eavesROP: Listening for ROP Payloads in Data Streams (preliminary full version)

Jämthagen, Christopher; Karlsson, Linus; Stankovski, Paul; Hell, Martin

2014

eavesROP: Listening for ROP Payloads in Data Streams

(preliminary full version)

Christopher Jämthagen, Linus Karlsson, Paul Stankovski, Martin Hell

Dept. of Electrical and Information Technology, Lund University,
P.O. Box 118, 221 00 Lund, Sweden
{christopher.jamthagen,linus.karlsson,paul.stankovski,martin.hell}@eit.lth.se

Abstract. We consider the problem of detecting exploits based on return-oriented programming. In contrast to previous works we investigate to which extent we can detect ROP payloads by only analysing streaming data, i.e., we do not assume any modifications to the target machine, its kernel or its libraries. Neither do we attempt to execute any potentially malicious code in order to determine if it is an attack. While such a scenario has its limitations, we show that using a layered approach with a filtering mechanism together with the Fast Fourier Transform, it is possible to detect ROP payloads even in the presence of noise and assuming that the target system employs ASLR. Our approach, denoted eavesROP, thus provides a very lightweight and easily deployable mitigation against certain ROP attacks. It also provides the added merit of detecting the presence of a brute-force attack on ASLR since library base addresses are not assumed to be known by eavesROP.

Keywords: Return-Oriented Programming, ROP, Pattern Matching, ASLR

1 Introduction

Buffer overruns [3] have for a long time been a common source of software vulnerabilities. The buffer overrun vulnerability may be exploited to perform a code injection attack, where the goal is to inject arbitrary data and replacing the return address with the address of the injected data. There are several well-known and widely used mitigations against this approach. Since the injected code should not be executable, but rather considered as data, the memory pages corresponding to this data can be marked as non-executable. Data Execution Prevention (DEP) [19] is a hardware supported OS feature implementing this idea. A similar approach is the $W \oplus X$ security feature [33] in which memory pages are either writeable or executable, but not both. While this will prevent the classic code injection attacks, it will not prevent code reuse attacks. In these attacks, the adversary will not inject the code to be executed, but will instead direct the program flow to code that is already loaded by the process, typically a shared library. One example of a code reuse attack is the return-to-libc attack [7] in which the attack points to existing code in libc, a library used by many programs.

A more advanced code reuse attack is Return-Oriented Programming (ROP), in which the attacker identifies small pieces of usable code segments, called *gadgets*, and

chains them together using a `ret` instruction. A `ret` instruction will pop an address from the stack and continue execution at that address.

One available countermeasure against code injection, which can also be applied to prevent code reuse attacks, is Address Space Layout Randomization (ASLR) [31]. ASLR will randomize the base address of the program's text, stack, and heap segments and the adversary will not know at which address a library starts, or where the gadgets will be located. However, it has been shown that ASLR can sometimes be bypassed [37]. The addresses of certain instructions can be leaked through other vulnerabilities and in some cases it is feasible to brute force the start address of a library, thereby succeeding with a ROP attack even in the presence of ASLR.

There have been several proposed defenses against ROP attacks, all taking slightly different approaches and using different assumptions. A typical mitigation is to identify some specific features in the attack that distinguishes it from benign code execution and then build a mitigation technique based on those distinguishing features [10,11,15,17,30,32]. Another approach is to rewrite libraries or targeted code such that it is not usable in an attack [23,27] or to randomize addresses which are needed by an attacker [18,20,29,45].

Instead of detecting the attacks on the target systems, another goal may be to detect ROP attacks in data. In [32] data was scanned and possible expoits were speculatively executed in order to determine if they were exploits. This requires a snapshot of the virtual memory of the process that is protected. In [40] the authors consider a detection approach where documents are analyzed to find ROP attacks. Documents are collected and sent to a separate virtual machine, where they are opened in their native application, and a memory is then analyzed for ROP payloads.

In this paper we present eavesROP, which is a more lightweight approach where no execution takes place. We try to identify ROP payloads by looking at network traffic only, i.e., we do not make any modifications to machines, programs, libraries or operating systems; nor do we try to execute any of the received data. We do not even require any kind of access to the machines. Scenarios could be an implementation in a gateway to a corporate network, ROP payload detection in switches or at an ISP before data is forwarded to the end user. The question that we try to answer is: How much information can we deduce by just looking at the data? We target ROP exploits where gadget addresses can be explicitly found in the data sent to the application. We assume that ASLR is enforced by the operating system, and that the attacker has information about the location of libraries, either through information leakage or a brute-force attack. Of course, our detection mechanism has no such information. We show how to filter out possible ROP payloads and how to determine if the candidate payload is a ROP attack or not. Even with just a moderate number of gadgets, we can detect the payload efficiently. This is true even if there is a large amount of noise present.

We have tested eavesROP using several available exploits and it is able to detect all tested exploits with no false negatives.

2 Background

In this section we provide the necessary background on return-oriented programming and address space layout randomization.

2.1 Return-Oriented Programming

Return-oriented programming [36] is an exploitation technique that allows for arbitrary code execution without having to inject code into the vulnerable process. To achieve this, an attacker constructs a payload of addresses, each pointing to a small sequence of instructions reachable and executable by the affected process. These instruction sequences are called gadgets and typically consist of very few instructions, ending with a return instruction (`ret`). This return instruction will pop the next dword from the stack, put it into the instruction pointer register (EIP) and continue execution at the next gadget. Gadgets do not have to be aligned with the intended instructions. Any byte that represents the opcode of a `ret` can potentially be used as a gadget.

Not only `ret` instructions can be used in these types of attacks. It is also possible to use jump-based instructions as in [5,9]. We will not consider these type of attacks in this paper, but note that it would be possible to extend our algorithms to detect these gadgets as well.

As an example [25], one frequently used ROP exploit pattern disables DEP, which makes normal code injection is possible. Figure 1 shows the layout of the payload, residing on the stack, and the corresponding gadgets found in a library. `DEP call` represents the address of the function to disable DEP for the current process.

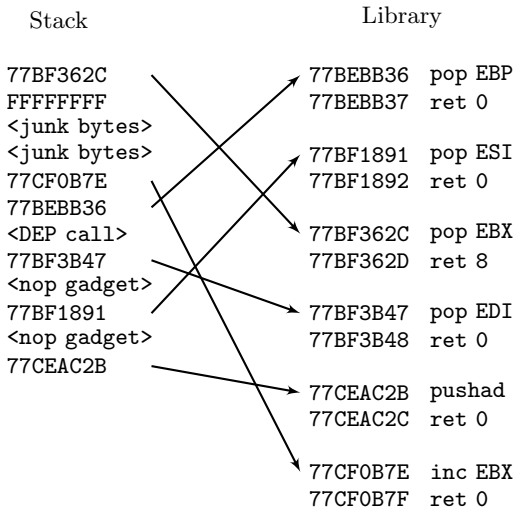

Fig. 1. An example of a ROP exploit. The addresses to gadgets are located on the stack, together with necessary integers and junk. The junk bytes are used when the `ret` instruction pops several values from the stack before continuing execution.

This exploit includes six instructions and if used as in the referenced exploit, where each instruction is included in its own gadget, we will have six gadgets each consisting of one instruction. As will be shown later, we are able to detect this payload.

2.2 Address Space Layout Randomization (ASLR)

ASLR protects from buffer overflow attacks by randomizing the location of the stack, the heap and the location of all dynamically loaded libraries. The term was coined by the PaX project [33] which also has a well-known implementation.

Following the introduction of ASLR in Windows XP SP2 (2004) and the Linux Kernel (since version 2.6.12, 2005) built-in ASLR support, writing exploits has become much more difficult. Since then, the number of base addresses that are randomized has increased, and so has the entropy of the randomization.

However, the efficiency of ASLR is limited. First, some small amount of code is not randomized, leaving the possibility to still use gadgets in the code where the location is predictable. Even though this code is rather small, it has been shown that it is possible to find usable gadgets in it [34]. Randomizing the application code is one kind of protection against these attacks [45]. Another aspect of ASLR, as was shown in [37], is the limited entropy in the address space, which makes it possible to brute-force absolute locations.

In addition to brute-forcing the ASLR, it has been shown that information leakage can occur through e.g., buffer and heap overrun bugs [16,43] and other types of vulnerabilites [35,39]. This could give an attacker at least partial information about the location of ASLR-affected code. Being able to read a return address on the stack is enough to deduce information about the base address of a DLL.

The exact means by which an attacker bypasses ASLR, may it be through brute force or information leakage, are independent of our payload detection algorithm.

3 Overview of Approach

In this section, we give a brief overview of our eavesROP. The entire exploit payload detection mechanism can be regarded as a black box device that takes a data stream as input and raises an alarm if it finds a ROP payload. The analysis of the data is based on the property that a payload consists of several 4-byte aligned addresses to gadgets within one library or chunk of executable code. Note that not all gadgets have to be in the same library. We only require a certain number (T) to be in one known library. The rest can be located anywhere.

The analysis performed inside the black box is divided into four layers, each with a distinct task, see Figure 2.

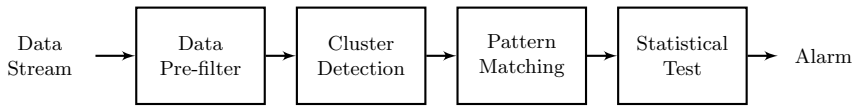

Fig. 2. Data flow overview.

Optional Data Pre-Filter The optional data pre-filter is a simple filter designed to discard all data that is unlikely to represent memory addresses.

Even though eavesROP is not very noise sensitive for detection, its performance is sensitive to data with very small variations, as this type of data could potentially represent memory addresses that are closely grouped. Plain text data has this near-recurring property and can thus for performance reasons be thrown away in a data pre-filter. A well-designed data pre-filter will significantly lower the processing requirements of the exploit payload detection mechanism since fewer blocks of data will be passed on to the next layer—the address cluster detection layer.

Cluster Detection An actual exploit payload will contain several gadget addresses that lie close together with respect to the entire addressable memory space. The purpose of the address cluster detection is to find and isolate the congested parts of the memory space for further processing.

The address cluster detection layer can be viewed as a transformation from data space to address space. Given a data window of some size, the clustering algorithm detects address windows that may contain dense memory activity. Thus, this layer also has the task of aligning data into 4-byte chunks since ROP gadgets are called using a 4-byte address, assuming a 32-bit architecture. After alignment, a second filter can be used in order to sort out certain addresses.

Based on clustered address windows, this layer will output a binary address vector, P_{obs}, of size L, i.e., the size of the largest targeted library. These vectors indicate addresses found in the data.

Pattern Matching The vector P_{obs} from the previous step is matched with binary library vectors. The relative distances of the memory addresses in an address window form a very distinct pattern. This pattern is matched with the gadget address patterns of libraries P_{lib}.

Without ASLR, this pattern matching could be trivially achieved by simple constant-time lookups into a suitable hash table (for example, see [28]). However, when we allow ASLR, the precise memory location of the library is unknown, making pattern matching more complex. By using the Fast Fourier Transform we can compute the maximum matching between P_{obs} and P_{lib}.

Statistical Test An address window containing only gadget addresses will give a perfect match. However, since we do not know the size of a payload, an address window may contain addresses that are just noise. The goal of the statistical test is to minimize false positives (α) and false negatives (β) by using a threshold value for the maximum overlap between P_{obs} and P_{lib}. This threshold will depend on the Hamming weight of P_{obs} and the targeted values of α and β.

4 A More Detailed Description

In this section we give a more detailed overview of the different parts of eavesROP.

4.1 Optional Data Pre-Filter

Certain input data can be expected to exhibit properties that make them look like addresses close to each other in the memory space—thus looking like ROP payloads—even though the data is actually non-malicious. Our goal is to filter out these addresses before they reach later steps in the algorithm, to reduce the total computational overhead of our system.

Of special interest are printable ASCII characters, not only because much data is readable text, but also because large portions of adjacent ASCII data may—when combined into 32-bit words—look like adjacent addresses. Filtering is however a trade-off between performance and false negatives. There are techniques to make ROP payloads printable [24]. Such a payload would be removed if a filter for printable characters is enabled. This is why the filtering step should be considered optional.

If the pre-filter is enabled, it removes blocks of UTF-8 strings. In our implementation, we define a block as a sequence of five or more adjacent, printable UTF-8 characters. A printable UTF-8 character is defined as all ASCII characters in the range from `0x20` up to and including `0x7e`. We also include the `0x09`, `0x0a` and `0x0d` for tab separator, line feed, and carriage return, respectively. Even though the last three are not printable characters, they occur frequently in the same context as the printable characters and whitespace. All valid multi-byte encoded UTF-8 characters are also considered printable.

When a matching block is found, the complete block is removed from the input. This leads to potential noise as non-adjacent bytes become adjacent after the data between them is removed. This does, however, only affect a few addresses, which does not cause any problems in practice since our ROP pattern matching is very precise and noise tolerant.

As our approach is modular, it is possible to add other heuristic pre-filters. Possible ideas may be to filter out specific network packets, such as ARP-requests. Even when using a very aggressive filter, if one or a few gadget addresses are filtered out, the detection will still succeed as long as there are enough gadgets left in the data that passes the filter.

4.2 Cluster Detection

We let M denote the maximum size of a ROP payload in 4-byte words that our detection is guaranteed to support. A naïve approach to detect the gadget addresses is to pick M words of data, map them to P_{obs} and match this vector with a known P_{lib}. Doing this byte by byte in the data would produce the correct maximum matching, but it is a very slow approach. Moreover, all words but one will repeat every 4 bytes. Another problem is that the addresses contained within the data window can be spread out over the entire ASLR address space (N bytes), making P_{obs} very large. We propose to use an algorithm that is much more efficient, and will still always find the correct maximum matching.

Instead of considering M addresses, we pick a data window of size $D = 2M$. Thus, we consider twice as much data as the maximum payload, but in return we consider $M + 1$ possible payloads simultaneously. This is illustrated in Figure 3. Doubling the data window size introduces a little more noise (more data in one window), so a few more

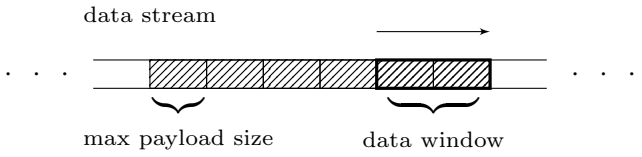

Fig. 3. Data window progression.

data windows will pass the cluster detection stage, but this effect is marginal compared to the significant gain in processing efficiency.

When the data window slides over the next data chunk of size M, we begin by extracting potentially viable addresses. As the offset of a ROP payload in the data buffer is unknown, but we know that each address is four bytes and addresses are aligned inside a payload, we create four lists, one for each offset, see Figure 4.

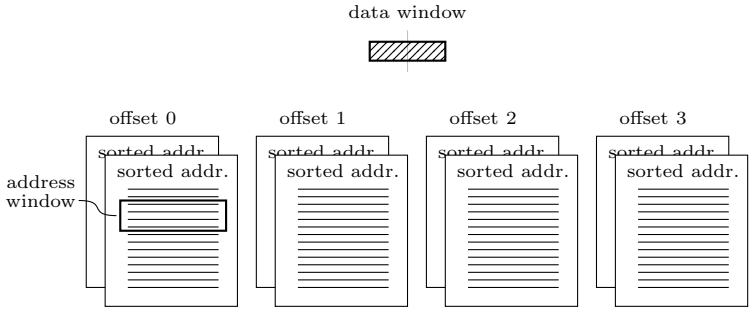

Fig. 4. Memory address pattern extraction from data window.

As Figure 4 further illustrates, we need to keep track of eight such address lists, four for each of the two M-word data chunks covered by the D-word data window. Separating the lists per data chunk allows for incrementing the data window in steps of M words, while reusing the four lists corresponding to the previous M words.

The four new address lists are sorted using an efficient linear-time sorting algorithm such as bucket sort [12]. Such efficient sorting is possible since all addresses are of the same size. Once sorted, we slide an address window of size L (same size as executable part of the largest library) down the combination of the two lists for each offset. Since each of the two lists is individually sorted, it is trivial to traverse the combination in sorted order efficiently. The time complexity for this is linear in the number of stored addresses, i.e., linear in D.

Let T be a threshold value that determines the minimum number of gadgets in an exploit that we want to be able to detect. A small T leads to detection of more exploits, but it also results in more pattern matching, slowing down the detection algorithm. In practice, the lowest value that our algorithm can handle is $T \approx 6$, depending on the

Algorithm 1 – Address Pattern From Data Stream

Input: data stream, maximum payload size in words M, address window weight threshold T, size of library L, word size in bytes n.
Output: set of P_{obs} (address window patterns).

$pos = 0$; /* current byte position in data stream */
$A_{(0,0)} = \ldots = A_{(0,n-1)} = \emptyset$; /* two address lists per offset */
$A_{(1,0)} = \ldots = A_{(1,n-1)} = \emptyset$;
while (data stream not exhausted) {
  for (each byte offset $i \in \{0, \ldots, n-1\}$) {
    $A_{(1,i)} = M$ words from data stream starting at $pos + i$;
    sort $A_{(1,i)}$;
  }
  $pos = pos + nM$;
  for (each offset $i \in \{0, \ldots, n-1\}$) {
    slide address window of size L over $A_{(0,i)} \cup A_{(1,i)}$ and find clusters;
    $A_{(0,i)} = A_{(1,i)}$;
  }
}

instruction size of each gadget (see Section 4.3) and the error probabilities (see Section 4.4).

If we find an address window that contains at least T unique addresses, the binary vector P_{obs} is constructed by entering a '1' in each position corresponding to an address in the address window. To minimize redundant checks, P_{obs} is normalized to always start with a '1'. Then we proceed to perform pattern matching via FFT, as described in Section 4.3. Algorithm 1 summarizes the cluster detection procedure.

4.3 Pattern Matching

In this section we give more details on the pattern matching layer. In order to pattern match the P_{obs} vector we first need to construct a vector P_{lib} of gadgets.

Identifying Gadgets in a Library In order to find all possible gadgets in a library, the executable part of it is scanned for the opcode of different types of return instructions, namely `0xC2` (`retn imm16`), `0xC3` (`retn`), `0xCA` (`retf imm16`) and `0xCB` (`retf`). For each position of these bytes in the library we search backwards one byte at a time and try to assemble a legal instruction flow ending with the return. We define the *entry zone*, z, as the number of instructions we allow for each gadget, not including the return instruction. This means that we can find many gadgets ending at the same return instruction due to the possibility of instruction overlapping in the x86 architecture.

The starting byte of every possible gadget is used to construct the binary vector P_{lib}. This is the vector that is used for pattern matching with P_{obs}, which is the output of the address cluster detection algorithm.

To understand how the gadget structure in a library is translated into a binary pattern, consider the following sequence of nine bytes (hex):

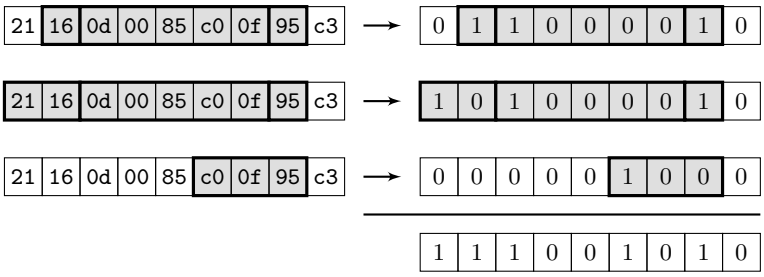

Fig. 5. Translation of maximal length gadget sequences to binary pattern.

```
21 16 0d 00 85 c0 0f 95 c3
```

Using an entry zone of size $z = 3$ (at most three instructions), we construct maximal gadget chains by interpreting the bytes preceding the return instruction `c3` as consecutive instructions. There are three possible maximal gadget chains in the above byte sequence, as illustrated in Figure 5.

The top two gadget chains are both of length three. While the top chain begins with a single-byte instruction `16`, the second chain extends this to a two-byte instruction `21 16`. The third chain is of length 1, but it is maximal since it cannot be further extended.

A sequence of bytes belonging to a library is translated into a binary pattern according to the following rules. Every byte position is designated a '0' or a '1'. A '1' is assigned if any maximal gadget chain has an instruction that begins at that byte position. If not, a '0' is assigned. Note that any unique sequence of valid instructions leading to a return counts as a gadget. Counting the instruction subsets of the top two chains, there are five gadgets in Figure 5. Algorithm 2 summarizes the construction of a P_{lib} vector.

As an example, Table 1 shows the number of gadgets, G, found in libc given the size of the entry zone. It should be noted that the gadget identification can be greatly optimized by filtering out useless gadgets.

Pattern Matching via FFT Perfect pattern matching can be performed efficiently using a Fast Fourier Transform (FFT). Pattern matching, here, means that we want to find the maximum weight of the overlap between two patterns that are overlaid, possibly displacing the patterns linearly with respect to one another. We also want this matching to be perfect, which is to say that all actual gadget addresses that are used in an exploit will be counted. All actual gadget addresses in an exploit will, in the general case, contribute positively to the weight of the maximal pattern match.

Table 1. The number of gadgets in libc for some choices of entry zone.

entry zone (z)	1	3	5	7
Number of gadgets (G)	12790	36113	57324	76796

Algorithm 2 – Gadget Pattern From Library

Input: entry zone z, library file f.
Output: library gadget pattern P_{lib}.

```
P_lib = (0, ..., 0); /* same length as f */
for (every byte position i in f) {
    if (byte i in f is not a return opcode) continue;
    /* disassembly */
    G = set of maximal gadget chains of length ≤ z ending at byte i;
    for (every maximal gadget chain g in G) {
        for (every instruction j in g) {
            k = location of first byte position in instruction j in f;
            P_lib[k] = 1;
        }
    }
}
return P_lib;
```

Recall that L denotes the maximum size of the executable part of the libraries. Focusing on one such library, P_{lib} is a binary vector of length L. Correspondingly, P_{obs} is a binary vector of length L from the address clustering detection step.

If both patterns are aligned, the maximum matching can be calculated as the dot product between P_{lib} and P_{obs} according to

$$P_{\text{lib}} \cdot P_{\text{obs}} = \sum_{i=0}^{L-1} P_{\text{lib}}[i] P_{\text{obs}}[i].$$

However, we have no way of knowing if the alignment is correct, so we rather need to try all alignments to see which one produces the highest fit. That is, we need to calculate the dot products for all possible shifts of the two patterns. This can be accomplished by using the Fast Fourier Transform (FFT). The FFT computes the circular discrete convolution c of two vectors a and b of length L,

$$c\,[t] = (a * b_L)\,[t] = \sum_{i=0}^{L-1} a\,[i]\, b\,[(t-i) \mod L]\,. \tag{1}$$

For this to be applicable to our situation, we need to adjust two things. First of all, we need to reverse one of the vectors, say P_{lib}. Secondly, since indices in Eq. (1) are taken modulo L, we need to pad both P_{lib} and P_{obs} with zeros to double length. Without this zero padding, the tails and fronts of the two vectors will contribute to the maximum matching in an undesirable way, effectively bringing more noise into our result.

The FFT approach (see [6]) has time complexity $O(L \lg L)$, compared to $O(L^2)$ for the naïve approach. Letting $\mathcal{F}$ denote the FFT version of the Discrete Fourier Transform (DFT), we may compute c as

$$c = \mathcal{F}^{-1}\left(\mathcal{F}\,(a) \odot \mathcal{F}\,(b)\right),$$

where $\odot$ denotes componentwise multiplication.

We let a and b be the vectors P_{lib} and P_{obs} respectively after the zero padding as described above. The weight of the maximum matching is given as the maximum component of c,

$$c_{\max} = \max_i c\,[i]\,. \tag{2}$$

Note that P_{lib} is known beforehand, so we can precompute $\mathcal{F}\,(a)$ for efficiency. The entire pattern matching procedure is illustrated in Figure 6.

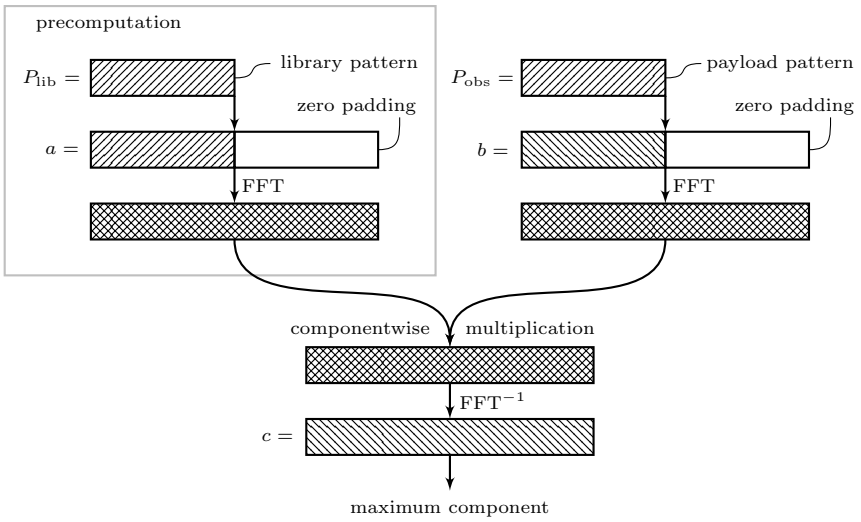

Fig. 6. An overview of FFT pattern matching.

4.4 Statistical Test

The maximum overlap is given by the maximum value of the inverse Fourier transform as given in Eq. (2). In order to find an expression for the number of overlaps we make the following approximations.

- Locations corresponding to gadgets in P_{lib} are uniformly distributed.
- The entries in the convolution vector c are approximated as independent events, all with the same probability.

Using these approximations, the number of overlaps between P_{lib} and P_{obs} is binomially distributed, $X^{(w)} \sim \text{Bin}(w, \frac{G}{L})$, where w and G denote the Hamming weights of P_{obs} and P_{lib}, respectively. Recall that G should here be understood as the number of gadgets in a library for a given entry zone and w is the number of addresses in an address window. Thus, the probability that there are s overlaps is given by

$$\Pr(X^{(w)} = s) = \binom{w}{s} \left(\frac{G}{L}\right)^s \left(1 - \frac{G}{L}\right)^{w-s}, \tag{3}$$

with expected value and variance given by

$$E(X^{(w)}) = \frac{wG}{L},$$
$$V(X^{(w)}) = \frac{wG}{L}(1 - \frac{G}{L}).$$

Since P_{lib} and P_{obs} are convolved, each convolution consists of L such binomially distributed samples. In order to find the probability distribution for the maximum value of the convolution array, we write the probability that any single value is at most s as

$$\Pr(X^{(w)} \leq s) = \sum_{t=0}^{s} \binom{w}{t} \left(\frac{G}{L}\right)^t \left(1 - \frac{G}{L}\right)^{w-t}.$$

The probability that all values are at most s is then, using the second approximation above, $\Pr(X^{(w)} \leq s)^L$. From this it follows that the probability distribution for the maximum value of the convolution vector $c_{\max}^{(w)}$ is given by

$$f_{c_{\max}^{(w)}}(s) = \Pr(c_{\max}^{(w)} = s) = \Pr(X^{(w)} \leq s)^L - \Pr(X^{(w)} \leq s-1)^L \tag{4}$$

with cumulative distribution function

$$F_{c_{\max}^{(w)}}(s) = \Pr(c_{\max}^{(w)} \leq s) = \Pr(X^{(w)} \leq s)^L.$$

Simulations show that the approximations give a distribution that is very close to the real distribution. For detailed numbers, see Table 5 in the Appendix.

A threshold value for $c_{\max}^{(w)}$ is chosen, denoted $\hat{c}_{\max}$. If $c_{\max}^{(w)} \geq \hat{c}_{\max}$ the payload is considered a ROP. Associated with this decision are false positives and false negatives. The false positive rate, denoted α, is defined as the probability that non-malicious data is considered malicious (i.e., a ROP payload) while the false negative rate, denoted β, is the probability that a malicious payload is mistaken for non-malicious data. To write expressions for α and β, let the Hamming weight w of P_{obs} be written as $w = w_G + w_N$, where w_G is the number of ROP gadgets and w_N is the number of noise addresses. The distribution of $c_{\max}^{(w)}$ for non-malicious data is given by Eq. (4). The value of $c_{\max}^{(w)}$ for a ROP payload is given by

$$c_{\max}^{(w)} = w_G + X^{(w_N)},$$

where $X^{(w_N)}$ is distributed according to Eq. (3). Now, we can write the two error probabilities as

$$\begin{aligned} \alpha &= \Pr(c_{\max}^{(w)} \geq \hat{c}_{\max}) = 1 - \Pr(c_{\max}^{(w)} \leq \hat{c}_{\max} - 1) \\ &= 1 - \Pr(X^{(w)} \leq \hat{c}_{\max} - 1)^L \\ \beta &= \Pr(X^{(w_N)} < \hat{c}_{\max} - w_G) = \Pr(X^{(w_N)} \leq \hat{c}_{\max} - w_G - 1) \end{aligned} \tag{5}$$

The false positives rate α is only for one library. If we want to test the payload against a set of ℓ libraries, the total false positive rate α_ℓ is given by

$$\alpha_\ell = 1 - (1 - \alpha)^\ell.$$

By choosing $\alpha = 0.0001$ we allow $\ell = 100$ libraries to be supported, still keeping the total false positive rate α_ℓ below 0.01. (We assume here that all libraries are of approximately equal size.) to be the maximum size of all libraries.) We let the false negative rate $\beta = 0.01$ since this is not affected by multiple libraries (the payload will only match one library). Using these values for α and β allows us to compute the threshold $\hat{c}_{\max}$ and the minimum number of gadgets w_G that are required for successful detection. Table 2 gives these numbers for $z = 3$ and some different choices of w. Results for other values of z can be found in the Appendix. Note that for all values w such that $\hat{c}_{\max} = w_G$, we have $\beta = 0$. Thus we will only obtain false negatives for very large noise values.

Table 2. Threshold $\hat{c}_{\max}$ and minimum number w_G of gadgets needed for ROP payload detection in an address window of weight w. Error rates $\alpha \leq 0.0001$ and $\beta \leq 0.01$. The example library used is libc 2.18 of size $L = 1224144$.

	entity	values								
weight of P_{obs}	w	7	10	15	20	25	30	50	100	200
threshold	$\hat{c}_{\max}$	7	8	9	10	11	12	15	20	27
min num gadgets	w_G	7	8	9	10	11	12	15	20	26

The standard deviation of Eq. (4) turns out to be very small, with almost all probability mass concentrated to only a few values for s. This makes the detection algorithm efficient, allowing us to choose small error rates while still requiring few gadgets to succeed, even in the presence of a large amount of noise.

It can be noted that the required number of gadgets w_N is very close to the threshold $\hat{c}_{\max}$. This is because the expected number of overlaps stemming from noise at a given offset in the convolution is very small. Only when there are relatively many gadgets and/or the noise is large, the noise will be able to contribute enough to relax the requirement on the number of gadgets needed. Thus, as a rule of thumb, the number of gadgets required for successful detection is approximately equal to the threshold

$$w_G \approx \hat{c}_{\max}.$$

The false positive rate has been simulated using the data from Table 4. The simulations indicate that the actual false positive rate is slightly larger than that given by Eq. (5). This is not surprising, since the theoretical model assumes that gadget addresses are uniformly distributed. Due to data redundancy and the proximity coupling between gadgets and return instructions, a slightly larger $c_{\max}^{(w)}$ is expected. Still, according to our simulations, increasing the threshold value by 2 will remove virtually all false positives. This shows that the theoretical model is adequate.

5 Performance

The performance of eavesROP depends on the parameters used in the various stages of the system. All simulations have been performed on an Intel Core i7 4770 @ 3.4 GHz with 16 GB of RAM.

A more aggresive filtering in each step will reduce the amount of data sent to the next stage, which will increase the overall performance. This is illustrated in Table 3, where the throughput and input/output size ratio is given for various types of input data when passed through the data pre-filter.

Table 3. Performance of data pre-filter.

type of data	throughput (MiB/s)	input/output size ratio
random	34.7	0.965
web (HTML, JPG,...)	52.7	0.068
mp3	39.5	0.956
pdf	38.6	0.811
mkv (H.264/MPEG-4)	34.5	0.965

After the optional data pre-filter—which may have reduced the total amount of data—the data is passed to the cluster detection step. This step has a throughput of around 10 MiB/s. The output of the cluster detection step is multiple matched windows, i.e. multiple P_{obs}. Table 4 shows how many P_{obs} vectors that are passed to the pattern matching layer, for some different types of data and different choices for T and D.

Table 4. Number of matching address windows per GiB of input data, for a data window of size D, and with at least T addresses within distance $L = 1224144$, for different types of data. L is here the size of libc 2.18.

	$D = 50$			$D = 200$			$D = 1000$		
type of data	$T = 6$	8	10	$T = 6$	8	10	$T = 6$	8	10
random	0	0	0	12	0	0	24749	53	0
web (HTML, JPG,...)	1795	689	590	5589	1878	1208	40795	8007	3292
mp3	42	8	2	631	106	10	162014	8472	1023
pdf	4068	248	61	34718	5266	1316	1011850	176992	45289
mkv (H.264/MPEG-4)	354	2	0	513	81	66	35545	841	125

Each P_{obs} outputted from the cluster detection stage will be passed to the pattern matching step. Each pattern matching sequence takes roughly 1 second using FFT implemented in software. If necessary, this step could be accelerated using a hardware FFT implementation.

All parts of eavesROP have been implemented and tested using real-world exploits. We are able to detect all exploits of at least 6 gadgets, using a threshold value $\hat{c}_{\max} = 6$, for example [8] and [41].

6 Strengths and Limitations

In this section we enumerate and clarify the different strengths and limitations in our ROP payload detection approach.

6.1 Strengths

An important feature of the detection mechanism is that it works even when ASLR is enabled on the targeted systems, assuming the attacker manages to bypass ASLR through e.g., information leakage or brute force.

While we have described the address cluster detection algorithm for 32-bit systems, it can also be applied to 64-bit systems. In this case brute-force attacks are out of scope due to the large entropy of ASLR, but absolute addresses could still be found using information leakage. The main modification is that the address cluster detection must consider eight offsets instead of four, but in return there will be much fewer addresses passing the cluster detection due to the large address space.

A distinguishing feature is the ability to detect ASLR brute-force attempts. As our detection mechanism only considers differences between gadget addresses, and not the addresses themselves, a probabilistic attack attempt will be detected as a ROP payload.

In the optional data pre-filter, we only apply one very simple UTF8-filter. However, filtering options are abundant, and it is easy to imagine other ad hoc filters which will significantly reduce the computational overhead of the detection.

While most examples have focused on one library, it is very cheap to add support for a large set of libraries. It may also be noted that pattern matching over a set of libraries is inherently parallelizable and can also take advantage of dedicated hardware for computing the FFT.

Last but not least, the modular structure of eavesROP provides a flexible framework that is easily adaptable to the characteristics of the target network. This makes it possible to tailor the system to match the desired detection and performance requirements.

6.2 Limitations

Since we do not have access to the target machine, but only consider a stream of bytes in our search for a ROP payload, the detection mechanism has some limitations.

First, we need to know the libraries and binaries that can be used in an attack. In general, this could be any library or binary, but by choosing the most commonly used ones, it is still possible to detect a significant fraction of attacks. The FFT can be precomputed for each library and the online time for each library will be limited to a componentwise multiplication of two vectors and an inverse FFT computation.

The limited size D of the data window makes it possible to utilize the `ret imm16` instruction which pops `imm16` bytes from the stack. Using these returns with a large `imm16` would leave very sparsely located gadget addresses in the data window. This could potentially avoid detection. However, it should be noted that we only require T gadgets in the window of max payload size so the detection does in general not require all gadgets to succeed.

Since we do not execute any potential exploits, we will not be able to detect exploits that obfuscate the gadget addresses such that they are not visible in the data sent on the

network. Such obfuscation would include e.g., polymorphic ROP attacks [24], or gadget addresses generated on the client-side using JavaScript or ActionScript.

As the FFT is rather computationally intensive, it would also be possible to mount a denial of service attack by sending data that would be interpreted as adjacent addresses, triggering false positives.

7 Related Work

There are several works describing defenses against ROP attacks. They can be categorized using several metrics or properties, see e.g., [11].

The problem of detecting ROP payloads in arbitrary data was previously considered in [32]. In that paper the gadgets were assumed to reside in the non-ASLR code segments. We do not make that assumption and allow the gadgets to reside anywhere in memory. Moreover, in [32] the executable memory segments of a process is needed and potential gadgets are speculatively executed in order to decide whether they are ROP payloads or benign data. We do not require any information regarding a process' memory segments and we do not care about the semantics of a gadget. On the other hand, we need to target a specific set of libraries in which gadgets can be used.

Another detection approach is described in [40], where the authors consider an approach where documents are analyzed to find ROP attacks. The detection assumes that documents are collected and sent to a separate machine, where they are opened in their native application inside a virtual machine. A memory dump of the application is then taken, and the dump is analyzed for ROP payloads by the virtual machine's host. In the same way as eavesROP this does not require any modification of the target machine, and ASLR can be enabled.

ROP is a type of control-flow hijacking attack in which the attacker changes the intended control-flow of the program. Control Flow Integrity (CFI) [1,2] can be used to stop these types of attacks, including ROP. While this is a robust counter-measure for many attacks, properties such as overhead and complexity has limited its adoption. Still, it is a promising mitigation approach which has been given much attention and several aspects of CFI have been considered recently [4,46,47]. While eavesROP is not intended to be as robust as CFI, it has instead a very low complexity and does not need to be implemented on the target systems.

There has also been much work on how to mitigate the source of the exploit, namely the presence of a buffer overrun vulnerability. Well known defenses include StackGuard [14], which is a compiler extension that places a canary value on the stack. Before returning from a function, the integrity of the canary is verified. Stack Shield [42] makes a copy of the return address and saves it in the beginning of the data segment. Before returning from the function, the return address is compared with the saved value. These mitigations have been followed by other compiler based solutions, e.g., PointGuard [13]. An overview and comparison of several buffer overflow prevention implementation and their weaknesses can be found in [38].

Malicious code scanners [21,22,26] have been used to detect malicious code using pattern matching. However, since the ROP payload does not include the malicious data, just addresses to it, they can not be used to detect ROP attacks [44]. Our detection

mechanism looks for the addresses to gadgets and can be seen as an ad hoc mitigation against ROP attacks, based on ideas from malicious code scanners.

8 Conclusions

We have investigated to which extent it is possible to detect a ROP payload by only analysing data, and assuming that ASLR is used on the target system. If we have the set of libraries and binaries that can be used to find gadgets, we show that it is possible to detect a ROP payload even in the presence of noise and by applying suitable data filters and the Fast Fourier Transform the detection has acceptable performance. The exact performance will depend on the type of data and the number of gadgets that are required for an exploit to be detected depends on the maximum allowed size for the payload and the amount of noise.

References

1. M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353. ACM, 2005.
2. M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), 2009.
3. Aleph One. Smashing the stack for fun and profit, phrack, 49, 1996.
4. T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11. ACM, 2011.
5. Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.
6. R. Bracewell. *The Fourier Transform and its Applications*. McGraw-Hill Series in Electrical and Computer Engineering. McGraw-Hill Science/Engineering/Math; 3 edition, June 1999.
7. c0ntex. Bypassing non-executable-stack during exploitation using return-to-libc. Available at: `http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf`.
8. Lorenzo Cantoni. BigAnt Server 2.52 SP5 - SEH Stack Overflow ROP-based exploit (ASLR + DEP bypass). Available at: `http://www.exploit-db.com/exploits/22466/`.
9. Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.
10. P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *Information Systems Security*, volume 5905 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009.
11. Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R.H. Deng. ROPecker: A generic and practical approach for defending against ROP attack. In *NDSS*. Research Collection School Of Information Systems, 2014.
12. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. MIT Press, 2009.

13. C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguardtm: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 91–104. USENIX Association, 2003.
14. C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 63–78. USENIX Association, 1998.
15. L. Davi, A.R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, 2011.
16. T. Durden. Bypassing PaX ASLR protection, phrack, 59, 2002.
17. I. Fratric. Ropguard: Runtime prevention of return-oriented programming attacks, 2012.
18. A. Gupta, S. Kerr, M.S. Kirkpatrick, and E. Bertino. Marlin: Making it harder to fish for gadgets. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12. ACM, 2012.
19. R. Hensing. Understanding DEP as a mitigation technology. Available at: `http://blogs.technet.com/b/srd/archive/2009/06/12/understanding-dep-as-amitigation-technology-part-1.aspx`, 2009.
20. J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J.W. Davidson. Ilr: Where'd my gadgets go? In *Security and Privacy (SP), 2012 IEEE Symposium on*, 2012.
21. H.A Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th Conference on USENIX Security Symposium*. USENIX Association, 2004.
22. C. Kreibich and J. Crowcroft. Honeycomb: Creating intrusion detection signatures using honeypots. *SIGCOMM Computer Communications Review*, 34(1):51–56, Jan 2004.
23. J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10. ACM, 2010.
24. Kangjie Lu, Dabi Zou, Weiping Wen, and Debin Gao. Packed, printable, and polymorphic return-oriented programming. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection*, volume 6961 of *Lecture Notes in Computer Science*, pages 101–120. Springer Berlin Heidelberg, 2011.
25. negux. DEP bypass with ROP. Available at: `http://www.exploit-db.com/exploits/24944/`, 2013.
26. J. Newsome, B. Karp, and D. Song. Polygraph: automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, pages 226–241, May 2005.
27. K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 49–58. ACM, 2010.
28. R. Pagh and F. F. Rodler. Cuckoo hashing. In *Journal of Algorithms*, volume 51, pages 122–144, Duluth, MN, USA, 2004. Academic Press, Inc.
29. V. Pappas, M. Polychronakis, and A.D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2012.
30. V. Pappas, M. Polychronakis, and A.D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, 2013.
31. PaX Team. Address space layout randomization. Available at: `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

32. M. Polychronakis and A.D. Keromytis. ROP payload detection using speculative code execution. In *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, MALWARE '11. IEEE Computer Society, 2011.
33. The PaX project. Available at: `http://pax.grsecurity.net/`, 2000.
34. E.J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of USENIX Security 2011*, 2011.
35. F. J. Serna. CVE-2012-0769, the case of the perfect info leak. Available at: `http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf`, 2009.
36. H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561. ACM, 2007.
37. H. Shacham, M. Page, N. Pfaff, E.J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307. ACM, 2004.
38. P. Silberman and R. Johnson. A comparison of buffer overflow prevention implementations and weaknesses. *IDEFENSE, August*, 2004.
39. K.Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. 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, May 2013.
40. Blaine Stancill, Kevin Z. Snow, Nathan Otterness, Fabian Monrose, Lucas Davi, and Ahmad-Reza Sadeghi. Check my profile: Leveraging static analysis for fast and accurate detection of ROP gadgets. In SalvatoreJ. Stolfo, Angelos Stavrou, and CharlesV. Wright, editors, *Research in Attacks, Intrusions, and Defenses*, volume 8145 of *Lecture Notes in Computer Science*, pages 62–81. Springer Berlin Heidelberg, 2013.
41. Sud0. Audio converter 8.1 0day stack buffer overflow PoC exploit ROP/WPM. Available at: `http://www.exploit-db.com/exploits/13763/`.
42. Vendicator. Stack shield: A "stack smashing" technique protection tool for linux. Available at: `http://www.angelfire.com/sk/stackshield/`, 2000.
43. P. Vreugdenhil. Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. Available at: `http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf`, 2010.
44. X. Wang, C.C. Pan, P. Liu, and S. Zhu. Sigfree: A signature-free buffer overflow attack blocker. *IEEE Transactions on Dependable and Secure Computing*, 7(1):65–79, Jan 2010.
45. R. Wartell, V. Mohan, K.W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.
46. C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13. IEEE Computer Society, 2013.
47. M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13. USENIX Association, 2013.

A Histogram overlap count

Table 5 shows the simulated and theoretical distributions for $c_{\max}^{(w)}$ when using random data. The theoretical distribution is given by (4) and the simulations are performed by taking 10,000 arrays (P_{obs}) with Hamming weight w, uniformly distributed over the array. Using FFT, the maximum overlap between the array and an array corresponding to libc with entry zone 3 is computed. A histogram for the 10,000 samples is used to illustrate the probability distribution.

Table 5. Histogram comparison of maximum overlap ($c_{\max}^{(w)}$) for various address window weights over random data and libc. Entry zone size is 3 instructions, sample size 10,000.

weight w																		
10	overlap	0	1	2	3	4	5	6	7	8	9	10						
	simulation	0	0	0	0	1672	7864	458	6	0	0	0						
	theory	0	0	0	0	22	8559	1392	26	0	0	0						
20	overlap	0	1	2	3	4	5	6	7	8	9	10	11	12	...	19	20	
	simulation	0	0	0	0	0	43	6815	2957	176	8	1	0	0	...	0	0	
	theory	0	0	0	0	0	0	2686	6691	597	24	1	0	0	...	0	0	
30	overlap	0	1	2	3	4	5	6	7	8	9	10	11	12	13	...	29	30
	simulation	0	0	0	0	0	0	30	5473	4077	392	26	2	0	0	...	0	0
	theory	0	0	0	0	0	0	0	1007	7441	1446	100	6	0	0	...	0	0
40	overlap	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...	39	40
	simulation	0	0	0	0	0	0	0	52	5409	4026	474	36	3	0	0	...	0	0
	theory	0	0	0	0	0	0	0	0	847	7100	1866	173	12	1	0	...	0	0
50	overlap	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	49	50
	simulation	0	0	0	0	0	0	0	0	240	5886	3343	487	41	3	0	0	...	0	0
	theory	0	0	0	0	0	0	0	0	0	1192	6736	1847	204	18	1	0	...	0	0

B Entry-zone

Table 6 gives the required threshold $\hat{c}_{\max}$ for a given weight w of P_{obs} when error rates are fixed to $\alpha \leq 0.0001$ and $\beta \leq 0.01$.

Table 6. Threshold $\hat{c}_{\max}$ and minimum number w_G of gadgets needed for ROP payload detection in an address window of weight w. Error rates $\alpha \leq 0.0001$ and $\beta \leq 0.01$.

entry zone	entity	values								
1	w	6	10	15	20	25	30	50	100	200
	$\hat{c}_{\max}$	6	7	7	8	9	9	11	13	17
	w_G	6	7	7	8	9	9	11	13	17
3	w	7	10	15	20	25	30	50	100	200
	$\hat{c}_{\max}$	7	8	9	10	11	12	15	20	27
	w_G	7	8	9	10	11	12	15	20	26
5	w	8	10	15	20	25	30	50	100	200
	$\hat{c}_{\max}$	8	9	11	12	13	14	17	24	35
	w_G	8	9	11	12	13	14	17	24	33
7	w	9	10	15	20	25	30	50	100	200
	$\hat{c}_{\max}$	9	10	11	13	14	15	19	27	40
	w_G	9	10	11	13	14	15	19	26	36