GLYPH: Efficient ML-based Detection of Heap Spraying Attacks

Fabio Pierazzi*, Stefano Cristalli†, Danilo Bruschi†, Michele Colajanni‡, Mirco Marchetti‡, Andrea Lanzi†
*King's College London, UK – fabio.pierazzi@kcl.ac.uk
†University of Milan, Italy – {stefano.cristalli, danilo.bruschi, andrea.lanzi}@unimi.it
‡University of Modena and Reggio Emilia, Italy – {michele.colajanni, mirco.marchetti}@unimore.it

***Abstract*—Heap spraying is probably the most simple and effective memory corruption attack, which fills the memory with malicious payloads and then jumps at a random location in hopes of starting the attacker's routines. To counter this threat, GRAFFITI has been recently proposed as the first OS-agnostic framework for monitoring memory allocations of arbitrary applications at runtime; however, the main contributions of GRAFFITI are on the monitoring system, and its detection engine only considers simple heuristics which are tailored to certain attack vectors and are easily evaded. In this paper, we aim to overcome this limitation and propose GLYPH as the first ML-based heap spraying detection system, which is designed to be effective, efficient, and resilient to evasive attackers. GLYPH relies on the information monitored by GRAFFITI, and we investigate the effectiveness of different feature spaces based on information entropy and memory n-grams, and discuss the several engineering challenges we have faced to make GLYPH efficient with an overhead compatible with that of GRAFFITI. To evaluate GLYPH, we build a representative dataset with several variants of heap spraying attacks, and assess GLYPH's resilience against evasive attackers through selective hold-out experiments. Results show that GLYPH achieves high accuracy in detecting spraying and is able to generalize well, outperforming the state-of-the-art approach for heap spraying detection, NOZZLE. Finally, we thoroughly discuss the trade-offs between detection performance and runtime overhead of GLYPH's different configurations.**

Keywords*—heap spraying; memory exploitation; machine learning; memory monitoring; detection.

I. INTRODUCTION

Memory corruption vulnerabilities are currently one of the biggest threats to software and information security. In this field, we have witnessed a constant arms race over the past decade, with system designers of compilers and operating systems on one side, and attackers on the other. Over the years, the former have introduced many new security features to increase the complexity of exploiting memory corruption vulnerabilities [8, 13, 46, 47, 57]. This list includes stack canaries [18], data execution prevention (DEP), Address Space Layout Randomization (ASLR) [9, 34], and Control Flow Integrity [3] just to cite some of the most popular solutions.

ASLR is certainly one of the most common and successful techniques adopted by modern operating systems due to its relatively high performance and low overhead. Among several attacks against such a defense mechanism, the most simple and effective one is to fill the memory with tens of thousands of identical copies of the same malicious code, and then jump to a random memory page, hoping to land in one of the pre-loaded memory areas. This makes this payload delivery technique, called *spraying*, one of the key elements used in most of the recent memory corruption exploits [23, 26].

Researchers have been looking for approaches to mitigate this technique. Unfortunately, the few solutions proposed so far [e.g., 22, 24, 53] were all tailored to defend a particular application (typically the JavaScript interpreter in Internet Explorer) using a given memory allocator in a specific operating system, and against a single form of heap spraying. This made these solutions difficult to port to other environments, and unable to cope with all possible variations of heap spraying attacks. In fact, the original heap spraying attack is now just the tip of the iceberg. The technique has rapidly evolved in different directions, for example by taking advantage of Just In Time (JIT) compilers [26], focusing on the allocation of pools in the OS kernel, or relying on stack pivoting to spray data instead of code [51].

Recently, GRAFFITI [19] has been proposed as a hypervisor-based memory monitoring solution to aid detection and prevention of all known variations of spraying attacks. In particular, by leveraging a novel micro-virtualization technique, this system proposes an efficient OS-agnostic framework to monitor memory allocations of arbitrary applications. GRAFFITI offers the first general and portable solution for efficiently monitoring the memory behavior; the system is modular, and relies on a set of plugins to detect suspicious patterns in memory at runtime. However, the detection heuristics provided in the original paper [19] are just an example for the use of the GRAFFITI system, and they do not provide any generic defense since they are very specific to a particular attack vector and are trivial to evade. Indeed, as the authors of [19] highlight, the proposed detection heuristics were not part of the main contributions, which instead focused on designing a framework for efficiently tracking the memory page allocation.

In this paper, we propose GLYPH *as an extension of the detection engine of* GRAFFITI [19]. In particular, we investigate the problem of designing resilient detection techniques against heap spraying attacks. To this end, we evaluate whether machine learning techniques can effectively detect heap spraying by monitoring memory pages at runtime.

To perform our analysis, we generate representative memory dumps of benign and malicious processes. We conduct our experiments on Windows 7 (32-bit) and Internet Explorer

11, and generate a dataset by running and dumping a total of 175 benign processes dumps, 160 malicious (sprayed) processes and 80 mixed (benign+malicious navigation) with different settings, including a mix of manual and automated Web navigations. In particular, we analyze the effectiveness of feature spaces based on memory n-grams and on information entropy: we compare the two methods, by using representative ML algorithms applied to the context of spraying attacks in memory.

Our results show that there is a trade-off between the runtime overhead and the effectiveness of the two feature spaces: entropy features are faster and more agnostic, but slightly less precise; n-grams are slower to compute and require some a priori attack knowledge, but are more effective for detection. To avoid overfitting and show the resilience of our system against spraying attacks we also perform selective hold-out experiments that simulate an adaptive attacker using different spray variants. Finally, we show how our machine learning techniques outperform NOZZLE [53], a state-of-the-art heap spraying detection mechanism.

In summary, we extend the original paper of GRAFFITI [19] *and make the following novel contributions:*

- We propose GLYPH, which is—to the best of our knowledge—the first system to explore the use of ML techniques for heap spraying detection. GLYPH extracts features from the page-level runtime memory monitoring of GRAFFITI [19]. We present solutions to several design and implementation challenges we have tackled to choose the appropriate ML algorithms and feature spaces in order to make GLYPH accurate in its detection, while containing its runtime overhead (§III).
- We build a representative dataset featuring a comprehensive set of heap spraying attack vectors and scenarios (§IV). On this, we perform a thorough experimental evaluation, which considers also mimicry and evasive attack vectors, to identify two optimal configurations of GLYPH which offer a trade-off between runtime overhead and detection performance (§V): one based on memory n-grams, slower and requiring some a priori attack knowledge, but more effective; one based on entropy, faster and more agnostic (i.e., not requiring a priori knowledge), but less precise.
- We experimentally show that the two best configurations of GLYPH outperform NOZZLE [53], the state-of-the-art system for heap spraying detection, in terms of both detection performance and runtime overhead (§V-F).

The remainder of the paper is structured as follows. Section II discusses some background information on heap spraying and the GRAFFITI framework [19]. Section III describes the design of GLYPH, along with detailed reasoning for the choice of the feature spaces and ML algorithms evaluated. Section IV shows how we create a representative dataset of heap spraying attacks for our evaluations. Section V presents the thorough experimental evaluation, which considers also attackers using evasive variants of heap spraying attacks, and compares the performance of GLYPH with respect to the state of the art. Section VI presents a discussion on main findings and some limitations of our analysis. Section VII compares GLYPH with related work, and Section VIII discusses conclusions and future work.

II. BACKGROUND

Heap spraying is a payload delivery technique that was publicly used for the first time in 2001 in the telnetd remote root exploit [44] and in the eEye's ISS AD20010618 exploit [40]. The technique became popular in 2004 as a way to circumvent Address Space Layout Randomization (ASLR) in a number of exploits for Internet Explorer. Since 2004, spraying attacks have evolved and became more reliable thanks to improvements proposed by Sotirov [58] and Daniel et al. [20] for precise heap manipulation. Spraying can now be classified into two main categories, based on the protection mechanisms in place on the target machine: Code Spraying and Data Spraying. If Data Execution Prevention (DEP) is not enabled, the attacker can perform the exploit by directly spraying the malicious code (e.g., the shellcode) into the victim process memory. On the other hand, when the system uses DEP protection, the attacker would not be able to execute the injected code. To overcome this problem, two main approaches have been proposed: (a) perform the heap spraying by taking advantage of components that are not subjected to DEP, such as Just in Time (JIT) compilers, or (b) inject plain data that points to Return Oriented Programming (ROP) gadgets. While the internal details between the three aforementioned approaches may be quite different, what is important for our research is that all these techniques share the same goal: to control the target dynamic memory allocation in order to obtain a memory layout that allows arbitrary code execution in a reliable way.

It is important to note that spraying is still a valuable technique in x86_64-based operating systems as well. In particular, this is the case for use-after-free vulnerabilities—but spraying can still be used in conjunction with vulnerabilities in the ASLR implementation [16] or other particular vulnerabilities [e.g., 23], or because of the wide adoption of 32-bit processes in 64-bit operating systems (as recently shown by Skylined [56]).

Our research devises an ML-based detection engine, GLYPH, that can be embedded in the GRAFFITI framework for runtime detection of heap spraying attacks. GRAFFITI [19] is a system designed to support detection and prevention of spraying attacks by monitoring individual applications running on any operating system. GRAFFITI is based on a custom hypervisor, implemented using hardware virtualization technologies, which runs below the operating system, intercepting all memory allocations performed by programs. These allocations are constantly monitored, and per-process profiles are built. Based on heuristics, such as the exceeding of an allocation threshold over a specified amount of time, GRAFFITI triggers the detection engine to check for the presence of an attack pattern. The GRAFFITI system is modular, and relies on a set of plug-ins to detect suspicious patterns in memory at runtime. The detection algorithms in GLYPH can be attached as plugins inside the GRAFFITI framework. The original paper [19] reported only simple detection heuristics that are very trivial to evade; indeed, its main contributions where on the design of a

framework for efficient tracking of memory page allocations. In this paper, we investigate GLYPH as an extension for the detection engine of GRAFFITI, for effective and efficient ML-based heap spraying detection.

III. Heap Spraying Detection

There are several design requirements for GLYPH, our detection system. **R1**: GLYPH should rely only on the memory information monitored by GRAFFITI [19]. **R2**: GLYPH should provide an efficient detection phase which is feasible to operate at runtime in the end-user machine, with a system overhead in line with that of GRAFFITI. **R3**: GLYPH should be general in the detection of heap spray attacks, using learning and features that do not overfit specific spray characteristics. **R4**: GLYPH should achieve high detection performance against a comprehensive dataset of spraying attack vectors.

These requirements have guided the design and evaluation of GLYPH, and the creation of an appropriate and representative dataset of memory dumps. In this section, we first provide a high-level overview of the different features involved in the detection process, and then we describe the algorithms used by GLYPH to perform detection of sprayed processes. Section V will present a thorough evaluation which identifies the best configurations of GLYPH.

A. Threat Model

We assume that the objective of the attacker is to trick the victim into opening a compromised Web page with malicious JavaScript that performs a heap spraying. As attack vector, the attacker mostly relies on phishing (e.g., a link to the malicious page in an email or social network message). There is a chance that a benign website is compromised by the attacker with stored XSS, so that the heap spraying begins while the victim visits the benign website.

There are two main settings for our threat model depending on the victim's memory context when the click on a malicious link happens: if a new browser process is opened (e.g., a new tab, or a new instance of the browser), then the spray occurs in a newly initialized—clean—process (without bytes from prior benign navigation); if the link is opened after some benign navigation of the victim, and an already-opened browser process is in use (e.g., a new tab is opened with a new thread), then the spray will occur in a page which already contains benign navigation history. Browsers like Internet Explorer create a new process for each newly opened tab, while others (e.g., Firefox) may either create new processes or new threads (depending on the amount of memory already in use by the browser); in general, GRAFFITI cannot distinguish between the two cases a priori, and hence GLYPH needs to achieve high detection performance in both scenarios.

We also assume that the attacker will try to evade detection through multiple orthogonal approaches. Existing metamorphic and polymorphic algorithms can automatically generate pseudo-random attack payloads that easily evade all detectors based on *signatures*. Behavioral detection approaches that try to differentiate between benign Web browsing activities and heap spraying attacks based on the *memory allocation rate* can be evaded by attacks that gradually deploy the heap spraying payload to mimic the memory allocation rate of a Web browser that renders legitimate pages [19]. Detectors based on machine learning approaches can sometimes be fooled by attacks that manage to include some *benign background noise* in the navigation. As an example, consider a malicious script with a time- or logic-bomb that is triggered only after the victim has done some benign navigation. Our detector GLYPH is designed to be resilient against *all these mimicry attempts* since it relies on features that are necessarily affected by heap spraying attacks, the size of which has to be relatively large by design (in the order of hundreds of megabytes) to achieve a sufficient success probability of the exploit. In particular, we also identify the most resilient configurations of GLYPH via experimental evaluation in §V.

B. Detection Task

We aim to design GLYPH as a system to detect whether heap spraying is occurring within a monitored process. More formally, we are interested in a *binary classification* task in which the detection algorithm $f : \mathcal{X} \longrightarrow \{0, 1\}$ takes a feature vector $\boldsymbol{x_i} \in \mathcal{X} \subseteq \mathbb{R}^m$ extracted from process P as input, and outputs label $\hat{y} = 1$ if the process is being *sprayed* (or label $\hat{y} = 0$ if the process is *clean*).

We rely on *supervised classification* and not on *anomaly detection* because we design and build a representative dataset of memory processes corresponding to benign navigations and heap spraying attack variants (§IV). It is well known that supervised classification has better performance than anomaly detection when a dataset representative of all classes (in our case, two classes: clean and sprayed) is available [11, 15]. The construction of a representative dataset is also related to evaluate satisfaction of requirement R4.

Machine learning algorithms mostly work on vector data as input. Hence, we first define a mapping between a memory process and a feature space $\mathcal{X}$. Since the heap spraying affects the process *memory*, which can be monitored at page-level by GRAFFITI [19], we extract features from the memory content (R1). In particular, we model a process as a memory object divided into pages, where each page is represented as a sequence of bytes, i.e., integer values between 0 and 255. We consider bytes because they are the units of assembly instruction, which will also be the object of spraying attacks.

More formally, the feature embedding $\varphi : \mathcal{P} \longrightarrow \mathcal{X}$ takes as input a process $P_i \in \mathcal{P}$ (set of memory pages of process i, represented as sequences of bytes) and outputs an m-dimensional feature vector $\boldsymbol{x}_i \in \mathcal{X} \subseteq \mathbb{R}^m$ (where m depends on the specific embedding—as explained later).

As in traditional machine learning [11], a model is learned for f through *training* on a set of labeled examples $z_i = (\boldsymbol{x}_i, y_i)$ corresponding to a process P_i with feature vector $\boldsymbol{x}_i$ (derived from embedding φ) and binary label y_i (0 if clean, 1 if sprayed). We use machine learning instead of heuristics and static thresholds [19] in order to learn more complex and general models that can effectively distinguish between clean and sprayed processes.

The following subsections describe feature embeddings φ in GLYPH, based on information entropy and n-grams. The

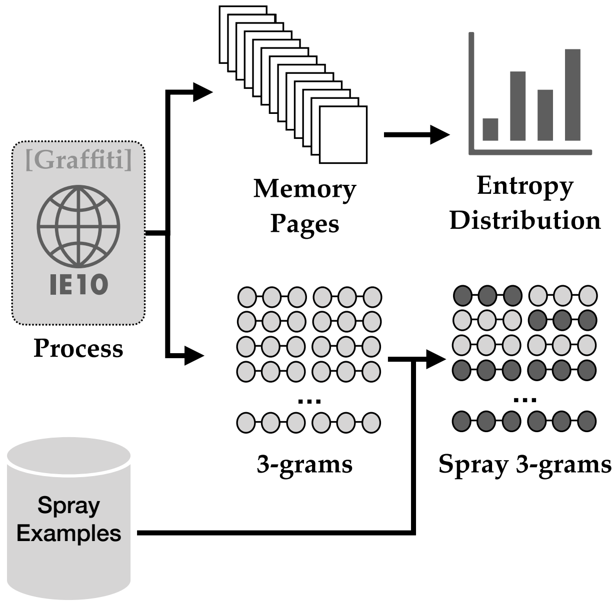


Fig. 1. Feature extraction overview. GLYPH extracts features for information entropy and memory byte n-grams from a process monitored by GRAFFITI.

intuition is that these should capture the changes introduced by heap spraying activity. The major challenge is to design an embedding that is at the same time fast to compute (R2), generalizable (R3), and effective in terms of detection performance (R4).

C. Feature Embedding: Information Entropy

In information theory, *entropy* is the average rate at which an information is produced from a stochastic source of data [54]. The intuition to consider entropy-based features is that heap spraying leaves an anomalous distribution of entropy within the memory of a process, due to the repetition of both NOP sleds and spraying of the same shellcode in multiple pages of the memory. Repetition of the same pattern, such as NOP sled and shellcode, will reduce the entropy of the process memory to a value closer to 0; such event does not happen in a benign process memory page that in general contains different information.

More formally, to have a value comprised between 0 and 1, we refer to the following definition of *normalized information entropy* corresponding to the memory of a process $P \in \mathcal{P}$:

$$H(P) = \frac{-\sum_{i=1}^{N} Pr_i \cdot log_b(Pr_i)}{log_b(N)} \tag{1}$$

where N is the total number of *bytes* within the memory of process P, and Pr_i is the probability of occurrence for the i-th byte value. For the sake of simplicity, in the remainder of this manuscript we refer to $H(P)$ as just *entropy*. We consider byte-level granularity because it is the minimum unit of assembly instructions, where the average instruction length is about 3 bytes [55]. Since the definition in Eq. 1 is divided by a normalizing factor, it is constrained as follows:

$$0 \leq H(P) \leq 1 \tag{2}$$

We now need to define a feature embedding φ based on entropy that extracts a numerical vector from the memory of a process P_i. A first option could be to directly extract a single entropy value $H(P_i)$ for the whole process P_i. However, only massive heap spraying attacks would cause a deviation across the whole process entropy—thus easing attacker evasion for lower intensity spraying that occurs, for example, in most of the heap spraying attacks on 64-bit architectures. Moreover, a single value for the whole process entropy would hardly be representative of the process itself, to distinguish between clean and sprayed memory. Hence, we rely on the fact that GRAFFITI is able to monitor process memory at page level to instead compute the *entropy distribution* of the memory pages. In other words, we compute the entropy value for each single page $p_j \in P_i$ as $H(p_j)$, and then consider the entropy distribution of:

$$\cup_{p_j \in P_i} H(p_j) \tag{3}$$

Figure 1 summarizes the feature extraction and embedding process. GRAFFITI monitors all the memory pages of process $P_i \in \mathcal{P}$, GLYPH computes $H(p)$ for each page $p \in P_i$. Since GRAFFITI monitors individual updates to pages, if a page p is modified, $H(p)$ is recomputed for that page. Then, the entropy distribution of all memory pages of P_i is approximated as a histogram with B bins. Using histograms is a common way to discretize a distribution as the frequency of object occurrences within a certain range of values [11, 15]. Finally, each process P_i is associated with a histogram representing the entropy distribution. We heuristically determine on a validation set that B=20 bins allows for a good representation of the entropy distribution offering good differentiation between benign and malicious processes.

The output of this process is a B-dimensional feature vector $\boldsymbol{x}_i$ corresponding to process P_i, where each element $x_j \in \boldsymbol{x}_i$ is the frequency of the occurrences in the j-th bin of the entropy distribution histogram. In the training set, each feature vector will be associated with label 0 if the process is clean (i.e., negative result), and with label 1 if the process is sprayed (i.e., positive result).

D. Feature Embedding: N-Grams

GLYPH also considers a feature embedding based on n-grams, under the intuition that they can capture anomalous byte distributions in the memory of a process. Features based on n-grams have been extensively and successfully adopted for the identification of malicious programs (e.g., [49, 61]); however, in our setting, it is not possible to trivially use the solutions proposed in past literature. The motivation is that, unlike traditional methods working on source code or code-specific memory regions [e.g., 49], here we are interested in looking at the process-wide system memory of each application, as GRAFFITI monitors it [19]. We observe that, unlike for entropy, we perform the n-gram analysis process-wide (i.e., not per-page). The motivation for this choice is that n-grams capture

the frequency of specific bytes sequences, whereas entropy is more content-agnostic; as an example, two memory pages may have the same entropy value while containing different n-grams. Consequently, analyzing n-grams process-wide allows the system to effectively identify *specific* byte sequences which are prevalent in sprayed processes (e.g., shellcode bytes can occur across among memory pages), whereas for entropy it is more appropriate to analyze the per-page entropy distribution within the whole process (because otherwise if we use a single value of entropy for the whole process memory, we would be able to recognize only extreme memory perturbations of the attacker).

Analyzing the entire process memory causes a rapid *n-gram state-space explosion* [61], because all possible n-grams become likely and occur at least once in each process; for example, with just 2-grams we have 2^{256} alternatives (where $n = 2$ is the n-grams, and 256 are the possible byte values), which is not feasible to compute. This is different from code abstractions where there are a fairly limited set of instructions (i.e., a limited set of possible n-grams); in past literature [49], pruning through feature selection has been adopted to reduce the problem complexity, but the feature selection could be applied only after obtaining the full n-gram feature matrix, which is not feasible in our case, as in the worst case for each n we consider we would have n^{256} possible features. In short, we need to design a fast yet effective solution that can select relevant n-grams in advance to be used for heap spraying detection (R2, R4).

First, for computational efficiency we need to constrain our analysis to a fixed value of n for the collection of n-grams. It is not feasible in our setting to collect n-grams for multiple values of n (e.g., n={2,3,4,5}). Intuitively, heap spraying will leave shellcode in the memory (at multiple locations) consisting of a certain sequence of instructions which would appear anomalous with respect to a clean process memory. Hence, we choose to consider the average Intel assembly instruction length, which is n=3 [30]. This allows us to capture the most frequent instructions within the memory of the process, and also take into account NOP sleds and data/payload bytes distribution.

In order to further lower the computational complexity for training the model, and to reduce the chance of learning artifacts from the training data, we first extract all possible 3-grams from a representative spray dataset (§IV)—where each 3-gram appears either in one of the NOP sleds or one of the shellcode samples. The result is a set of about 100K 3-gram features, which also represent the feature space $\mathcal{X}$ considered for the 3-gram feature embedding.[1] For each process memory P_i, we extract all the 3-grams in list n_i; then, we project the 3-grams n_i onto the feature space $\mathcal{X}$ (i.e., the set of 100K 3-grams within any spray). The 3-grams that are present in $\mathcal{X}$ but not in the list n_i (i.e., absent from the process P_i) are assigned frequency 0.

Figure 1 presents a summary of the feature extraction process for n-grams. Here, 3-grams are extracted from a whole process P_i monitored by GRAFFITI—this is immediately obtained as a concatenation (not necessarily ordered) of pages within the memory of a process. The page-level monitoring of GRAFFITI is useful for an efficient update of the n-gram feature vector (i.e., by changing only the 3-gram frequencies corresponding to a modified memory page).

The output of this process is a feature vector $\boldsymbol{x}_i$ where each element $x_j \in \boldsymbol{x}_i$ is the absolute frequency of the i-th 3-gram in the feature space $\mathcal{X}$, represented by possible 3-grams in representative heap sprayings.

E. ML Algorithms

We rely on supervised learning algorithms to distinguish between clean and sprayed processes.

Choice of the Algorithms. The famous "No Free-Lunch" theorem of Machine Learning posits that there is no specific algorithm that is suited for all tasks and datasets [28]. Moreover, if two models can achieve the same performance, the simpler model is always to be preferred—because they are easier to explain, and because they reduce chances of overfitting (i.e., they tend to generalize better than complex ones). We decide *not* to rely on deep learning algorithms because they require high computational resources, lots of training data, and are more challenging to explain [11, 60]. Other algorithms such as k-NearestNeighbor do not make assumptions on the structure of the data, but have very high testing cost—which is not feasible with the online requirements of our setting. Hence, in our scenario, we consider and evaluate the following two supervised learning algorithms to be adopted on top of the entropy and n-gram feature embeddings [11]: *Support Vector Machine* (SVM) and *Random Forest* (RF).

The intuition behind the choice of SVM and RF is as follows. SVM is known to perform well in high-dimensional feature spaces [25, 61], which is especially the case for the n-gram feature space. RF is designed to intrinsically reduce overfitting and improve generalization capabilities. Moreover, once a model is learned, the *test-time overhead* of both SVM and RF is negligible; in other words, once a model is trained, detecting if a feature vector (i.e., a process) is sprayed or not (i.e., label $\hat{y} = 0$ or $\hat{y} = 1$) is performed in a negligible time (some other ML models, like k-NearestNeighbor, have negligible training time, but huge test-time overhead [11]). Moreover, SVM and RF have been successfully used in many contexts related to malware detection [5, 14, 25, 37, 48].

In the remainder of this section, we briefly describe the details of SVM and RF, and explain the main hyperparameters that we consider in our evaluation for tuning the classifiers.

Support Vector Machine. SVM aims to identify the optimal *separation hyperplane* between two classes; in our case, between clean and sprayed processes. We consider the same definition of SVM hyperplane optimization (i.e., to determine the optimal slope of the hyperplane) as in [5, 11, 21], which

[1] We also considered and evaluated using only the "top-k 3-grams" for each process (e.g., k=1,000 and k=10,000), to create a unique feature space representation. However, we have experimentally verified that such a representation was causing the ML models to learn artifacts (e.g., giving high importance to n-grams unrelated to the spray attack vectors); this was probably also related to the "top-k 3-grams" not being necessarily related to heap spraying for the more evasive attack scenarios we evaluated (see §IV).

is formalized as follows:

$$\min_{\boldsymbol{w},b} \left\{ \underbrace{\frac{1}{2}\boldsymbol{w}^\top \boldsymbol{w}}_{R(f)} + C \underbrace{\sum_{i=1}^{n} \max(0, 1 - y_i f(\boldsymbol{x}_i))}_{L} \right\} \quad (4)$$

where R is the l_2-regularization term (used to increase generalization), and L is the Hinge loss function. The weights vector $\boldsymbol{w}$ and the bias b determine the slope and the intercept of the separating hyperplane. The hyperparameter C is used as a balance factor for the importance of the regularization term. The decision function $f(x_i)$ is defined as:

$$f(\boldsymbol{x}_i) = \boldsymbol{w}^T \cdot \boldsymbol{x}_i + b \quad (5)$$

and is used to predict the class of a test object x_i. In particular, if $f(x_i) \geq 0$ then $\hat{y}_i = 1$, otherwise $\hat{y}_i = 0$. We recall that SVM is known to perform well for malware detection and in high-dimensional feature spaces [5, 21, 48], the latter being the case especially for GLYPH's n-gram feature embedding.

Random Forest. RF is an ensemble algorithm based on Decision Trees (DTs). Starting from the feature matrix X, RF spawns a set (forest) of k Decision Trees; each tree contains a random subset of p variables, $X_p \subseteq X$ (i.e., features); each tree is then built on its set X_p, and multiple progressive splits are created through information theoretic criterions (e.g., Gini) that reduce *impurity* in the dataset [11]. The splits are performed to maximize the Mean Decreased Impurity before and after splitting a node into two leaves, and we try different values of maximum tolerated depth. In the training phase, RF also uses *bootstrap aggregating* (*bagging*), which consists in sampling randomly with replacement elements from the training set to build each tree (i.e., so that some elements are repeated).

The main hyperparameter of RF that we consider are the *number of trees* k, the *maximum tree depth* m, and the *maximum number of leaf nodes* l: the higher the value of k, the better the generalization capability of the learning algorithm; lower values of m and l may increase generalization, but at the risks of underfitting. RF minimizes chances of overfitting by design [27] (more formally, it reduces the *variance* of the DT algorithm while retaining the same *bias*).

We recall that RF has shown good generalization and classification capabilities in the malware domain [14, 37, 48], but has never been tested for heap spraying detection. Moreover, its design is useful to limit chances of overfitting our dataset during experiments.

IV. DATASET

We design and create a representative dataset of Web browser process pages—benign (clean), malicious (sprayed), and mixed (sprayed after clean navigation)—to evaluate GLYPH's detection performance and runtime overhead (§V).

We use GRAFFITI [19] to monitor the memory of the process *Internet Explorer 11* on a Windows 7 32-bit virtual machine.[2] We rely on METASPLOIT as a tool to generate a representative variety of *heap spraying* attacks; in particular, we generate a set of 160 Web pages containing malicious JavaScript code that performs the heap spraying. We do not consider "packing" as an obfuscation technique [10] since in the heap spraying context it reduces the probability of jumping to the correct instruction and consequently reduces the probability of a successful attack. Instead, we consider any transformations of shellcode due to metamorphic and polymorphic techniques as provided by the METASPLOIT tool.

[2] Although our experiments are only on 32-bit architectures, they generalize also for 64-bit architectures as explained in §VI.

TABLE I. DATASET COMPOSITION.

Type	Navigation	Processes	Total
Benign (Clean)	Automated	55	175
	Manual	120	
Malicious (Sprayed)	Automated	160	160
	Manual	–	
Mixed	Automated	71	80
	Manual	9	

TABLE II. HEAP SPRAYING ATTACKS PARAMETERS.

Parameter	Values
NOP sled type	SIMPLE, COMPLEX
NOP length	50K, 100K, 150K, 200K
Block size	100, 500, 1000, 2000
Shellcode payloads	*bind_tcp*, *download_exec*, *format_all_drives*, *adduser*, *powershell_bind_tcp*

To successfully spray the heap in Internet Explorer using JavaScript, we had to face two challenges. (1) Simple memory allocations in JavaScript do not reliably produce the expected result for heap spraying. As an example of the issue, a string allocation does not always correspond to an actual heap allocation, due to the use of cached free memory blocks in Internet Explorer's custom memory allocator. This problem is extensively discussed in [58]. (2) Some shellcode examples are detected and stripped from the heap by a defense mechanism of Internet Explorer, which needs to be bypassed to produce realistic process memory dumps.

To solve the first problem we used a JavaScript library called *HeapLib* [58], that allowed us to generate spraying payloads with specific memory layouts, effectively solving the problems mentioned in §II. In particular, we were able to allocate large contiguous memory regions in the heap, containing multiple copies of the same payload (a shellcode preceded by a NOP sled). Since the original version of HeapLib was engineered only for Internet Explorer up to version 8, we used a modified version by Chris Valasek, targeting versions 9 to 11 [59].

The second problem is due to internal protection mechanisms in Internet Explorer which detect malicious code via static analysis at runtime and remove it for preventing exploitation. We managed to overcome the issue by crafting shellcode that was not detected by this system. Specifically, we used METASPLOIT's payload encoder *x86/alpha_mixed* to produce payloads and complex NOP sleds. Such encoder transforms the desired payload, producing one with equivalent functionality but made only with bytes that are both x86 instructions and alphanumeric characters (with a small non-ASCII, binary preamble). This was enough to evade the static analysis defenses of Internet Explorer (see §II); the same malicious code that would get stripped when written to the

heap *always* appeared in its full form after encoding.

To produce our dataset, we crafted an HTML template containing the JavaScript calls to HeapLib, with placeholders for the actual spraying payload. A script iteratively calling METASPLOIT generated all the different payloads for testing, with varying parameters. Four parameters could be varied for each payload:

1) *numbler_of_blocks*: the number of repetitions of the payload formed by NOP sled plus shellcode;
2) *nop_style*: the type of NOP sled, either *simple* (just the byte `0x90` repeated), or *complex* (pseudo-random sequences of bytes representing x86 instructions, which are always different from one another in memory, but nevertheless semantically equivalent to no-ops at runtime [2, 32]);
3) *nop_length*: the number of bytes composing the NOP sled in each block;
4) *payload*: the malicious code at the end of each block, chosen from a list of codes available in METASPLOIT.

To extract memory dumps from running processes, we instrumented GRAFFITI. We implemented a new command in the tool which, upon invocation, traverses the pages allocated by a process, and dumps their binary content to file. This command can be invoked right after an heap spraying has been performed on Internet Explorer. While GRAFFITI runs the dump, the execution of the entire operating system is momentarily stopped, resulting in a precise snapshot of the process memory at a given time. Once the dump is finished, the execution is resumed.

Table I reports the dataset composition of memory snapshots that we consider when evaluating GLYPH. We recall that each memory snapshot has average size of 200MB and has on average 50K memory pages of 4KB. The 175 benign processes are derived from both automated random navigation with AUTOIT [1] and manual navigations of Alexa Top-1,000 Web sites, with an average navigation time of three minutes. Each benign process results from the navigation of a few randomly chosen websites from the Alexa Top-1,000; the choice falls within varying categories such as news, e-commerce, social networks (e.g., Twitter), streaming (e.g., YouTube). In the manual navigations, we simulated both lightweight navigation (e.g., news websites) and more memory-consuming usage (e.g., downloads of large files, and YouTube HD streaming). This data collection of benign processes ensures varying statistics in memory usage of legitimate navigations. The 160 "sprayed" processes correspond to the threat model in which a user clicks on a link (e.g., within a phishing email or social media message) and opens a malicious page directly; in particular, the sprayed processes navigate to a Web page that contains JavaScript heap spraying attacks of different types. The 80 "mixed" processes correspond to the threat model in which the heap spraying happens after some benign navigation within the same process (and this is also one of the possibly evasive strategies that the attacker can rely on).

Table II summarizes the parameters that we have varied to obtain different heap spraying JavaScript-based attacks through METASPLOIT. The SIMPLE NOP corresponds to NOP sleds with 0x90 values; the COMPLEX NOP sled corresponds to a pseudo-random sequence of operations that are overall semantically equivalent to 0x90 but which may look legitimate in assembly code. For each of these two cases, we consider different NOP lengths: 50K, 100K, 150K and 200K—measured in bytes. The block size represents the number of repetitions of the spraying pattern. We also consider different shellcode payloads—to ensure that our detection capability does not overfit a specific shellcode pattern. The total number of combinations is 160 because it is the cartesian product of all the elements in Table II: (2 NOP sleds types) $\times$ (4 NOP length) $\times$ (4 block sizes) $\times$ (5 shellcode payloads).

V. EXPERIMENTAL EVALUATION

We aim to evaluate the effectiveness of GLYPH with the different feature embeddings and algorithms introduced in §III. The experiments are aimed at answering the following questions. (**RQ1**) Can GLYPH detect heap spraying? (**RQ2**) What is the best combination of features/algorithms in GLYPH for detecting heap spraying, and with which trade-offs? (**RQ3**) Is GLYPH's overhead small enough to allow for runtime detection? (**RQ4**) Is GLYPH robust to evasive attackers, which try to perform new spray variants of attacks or which rely on benign background navigation?

A. Experimental Settings

We perform feature extraction and embedding according to §III on the dataset described in §IV. We implement GLYPH as a Python3 prototype, relying on several libraries: *sklearn* for machine learning algorithms; *entropy.shannon_entropy* for the computation of normalized Shannon entropy; *nltk* for the efficient extraction of n-grams. Our experiments are conducted on a VM with the following characteristics: Internet Explorer 11 on Windows 7 32-bit, 4GB of allocated RAM.

Experiments. We conduct five main experiments. First, we consider the overall performance on detecting benign and malicious pages through 10-fold cross-validation, as traditionally done in the machine learning community [11], to simulate a stationary setting in absence of concept drift [48].

Second, we evaluate the robustness of our approach in presence of an evasive attacker that introduces new variants of spray attacks (Table II); in particular, we simulate evasive attacks by performing selective hold-out validations, in which malicious processes with particular variants of heap spraying attacks are removed from the training set and used only in the testing set. More formally, we consider the following hold-out settings (i.e., where each setting corresponds to one in which that particular type of heap spraying is used only in the testing set):

A) NOP complex;
B) Small blocks (size $<$ 2,000);
C) Small NOPs (length $<$ 200K);
D) Shellcode set$_a$ (*adduser*, *format_all_drives*);
E) Shellcode set$_b$ (*bind_tcp*, *download_exec*);
F) NOP complex + Shellcode set$_a$;
G) NOP complex + Shellcode set$_b$.

It is important to note that we do not combine the small NOPs with small blocks, since we believe that this combination

is not representative of the heap spraying attack. In fact if the attacker uses very small blocks and small NOPs, the probability of attack success highly decreases [19]. In the normal heap spraying context the attacker needs to spray a huge number of the memory pages with relatively large NOP sleds and blocks, otherwise they could not be sure to land in the right memory location and execute the injected exploit code.

Third, we evaluate an evasive setting in which the spray occurs after some benign navigation (§III-A); this is evaluated through testing on the *mixed* processes (Table I).

Fourth, we evaluate the detection time overhead of GLYPH's prototype to evaluate the feasibility of runtime deployment.

Given these four experiments, we devise the best configurations of features and algorithms for GLYPH deployment, and discuss the trade-offs of the suggested configurations.

Finally, we compare the performance of GLYPH with the state-of-the-art for heap spraying detection: NOZZLE [53].

We observe that, in general, we train the SVM and RF algorithms only on (subsets of) benign and malicious processes, and use the mixed ones—where the spray occurs after benign navigation—only for testing scenarios (see also §III-A). The reason for this choice is associated with the unavoidable risk of overfitting and learning artifacts if mixed processes were included into the training set: while training on purely benign and purely malicious examples has the potential to highlight the real characteristics that distinguish heap sprayed pages from benign ones, training on mixed pages may lead the classifier to learn artifacts that are more related to differences in various types of benign traffic instead of capturing the salient characteristics of sprayed pages. Moreover, it would be practically challenging to generate a comprehensive dataset of mixed traffic which would allow the model to generalize, as one should consider at least the cartesian product of all benign and malicious alternatives, which would correspond to hundreds or thousands of terabytes of training data, which would be problematic to handle and process (see also §IV).

Performance Metrics. We report the performance in terms of True Positive Rate (TPR) and False Positive Rate (FPR). All scores are considered with respect to y=1 as positive class (corresponding to a *sprayed* process), and y=0 as the negative class (corresponding to a *clean* process).

Reducing Overfitting. The total number of elements within the dataset is 415, hence a reader may think that the results of our experiments may not generalize well, and that we may be overfitting our dataset. First, this dataset corresponds to a total of approximately 21,000,000 memory pages of 4KB, grouped into 415 heterogeneous processes. Second, we carefully considered 160 different attack scenarios for heap spraying, with varying characteristics—which cover different types of attackers and types of attacks (see Table II). Third, the benign processes contain both manual and automated navigation of the Alexa Top-1,000 domains for an average time of three minutes. While the dataset size is partially limited by storage space required to dump the processes,[3] we believe it constitutes a relevant sample set for the heap spraying scenario. Nevertheless, to reduce the risk of overfitting with GLYPH on our 415 processes, we rely on several mitigations that are commonly used in the ML literature [11]: (i) we consider hold-out settings in which some spraying examples are entirely absent from the training set; (ii) we perform the hyperparameter tuning to increase generalization [11], i.e., by varying the number of trees k and maximum depth m in RF (without restricting maximum number of leaf nodes l), and by employing an l_2 regularizer term in the SVM (by adjusting the C hyperparameter).

Hyperparameter Tuning. Our full dataset $\mathcal{D}$ consists of three types of processes: benign, malicious, and mixed (see Table I). For hyperparameter tuning, we consider $\mathcal{D}'$ consisting *only* of benign and malicious processes ($\mathcal{D}' \subset \mathcal{D}$); the subset $\mathcal{D}'$ does *not* contain mixed processes, which are used later in this paper solely for testing purposes. We then randomly split $\mathcal{D}'$ into 80% training Tr and 20% hold-out testing Ts. To find the best hyperparameters, we perform a grid-search within Tr (i.e., without Ts) with the following values: for linear SVM, $C = \{0.001, 0.01, 0.1, 1, 10, 100, 1000\}$; for RF, *number of trees* $k = \{10, 100, 1000\}$, *maximum depth* $m = \{32, 64, 128\}$, without limiting the *maximum number of leaf nodes* l. For each hyperparameter combination, we perform a nested 10-fold cross-validation (CV) within the training set Tr (i.e., without Ts), and obtain the *average TPR performance* at 0.1% FPR. The best average TPR at 0.1% FPR on the validation sets is achieved with the following hyperparameters: for SVM, **C=10**; for RF, **k=1,000**, **m=128**, **l=unrestricted**. To check for possible overfitting, we finally test with our hyperparameters on the 20% Ts set (which was not involved in the hyperparameter tuning), and we obtain on Ts almost the same performance of the nested 10-fold CV on Tr: more specifically, a performance within a ± 0.005 difference, which suggests lack of overfitting [11]. *We maintain these hyperparameters throughout the experiments.*

B. 10-fold CV Detection Performance

We first perform 10-fold cross-validation to evaluate detection performance considering the dataset of benign and malicious processes (without "mixed" processes, yet). This scenario is representative of a stationary setting in which training and testing set come from the same distribution (i.e., in the absence of concept drift [48]). Table III summarizes TPR and FPR with these hyperparameters. We can observe that all settings have high TPR, but entropy-based detectors have 1.1% FPR, and SVM on n-grams has a few false negatives leading to a TPR of 96.9%. It is good to observe that there are no false positives (FPs) with n-gram features, meaning that any heap spraying alert would correspond only to real threats. These 10-fold CV results simulate a scenario in which the attacker only performs minor variations of the known sprays

[3] We snapshot and dump process memory to ensure repeatability of experiments in different settings, and our dataset for 415 snapshot is about 60GB (compressed) and over 100GB (uncompressed), where the size of each process snapshot varies from about 100MB to 580MB (uncompressed).

TABLE III. 10-FOLD CV PERFORMANCE OF HEAP SPRAYING DETECTION (BENIGN AND MALICIOUS PROCESSES).

Features	Alg.	TPR	FPR
entropy	SVM	100.0%	1.1%
	RF	100.0%	1.1%
n-gram	SVM	96.9%	0%
	RF	100.0%	0%

TABLE IV. DETECTION PERFORMANCE OF EVASIVE HEAP SPRAYING VARIANTS (SELECTIVE HOLD-OUT).

#	Evasive Variants	Features	Alg.	TPR	FPR
A	*NOP complex*	entropy	SVM	0.0%	0.0%
			RF	100.0%	0.0%
		n-gram	SVM	0.0%	0.0%
			RF	100.0%	0.0%
B	*Small blocks*	entropy	SVM	58.3%	0.0%
			RF	58.3%	0.0%
		n-gram	SVM	66.6%	0.0%
			RF	100.0%	0.0%
C	*Small NOPs*	entropy	SVM	79.2%	0.0%
			RF	96.7%	0.0%
		n-gram	SVM	91.7%	0.0%
			RF	100.0%	0.0%
D	*Shellcode set*$_a$	entropy	SVM	100.0%	0.0%
			RF	100.0%	0.0%
		n-gram	SVM	100.0%	0.0%
			RF	100.0%	0.0%
E	*Shellcode set*$_b$	entropy	SVM	100.0%	0.0%
			RF	100.0%	0.0%
		n-gram	SVM	100.0%	0.0%
			RF	100.0%	0.0%
F	*NOP complex + Shellcode set*$_a$	entropy	SVM	0.0%	0.0%
			RF	100.0%	0.0%
		n-gram	SVM	0.0%	0.0%
			RF	100.0%	0.0%
G	*NOP complex + Shellcode set*$_b$	entropy	SVM	0.0	0.0%
			RF	100.0	0.0%
		n-gram	SVM	0.0	0.0%
			RF	100.0	0.0%

(in our case, the ones in Table II). We have further investigated the false negatives (FNs) and false positives (FPs) obtained in Table III. The FNs associated with the SVM on n-grams correspond to malicious dumps with only 100 blocks that use NOP complex—this represented a challenging scenario that the SVM is not able to detect. The FPs obtained by both SVM and RF on entropy are due to benign memory pages that have an entropy pattern very similar to some of the malicious pages in the training set. We will better discuss how to avoid such FNs and FPs in the next section.

C. Security Analysis & Performance

The goal of the next experiments is twofold. On the one hand, we want to show that our system does not present overfitting and it is resilient enough for detecting a large spectrum of heap spraying attacks; on the other hand, we want to introduce a security analysis of the system where the attacker designs some components of the attack vector that are able to impact on the detection rate.

Evasive Heap Spray Attack Variants. We now consider an experiment in which we selectively remove some attacks from the training set. In other words, these experiments aim to evaluate if the model generalizes well to novel, evasive attack variants by training the model on partial knowledge only. Moreover, it is useful to verify if the results obtained in Table III still hold.

TABLE V. DETECTION PERFORMANCE ON EVASIVE HEAP SPRAYING AFTER BENIGN NAVIGATION (MIXED PROCESSES).

Features	Alg.	TPR	FPR
entropy	SVM	41.3%	0.0%
	RF	11.3%	0.0%
n-gram	SVM	100.0%	0.0%
	RF	0.0%	0.0%

Table IV reports the GLYPH results for entropy and n-grams in the different selective hold-out scenarios, with SVM and RF algorithms. Green cells correspond to optimal detection performance (i.e., 100% TPR), and red cells correspond to the lowest detection rate (which corresponds to the maximum evasion for an attacker). The results here are reported in terms of TPR and FPR. We remark that the *evasive variants* column shows which elements are removed (i.e., hold-out) from training and then used only for testing. From Table IV we make the following conclusions:

- GLYPH's approach is always independent of the shellcode payloads considered. GLYPH is also able to detect as sprayed pages that contain shellcode samples that were absent in the training. This is shown by the perfect detection rate (TPR) in scenarios D and E.
- We can see that RF outperforms SVM, likely due to its intrinsic generalization capabilities (§III-E).
- The best performance of GLYPH is achieved by RF on the n-grams feature space.
- The most challenging scenario to detect is for SVM when all NOP complex examples are removed entirely from the training set (configurations A, F, and G). Nevertheless, this represents an extreme scenario, and it is sufficient to add at least a few NOP complex examples in the training set to take their characteristics into account. On a related note, it is interesting to observe that in the configurations G and F, the detection rate is not affected by the use of different payloads.

Evasive Benign Background Noise. We now evaluate how performing benign navigation within the same process that is then sprayed can affect the detection capability of GLYPH (§III-A). In particular, we train on all *benign* and *malicious* process memory dumps (§IV), and test on the *mixed* ones. Table V reports the results on this setting corresponding to evasive background noise. We can observe that the overall detection capability of GLYPH is lower than that of other scenarios. The SVM on n-grams frequencies achieves a perfect detection rate of 100%, whereas SVM on entropy has only 41.3% TPR. On the other hand, the RF has very poor performance. The high performance of the SVM is likely motivated by the fact that the weights vector $\boldsymbol{w}$ of the SVM determines a hyperplane that is somewhat similar to applying a weighted threshold on each feature value (§III-E); this implies that, despite the benign background noise, the SVM may be detecting suspicious bytes distributions associated with spray activity. On the other hand, the RF learns decision trees to

TABLE VI. GLYPH'S DETECTION PERFORMANCE IN ALL SETTINGS AFTER USING MAJORITY VOTING ENSEMBLE OF SVM AND RF.

#	Experiment	SVM+RF (entropy)		SVM+RF (n-gram)	
		TPR	FPR	TPR	FPR
-	10-fold CV Detection Performance	100.0%	2.4%	100.0%	0.0%
A	Hold-out: Unseen *NOP complex*	100.0%	0.0%	100.0%	0.0%
B	Hold-out: Unseen *Small blocks*	100.0%	0.0%	100.0%	0.0%
C	Hold-out: Unseen *Small NOPs*	96.7%	0.0%	100.0%	0.0%
D	Hold-out: Unseen *Shellcode* set_a	100.0%	0.0%	100.0%	0.0%
E	Hold-out: Unseen *Shellcode* set_b	100.0%	0.0%	100.0%	0.0%
F	Hold-out: Unseen *NOP complex + Shellcode* set_a	100.0%	0.0%	100.0%	0.0%
G	Hold-out: Unseen *NOP complex + Shellcode* set_b	100.0%	0.0%	100.0%	0.0%
-	Hold-out: Unseen *Evasive Benign Background Noise*	41.3%	0.0%	100.0%	0.0%

determine what is benign and what is malicious from the training data, and has no mixed examples; hence, once decision branches suggesting that a process is benign are taken (due to the benign background noise), there is no rule that can reconnect the RF to a malicious process. We recall that we cannot train our classifiers on mixed processes in order to avoid overfitting and to avoid learning artifacts associated with different benign navigations (see §V-A).

It is interesting to observe that the results in Table V seem to be the opposite of the results in the previous experiments in Table IV for scenarios A, F and G, in which SVM has 0% TPR and RF has 100% TPR. The performance of Table IV is mostly caused by the holding out of NOP complex sleds from the training set: in this scenario, the RF is still able to generalize maliciousness by following branches likely indicating malicious bytes 3-grams or entropy distributions, but the SVM does not learn a proper hyperplane orientation due to the lack of NOP complex objects in the training set; moreover, without NOP complex sleds in the training set, the SVM may have been over-emphasizing the presence of "simple NOP sleds" as indication of maliciousness. On the contrary, in Table V the SVM based on n-grams is able to capture the maliciousness of all sprayed processes, whereas the RF does not. As mentioned before, this is likely because RF takes some initial split choices based on presence/absence of some benign bytes. These results show that RF and SVM have complimentary detection and robustness properties in our scenario, which inspires us to combine them in an ensemble algorithm.

Best Configurations of GLYPH. The detection performance results obtained above, with low false positives, suggest two main configurations on which GLYPH can be used: one based on entropy and one based on n-grams. Each configuration must run in an *majority voting ensemble* [14] with RF and SVM: in particular, a process is marked as sprayed by the detector if at least one classifier (RF or SVM) marks it as malicious. Table VI reports the performance obtained with such *majority voting ensemble* on the two feature spaces. In particular, optimal performance is achieved with SVM+RF on n-grams, for which we recall that the feature space requires knowledge of possible spray 3-grams (§III-D).

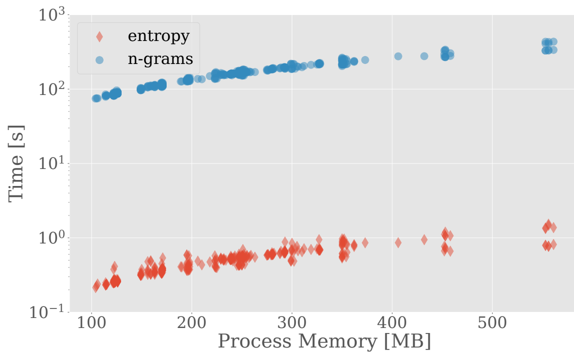
D. Runtime Detection Overhead

The decision time of SVM and RF algorithms used in GLYPH is negligible as they are inference-based models [11]. However, when new pages are created in the process memory, the features need to be extracted again; hence, to determine feasibility of detection time, it is crucial to determine feature extraction times in GLYPH. We recall that R2 (§III) requires that detection time is inline with that of GLYPH so that runtime detection could be feasible.

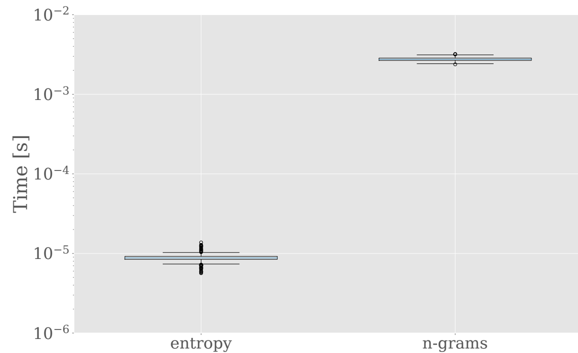
Figures 2 report detailed times for feature extraction on our Python3 prototype for both entropy and n-gram feature spaces. Figure 2a reports a scatterplot of the extraction times, when considering the feature extraction for the whole process memory. The X-axis represents the process memory size in MBs, whereas the Y-axis is the total extraction time expressed in seconds. We highlight that the Y-axis is in logarithmic scale. The entropy feature extraction is almost three orders of magnitude faster than that of n-grams. In particular, for all processes in our dataset, the entropy feature extraction is always approximately below 1 second. Conversely, the extraction of n-grams on a full process may even take a few minutes for the larger ones.

Figure 2b reports the time required to extract entropy and n-gram features at a page-level perspective. We recall that GRAFFITI monitors pages of 4KB. Figure 2b reports two boxplots, one for each feature type; the Y-axis is in logarithmic scale and reports the time in terms of seconds. We can observe that even at page-level the entropy features are almost three orders of magnitude faster to extract than n-gram features. The fact that the boxplots are compact implies that the extraction time is approximately stable between multiple runs. Hence, we can see that the extraction time is about $10\mu s$ per-page for entropy, and about $3ms$ per-page for n-grams. *It is immediate to derive that a computer can process up to 100,000 pages modified per second (on each vCPU core) for the entropy features, whereas the n-grams will support processing of about 300 pages modified per second (on each vCPU core).*

It is important to highlight that the process-level extraction time in Figure 2a is relevant only at startup time. In an online context, GRAFFITI monitors the memory pages, and can update the feature vectors of a monitored process by changing only the features of the modified pages. Let us consider an example of "feature vector update" between time t and $t + 1s$. If k pages are modified within this time interval, GLYPH must update the feature vector according only to the changes that occur in these k pages. GLYPH considers k pre-modification pages (i.e., their content at time t) and k post-modification pages (i.e., their content at time $t + 1$). To update the feature vector, it is sufficient for GLYPH to *subtract* the feature values corresponding to k pre-modification pages and *add* the feature values corresponding to k post-modification pages. In this



(a) Process-level extraction times (Y-axis: log scale)



(b) Page-level extraction times (Y-axis: log scale)

Fig. 2. Feature extraction times for entropy and n-grams features. The left figure reports a scatterplot of the extraction times as a function of the process memory size. The right figure reports the boxplot distribution of the feature extraction costs. The entropy extraction is more than two order of magnitudes faster than the n-grams.

way, the feature vector of process P_i can be tested again, to see whether GLYPH identifies it as *sprayed* or *clean*. In other words, to evaluate feasibility for the online context, only the per-page extraction performance matters (Figure 2b). The number of pages for which features are extracted can also be regulated by adjusting the threshold for the *security mode* of GRAFFITI [see 19].

E. GLYPH *Best Configurations and Trade-Offs*

The results show that there are two best configurations for GLYPH based on a majority voting ensemble of SVM+RF. A first mode will work on SVM+RF ensemble with entropy, because of its fast processing speed, and with no a priori knowledge of spray attacks needed, and will raise an alert if heap spraying is detected—with the risk of a few false positives and some false negatives. A possible response to an alert could be to kill the process or signal the user. A second mode with perfect detection rate (i.e., TPR) and no false positives is that achieved by SVM+RF ensemble with n-grams, despite requiring some a priori attack knowledge and the higher runtime overhead required to extract the n-grams. Despite its runtime overhead, we believe this latter mode is the one recommended for the following reasons: our prototype is in Python3, so it is reasonable to assume that a computational speedup may be achieved with an optimized implementation and that the *security mode* of GRAFFITI (see [19]) can reduce the number of pages to be processed per second by prioritizing only *suspicious pages* with memory allocation patterns similar to those of heap spraying.

F. Experimental Comparison with State-of-the-Art

In this section we compare GLYPH, built on top of GRAFFITI, with the existing state-of-the-art for heap spraying detection: NOZZLE [53]. This experiment is performed on two machines, equipped with an Intel Core i5-2500 @ 3.3 GHz and 8GB of RAM, running Windows 7 Professional 32bit and Debian Wheezy 32bit (kernel 3.2), respectively. In the experiment we first compare the efficiency of the two systems applied to Internet Explorer during the average user system workload. We chose the IE since NOZZLE is designed to protect the Internet Explorer application.

GLYPH is embedded in GRAFFITI and is designed to be adaptive. Consequently, the only part that is always active is the Memory Tracer of GRAFFITI. GLYPH uses GRAFFITI's micro-virtualization solution that confines the overhead to a single process and allows our system to monitor an arbitrary number of different applications without any increase in the overhead of the rest of the system. NOZZLE is instead designed to protect only the Web Browser, and it has been specifically designed to be integrated into the JavaScript allocation engine.

During normal operation of our system, the tracker overhead is negligible, and it is only noticeable when the monitored application allocates tens of megabytes of memory at a time—typically at start-up or when a large document is opened [19]. On top of this small overhead, each application can observe a different overhead when GRAFFITI switches to security mode and enables the GLYPH algorithm detection modules to scan the application memory. The frequency at which this happens depends on the value of the activation threshold. We perform an experiment aimed at measuring this overhead. In the experiment we asked some users to surf the Web by using Internet Explorer 8 on Windows 7 with our detection system activated. We choose Internet Explorer 8 since this application usually consumes a large amount of memory and represents one of the main targets of spraying attacks. To mimic a realistic behavior, the users kept a tab open on Gmail, and then alternately opened three other tabs performing memory intensive activities: watching videos on YouTube, browsing Facebook, and checking hundreds of pictures on 9gag.

To have an overhead comparison with NOZZLE, we follow the NOZZLE approach and select a sampling rate of 10% (number of pages checked by our detection module over the total number of pages allocated). As a reference, with this value NOZZLE introduces an overhead of 20% to Internet Explorer. Instead the overhead obtained with GLYPH algorithm

in the worst case was 8% with entropy detection enabled and 10% with n-grams approach enabled by checking more than 5,000 memory pages. This result shows the GLYPH algorithm outperforms NOZZLE.

From an false positive of view, NOZZLE shows an average of 10% FPR on certain websites belong to Alexa Top-150 domains. Most of the false positives are coming from the fact that the heap object contains some data page that can be interpreted as a shellcode attack vector. GLYPH is more accurate and on the Top-1,000 Alexa domains has an FPR between 1.1% and 2.4% when using entropy features, and an FPR of 0% when using n-grams. It is important to note that from a design point of view our system is more agnostic compared with other state-of-the-art methods, GRAFFITI's heuristics included. Moreover our system is more resilient to mimicry attacks, as showed in §V-C, since it is not related to a specific attack exploitation technique and not affected by benign background noise, and can be used to defend any application that runs in any Operating System.

VI. DISCUSSION

A. Heap Spraying on 64-bit Architectures

For our experimental evaluation, we focused on the Intel x86 32-bit architecture. However, our results can be generalized, and our work could be also used for detecting heap spraying attacks on 64-bit architectures.

Applying heap spraying techniques on 64-bit processes is generally harder due to the increasing amount of variability created by randomization techniques (e.g., ASLR). On 64-bit platforms the use of memory randomization techniques makes the address space to be sprayed larger and hence the attack, in general, is not feasible anymore. One example in this direction is Windows 8, which uses two major changes to make heap spraying more challenging on 64-bit architectures. First, it uses HiASLR that enables greater entropy for ASLR. On 64-bit platforms, HiASLR introduces a 1TB range of possible addresses for the base of the heap. This makes it harder to predict the address of memory objects on the heap. Second, Windows 8 makes allocations non-deterministic: when you allocate an object using the default allocator, the position that is used is randomized (i.e., no longer deterministic), introducing fine-grained randomization at the individual object level.

However, there are several attack examples for bypassing such protections. For instance, when the attacker has partial knowledge of a pointer value (or where some object could be located in memory) [23]. In this particular attack case (e.g, Internet Explorer 11 on 64-bit architecture), the attacker uses a heap spray to make the exploit reliable. More precisely, the attacker triggers a write to address A+256MB, where A is the address of some heap object. Due to ASLR, the attacker cannot predict the exact value of A's memory address, and due to the 64-bit heap, they cannot spray enough memory to fill all of the heap—however, it is enough to spray around 256MB of data into the heap. This makes it likely that the address of a random heap object, plus 256MB, will land in the sprayed region. With partial knowledge of such an address, the attacker can mount a successful attack [23]. Heap spraying on 64-architecture can still be useful for the attacker if he can make a vulnerable application dereference memory at a valid heap address plus a large offset. For instance, consider the buggy code *do_something_with(a[i])*, where i might be an offset that points past the end of the array. Other examples of heap-spraying attacks on 64-bit architectures are reported by Fratric [23] and Gawlik and Holz [26].

In all such cases our system can be used to detect heap spraying also on 64-bit architectures by using the same design principles introduced in this paper.

B. Dataset

Our dataset consists of various samples which represent realistic memory dumps obtained from heap spraying attacks and benign navigation. Like any dataset, it has limitations in both structure and variety, due to experimental choices and technological reasons. In particular, we elaborate more on the following aspects:

- **Variety of sources**: all of our payloads were built using METASPLOIT, for producing both the NOP sleds and the malicious code. We lack samples from real attacks with payloads coded manually or with different tools, mainly due to the problem of finding and validating a sufficient number of working samples to fit in our machine learning approach.
- **Variety in structure**: we varied the structure of our payloads in four different parts: (1) NOP sled type, (2) NOP sled length, (3) number of repetitions of the malicious payload composed by NOP sled + shellcode, and (4) shellcode type. Although we think that this produces a representative range of variations in our dataset, it is possible to imagine additional changes and combinations for heap spraying payloads.
- **Malicious code form**: in order to bypass the shellcode checks in Internet Explorer, we used a component in METASPLOIT for ASCII-encoding all our malicious payloads (see Section IV). Although this technique has allowed us to effectively bypass the aforementioned checks, we must observe that it reduces the number of machine instructions that can appear inside a malicious payload: besides a prologue composed of binary instructions, the rest of the payload is obviously limited to machine instructions that are also ASCII characters. It is important to note that the metamorphic and polymorphic transformations are independent of the ASCII encoding as shown in [7] where the authors describe a technique for turning an arbitrary ARMv8 code into alphanumeric (ASCII) executable code. The technique is generic and may well apply to other architectures.
- **Payload structure**: finally, all of our samples are in the classic form of a single block of NOP sled plus shellcode, repeated many times. Although this is a valid model for heap spraying attacks, based on real-world exploit code, we cannot exclude the possibility of different, more elaborate payloads. We have made an effort to generalize by using complex NOP sleds always made of different instructions, but attackers could find

other ingenious techniques to vary the structure of the payload, for instance by slightly modifying the single block at each repetition.

In summary, we believe that these limitations are natural, as attackers have a broad range of techniques for constructing attack variations, given a model. However, as shown in §V, we believe that we have demonstrated GLYPH to be agnostic and resilient with respect to the variations we introduced, and successful in classifying samples from a realistic attack model.

VII. RELATED WORK

We compare our work with the state of the art of heap/JIT/data spraying defenses, and of machine learning techniques for detecting malicious code.

A. Heap Spraying

Researchers have proposed several approaches for detecting heap-spraying attacks [22, 24, 53]. For example, Egele et al. [22] used x86 emulation techniques to defend Web browsers against drive-by download attacks that use heap-spraying code injection. The authors proposed to check for the presence of shellcode by monitoring all strings allocated by the JavaScript interpreter. Their goal is similar to that of NOZZLE [53], which uses static analysis of the objects on the heap to detect heap-spraying attacks. In particular, NOZZLE scans memory objects looking for a sequence of instructions that includes a NOP sled and ends with malicious shellcode. However, as the authors point out, the tool has several drawbacks. For example, attackers can evade detection by using large NOP sleds. Moreover, NOZZLE is also specific to the JavaScript Engine Memory Allocator and it cannot be applied to a generic application. Another work to defend against heap spraying attacks is BuBBLE [24]. In this case, the authors start from the assumption that an attacker needs to spray a large part of the heap memory with homogeneous data (i.e., NOP sled). BuBBLE breaks such an assumption by inserting special values at random positions inside strings before storing them in memory, and removing them when a string is used by the application. Again, this solution is specific to the JavaScript language and cannot be easily ported for the protection of other applications.

Thanks to GRAFFITI [19], our approach is different from the previous ones since it does not require knowing how the memory allocator of a particular interpreter engine works, and consequently it does not require access to source code and is OS-agnostic. Moreover, it can protect any system application as well as kernel subsystems without any assumption about the internals of the protected component.

This paper proposes GLYPH as an extension of the detection engine against heap spraying attacks offered by the original GRAFFITI paper [19]. GRAFFITI uses different modules and specific heuristics for detecting attacks. In particular the detection modules are composed by a Malicious Code Detector engine, a component for detecting self-unpacking shellcode, and several heuristics for activating checks based on memory allocation rates performed by monitored processes. Such heuristics and components are not the main contribution of GRAFFITI, and present some potential limitations as described by the authors in the Security Analysis section of the GRAFFITI paper [19]. Moreover such detection techniques need to be calibrated for avoiding false positives and false negatives. Our agnostic approach instead allows GLYPH to overcome the specificity of GRAFFITI's heuristics by providing a more generic detection technique that is independent on the attack vector itself. By using such mechanism we can detect a large spectrum of heap spraying attacks without knowing how the attack vector is constructed.

B. JIT Spraying

Bania [6] proposed a detection technique based on the fact that in order to force the JIT compiler to generate code, an attacker should use ActionScript arithmetic operators. However, it is not mandatory for JIT spraying attacks to use arithmetic operations. Another JIT spraying defense has been proposed by Hu et al. [29]. This solution consists of a kernel patch, JITsec, that tests for several conditions when a system call is invoked. In particular, the authors argue that an application can maintain its security properties and execute code from the stack and heap by decoupling sensitive code from non-sensitive code and allowing the latter to run from writable memory pages. As a result, such a detector only identifies attacks that directly issue system calls. Mimicry and ROP attacks are therefore not covered by this model. JITDefender [17] is another work based on hardware assisted technologies which aims at defeating JIT Spraying attacks. The system protects the Virtual Machine dynamic memory pages created by the JIT-Compiler and allows for the execution of the pages requested by the VM only. This approach is strictly VM dependent, and can only detect JIT-spraying attacks. Our solution is agnostic to the type of attack, and therefore can successfully detect JIT-spraying attacks without any assumption about the instructions that are used by the attacker.

Finally, Lobotomy [31] proposes mitigating JIT spraying attacks by applying the principle of least privilege to the Firefox JIT engine: by splitting the compiler and executor modules of the engine, to greatly reduce the amount of code that needs to access writable and executable pages. The main drawbacks of Lobotomy, with respect to our approach, are: 1) its overhead, which is higher than ours, and 2) the need to redesign the JIT engine of the protected process. The latter is particularly hindering because it greatly limits the portability of Lobotomy to other JIT engines. On the contrary, GLYPH can seamlessly protect any program, without modifying any of its inner components.

C. Data Spraying

Several defensive solutions have been proposed to avoid pivoting-based techniques [43, 51, 52]. One of the most deployed is part of EMET [43], a solution designed by Microsoft. EMET is a utility that helps to prevent vulnerabilities in software from being successfully exploited. Among other features, EMET also addresses the problem of stack pivoting

attacks by checking if the stack pointer points outside of a process stack boundaries whenever a dangerous API is invoked. However, several researchers proved that it is possible to bypass the EMET technology in many ways [38, 39, 41]. The impact of these studies shows that technologies that operate at the same level of execution of the malicious code need to be extensively tested and carefully designed to offer the desired protection and avoid possible bypasses. Consequently, these studies also show the importance of designing reference monitors that operate at a lower level (e.g., at the hypervisor level) such as GRAFFITI and GLYPH to avoid these trivial attacks. Moreover, Microsoft recently introduced two new countermeasures to hinder browser exploitation: isolated heap and delayed free [33, 45]. Both these techniques raise the bar for use-after-free attacks; as stated by the Fortinet Labs researchers [42], they also make heap manipulation harder, but they are not a general solution as they protect only the Internet Explorer browser.

D. Machine Learning in Malware Analysis

Machine learning has been extensively and successfully applied to malware analysis, both in desktop [4, 35, 36, 49] and mobile [5, 12, 14, 37] settings.

N-grams have been explored extensively as a way to characterize and capture malicious (short) sequences of code instructions or bytes [e.g., 49, 61]. The main challenge to tackle is related to the high dimensionality of the feature space generated by n-grams. One approach to tackle the high-dimensionality problem is to perform *feature selection* after computing the full feature matrix [e.g., 49], so that it is more efficient to train and run a detection algorithm; however, in our setting it is not feasible to precompute the whole feature space since n-grams derived from process memory, and the possible combinations, are exponentially greater than those that can be found in source code and binaries, quickly leading to out-of-memory errors during feature extraction. Another approach relies on *bloom filters* to approximate the content of a memory process [e.g., 61]; however, this approach loses information of the bytes that cause the classification, we did not want to have n-gram collisions due to the nature of bloom filters. Hence, we decided to apply a *mask* of n-grams found in a high variety of spray attacks (§IV); this allowed us to prioritize the relevant n-grams that may distinguish between clean and sprayed processes.

When a malware is heavily obfuscated or encrypted, n-grams may not be sufficient by themselves. Hence, many approaches based on Shannon entropy [54] have been explored for malware detection, in particular for packed and encrypted malware [e.g., 12, 36]. Under the intuition that a spray will also affect the bytes distribution in the process memory—given the implicitly disruptive nature of a spraying attack [19]—we decided to explore the use of entropy as a feature for anomaly detection as well.

To the best of our knowledge, this is the first paper that explores the use of machine learning techniques for detecting heap spraying.

VIII. CONCLUSION

This paper extends GRAFFITI [19] by proposing GLYPH, which for the first time explores the use of machine learning for heap spraying detection via runtime page-level memory monitoring. Evaluations on a representative dataset of more than 400 process dumps demonstrate GLYPH's efficiency, effectiveness, and resiliency against different heap spraying attack strategies. In particular, we identify two major configuration modes for GLYPH: one based on information entropy, which supports very fast execution, and does not require a priori knowledge on heap spray variants, but suffers from false positives; one based on memory n-grams, which is more computationally demanding, requires some a priori knowledge on the heap spray attack variants, but achieves perfect accuracy. We show that both modes outperform NOZZLE [53] in terms of both detection performance and runtime overhead.

Future work may explore the feasibility of generating problem-space adversarial ML attacks [50] in the context of heap spraying and other memory corruptions. That is, adversarial ML attacks that do not work solely on the feature space, but for which also a feasible and inconspicuous real-world exploit can be generated and executed to evade the ML-based detection classifier. Moreover, it would be interesting to explore higher-level abstractions of process memory which could provide more explainable predictions.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments which helped improve the quality of this work. This project has received funding from the Italian Ministry of Foreign Affairs and International Cooperation (grant number: PGR00814).

REFERENCES

[1] Autoit. https://www.autoitscript.com/site/autoit/.
[2] NOP Generators in MetaSploit. https://www.coursehero.com/file/p48l4qq/NOP-Generators-Metasploits-NOP-generators-are-designed-to-produce-a-sequence-of/, Visited May 2020.
[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 2009.
[4] B. Alsulami, A. Srinivasan, H. Dong, and S. Mancoridis. Lightweight behavioral malware detection for windows platforms. In *MALWARE*. IEEE, 2017.
[5] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2014.
[6] P. Bania. JIT spraying and mitigations. *CoRR*, 2010.
[7] H. Barral, H. Ferradi, R. Géraud, G. Jaloyan, and D. Naccache. ARMv8 Shellcodes from 'A' to 'Z'. *CoRR*, abs/1608.03415, 2016.
[8] E. Berger and B. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *SIGPLAN*, 2006.
[9] E. Bhatkar, D. C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proc. of USENIX Security Symposium*, 2003.
[10] L. Bilge, A. Lanzi, and D. Balzarotti. Thwarting real-time dynamic unpacking. In *Proc. of the European Workshop on System Security (EUROSEC)*, 2011.
[11] C. M. Bishop. *Pattern Recognition and Machine Learning*. 2006.
[12] G. Canfora, F. Mercaldo, and C. A. Visaggio. An HMM and structural entropy based detector for android malware: An empirical study. *Computers & Security*, 2016.

[13] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2012.

[14] T. Chakraborty, F. Pierazzi, and V. S. Subrahmanian. EC2: Ensemble Clustering and Classification for Predicting Android Malware Families. *IEEE Trans. Dependable and Secure Computing (TDSC)*, 2020.

[15] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 2009.

[16] L. Chen and Q. He. Shooting the osx el capitan kernel like a sniper. https://speakerdeck.com/flankerhqd/shootingthe-osx-el-capitan-kernel-like-asniper.

[17] P. Chen, Y. Fang, B. Mao, and L. Xie. JITDefender: A Defense against JIT Spraying Attacks. In *Future Challenges in Security and Privacy for Academia and Industry*. Springer Berlin Heidelberg, 2011.

[18] 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 *Proc. of USENIX Security Symposium*, 1998.

[19] S. Cristalli, M. Pagnozzi, M. Graziano, A. Lanzi, and D. Balzarotti. Micro-Virtualization Memory Tracing to Detect and Prevent Spraying Attacks. In *Proc. of USENIX Security Symposium*, 2016.

[20] M. Daniel, J. Honoroff, and C. Miller. Engineering Heap Overflow exploits with Javascript. In *USENIX Security Symposium*, 2008.

[21] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, machine learning can be more secure! A case study on android malware detection. *IEEE Trans. Dependable and Secure Computing (TDSC)*, 2017.

[22] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proc. of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Springer, 2009.

[23] I. Fratric. Exploiting internet eplorer 11 64-bit on windows 8.1. http://ifsec.blogspot.com/2013/11/exploiting-internet-explorer-11-64-bit.html, 2013.

[24] F. Gadaleta, Y. Younan, and W. Joosen. BuBBle: A Javascript Engine Level Countermeasure against Heap-Spraying Attacks. In *Engineering Secure Software and Systems (ESSoS)*. Springer Berlin Heidelberg, 2010.

[25] H. Gascon, S. Ullrich, B. Stritter, and K. Rieck. Reading between the lines: Content-agnostic detection of spear-phishing emails. In *Proc. of Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Springer, 2018.

[26] R. Gawlik and T. Holz. Sok: Make jit-spray great again. In *Proc. of USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[27] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning (ELS)*. Springer, 2009.

[28] Y.-C. Ho and D. L. Pepyne. Simple explanation of the no-free-lunch theorem and its implications. *Journal of Optimization Theory and Applications*, 2002.

[29] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proc. of Int. Conference on Virtual Execution Environments*. ACM, 2006.

[30] K. R. Irvine et al. *Assembly language for Intel-based computers*. Citeseer, 2003.

[31] M. Jauernig, M. Neugschwandtner, C. Platzer, and P. M. Comparetti. Lobotomy: An architecture for jit spraying mitigation. In *Proc. of the Internationalence on Availability, Reliability and Security (ARES)*, 2014.

[32] D. Kennedy, J. O'gorman, D. Kearns, and M. Aharoni. *Metasploit: the penetration tester's guide*. No Starch Press, 2011.

[33] M. Labs. Isolated heap and friends - object allocation hardening in web browsers. https://labs.mwrinfosecurity.com/blog/2014/06/20/isolated-heap-friends---objectallocation-hardening-in-web-browsers/.

[34] L. Li, J. E. Just, and R. Sekar. Address-space randomization for windows systems. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2006.

[35] M. H. Ligh, A. Case, J. Levy, and A. Walters. *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons, 2014.

[36] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 2007.

[37] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2017.

[38] Bromium Labs. Bypassing EMET 4.1. http://bromiumlabs.files.wordpress.com/2014/02/bypassing-emet-4-1.pdf, 2014.

[39] Duo Security. Wow64 and so can you bypassing emet with a single instruction. https://duo.com/assets/pdf/wow-64-and-so-can-you.pdf.

[40] eEye Research. Microsoft internet information services remote buffer overflow (system level access). https://web.archive.org/web/20061026101830/http://research.eeye.com/html/advisories/published/AD20010618.html, 2006.

[41] FireEye. Using EMET to disable EMET. https://www.fireeye.com/blog/threatresearch/2016/02/using_emet_to_disabl.html, 2016.

[42] Fortinet Labs. Is use-after-free exploitation dead? the new ie memory protector will tell you. http://blog.fortinet.com/.

[43] Microsoft. The enhanced mitigation experience toolkit. http://support.microsoft.com/kb/2458544, 2017.

[44] Team Teso. Exploit "7350854.c". https://www.exploit-db.com/exploits/409/, 2001.

[45] Trendmicro Labs. Mitigating UAF Exploits with Delay Free for Internet Explorer. https://blog.trendmicro.com/trendlabs-security-intelligence/mitigating-uaf-exploits-with-delay-free-for-internet-explorer/, 2014.

[46] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *Communication of the ACM (CACM)*, 2008.

[47] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2010.

[48] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. *Proc. of USENIX Security Symposium*, 2019.

[49] R. Perdisci, A. Lanzi, and W. Lee. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2008.

[50] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. Intriguing Properties of Adversarial ML Attacks in the Problem Space. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2020.

[51] A. Prakash and H. Yin. Defeating ROP Through Denial of Stack Pivot. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2015.

[52] R. Qiao, M. Zhang, and R. Sekar. A Principled Approach for ROP Defense. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2015.

[53] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In *Proc. of USENIX Security Symposium*, 2009.

[54] C. E. Shannon. Prediction and entropy of printed english. *Bell system technical journal*, 30(1):50–64, 1951.

[55] M. Sikorski and A. Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. No Starch Press, 2012.

[56] Skylined. Heap spraying high addresses in 32-bit chrome/firefox on 64-bit windows. http://blog.skylined.nl/20160622001.html.

[57] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*.

[58] A. Sotirov. Heap Feng Shui in Javascript. *Black Hat Europe*, 2007.

[59] C. Valasek. heapLib 2.0. https://ioactive.com/heaplib-2-0/, 2013.

[60] A. Warnecke, D. Arp, C. Wressnegger, and K. Rieck. Evaluating explanation methods for deep learning in security. *Proc. of IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[61] C. Wressnegger, G. Schwenk, D. Arp, and K. Rieck. A close look on n-grams in intrusion detection: anomaly detection vs. classification. In *Proc. of ACM Workshop on Artificial Intelligence and Security (AISEC)*, 2013.

Appendix

A. Symbol Table

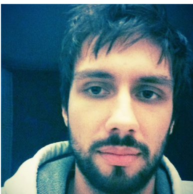
Table VII provides a reference for notation, acronyms, and major symbols used throughout the paper.

TABLE VII. Symbol Table.

Symbol	Description
SVM	(Linear) Support Vector Machine.
DT	Decision Tree.
RF	Random Forest.
SVM+RF	Ensemble classifier based on majority voting between SVM and RF. In practice, ensemble prediction corresponds to label 1 if at least one classifier predicts label 1.
C	SVM hyperparameter for regularization-loss trade-off.
k	Number of trees in the forest (RF hyperparameter).
m	Maximum tree depth (RF hyperparameter).
l	Maximum number of leaf nodes (RF hyperparameter).
$\mathcal{X}$	Feature space.
P_i	Process i. A process is represented as its sequence of bytes in RAM.
n_i	3-grams set of process P_i.
$\boldsymbol{x}_i$	Feature vector corresponding to process i. Vectors are represented in bold, and vector elements are in italic. For example: $x_j \in \boldsymbol{x}_i$ (element x_j that belongs to vector $\boldsymbol{x}_i$).
y_i	Ground truth label of process P_i. If $y_i = 0$, P_i is a benign process; if $y_i = 1$, P_i is a sprayed process.
$\hat{y}_i$	Predicted label of process P_i. If $\hat{y}_i = 0$, P_i is predicted as a benign process; if $\hat{y}_i = 1$, P_i is predicted as a sprayed process.

Biographies

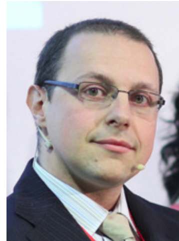
Fabio Pierazzi is a Lecturer (Assistant Professor) at the Department of Informatics at King's College London. His research interests lie at the intersection of AI and cybersecurity, with particular focus on intrusion detection, adversarial ML, and systems security. He completed his Ph.D. in Computer Science in 2017 at the University of Modena, Italy. He spent most of 2016 as a visiting scholar at University of Maryland, College Park (US), and held a two-year PostDoc in the UK at the Systems Security Research Lab (S2Lab). Home page: https://fabio.pierazzi.com



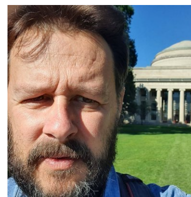
Stefano Cristalli got his Ph.D in computer science from University of Milan (2019). During his Ph.D he has worked on several security projects with the aim of protecting applications from sophisticated attacks. In particular his main area of research deals with software protection, program analysis and automatic exploit generation.

Danilo Bruschi is Full Professor in Computer Science at University of Milan where he leads the Security Lab called Laser. He received a Ph.D. in Computer Science from University of Milan, and he was a honorary fellow at University of Wisconsin, Madison. He is one of the pioneers of the systems security field in Italy. His research interests cover several area of the cyber security such as: System Security, Operating System, Computer Forensics. Homepage: http://bruschi.di.unimi.it/bruschi/Danilo_Bruschis_Home.html

Michele Colajanni is Full Professor in Computer Engineering at the University of Modena and Reggio Emilia since 2000. He received a Master degree in Computer Science from the University of Pisa, and a Ph.D. degree in Computer Engineering from the University of Roma in 1992. He manages the Interdepartmental Research Center on Security and Safety (CRIS). His research interests include the security of large scale systems, performance and prediction models. Homepage: https://weblab.ing.unimo.it/people/colajanni/



Mirco Marchetti is an Associate Professor at the Department of Engineering "Enzo Ferrari" of the University of Modena and Reggio Emilia (Italy). He received a Ph.D. in Information and Communication Technologies in 2009. His research interests include all aspects of system and network security, security for cyber-physical systems and automotive, cryptography applied to cloud security and outsourced data and services. Homepage: https://weblab.ing.unimore.it/people/marchetti



Andrea Lanzi is an Associate Professor in Computer Science at the University of Milan. He is interested in several aspects of Cyber Security. In particular, his main area of research deals with Host Intrusion Detection Systems (HIDS), memory errors exploitation, reverse engineering, malware and forensic analysis. In recent years he has mainly studied the application of emulation/virtualization and compiler techniques for malware analysis and detection in Android context. In addition he has been working on analyzing large-scale security malware datasets to investigate the behavior of current cyber threats. Homepage: http://lanzi.di.unimi.it