Statistical Measurement of Production Environment Influence on Code Reuse Availability

Étienne Louboutin, Jean-Christophe Bach, Fabien Dagnat

Statistical Measurement of Production Environment Influence on Code Reuse Availability

Étienne Louboutin
Ecole Navale, IMT Atlantique
F-29240 Brest, France
Email: etienne.louboutin@imt-atlantique.fr

Jean-Christophe Bach, Fabien Dagnat
IMT Atlantique, Lab-STICC, UMR6285
F-29238 Brest, France
Email: {jc.bach,fabien.dagnat}@imt-atlantique.fr

Abstract—**Return-oriented-programming is widely used for software exploits, and ten years after its academic description, little to no protection is deployed most of the time. Performance trade-offs or insufficient protection often results in no protection deployment. Address space layout randomisation is a basic protection that just increases the complexity of writing attacks but does not prevent code-reuse exploits. Its overhead is negligible enough to justify its deployment. These protections come after software development, and are implemented in the compiler or via binary modification. Usually, each binary is either critical and protected or not critical and not protected. This decision results from a usage criterion, like `gzip`, or if it exposes network interfaces, like `apache`. In this paper, we go through multiple views to expose elements that make it possible to compare binaries with respect to their available code-reuse components. We look at these elements to underline what part of the production process of a binary can increase or decrease its quantitative inclusion of code reuse components. With this evaluation, we expose certain disparities introduced by production tools, by the language used to write applications or even because of the targeted platform. We also show how hardware architectures affect this statistical measurement.**

Keywords–Return-Oriented-Programming; ROP; Code-reuse exploits.

I. Introduction

With hardware protection against code injection, software exploitation is widely based on code reuse. Starting with return-to-libc then generalised with Return Oriented Programming (ROP) [1], the class of code reuse attacks allows an intruder to recreate any arbitrary program by hijacking the control flow of a host application. To construct such an exploit, the needed instructions have to be found in the target binary. A group of instructions used during such a hijack is called a gadget. A more precise description of a gadget is provided in Section II. Address Space Layout Randomisation (ASLR) makes such a task more difficult, since it allows memory layout to be randomised when an application is started. The search of useful instructions must be done at runtime. But ASLR has been shown as not to be efficient enough for full protection, and can be bypassed, for example with blind ROP [2]. In a common playground, such as a JavaScript jail in a browser, process memory reading cannot be prevented and ASLR becomes less effective, as shown for example in the Spectre exploit [3].

For x86, solutions have been proposed [4]–[6] to protect an application against these attacks. These solutions guarantee that the execution will follow a legitimate control flow. However, they introduce more overhead in execution time than what can be accepted for general-purpose programs. Protecting only the relevant part of a program is an appealing way to reduce the induced overhead, which is done to some extents in [6]. The authors propose different selection criteria for level of protected code pointers and arbitrary jumps, giving some trade-off between performance and security.

Despite all this work, we lack a way to measure the effectiveness of these types of protection on security. It is difficult to compare two binaries, protected or not, with regard to a notion of ROP-class sensitivity. From a performance perspective, unified benchmarks are commonly used to evaluate the costs induced by the deployment of a protection. The efficiency of a given protection is more difficult to measure. Nothing exists to measure effectiveness beyond trying to write exploits, manually or with human intervention and this is hardly scalable. We develop this idea in Section III.

More generally, during the creation of an application, a lot of choices must be made. For example, we have to choose the language to write our application, the operating system it will be deployed on and, in some case, the hardware the application will support. The influence of such choices on the availability of control flow hijacks in the final binary is not known.

Brown *et al.* [7] have highlighted how debloating tools affect sensitivity to control flow hijack. They have shown that using the gadget number is not enough to define a security metric. Furthermore, they propose a binary production process that relies on a human to validate a significant security improvement. Such a process clearly does not scale and cannot be integrated in a software-automated build process.

While debloating influences the available gadgets in an application, this is probably the case as well for other steps of binary production. We want to know which tool or technology choices (compiler, language, etc.) have an effect, positive or negative, on these available gadgets. Instead of finding and exploring any combination of possible production tools for an application, we chose to select a wide range of systems. On those systems, we analyse available binaries to see if we can characterise the production process by the resulting available gadgets. We explore how different production setups affect the notion of gadget density. We investigate what is available to an attacker to craft a control flow hijack payload and what characterises a given binary with regard to code reuse exploits. The objective is to extract information from deployed binaries on living systems to provide recommendations for building applications that are more resistant to code reuse attacks. We study how the target execution environment for binaries affects the quantity and diversity of elements presented to write an exploit.

Then, the influence of environment on the quantitative measurement of code reuse availability is identified. The goal is to characterise how different steps of a process, from software creation to execution, could be leveraged to reduce the number of ROP components exposed by an application. This paper presents the method we used to define this quantitative measurement, which allows us to distinguish which binaries provide the most elements to write code reuse exploits.

This article starts by explaining how control flow hijack attacks are written in Section II, with a focus on the basic elements that compose these attacks. We then continue with the methodology used to retrieve these gadgets from binaries in Section IV. In Section V, we explore how gadgets are distributed among analysed binaries, in order to identify those that are used most often and study their diversity. Then, we analyse the disparity in binaries, given their hardware architecture or system environment (for example, the runtime Linux distribution). In Section V-D, we highlight which binary production steps influence the availability of control flow hijack components. In the end, we also show that a ROP chain crafted to target a given application built on two similar systems is unlikely to work on both.

II. Control flow hijacking

The idea behind control flow hijacking is the use of hardware processor operational behaviour to trick it out of the normal flow. One known method to hijack control flow is ROP, first described by Shacham [1]. ROP is a paradigm which allows generating a completely new application using the existing set of instructions of a given software. Exploits written with ROP need an entry point to start the attack, as detailed at the end of this section, in the threat model paragraph. Examples given in this section use the x86 family architecture but other architectures can be targeted by these attacks.

A program can be decomposed into multiple sequences of instructions linked by control flow instructions defining where the execution continues. These instructions can be function calls, system calls, jumps or returns.

During a control flow hijack, addresses used by control flow instructions are corrupted to divert the flow toward `libc` functions – for `return-to-libc` attacks. Return oriented programming uses small subsets of available code instead of full functions. These subsets are called gadgets. For example, `mov rdi, qword ptr [rbx]; call rdi` is a gadget found in some x86_64 applications.

ROP is the construction of an application by chaining gadgets together, effectively using only present and legitimate code. A ROP chain is created by corrupting the memory with a sequence of addresses pointing toward such elements. If the execution stack is corrupted, a `return` instruction is used to chain the gadgets. On hardware architecture without this `return` instruction, other instructions are used to build similar hijacks of a program execution. These constructions are shown for x86 and SPARC in [8] and for ARM in [9].

While the term ROP is used only for `return`-terminated gadgets, COP (Call-Oriented Programming) is for `call`-terminated ones and JOP (Jump-Oriented Programming) is for `jmp`-terminated ones. We also consider system call gadgets in our statistical analysis, amongst all other gadgets. Non-control data flow hijacks are out of the scope of this paper.

Threat Model: In the context of an attack following the ROP paradigm, few basic hypotheses are made on what intruders can do. The capabilities given to them are arbitrarily read in the process binary, which is not a strong hypothesis. For writing, we assume W XOR X is enforced, meaning that writing and executing are mutually exclusive. We also make the assumption that the executable part of a given program cannot be corrupted, but any write that does not violate this property is available. We also assume that a memory corruption allowing a ROP chain execution to be started is available. If the application is written in a memory-safe language, a side channel attack – either hardware or software – can be used to initiate the chain. We also make the assumption that attackers have an idea of what they attack, and have some knowledge on which gadgets they can find, expecting that hardware architecture is known.

III. Related work

In a first approach, we looked for a measurement of ROP effectiveness, apart from Turing-completeness of the set of gadgets found, which has been demonstrated if code base is sufficient enough, like the standard C library [8]. The objective here is to look at how different protections measure their results, not for performance overhead introduced but regarding ROP gadget availability.

Schwartz *et al.* proposed Q [10], a tool that hardens any ROP exploit to be resistant to ASLR. The tool effectiveness is proven by testing on which program they can construct a chain. Based on semantic analysis of gadgets without side effects, they managed to construct a chain automatically on a large set of `/usr/bin` of a given Linux system. However, the only metric that is used to measure the sensitivity of a given binary is the success of their tool to craft a chain. They also provide a statistical study on semantic gadgets available in their surveyed binaries, with just a short discussion.

Dullien *et al.* proposed another tool to look for gadgets in cross-platform environments [11]. The effectiveness of their solution is demonstrated by checking the Turing-completeness of the gadget set found in one binary on three different ARM platforms. The chosen binary is a core library linked with most applications and no other measurement is proposed.

Keromytis *et al.* published a protection against ROP [12] on a part of the binary. The published tool is evaluated in terms of both performance and security. The efficiency of the protection is tested using known software to create ROP payloads and gadgets, Q [10] and Mona [13]. They used two different results to evaluate the effectiveness of the protection. The first one is the ability to craft a chain automatically with these tools on the protected binary. The second one is the count of gadgets which were found by the two softwares and which are removed in the protected part of the binary.

In a similar way, published protections like [14], [15] or debloating techniques [16] often use either automatic crafting failure as an effectiveness measurement, with either Q [10], ROPgadget [17] or a custom tool. The default crafted chain is a shell spawning, but the failure or success of the craft on a given binary does not provide much information about its protection. Another chain, which brings as much harm, could be potentially (hand)crafted without being detected by this method of evaluation. The second method used to evaluate protections is the enumeration of available gadgets and reduction observed before and after the protection in question is applied. To do so, either Q output is used or custom processes are built. Even when a common tool is used, methods of comparison differ, implying some lack of common ground with respect to security benchmarks.

IV. Gadget density measurement

A gadget is a sequence of instructions terminated by a jump, as defined earlier. For our analysis, a gadget is at most five instructions long, including the jump, following [8]. Furthermore Homescu *et al.* [18] have shown that one-byte instructions can be enough to achieve Turing-completeness. Therefore, all subsequences of a gadget are relevant. So for a five-instruction long gadget, all sub-gadgets of one up to five instructions are counted as different gadgets. As a result, each control flow instruction can generate up to five different gadgets. A gadget of size one is limited to a control flow instruction. For example, `call rdi;` can be used alone if a preceding gadget in the chain already set the content of the register `rdi` to a desired value. We also want to keep them for checking control flow instruction diversity. The basic `ret` instructions are not considered, as such gadgets are not relevant (i.e., they do nothing).

In this article, gadget classification is purely based on opcodes. Two gadgets that are similar semantically but different syntactically are considered distinct in the measurement. For example, `pop rax; ret` and `pop rbx; ret` are distinct.

The semantic analysis of gadgets is outside of the scope of this study. It has been demonstrated [10] that some arithmetic gadgets can be chained in order to create missing stores, load or any other needed gadgets. All gadgets are treated equally in the scope of this analysis whether they produce side effects or are just not usable at all.

Different metrics are used in this comparison. To fairly compare binaries of different sizes, the number of gadgets found in a binary is normalised with the size of its executable section. We define two densities with these measurements: unique gadget density, which represents the number of distinct gadgets present in a binary, and total gadget density, which includes all occurrences of each gadget in an executable file.

These metrics are used to identify how easy it is to attack a binary using control flow hijacking, given its production context. The context is composed of hardware architecture at first, completed by its target environment and its creation process. This creation process is decomposed to identify the role that each step plays in the evolution of the gadget density of a binary.

Methodology: The search for gadgets in a binary is done using a tool, ROPgadget [17], based on the library `Capstone` for assembly parsing. Even though some architectures are not supported by the highest-level tool we used, adding support for more instructions in the future is not excluded, as the library supports many more hardware platforms. On top of that, we have developed software for data aggregation and process automation, which enables an easy process to integrate more binaries for analysis, or to extract some data subsets.

V. Experiments

This section contains an analysis of how gadgets are distributed in a binary, and of how different binaries react regarding the characteristics of gadgets exposed to an attacker.

Analysed binaries come from different Linux distributions and platforms. All binaries from `/usr/bin` are used. Most systems have various applications installed, from system utilities to window managers or web servers. We tried to have a lot of diversity in target usage of these applications to avoid any bias that may exist, for example due to applications needing more I/O accesses. All analysed binaries are the executables without their libraries, except if they are statically linked. If a program is statically compiled, we did not try to isolate what comes from the linked libraries and what is specific to the application, and we analyse it as a single binary. We excluded any standard libraries in this analysis, and other dynamically linked libraries. Given that `libc` and other libraries are so heavy, we assume that if someone wants to protect an application, they will either build without relying on such libraries, or will use static compilation. We focus on binary specific gadgets, to avoid an already known Turing-complete set.

The Linux distributions and platforms analysed – with the snapshot date when relevant – are the following: Fedora 26 (2018-01-19), Ubuntu 16.04 (2017-10-10), Debian Testing (2018-01-15), Debian 9.3 (2018-01-12), Gentoo (2017-09-07), Arch (2017-11-23), Buildroot, RISC & CISC (2018-02-05), Tar 1.30 (multiple compilation flag combinations) and Firefox (all release versions from 4.0 to 65.0b9).

Around 10 500 binaries went through this process, which was run on a dedicated platform. The dataset was created on an Intel® Xeon® CPU E5-2640 clocked at 2.4 GHz. Parsing the whole set of binaries takes around 48 hours to complete.

A. Binary Gadget Density Measurement

There is a correlation between the size of a binary and its number of gadgets. As shown in Figure 1, both unique gadgets and total gadgets increase quite linearly on a log scale with the size of the executable section (ES) of a binary. From really small binaries (less than 1 kB of ES), to large ones (more than 10 MB of ES), executable size is correlated with new gadgets, despite greater variation on small binaries. Analysed binaries present non-negligible variations in gadget count, in a window which is not correlated with ES size.

The interesting part is that while increasing in size, the number of unique gadgets keeps increasing. Intuitively, most gadgets would already be present in a binary when ES is above a certain threshold, and a slower increase or stagnation would be observed. Such behaviour is not observed, as binaries keep introducing new gadgets regularly as they grow in size. A normalised number of gadgets relative to its ES, in kB, has

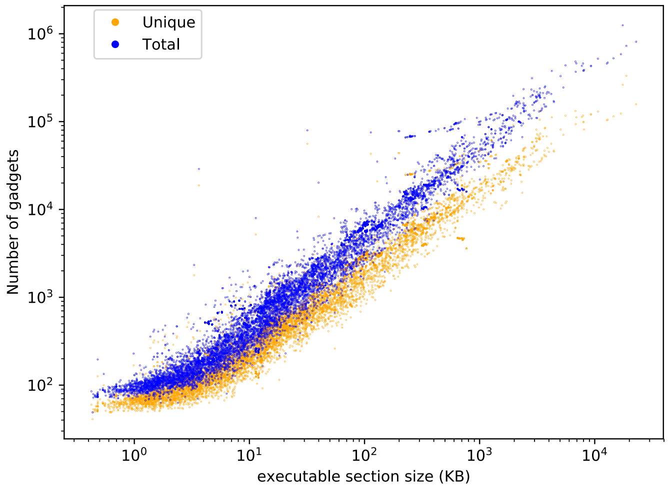

Figure 1. Gadgets found depending on the size of the executable sections

been taken as a first quantitative indicator for comparison of multiple binaries regarding ROP-class availability.

Both these total and unique gadget densities increase with size, as shown in Figure 2. Two tendencies stand out. First, a group of binaries has a ratio between the two densities around 1, meaning each gadget is rarely present more than once. Second, the binaries have a ratio that increases with their ES size. Amongst all these binaries, some have an extreme ratio, like `Quasselcore`, coming from Gentoo Linux, with a ratio of 6.2. The most impressive one is `gregorio`, from Arch Linux, which has a ratio of 8.89, around 3 times the average of the binaries with similar ES size. For example, coming from a different platform, ARM32, `grep` has a ratio of 1.17. Such a disparity in density ratio is limited to neither architecture change nor binary size. For instance, a gap is observed between `clang` (Arch Linux, x86_64), and `darcs` (Ubuntu 16.04, x86_64) with 5.2 and 2.2 for 23 MB and 19 MB of ES size, respectively.

On the code reuse availability that a binary could present, all binaries are not equal regarding what they provide to an attacker to craft an exploit. There are some extreme cases, but the global distribution shows a lot of binaries outside common trend. Moreover, an application can provide more or fewer gadgets. For example, `screen`, a binary available on all Linux systems analysed, does not have the same number of gadgets, either total or unique, and even has a changing ratio. On three architectures (SPARC v8 and leon, ARM32), the density ratio of this binary is around 1.20, 2.1 for ARM64, while it is around 2.50 on i386 & x86_64 architectures. These disparities are explored in the following sections, to observe how either system, in Section V-D, or hardware targets, in Section V-C, affect software on its control flow hijack elements.

B. Gadget representation

This section studies the diversity of gadgets amongst binaries on the x86_64 architecture. This architecture is selected because it represents 10 100 binaries out of 10 516. The goal is to determine how different a ROP chain would have to be, to execute the same attack on two distinct binaries.

Most available gadgets across all binaries are just manipulations that pop stack values into registers, *i.e.*, **`pop`** `r14;` **`pop`** `r15;` `ret`. Such gadgets are available in most binaries. There is only around 1 % of our binaries that do not contain these gadgets. Most gadgets are not shared amongst the binaries, with only 16 % of them in 2 or more binaries.

In our sample, some binaries expose a lot of the same gadgets. One gadget is used extensively by a few binaries, by a large margin compared to the other most popular gadgets. This gadget comes from binaries that share an interesting property, namely they are written with the Qt framework (like `Quasselcore`).

To evaluate the predominance of ROP gadgets in a binary compared to other kinds of gadgets, we isolated the ones that are terminated by any instruction except `return`. An excerpt of the results is shown in Table I. For this categorisation of gadgets, few stand out for their statistical usage. The two most used gadgets come exclusively from binaries of programs written in the programming language OCaml, with an above-average frequency. No C/C++ compiler generated these gadgets, nor did any other compiler used to produce one of the analysed binaries. We identified some but not all compilers used to generate our binaries, including `go`, `rust`, `gcc` or `clang`. This gadget can be used as a signature of OCaml binaries in our dataset.

The presence of these particular behaviours is not limited to these compilers and libraries. In a first attempt to identify software engineering choices influencing the gadget density, the production process of `tar` and a small graphical game have been modified to ensure a given compilation option set. Two compilers were used (`gcc` and `clang`) with the usual options. In this limited sample, the influence of a compilation option on gadget density does not depend on the compiled application. The effect of a given option has a similar effect on both applications. The choice of a compiler did result in different behaviour. This implies that the compilation process (the choice of the compiler and its options) cannot be neglected when designing an application to be less sensitive to code reuse, and should not depend on the application.

Since these results only concern the x86_64 architecture, it is planned to check whether such behaviour can be observed with other hardware architectures. For now, our sample size of binaries from other architectures is too small and too specific to be relevant for a comparison with x86_64 figures.

C. Hardware Influence

This section shows the impact that hardware architecture has on different measurements of gadgets in binaries. In Section V-B, we explained that x86_64 binaries were overrepresented. Therefore some bias may exist in the given results.

TABLE I. MOST USED GADGETS NOT TERMINATED BY A RETURN

Gadget	Nb of occurrence	Nb binaries	Avg Occurrence
`mov` `rdi,` **`qword ptr`** `[rbx] ;` **`call`** `rdi`	43 737	54	809.94
`mov` `edi,` **`dword ptr`** `[rbx] ;` **`call`** `rdi`	43 202	54	800.04
`mov` `rdi, rbx ;` **`call`** `rax`	22 559	698	32.32
`mov` `edi, ebx ;` **`call`** `rax`	22 536	786	28.67
`add` `eax, edx ;` **`jmp`** `rax`	16 770	1173	14.30
`add` `rax, rdx ;` **`jmp`** `rax`	16 624	1168	14.23
`add` `edx, eax ;` **`jmp`** `rdx`	15 132	455	33.26
`add` `rdx, rax ;` **`jmp`** `rdx`	15 003	398	37.70

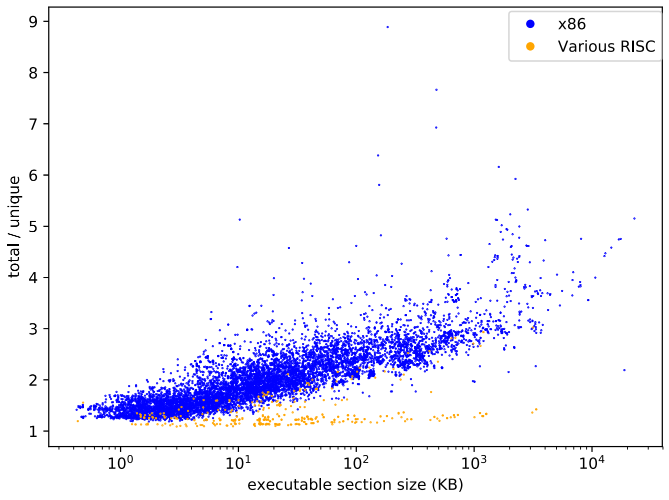

Figure 2. Ratio of total to unique gadgets by hardware architecture

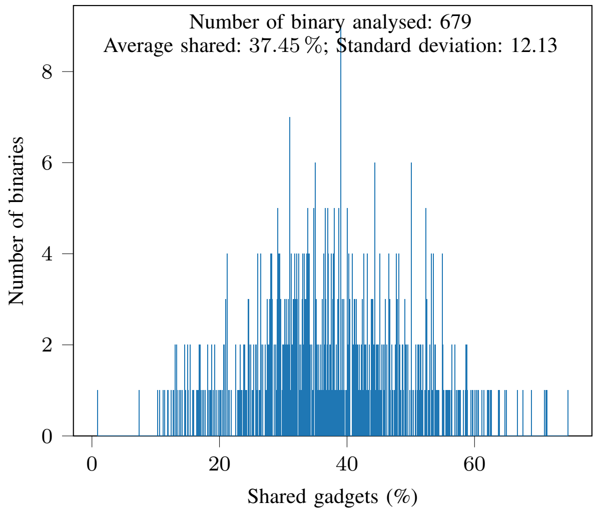

Figure 3. Percentage of shared gadgets on both Fedora and Ubuntu

Figure 2 presents the ratio between total and unique gadgets for all 10 516 binaries. It uses two colours to highlight the differences between the type of architectures, RISC –ARM (32 & 64 bits), SPARCv8– and CISC –x86 (32 & 64 bits). The main difference in behaviour between the two groups is the diversity of gadgets in RISC binaries. They rarely have a ratio above two, and the ratio does not increase with ES size. Binaries that target x86 family platform have a higher variance in gadget density ratio. The bigger the x86_64 binary ES size, the more often their gadgets appear.

`Buildroot` is a Linux distribution which targets embedded systems. A system can be built with a list of packages with a constant configuration from one hardware architecture to another. With a fixed compiler and its options, the only parameter that differs from build to build is the target architecture. Binaries are compared respectively on each architecture, to observe the differences in density.

Comparing these `Buildroot`, few differences are present between architectures. Sometimes the average of unique gadget density displayed by a system is more than twice the density on its counterpart, like between i386 and ARM32. An interesting result is that architecture family is not sufficient to classify a system with respect to its ROP availability. For example, even if there are more unique gadgets on i386 than ARM32, the comparison between their 64-bit counterpart gives an opposite result, with a lot more on ARM64. The availability of more instructions on ARM64 may be the reason for this evolution, but this is only a hypothesis that has to be explored further.

Even if what causes such behaviour is not known, it is certain that hardware architecture has a great effect on diversity and density in the quantitative measurement of ROP availability on these systems. Since gadgets cannot be compared across architectures on a syntax basis, semantics would be required to expand the discussion on hardware influence.

D. Disparities in deployment environments

In some Linux communities, security is the main focus, while in other performance or stability are privileged. This influences the choice of an application version and the production process of the binaries. For instance, in Fedora and Red Hat Enterprise [19], there are recommendations on which `gcc` compilation flags to use to publish a package. This section looks into different distributions to identify the influence that it can have on binary gadget density. We chose different types of distributions. Some use rolling release, where each application is updated with upstream updates, and others are stable or on a slow update cycle. We added some initially uninstalled packages to have better coverage for comparison.

For the purpose of comparison, only the content of `/usr/bin` is used in the five distributions. To compare two distributions, a binary is searched on both, and if it exists on only one, it is discarded from the result. If it is available on both distributions, the density of unique and total gadgets are compared. Then results are aggregated.

On average, only a small deviation can be observed between two distributions, around a few percent, and standard deviation between 25 and 30 percentage points increase or reduction in density on two distributions. Despite the majority of binaries showing little to no differences in density between two distributions, others bring up more questions, with up to three times more gadget density in one environment.

In some cases, there is only a small difference in version, like minor version or just a distribution patch that differs. On top of that, compilation flags and compilers can also vary between distributions. A lot of different behaviours are observed, since there is a huge sample of contributors involved in development and packaging. Compilation flags can be chosen for technical reasons, or due to distribution directives given to maintainers, or even users who change these themselves.

Having high average differences in density between two Linux distributions is not sufficient to evaluate the usability of a crafted chain on a similar binary from one to the other. A program available on both systems may contain different gadgets even if it exposes a similar density. Figure 3 presents the extent to which a Fedora desktop distribution and its Ubuntu counterpart share gadgets.

First, with only 37 % (around 600 gadgets) of shared gadgets on average between two distributions for a binary, the ability to create an easily distributable chain becomes compromised. An attack would probably have to use different gadgets for each target distribution. The low re-usability of a given chain is reinforced by the fact that only a few binaries between systems have up to 75 % shared gadgets, and way more than half of those analysed do not even reach 50 %. Few variations in a software production process can result in enough variations to reduce the portability of a code reuse exploit.

E. Results

These experiments have shown multiple parameters that have a significant influence on gadget density. A wide disparity of density amongst binaries is observed. Interestingly, the bigger a binary gets, the more unique gadgets it has.

We have also seen that despite gadgets being widely available on all hardware architectures, gadget availability is influenced by these architectures. Specifically, RISC architectures tend to have a gadget available only once per binary.

The results also highlighted the fact that the whole production environment has a role in the creation of gadgets. Some impactful steps are, amongst others, the choice of the source language, its compiler and the compilation options. This step can reduce or increase density, but most importantly they affect the variety and type of gadgets one can find in a binary. This fact is reinforced by the disparity in gadgets found in two binaries from the same application produced for distinct environments, shown in the last section. As a consequence, it is very unlikely to be able to port a given ROP chain at a low cost from a system to another.

VI. Conclusion and Future Work

In this paper, using a statistical analysis, we highlighted the influence of the binary production process on the number and density of gadgets. While this does not provide a direct security metric, it shows that code reuse has to be taken into account at an early stage of application design. Understanding what impacts the number of gadgets may lead to better protection, a more suitable protection, or one with less overhead.

The next step to expand this work is to consider the semantics of gadgets, to check whether each design decision has the same effect on density for gadgets with similar semantics. We have started to work on a measurement of semantic diversity, to ease the comparison of binaries. We also plan to improve the dataset of analysed programs, with more diverse source languages, and complete the dataset with more RISC binaries. We have seen limitations of our dataset with respect to hardware architecture in Section V-C, with the over-representation of x86_64. The diversity of gadgets has been inspected only with this architecture. We will expand this analysis to other available architectures. We plan to complete the study on compilation options, compilers and languages, started in Section V-B, with, for example, the addition of applications written in a memory-safe language like `rust`. The enhancement of the dataset is important to confirm what is shown on a small scale here. It would help confirm that other systems (POSIX compliant or not) behave differently than those in Section V-D. With enough information on what impacts the number of gadgets, an application could be built with guidelines to become less sensitive to code reuse exploits.

Acknowledgments

This research is supported by the Chair of Naval Cyber Defense, funded and supported by Ecole navale, ENSTA Bretagne, IMT Atlantique, Thales and Navale Group.

We also thank Pierre-Henri Horrein for his support during his work at IMT Atlantique.

References

[1] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in Proceedings of the 14th ACM Conference on Computer and Communications Security, ser. CCS ’07. New York, NY, USA: ACM, 2007, pp. 552–561.

[2] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, “Hacking blind,” in 2014 IEEE Symposium on Security and Privacy, May 2014, pp. 227–242.

[3] P. K. et al., “Spectre attacks: Exploiting speculative execution,” CoRR, vol. abs/1801.01203, 2018.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity,” in ACM Conference on Computer and Communication Security (CCS), Alexandria, VA, November 2005, pp. 340–353.

[5] T. Coudray, A. Fontaine, and P. Chifflier, “Picon: Control flow integrity on LLVM IR,” in Symposium sur la sécurité des technologies de l’information et des communications (SSTIC), 2015.

[6] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity.” in OSDI, vol. 14, 2014.

[7] M. D. Brown and S. Pande, “Is less really more? why reducing code reuse gadget counts via software debloating doesn’t necessarily lead to better security,” arXiv preprint arXiv:1902.10880, 2019.

[8] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” ACM Trans. Inf. Syst. Secur., vol. 15, no. 1, Mar. 2012, pp. 2:1–2:34.

[9] T. Kornau, “Return oriented programming for the arm architecture,” Master’s Thesis, Ruhr-Universität Bochum, 2010.

[10] E. J. Schwartz, T. Avgerinos, and D. Brumley, “Q: Exploit hardening made easy,” in USENIX Security Symposium, 2011.

[11] T. Dullien, T. Kornau, and R.-P. Weinmann, “A framework for automated architecture-independent gadget search,” in Proceedings of the 4th USENIX Conference on Offensive Technologies. Berkeley, CA, USA: USENIX Association, 2010.

[12] V. Pappas, M. Polychronakis, and A. D. Keromytis, “Smashing the gadgets: Hindering return-oriented programming using in-place code randomization,” in 2012 IEEE Symposium on Security and Privacy, May 2012, pp. 601–615.

[13] Corelan team, “Mona,” 2013, https://github.com/corelan/mona (last accessed on 18/09/2019).

[14] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, “Ilr: Where’d my gadgets go?” in 2012 IEEE Symposium on Security and Privacy, May 2012, pp. 571–585.

[15] X. Chen, H. Bos, and C. Giuffrida, “Codearmor: Virtualizing the code space to counter disclosure attacks,” in Security and Privacy (EuroS&P), IEEE European Symposium on. IEEE, 2017, pp. 514–529.

[16] G. Mururu, C. Porter, P. Barua, and S. Pande, “Binary debloating for security via demand driven loading,” arXiv preprint arXiv:1902.06570, 2019.

[17] J. Salwan, “ROPgadget tool,” 2012, http://shell-storm.org/project/ROPgadget (last accessed on 18/09/2019).

[18] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, “Microgadgets: Size does matter in turing-complete return-oriented programming,” in Proceedings of the 6th USENIX Conference on Offensive Technologies, ser. WOOT’12. Berkeley, CA, USA: USENIX Association, 2012, pp. 7–7.

[19] F. Weimer, “Recommended compiler and linker flags for gcc,” https://developers.redhat.com/blog/2018/03/21/compiler-and-linker-flags-gcc (last accessed on 18/09/2019).