryptAll` at line 2 where all return addresses remain encrypted. Function `decryptAll` restores return addresses to plaintext before function bar throws the exception, resulting in Fig. 6*b*. Line 3 throws an exception that propagates up to the handler in function `main`. The stack unwinds until the function main's frame appears at the top, as shown in Fig. 6*c*. The exception handler promptly encrypts the remaining return addresses at line 12 through `encryptRemainder`. The runtime stack reaches the state illustrated in Fig. 6*d* when stack unwinding completes.

```
void bar(void) {
  decryptAll(); // restoration
  throw "exception";
}
void foo(void) {
  bar();
}
int main(int argc, char *argv[]) {
  try {
    foo();
  } catch (const char* msg) {
    encryptRemainder(); // reencryption
    /* handles exception */
  }
}
```

Fig. 5 *Reencryption and housekeeping before and after stack unwinding*

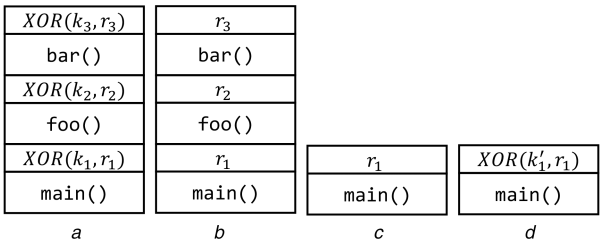


Fig. 6 *Sequence of stack unwinding by DRORA*
(a) Before calling function decryptAll, ***(b)*** After restoring return addresses to plaintext by function decryptAll, ***(c)*** After stack unwinding, ***(d)*** After reencrypting remaining return addresses by function encryptRemainder

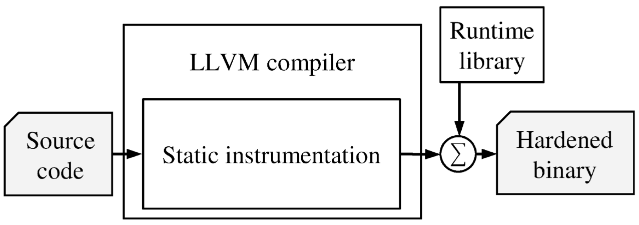


Fig. 7 *Overall workflow of DRORA*

5.4 Variations for improving performance

When the stack grows high, reencrypting all return addresses increases execution time. If execution speed is too slow to be practical, we may alter the design of DRORA for improved performance with a little sacrifice in security. The first variation reencrypts only the most recent return address at the top of the runtime stack. In general, we may apply reencryption to the $k \geq 1$ most recently pushed or randomly selected return addresses. Limiting the number of reencrypted return addresses improves runtime overhead, especially when the height of a runtime stack grows potentially high in recursion. This variation limits overhead to a fixed constant time.

Attacks cannot avoid encountering the currently active function that occupies the top stack frame. Based on this observation, we claim that the top return address has a higher probability of being attacked than others. Linear buffer overflow attacks can overwrite the top-most encrypted return addresses. The decryption before return would crash the target program even though adversaries do not intend to exploit the overwritten address. However, direct overwriting or probing a return address at a deep level in the runtime stack may have a chance of invalidating the reencryption of the top-most k return addresses.

Another variation works with a single dynamic key for all reencryptions. The single key variation reduces the size of the shadow stack as well as the time overhead for reencryption. Since every encryption of a return address shares a single key with each other, it is necessary to entirely update encryption states of the return addresses when the variation updates the key. In addition, it is mandatory to store the key in a dedicated register without the chance of spills to prevent leakage.

6 Implementation

DRORA consists of two components as shown in Fig. 7: One instruments programs with code fragments for dynamic reencryptions, and the other is a runtime library supporting the fragments. We implemented the components in the LLVM compiler infrastructure. LLVM driver applies dynamic reencryption as the last optimisation stage to prevent the instrumented code from being swept away by other optimisation passes.

The current implementation of DRORA is source-compatible, but DRORA's framework can be binary-compatible if a runtime instrumentation tool such as Pin [26] would intervene between binaries and processors. Nevertheless, dynamic instrumentation would be virtually impractical due to its high overhead. If source code is not available, binary editing may be feasible with the aid of a disassembler or decompiler. However, upward translation to assembly or source programs cannot be complete enough to guarantee source-compatibility for ×86 and similar whose instruction lengths are not uniform.

6.1 Static instrumentation module

We implemented the static instrumentation module as a class named `DRORA` shown in Fig. 8 and placed it in the directory `$LLVM/lib/Target/X86` for ×86-64-specific transformations. Class `DRORA` implements a virtual function `runOnMachineFunction` inherited from class `MachineFunctionPass`. For each function in source programs, DRORA calls the virtual function and inserts a code fragment for encryption or reencryption at every instrumentation point specified in Table 2. First of all, DRORA inserts the code fragment for the initial encryption of a return address in function prologue at line 7 in Fig. 8 and then iterates over instructions looking for the instrumentation points. Among those points, the call sites to functions `__cxa_throw`, `longjmp`, and `siglongjmp` attach calls to function `decryptAll` to prepare for stack unwinding. Similarly, DRORA inserts calls to function `encryptRemainder` after the calls to `__cxa_begin_catch`, `setjmp`, and `sigsetjmp` to reencrypt return addresses still left in the stack when stack unwinding has completed.

6.2 Runtime libraries

Since DRORA has two additional variations in Section 5.4, the runtime for DRORA is built as three static libraries. Linker selects one of them according to a compiler option `-fdrora`=k, $k = 1, 2, 3$. Each static library consists of three parts. The first part defines functions `reencrypt`, `decryptAll`, and `encryptRemainder` in Section 5.3.

The second part of the runtime defines *wrappers* for the risky functions and system calls in Section 5.1. A macro call `INTERCEPTOR(·, f, etc.)` defines the wrapper that intercepts the call to function `f` and checks whether function `f` writes into the stack. If it does, the wrapper calls function `reencrypt` to reencrypt all of the return addresses in the stack before invoking function `f`. Otherwise, the wrapper merely transfers control to function `f` without security checks. DRORA also wraps system call `fork` as shown at line 12 in Fig. 9 to reencrypt return addresses in the stack of a child process.

The last part of the runtime is responsible for initialisation of TLS and register `XMM15` reserved for the global key. DRORA enrols the initialisation routines in section.`preinit_array` of ELF format. This section defines an array of function pointers which the dynamic linker invokes before execution proceeds to function main. Meanwhile, to initialise TLS and `XMM15` after creating a thread, we define a wrapper for function `pthread_create` to create a thread that initialises TLS and transfers control to the intended thread's main.

```
class DRORA : public
    MachineFunctionPass
{
  void insertDRORAx64Prologue(
      Instruction &I);
  void InsertDRORAx64Return(Instruction
       &I);

  virtual void runOnMachineFunction(
      MachineFunction &F) {
    insertDRORAx64Prologue(F.begin());
    for(iterator I=F.begin(), E=F.end()
        ; I!=E; ++I) {
      if(I->isReturn())
        insertDRORAx64Epilogue(I);

      /* UnwindBegin: __cxa_throw,
          longjmp, siglongjmp */
      if(I->isCall() && UnwindBegin.has
          (I->getFuncName()))
        CreateCall("decryptAll")->
            insertBefore(I);

      /* UnwindEnd: __cxa_begin_catch,
          setjmp, sigsetjmp */
      if(I->isCall() && UnwindEnd.has(I
          ->getFuncName()))
        CreateCall("encryptRemainder")
            ->insertAfter(I);
    }
  }
};
```

Fig. 8 *Class* `DRORA`

A TLS has a shadow stack of 1 MB and two address constants bounding the stack segment. The compiler places data structure prefixed with `__thread` specifier in TLS. The initialisation routine reads process memory mapping information in `/proc/`*process-id*`/maps` and fills the area for a shadow stack with random bits from `/dev/random`. These random bits become the initial values of the private keys for return addresses. Also, the initialisation routine randomly selects the starting top position of the shadow stack and copies the address of the top frame into the lower half of register `XMM15`. The higher half contains the global key set by `/dev/random`.

6.3 Safety of encryption keys

DRORA encrypts each return address by using its private key stored in the shadow stack. In turn, DRORA also encrypts the private keys using a global key. To keep the global key secure, DRORA reserves two `XMM` registers. The upper half of 128-bit register `XMM15` stores the global key, and the lower half serves as the shadow stack pointer. Register `XMM14` is a temporary used when DRORA updates the global key. `XMM` registers do not spill except for context switching. To avoid conflicts with the C library in allocating `XMM` registers, we recompiled the GNU C library to reserve the register. Without reservation, the underlying LLVM compiler would freely generate code spilling the register into a stack frame of a user process where the confidentiality of the register cannot be guaranteed. If the registers were reserved, the kernel would spill the registers in the kernel stack. Consequently, the global key is secure as long as the kernel is secure.

7 Experiments

7.1 Effectiveness

To demonstrate effectiveness, we tested if DRORA could protect an Nginx web server from information leak attacks. We replicated CVE-2013–2028 [27], a stack buffer overflow in Nginx-1.4.0, which is also demonstrated by Blind ROP (BROP) [22]. BROP starts by disclosing a return address and continues the next stage of the attack using the leaked address. We confirmed that programs without any protection or with static encryption could not block the leaks of return addresses, as shown in Figs. 10*a* and *b*. Dynamic reencryption, on the contrary, prevents the exploit from incrementally reading more than 2 bytes of the encrypted return address, as demonstrated in Fig. 10*c*. The leaked bytes are reencrypted before brute-forcing the next byte. Attackers could guess all 6 bytes at once. Since the probability of correct guesses is very low, $1/2^{48}$, we also expect that reencryption can block attacks utilising crash-resistance as long as reencryption immediately precedes vulnerable operations.

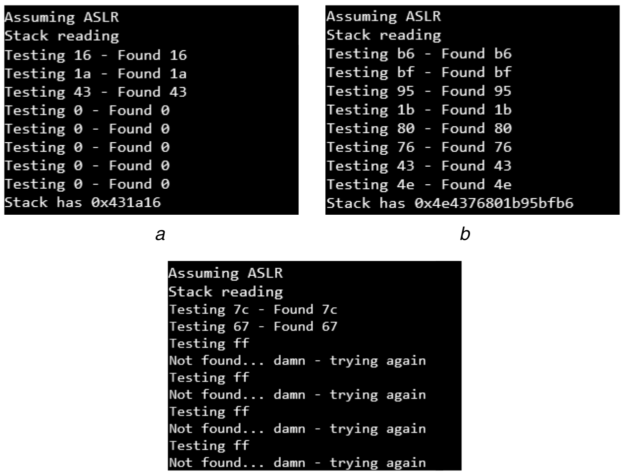
```
INTERCEPTOR(char *, strcpy,
            char *dest, const char *src
                )
{
  if(dest >= stack_lower &&
     dest <= stack_upper)
    reencrypt();

  char *ret = REAL(strcpy)(dest, src);
  return ret;
}

INTERCEPTOR(int, fork)
{
  int ret = REAL(fork)();
  if(!ret)
    reencrypt();

  return ret;
}

INTERCEPTOR(int, pthread_create,
            void *th, void *attr,
            void *(*start_routine)(void
                 *),
            void *arg)
{
  /* tci : {start_routine, arg} */
  int ret = REAL(pthread_create)(
    th, attr, drora_thread_start_func,
        tci
  );
  return ret;
}

void *drora_thread_start_func(drora_tci
     *tci) {
  /* initialise XMM15, shadow stack and
     stack boundaries */
  Initialize();
  return tci->start_routine(tci->arg);
}
```

Fig. 9 *Wrappers defined in DRORA's runtime*

7.2 Performance

We measured the performance overhead of DRORA and its two variants using SPEC CPU2006 for CPU-bound workloads and HTTP servers Apache and Nginx for IO-bound workloads. Since HTTP servers run perpetually, it is not appropriate to measure overhead in execution time. Instead, we measured the overhead in round-trip time (RTT) of processing a request with various configurations of the servers and a client.

Dynamic reencryption increases execution time by 11.6% on average for SPEC CPU2006. Contrary to our expectation, as observed in Fig. 11, the dominant source of overhead is the shadow stacks rather than runtime reencryption. Each column of the figure has the following three parts: the bottom parts represent the overhead for static encryption incurred when the addresses are pushed onto and popped from the runtime stack. The middle parts



```
Assuming ASLR
Stack reading
Testing 16 - Found 16
Testing 1a - Found 1a
Testing 43 - Found 43
Testing 0 - Found 0
Testing 0 - Found 0
Testing 0 - Found 0
Testing 0 - Found 0
Testing 0 - Found 0
Stack has 0x431a16
```

a

```
Assuming ASLR
Stack reading
Testing b6 - Found b6
Testing bf - Found bf
Testing 95 - Found 95
Testing 1b - Found 1b
Testing 80 - Found 80
Testing 76 - Found 76
Testing 43 - Found 43
Testing 4e - Found 4e
Stack has 0x4e4376801b95bfb6
```

b

```
Assuming ASLR
Stack reading
Testing 7c - Found 7c
Testing 67 - Found 67
Testing ff
Not found... damn - trying again
Testing ff
Not found... damn - trying again
Testing ff
Not found... damn - trying again
Testing ff
Not found... damn - trying again
```

c

Fig. 10 *Protection of Nginx against BROP using DRORA. Incremental brute-force memory reading discloses a return address when no encryption is in effect or encryption is static*
(a) No encryption, ***(b)*** Static encryption, ***(c)*** Dynamic reencryption

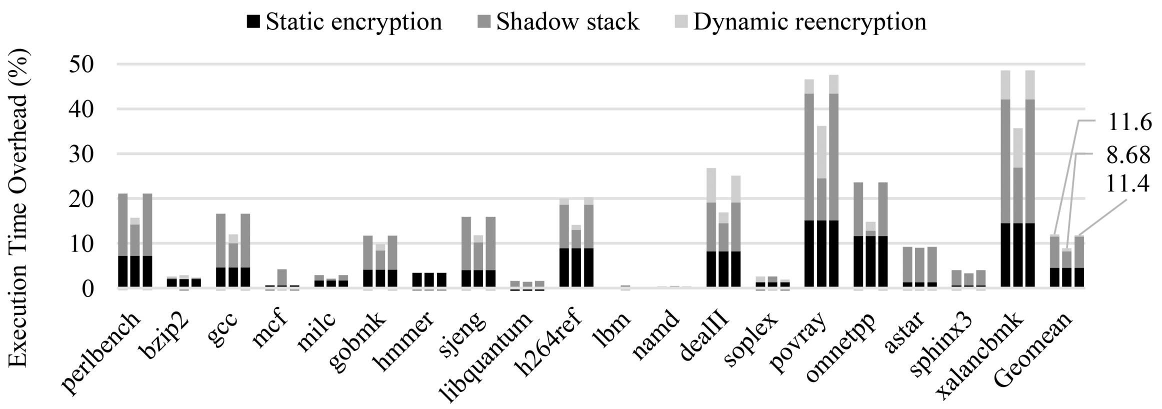


Fig. 11 *Execution time overhead of DRORA on SPEC CPU2006. Each benchmark has three bars. The first bar shows the overhead when every return address is dynamically reencrypted with its private key. The second bar shows the reduced overhead when return addresses share a single encryption key, and the last bar shows the overhead when only the return address in the top stack frame is dynamically reencrypted*

Table 3 Overhead in RTT of HTTP servers Apache and Nginx. The column 'DRORA' is the overhead when all return addresses are dynamically reencrypted. The column '1 Key' shows overhead when DRORA dynamically encrypts all return addresses with a single key. The last column 'Top' is the overhead for dynamically reencrypting only the return address in the top frame

Programs	RTT, µs	Overhead factors	Overhead, %		
			DRORA	1 key	Top
Apache	38.7	encryption	1.76	1.76	1.76
		shadow stack	1.54	1.51	1.54
		reencryption	0.82	0.22	0.06
		overall	4.12	3.49	3.36
Nginx	26.2	encryption	1.58	1.58	1.58
		shadow stack	0.83	0.56	0.83
		reencryption	0.65	0.31	0.15
		overall	3.06	2.45	2.56

denote the increased overhead due to the shadow stacks without dynamic reencryption. The entire height of each column is the overall overhead when dynamic reencryption is in effect.

We could lower the overhead by simplifying the structure of the shadow stacks though protection may become somewhat loose. Instead of assigning an exclusive key for a return address, we can adopt a single key for all return addresses. As a result, the overhead necessary for managing individual keys disappears. Among the three bars of each benchmark program in Fig. 11, the second one is the overhead when DRORA uses a single key for encryption of all return addresses.

Limiting dynamic reencryption to the return address in the top stack frame may reduce overhead at the sacrifice of security. However, the portion of the overhead by dynamic reencryption is not significant, and thus the effect is marginal. The third bar of each benchmark in Fig. 11 shows the overhead.

For web server benchmarking, we measured RTTs in a single host environment that executes both a server and client to minimise the distortion by network delays. An HTTP server is configured to launch W workers at most, and so is a client to generate C concurrent requests, where W and C are powers of 2 less than or equal to 64, giving 49 measured RTTs. DRORA increases the average RTT by 4.12% for Apache and 3.06% for Nginx. Table 3 shows the overheads and their ingredients of DRORA and its two variants.

Table 4 Densities of encryptions and reencryptions of the benchmarks SPEC, Apache, and Nginx. The density is the number of calls to the corresponding functions per second

Programs	Encrypt.	Reencrypt.	Overhead, %
SPEC	9.15×10^7	7.76×10^4	11.6
Apache	9.89×10^5	1.02×10^3	4.12
Nginx	4.08×10^5	1.51×10^3	3.06

On the other hand, the dynamic reencryption of code pointers would seldom affect the performance of web servers in a network environment. The average RTT increases by 0.65% for Apache and 0.36% for Nginx when we measured RTTs using two separate hosts for the server and client. More than 90% of an RTT comes from network delays, which hide the execution time increased by DRORA. Since network delays are unpredictable and larger than execution times on the hosts, the delays hide the overheads by DRORA.

The differences in overheads of SPEC, Apache, and Ngnix can be explained in terms of the densities of encryptions and reencryptions. The density of encryptions is the number of calls to the functions responsible for initial encryption of return addresses and the corresponding decryption. Reencryption density is defined similarly. Table 4 shows the measurements: The larger the numbers of encryptions and reencryptions are, the higher the overheads.

8 Related work

Program security depends on memory integrity. The in-depth survey reported in [28] provides a description and classification of memory attacks and mitigations. Protection of code pointers has been developed in two streams. One is based on monitoring control-flow integrity (CFI) [29], and the other is based on encryption [1–3]. CFI, in general, confirms whether critical control transfers remain inside a control-flow graph (CFG) deduced by static analysis. CFI requires no secrecy in protecting running programs. CFI is immune to information leak attacks for control-flow interceptions, but it has some weakness with issuing false negatives because a CFG may include edges that do not correspond to possible control transfers. Furthermore, CFI has high overhead in comparing control transfers with a CFG, whose complexity is not trivial. Researchers have attempted to reduce the overhead by increasing the monitoring granularity [30, 31], but the effectiveness diminishes, worsening the false negative defect [32–34].

Another stream has pursued randomisation that injects arbitrariness into memory. Randomisation hides security-critical information by deploying chaos in memory and equips programs with probabilistic defence. Blindly probing randomised memory would result in crashes due to illegal memory accesses. Randomisation applies to address spaces and bit-level representations of data values and instructions [35]. ASLR [16] introduced randomisation of memory segment locations. However, the 32-bit version of ASLR can only diversify 16–24 address bits due to alignment restrictions. Consequently, it does not have sufficient entropy to block brute-force attacks [21]. As an endeavour to increase the level of entropy, researchers worked on fine-grained ASLRs at the function [36], basic block [37], and instruction [38] levels. However, even the fine-grained address space randomisations are vulnerable to code pointer leaks [9], fault and timing analysis [7, 22], or crash-resistance [6]. These attacks can leak process memory layouts and proceed to subsequent control hijacking attacks.

Code pointer integrity (CPI) [39] maintains and hides the data structure containing the boundaries of code pointers in the huge address space provided by 64-bit microprocessors, though they currently activate 48 address bits out of 64 bits. Similarly, a shadow stack technique based on compilers [40] or binary rewriting [41] preserves the stack integrity by separating the return addresses and hides them in the vast address space. Although subverting information hiding in 48-bit space seemed improbable, attackers could find hidden memory regions and then corrupt or gather information from those regions. Specifically, Evans *et al.* [24] illustrated how to locate CPI meta-data via memory scans based on timing analysis [7]. Conti *et al.* [23] showed bypassing a shadow stack when there exists any reference pointing to the shadow stack in memory.

In addition to locations, bits representing instructions and data values can be randomised. Instruction set randomisation [42] encrypts instructions to prevent code injection attacks. PointGuard [2] encrypts code pointers whenever the pointers are assigned and decrypts the pointers before dereferencing them. Program-counter encoding [1] encrypts return addresses, and later was extended to include encryption of most code pointers [43]. Program-counter encoding decrypts the encoded return addresses and function pointers before being loaded into program counters. Simple pointer copies require neither encryption nor decryption. This can result in compatibility issues when return operations or function pointers become involved in pointer arithmetic, but there are no problems in practice as demonstrated by building the standard C library. The static part of the proposed method is similar to program-counter encoding reinforced by an exclusive key for the encryption of each return address. DSR [4] encrypts all types of data including both control and non-control data. However, DSR has weaknesses against information leaks because it stores the encrypted data and keys in memory. CCFI [3] puts a message authentication code (MAC) of a code pointer next to it and verifies that the MAC is intact when the code pointer gets assigned. CCFI is also vulnerable to information leaks since it allows reusing the MAC and its corresponding pointer at their original locations [8].

Information hiding by randomisation or encryption is vulnerable to attacks that bypass high entropy. Spraying attacks [44] lower the entropy of security sensitive metadata for each thread by spawning an enormous number of threads and filling the address space with the metadata. Oikonomopoulos *et al.* [45] show that the sizes of unallocated memory holes in the address space can be used as offsets to lower the entropy of sensitive metadata. BROP [22] incrementally brute-forces the stack to disclose code pointers and blindly searches gadgets using the pointers. BROP takes advantage of the fact that a spawned child process starts with the same memory contents as its parent. Crash-resistant oriented programming [6] makes it possible for attackers to continue scanning the memory of the target program by leveraging their crash-resistance. Rudd *et al.* [8] showed that even encoded code pointers in memory could be leaked and reused by chaining them.

Dynamic rerandomisation can defeat the attacks that circumvent entropy barriers. TASR [12] and RuntimeASLR [11] both dynamically rerandomise an address space layout to mitigate information leakage. TASR incurs 30 and 2.1% overheads on average with -O2 and -Og flags, respectively, in SPEC CPU benchmarks [46]. RuntimeASLR incurs 0.5% overhead on the web service, but it is 12,218 times slower on average for SPEC CPU benchmarks due to its pointer tracking. CodeArmor [13] replaces code pointers with offsets to their targets and then rerandomises the code layout periodically. Shuffler [46] also periodically rerandomises the code layout and reencrypts return addresses using the XOR operation with a per-thread key. Both CodeArmor and Shuffler reduce their overheads to 6.9 and 14.9%, respectively, for SPEC CPU2006 by deploying asynchronous code rerandomisation. However, their periodic property cannot avoid time windows between two rerandomisations in which an attacker can launch an information leak attack. Note that a key in per-thread memory is vulnerable to memory disclosure. DRORA does not allow such time windows by invoking reencryption before or after vulnerable operations. Also, to prevent the leakage of encryption keys, DRORA encrypts the keys and randomises their location.

DynaGuard [14] extends StackGuard [20] to differentiate stack canaries among child processes spawned from a parent.

DynaGuard is semi-dynamic because it renews canaries only when a process forks a child, and the canaries of the child process stay unchanged during its lifetime. Isomeron [47] dynamically selects targets for calls and returns between two copies of the program code. However, recent work showed that crash-resistance [6] could bypass Isomeron since it keeps the same memory layout during the lifetime of the process. DRORA renews the encryption states of child processes to prevent brute-forcing a child based on crash-resistance.

9 Discussion

9.1 Merits

DRORA has an advantage of not increasing the number of code pointers over ASLR and its dynamic versions. Dynamic descendants of ASLR have to relocate numerous components and adjust pointers in accordance to the new locations of the components. To minimise relocation workload, ASLR-family generates position-independent code (PIC). Since PIC relies on indirect branches, it has more code pointers than non-PIC. Since code pointers are adversaries' primary targets, it is not desirable to increase the number of code pointers. Though DRORA is limited to return addresses, if it would be extended to include function pointers, DRORA's framework would not increase the population of code pointers.

DRORA can refine reencryption timing independently of other program components and without kernel support. In particular, a thread can reencrypt the return addresses in its stack while another continues execution in the same process. This feature can prevent attacks resorting to in-process crash-resistance by exploiting exception handling and multithreading. On the other hand, mitigations relying on kernel support, for example, ASLR's dynamic descendants [11, 12] cannot apply rerandomisation freely because it requires two times of context switching to and from the kernel: The kernel pauses a process, randomly relocates the components of the process, and finally lets the process resume execution. If rerandomisation were initiated at an arbitrary point as DRORA does, the overhead would increase further. Thus rerandomisation takes place only when the kernel is about to take or release control for unavoidable services. Process forking is a typical instance.

DRORA installs the data structures in the user space, hides them by random placement similar to ASLR and initialises surrounding locations by random bits. It has to protect its data area for itself. On the other hand, the approaches relying on kernel support have the advantage of using kernel memory which is, in general, regarded as being more secure than user memory space.

9.2 Limitations

DRORA does not cover function pointers including the pointers to virtual functions in C++. Considering that >85% of code pointers are return addresses in SPEC CPU2006, DRORA covers the most critical code pointer, but DRORA should extend to the rest of the code pointer. The function pointers in GOTs and virtual function pointer tables can also become moving targets similarly to return addresses. However, converting function pointers referenced in a chain of copy or arithmetic operations is not as straightforward as that of return addresses. The difficulty comes from tracking code pointers in C/C++ programs where type casting and aliasing occur unruly. We leave it as future work to expand dynamic code pointer reencryption for function pointers. Considering that the instruction mix of indirect branches referencing function pointers is <15% in SPEC CPU2006, the increased overheads would not hurt the practicality of DRORA.

Finally, in a multithreading environment, a return address might be overwritten by another thread between decryption and return instructions. However, this time-of-check-to-time-of-use attacks can hardly succeed because the attack should be conducted under precise timing. Besides, attackers should know when a process launches threads.

10 Conclusion

We have developed a method for dynamic reencryption of return addresses and implemented it in the LLVM compiler infrastructure. The method refreshes the encryption states of return addresses before or after vulnerable operations to mitigate the leakage of return addresses. The experimental results show that recent attacks with entropy reduction or crash-resistance cannot succeed in disclosing encrypted return addresses. It follows that a control interception by overwriting a return address becomes infeasible. Our implementation increases execution time by ∼11%. Dynamic reencryption brings forth synergy when it works together with ASLR. ASLR and dynamic reencryption together can thwart identifying a return address to attack from a memory layout and constructing the memory layout from the return address. Meticulous tuning of reencryption timing would lower the overhead further while increasing mitigation effects. It is also necessary to extend dynamic reencryption to code pointers other than return addresses so that dynamic reencryption can comprehensively convert all code pointers into moving targets.

11 Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2015R1D1A1A01057711 and NRF-2015R1A2A2A01003242).

12 References

[1] Lee, G., Tyagi, A.: 'Encoded program counter: self-protection from buffer overflow attacks'. Proc. Int. Conf. Internet Computing, Las Vegas, NV, USA, 2000, pp. 387–394
[2] Cowan, C., Beattie, S., Johansen, J.*, et al.*: 'Pointguard™: protecting pointers from buffer overflow vulnerabilities'. Proc. USENIX Security, Washington, DC, USA, 2003, pp. 91–104
[3] Mashtizadeh, A.J., Bittau, A., Boneh, D.*, et al.*: 'CCFI: cryptographically enforced control flow integrity'. Proc. Conf. Computer and Communications Security (CCS), Denver, CO, USA, 2015, pp. 941–951
[4] Bhatkar, S., Sekar, R.: 'Data space randomization'. Proc. Int. Conf. Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Paris, France, 2008, pp. 1–22
[5] Evans, D., Nguyen-Tuong, A., Knight, J.: 'Effectiveness of moving target defenses', in Jajodia, S., Ghosh, A.K., Swarup, V.*, et al.* (Eds.):'*Moving target defense*' (Springer, New York, NY, 2011, 1st edn.), pp. 29–48
[6] Gawlik, R., Kollenda, B., Koppe, P.*, et al.*: 'Enabling client-side crash-resistance to overcome diversification and information hiding'. Proc. Network and Distributed System Security Symp. (NDSS), San Diego, CA, USA, 2016
[7] Seibert, J., Okkhravi, H., Söderström, E.: 'Information leaks without memory disclosures: remote side channel attacks on diversified code'. Proc. Conf. Computer and Communications Security (CCS), Scottsdale, AZ, USA, 2014, pp. 54–65
[8] Rudd, R., Skowyra, R., Bigelow, D.*, et al.*: 'Address-oblivious code reuse: on the effectiveness of leakage-resilient diversity'. Proc. Network and Distributed System Security Symp. (NDSS), San Diego, CA, USA, 2017
[9] Snow, K.Z., Monrose, F., Davi, L.*, et al.*: 'Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization'. Proc. IEEE Security and Privacy, San Francisco, CA, USA, 2013, pp. 574–588
[10] Shacham, H.: 'The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)'. Proc. Conf. Computer and Communications Security (CCS), Alexandria, VA, USA, 2007, pp. 552–561
[11] Lu, K., Nürnberger, S., Backes, M.*, et al.*: 'How to make ASLRWin the clone wars: runtime re-randomization'. Proc. Network and Distributed System Security Symp. (NDSS), San Diego, CA, USA, 2016
[12] Bigelow, D., Hobson, T., Rudd, R.*, et al.*: 'Timely rerandomization for mitigating memory disclosures'. Proc. Conf. Computer and Communications Security (CCS), Denver, CO, USA, 2015, pp. 268–279
[13] Chen, X., Bos, H., Giuffrida, C.: 'Codearmor: virtualizing the code space to counter disclosure attacks'. Proc. IEEE European Symp. on Security and Privacy, Paris, France, 2017, pp. 514–529
[14] Petsios, T., Kemerlis, V.P., Polychronakis, M.*, et al.*: 'Dynaguard: armoring canary-based protections against brute-force attacks'. Proc. Annual Computer Security Applications Conf. (ACSAC), Los Angeles, CA, USA, 2015, pp. 351–360
[15] 'Security vulnerabilities published in 2017'. Available at https://www.cvedetails.com/vulnerability-list/year-2017/vulnerabilities.html, accessed May 2018
[16] 'Address space layout randomization'. Available at https://pax.grsecurity.net/docs/aslr.txt, accessed November 2017
[17] 'Data execution prevention, part 3: memory protection technologies'. Available at https://technet.microsoft.com/en-us/library/bb457155.aspx, accessed November 2017
[18] Lipp, M., Schwarz, M., Gruss, D.*, et al.*: 'Meltdown', ArXiv e-prints, 2018
[19] Kocher, P., Genkin, D., Gruss, D.*, et al.*: 'Spectre attacks: exploiting speculative execution', ArXiv e-prints, 2018

[20] Cowan, C., Pu, C., Maier, D., *et al.*: 'Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks'. Proc. USENIX Security, San Antonio, TX, USA, 1998, pp. 63–78

[21] Shacham, H., Page, M., Pfaff, B., *et al.*: 'On the effectiveness of address-space randomization'. Proc. Conf. Computer and Communications Security (CCS), Washington, DC, USA, 2004, pp. 298–307

[22] Bittau, A., Belay, A., Mashtizadeh, A., *et al.*: 'Hacking blind'. Proc. IEEE Security and Privacy, San Jose, CA, USA, 2014, pp. 227–242

[23] Conti, M., Crane, S., Davi, L., *et al.*: 'Losing control: on the effectiveness of control-flow integrity under stack attacks'. Proc. Conf. Computer and Communications Security (CCS), Denver, CO, USA, 2015, pp. 952–963

[24] Evans, I., Fingeret, S., González, J., *et al.*: 'Missing the point (er): on the effectiveness of code pointer integrity'. Proc. IEEE Security and Privacy, San Jose, CA, USA, 2015, pp. 781–796

[25] 'The GNU C library'. Available at https://www.gnu.org/software/libc/manual/, accessed November 2017

[26] Luk, C.K., Cohn, R., Muth, R., *et al.*: 'Pin: building customized program analysis tools with dynamic instrumentation'. Proc. Programming Language Design and Implementation (PLDI), Chicago, IL, USA, 2005, pp. 190–200

[27] 'CVE-2013-2028'. Available at http://cve.mitre.org/cgibin/cvename.cgi?name=CVE-2013-2028, accessed November 2017

[28] Szekeres, L., Payer, M., Wei, T., *et al.*: 'Sok: eternal war in memory'. Proc. IEEE Security and Privacy, San Francisco, CA, USA, 2013, pp. 48–62

[29] Abadi, M., Budiu, M., Erlingsson, U., *et al.*: 'Control-flow integrity'. Proc. Conf. Computer and Communications Security (CCS), Alexandria, VA, USA, 2005, pp. 340–353

[30] Zhang, M., Sekar, R.: 'Control flow integrity for COTS binaries'. Proc. USENIX Security, Washington, DC, USA, 2013, pp. 337–352

[31] Zhang, C., Wei, T., Chen, Z., *et al.*: 'Practical control flow integrity and randomization for binary executables'. Proc. IEEE Security and Privacy, San Francisco, CA, USA, 2013, pp. 559–573

[32] Davi, L., Sadeghi, A.R., Lehmann, D., *et al.*: 'Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection'. Proc. USENIX Security, San Diego, CA, USA, 2014, pp. 401–416

[33] Göktas, E., Athanasopoulos, E., Bos, H., *et al.*: 'Out of control: overcoming control-flow integrity'. Proc. IEEE Security and Privacy, San Jose, CA, USA, 2014, pp. 575–589

[34] Carlini, N., Wagner, D.: 'ROP is still dangerous: breaking modern defenses'. Proc. USENIX Security, San Diego, CA, USA, 2014, pp. 385–399

[35] Larsen, P., Homescu, A., Brunthaler, S., *et al.*: 'Sok: automated software diversity'. Proc. IEEE Security and Privacy, San Jose, CA, USA, 2014, pp. 276–291

[36] Kil, C., Jun, J., Bookholt, C., *et al.*: 'Address space layout permutation (ASLP): towards fine-grained randomization of commodity software'. Proc. Annual Computer Security Applications Conf. (ACSAC), Miami Beach, FL, USA, 2006, pp. 339–348

[37] Wartell, R., Mohan, V., Hamlen, K.W., *et al.*: 'Binary stirring: self-randomizing instruction addresses of legacy x86 binary code'. Proc. Conf. Computer and Communications Security (CCS), Raleigh, NC, USA, 2012, pp. 157–168

[38] Hiser, J., Nguyen-Tuong, A., Co, M., *et al.*: 'ILR: where'd my gadgets go?'. Proc. IEEE Security and Privacy, San Francisco, CA, USA, 2012, pp. 571–585

[39] Kuznetsov, V., Szekeres, L., Payer, M., *et al.*: 'Code-pointer integrity'. Proc. USENIX Operating Systems Design and Implementation (OSDI), Broomfield, CO, USA, 2014, pp. 147–163

[40] Chiueh, T.C., Hsu, F.H.: 'RAD: A compile-time solution to buffer overflow attacks'. Proc. Int. Conf. Distributed Computing Systems, Mesa, AZ, USA, 2001, pp. 409–417

[41] Prasad, M., Chiueh, T.C.: 'A binary rewriting defense against stack based buffer overflow attacks'. Proc. USENIX Annual Technical Conf. (ATC), San Antonio, TX, USA, 2003, pp. 211–224

[42] Kc, G.S., Keromytis, A.D., Prevelakis, V.: 'Countering code-injection attacks with instruction-set randomization'. Proc. Conf. Computer and Communications Security (CCS), Washington, DC, USA, 2003, pp. 272–280

[43] Lee, G., Pyo, C.: 'Method and apparatus for securing indirect function calls by using program counter encoding'. U.S. Patent US8583939 B2, November 2013

[44] Göktas, E., Gawlik, R., Kollenda, B., *et al.*: 'Undermining information hiding (and what to do about it)'. Proc. USENIX Security, Austin, TX, USA, 2016, pp. 105–119

[45] Oikonomopoulos, A., Athanasopoulos, E., Bos, H., *et al.*: 'Poking holes in information hiding'. Proc. USENIX Security, Austin, TX, USA, 2016, pp. 121–138

[46] Williams-King, D., Gobieski, G., Williams-King, K., *et al.*: 'Shuffler: fast and deployable continuous code re-randomization'. Proc. USENIX Operating Systems Design and Implementation (OSDI), Savannah, GA, USA, 2016, pp. 367–382

[47] Davi, L., Liebchen, C., Sadeghi, A.R., *et al.*: 'Isomeron: code randomization resilient to (just-in-time) return-oriented programming'. Proc. Network and Distributed System Security Symp. (NDSS), San Diego, CA, USA, 2015