Defense of the Clones: Securing Web Applications with Automatic Honeypot Generation and Deployment

Billy Tsouvalas
Stony Brook University
Stony Brook, NY, USA
vtsouvalas@cs.stonybrook.edu

Nick Nikiforakis
Stony Brook University
Stony Brook, NY, USA
nick@cs.stonybrook.edu

***Abstract*—In this paper, we introduce PARALLAX, an automatic, application-agnostic, and resource-efficient web application honeypot generation and deployment framework. PARALLAX can generate honeypot clones of any live LAMP stack, without interfering with the availability of the web application, and deploys the clones alongside the original web application. In the PARALLAX-based network deployment, all attackers are seamlessly and covertly redirected to the honeypot clone, while benign visitors may continue their interaction with the original web application, same as before. Alongside PARALLAX, we introduce three independent sensitive data detection schemes, which we employ to isolate and replace the sensitive data of the original web application on the honeypot clone. As we allow attackers full interaction with all parts of the honeypot clone, we replace the sensitive data on the honeypot with realistic, context-aware, synthetic data using an LLM to ensure that none of the sensitive data of the original web application are compromised by attackers. To evaluate PARALLAX, we deploy it in the wild for five open-source web applications, and we examine the honeypot generation and deployment performance, as well as the interaction of attackers with the honeypot clones. Lastly, to evaluate the deceptive capability of the synthetically generated data, we conduct a large-scale user study and evaluate how well humans are able to differentiate between real and synthetic sensitive data.**

I. INTRODUCTION

In the last two decades, web applications have become ubiquitous and account for the majority of online activity. Encompassing all interactive online tasks, and spanning from simple blogs and news outlets to more elaborate social media and e-commerce sites, web applications are critical elements of online infrastructure and business operations. As such, proactively protecting web applications and ensuring that malicious actors are not able to carry out their attacks is crucial and affects the entire online landscape.

To protect web applications against cyber attacks, defenders focus on detecting the perpetrator who is attempting to breach the system's defenses, and subsequently deny them further interaction with the web application. In this manner, after they have been detected, malicious actors are blocked and are disallowed to carry out their attack. However, both the detection of intruders, as well as the attempts of defenders to isolate or block attackers are not immune to error, false negatives, and general inaccuracies, which could lead to easily exploitable cracks in the defense of the system. While the weaknesses of different detection schemes are well understood, e.g., inability of signature-based detection to deal with novel attacks, blocking an attacker also introduces second order effects to the security posture of a web application, that are rarely considered.

Blocking a detected attacker, primarily refers to denying them access, i.e., based on an IP blocklist, or by employing some other fingerprinting or threat intelligence technique. However, attackers may simply use a different IP address, or hide their characteristics and attempt the attack again. Thus, blocking is only efficient against easily discouraged attackers, but not against sophisticated Advanced Persistent Threats (APTs). Furthermore, the overt and transparent blocking, inadvertently provides the attacker with target intelligence on the defenses employed by the system. Thus, by simply using fresh IPs, an attacker is able to iteratively gain crucial information on the security posture of the system, at very low cost. Overall, by using rather uncomplicated concealment techniques, attackers can render the defenders' intelligence irrelevant, while they are collecting information on the system's security posture, unwillingly offered by the defenders.

The monolithic nature of mitigating attacks by simply blocking malicious actors has been widely exploited by attackers, who use a multitude of different evasion techniques, ranging from rather uncomplicated IP address spoofing and rotation, and more elaborate fingerprint concelament methods. Such techniques have been employed in the context of many attack vectors, with incidents including large-scale availability attacks, i.e., the GitHub DDoS attack of 2018, which originated from tens of thousands of unique endpoints [1], [2], and even password brute force attacks targeting VPN devices and using more than 2.7 million source IP addresses [3]. In 2024, the US Justice Department managed to disrupt the 911 S5 botnet, which had infected over 19 million unique IP addresses and was offerring these IP addresses to cybercriminals for a fee [4]. As is readily apparent, attackers are heavily relying on using a large number of IP addresses to bypass both the detection and mitigation measures of defenders time and time again.

979-8-3315-8969-1/25/$31.00 ©2025 IEEE.

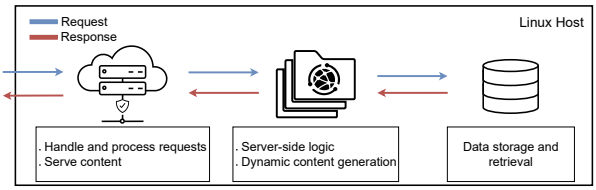

Fig. 1: LAMP stack request and response flow

Recognizing the aforementioned drawbacks of simply blocking attackers, in order to protect the web application, we propose PARALLAX, a deception-based approach, which consists of covertly redirecting the attackers away from the web application and allowing them to actively interact with a honeypot, which is a seemingly exact replica of the real web application. On that honeypot, all the sensitive data of the original database have been replaced with synthetic ones, thus not jeopardizing the data of the original web application. In this manner, we are able to conspicuously lead the attacker away from the real web application, and observe their behavior, while they are wasting time and resources on a honeypot version of the web application. PARALLAX is a resource-efficient honeypot generation and deployment framework, which isolates attackers on the honeypot clone, while it does not interfere with benign users. This paper makes the following original contributions:

- We introduce PARALLAX, an automatic, application-agnostic honeypot generation and deployment framework, which, given a live web application, generates its clone, and deploys it alongside the original web application.
- Along with PARALLAX, we present three different, automatic sensitive data detection schemes, which we employ to detect the original web application sensitive data. This sensitive data is subsequently replaced with realistic, context-aware, synthetic data for the honeypot clone.
- We demonstrate the security properties of PARALLAX by deploying five different PARALLAX-protected web applications in the wild.
- We evaluate the deceptive capability of the honeypot clones and the synthetically generated content in a large scale user study where real participants are attempting to distinguish real from synthetic data.

II. MOTIVATION AND BACKGROUND

Considering that blocking an attacker is the primary and, oftentimes, only means of attack mitigation employed by defenders, it comes as no surprise that attackers have found multiple ways to bypass it. Dynamic IP address rotation [5], [6] is often used by attackers to change their IP addresses frequently to evade blocking. These evasion techniques are carried out using VPN services [4], [6], large-scale botnets [3], [7], and even residential proxies, which allow attackers to hide behind the legitimacy of residential networks [8]. On the other hand, attackers also leverage more elaborate evasion techniques, such as breaking CAPTCHAs [9] and bypassing JavaScript challenges [10], as well as fingerprinting evasion methods, i.e., mimicking human behavior, and randomizing request patterns. Such techniques have been employed by attackers for a range of attack vectors, from content scraping [8], to large-scale credential stuffing and password brute-force attacks [3], [5], and even sweeping operational disruptions in the form of DDoS [2].

To counter IP address rotation, as well as more sophisticated mitigation evasion techniques, defenders have employed deception-based defenses, ranging from small-scale self-contained elements, i.e., honeytokens [11], [12], honeywords [13], and decoy files [14]–[18], all the way to high-interaction honeypots that emulate entire applications [19]–[21]. Along with countering the evasion techniques, by deploying realistic honeypots, defenders are able to keep attackers engaged and further fingerprint them, thus collecting more comprehensive threat intelligence. Moreover, by employing honeypots, attackers are unable to collect reliable or actionable information against the targeted system's defenses. Since all interactions with the honeypot are controlled by the defenders, the system is protected against target fingerprinting techniques i.e., probing the target system, or actively fingerprinting the intrusion detection rules. However, although honeypots offer comparative advantages to simply blocking attackers, the generation of realistic, enticing honeypots that attract attackers attention is not a trivial task.

Realistic high-interaction honeypots are more effective at deceiving attackers into believing that they are interacting with a real system. To capture accurate data on attacker behavior, the honeypot needs to be enticing and manage to actively deceive attackers. While honeypots that simulate or emulate web applications can effectively mimic real-world environments, they often lack in authenticity, offer limited dynamic interaction, and may inadvertently inform attackers of the deception in-play. We can alleviate all the aforementioned by using a clone of a web application as a honeypot.

Overall, using clones of web applications as honeypots offers several advantages. First, cloned web applications are exact replicas of the original production application and are more convincing to attackers than emulators or simulators. This authenticity allows for more accurate data collection on the attacker behavior during their interaction with the honeypot. Moreover, given that a honeypot clone is a web application in its own right, it implements the same dynamic behavior as the original web application, thus making it even more difficult for attackers to distinguish the honeypot from the real system. Thus, honeypot clones inherit the advantages of classic high-interaction honeypots, while improving on their drawbacks.

Although honeypot clones are a powerful weapon in the defenders' arsenal, their generation and deployment introduces several technical challenges. First, generating the honeypot clone from a production-grade web application is an involved procedure, given the many different and complex web applications, the non-standardization and variance of the different codebases, the necessary scalability and flexibility constraints, the multitude of different network and database deployment configurations, etc. Furthermore, considering that a honeypot

is a security resource whose value lies in being probed, attacked, or compromised [22], their generation needs to ensure that the honeypot is properly and sufficiently isolated from the production host system, in order to prohibit attackers from using the host system as a stepping stone to further attacks. Lastly, cloning a web application would mean that sensitive data is potentially copied over to the clone. It is crucial that any sensitive data of the original web application are removed or sanitized before the honeypot gets deployed, so as to avoid their accidental exposure.

Specifically, looking at LAMP stacks, the most popular web application framework, both the individual component complexity (Linux, Apache, MySQL, PHP), as well as the intricacy of the interactions between components make the task of generating a honeypot clone of the web application arduous. The challenge of isolating, extracting and cloning entire LAMP stacks at scale is further encumbered since a request to a LAMP-based web application needs to pass through all the layers of the stack to be served, i.e., the web server serving the request to PHP, PHP interacting with the database, the generated content sent back to the web server to finally reach the browser. A diagram depicting the request and response flows through the different components of a LAMP stack is provided in Fig. 1.

Considering the aforementioned challenges inherent in developing honeypot clones, we highlight that there exists no framework or software that can automatically and agnostically generate a honeypot clone of any given LAMP stack. Although dedicated application or host specific software, mainly in the form of plugins, are able to carry out web application cloning tasks (e.g., Duplicator [23], Migrate Guru [24], Jetpack VaultPress Backup [25], Softaculous [26]), we note that their primary objective is resource migration, and not honeypot deployment, while none of them can carry out the task agnostically.

Recognizing the complexity of LAMP stacks, as well as the aforementioned challenges regarding the generation and deployment of web application honeypots, in the present paper, we introduce PARALLAX, which automatically generates and deploys realistic, high-interaction honeypot clones of any *live* LAMP stack web application. PARALLAX is application-agnostic, requires zero downtime, and is resource-efficient, thus alleviating the scalability and complexity concerns of generating a clone of a LAMP stack. Furthermore, PARALLAX detects and replaces the sensitive web application data on the honeypot clone, and employs a containerized deployment for the clone to ensure sufficient resource isolation.

A. Research Questions

a) ***RQ1: Can LAMP stacks be cloned automatically, agnostically, and in a performant manner?****:* Given the complexity of LAMP stack web applications, we wish to evaluate whether generating and deploying honeypot clones of live web applications can be done without human intervention, independent of the application, and in a resource-efficient manner.

b) ***RQ2: Will real attackers engage with our* PARALLAX *generated honeypots?****:* Having generated and deployed the honeypot clones alongside the real web applications, is PARALLAX able to deceive attackers into actively engaging with the honeypots containing synthetic, context-aware data?

c) ***RQ3: Can humans differentiate between real sensitive information vs. synthetically generated ones?****:* We wish to evaluate the deceptive capabilities of PARALLAX, by determining whether human users can distinguish the real sensitive data from the synthetically generated, context aware ones, which we use to populate the honeypot clones.

III. DESIGN

PARALLAX is a deception-based security framework that can automatically generate and deploy honeypot clones of any live LAMP stack web application. The honeypot clone itself is a LAMP stack in its own right and is deployed alongside the original web application, in a "parallel world" setting, meaning that attackers are internally redirected to the honeypot clone, while benign users are directed to the real web application, same as before. In order to ensure that none of the sensitive data of the original web application can be accessed by attackers that are allowed to reach the honeypot clone, we detect and replace all sensitive database entries with realistic, context-aware, synthetic data. The main components of PARALLAX are the three; (i) the honeypot generation, (ii) the parallel world network deployment, and (iii) the sensitive data replacement taking place at the honeypot clone. In this section we outline each component and discuss the design goals of the system.

A. Honeypot Generation

The goal of this component is to generate a clone of a LAMP stack and leverage it as a honeypot. In terms of its design, our objective is for the honeypot generation to be automatic, application-agnostic, lightweight, and resource-efficient. Furthermore, the honeypot generation should not interrupt the availability of the original web application, or interfere with its functionality. In order to generate a clone of the original LAMP stack, we need three components: the web application directory, the database linked with the web appication, and the network configuration, i.e., the domain name, and the TLS certificates. PARALLAX can locate and extract these components agnostically, without interfering with the live web application.

B. Network Deployment

The objective of the honeypot clone deployment alongside the original web application entails the adaptation of the original network deployment to include a web server configuration that will not interefere with the benign visitors, but will seamlessly and covertly redirect the malicious and unwanted visitors to the honeypot clone. In order to achieve this, PARALLAX modifies the original web server configuration and employs a scalable and dynamic web server, which allows us to implement an intrusion detection mechanism and seamless redirection of the visitor to the appropriate instance of the web application, i.e.,

the real web application for benign users and the honeypot clone for attackers.

C. Sensitive Data Replacement

The parallel world network setting redirects attackers to the honeypot clone of the real web application. Although this version of the web application is an exact clone of the live LAMP stack, we want to remove any sensitive data from the database, to ensure that attackers can not access it. Considering that the honeypot clone needs to remain realistic and enticing to attackers, in order to encourage their further engagement, we replace the sensitive data with synthetic, context-aware data. First, we employ three different sensitive data detection schemes to locate the data we wish to replace; (i) heuristics and entropy, (ii) role-based, and (iii) Large Language Model (LLM). Then, we use an LLM to replace the sensitive data, and we deploy the updated database on the honeypot clone.

D. Threat Model

Given the application-agnostic nature of PARALLAX, it can be used to automatically generate and deploy a honeypot clone of any LAMP stack. Since it does not interfere with the live web application at any stage of the honeypot generation or the network deployment, it allows defenders to covertly install PARALLAX and benefit from a honeypot clone alongside the original web application. While all LAMP stacks could alleviate critical security issues by employing PARALLAX, it is best suited for web applications that are experiencing high volumes of malicious traffic.

E. System Properties

Automatic generation and deployment. PARALLAX does not require any user intervention during the generation or the deployment phases. The entire procedure is implemented in a single Ansible playbook, which automatically carries out all the necessary tasks.

Application-agnostic: The PARALLAX requirements are minimal and readily available to any web application administrator. The entire functionality only requires the IP address and root-level host credentials, along with the web application credentials. PARALLAX does not require any information regarding the deployed web application, and is fully application-agnostic, readily deployable for any live LAMP stack.

Minimal overhead. The only overhead that is introduced by PARALLAX is isolated in the one-time honeypot generation and deployment. Following the deployment of PARALLAX, the network configuration introduces no measurable overhead.

Non-interference. The generation and deployment procedure does not interfere with the live web application's availabilty.

Isolation/separation guarantee. Given the deployment of the honeypot clone using Docker containers on a separate host from the original web application, PARALLAX guarantees isolation of the original web application and its clone. Furthermore, PARALLAX automatically finds and extracts only those components that are used by the LAMP stack, not interfering with any other parts of the original web application host.

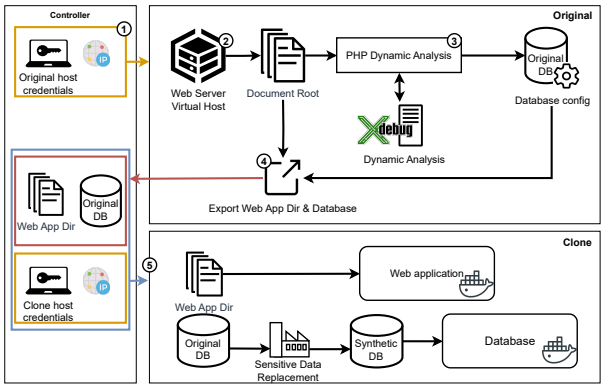

Fig. 2: PARALLAX honeypot generation pipeline

No special software/hardware necessary. PARALLAX does not require any special software or hardware, and uses open-source and readily available frameworks.

IV. ARCHITECTURE

In this section, we elaborate upon the architecture of the three main components of PARALLAX; (i) honeypot generation, (ii) network deployment, (iii) sensitive data replacement.

A. Honeypot Generation

The design goals of the honeypot generation component comprise of automatically and agnostically cloning any live LAMP stack, and employing it as a honeypot clone, deployed alongside the original web application. We want that the cloning procedure requires no human intervention, minimal requirements, zero downtime, and no interference with the availability of the original web application. Considering these objectives, in Fig. 2, we present a high-level diagram of the PARALLAX honeypot generation pipeline, and in the following sections, we describe in detail each part of the pipeline.

Requirements ①. The honeypot generation procedure assumes three hosts; a *Controller* host, the host of the original LAMP stack (*Original*), and the host where the honeypot clone will be deployed (*Clone*). Given the popularity of Linux-based websites, web servers, and LAMP stacks in general, we assume that the three hosts are all Linux-based [27]. The *Controller* represents the administrator of the web application, who wishes to enhance the web application defenses by using PARALLAX. Being the initiator and orchestrator of the entire honeypot generation process, the *Controller* needs to have privileged SSH access to the *Original* and the *Clone* hosts. Given the above, the entire honeypot generation process requires only the following:

- The usernane, IP address, and privileged credentials of the *Original* and *Clone* hosts
- The domain of the web application
- The web application credentials

The honeypot clone generation and deployment is a fully automatic procedure using infrastructure-as-Code playbooks [28]. Given the aforementioned requirements, the *Controller* can initiate the entire procedure through a single playbook, which

orchestrates every step of the generation and deployment of the honeypot clone, necessitating no further input from the user.

We underline that the honeypot generation is fully application-agnostic and does not need any information regarding the web application or the relevant database, i.e., web application name, database name, user, password, etc. Furthermore, the entire process is automatic and requires no human input. Given the above information, PARALLAX is able to generate a clone of any live LAMP stack web application, in a non-intrusive, non-interfering, covert manner.

Web Server Virtual Host ②. In every LAMP stack, the web server operates on the directory that contains all the web application files to serve active and passive content. We locate the web application directory starting from the web server. In particular, when working with a LAMP stack, we start from the enabled web server, i.e., Apache configuration files. These configuration files are the ones that the Apache web server is using to handle and process requests, and contain the web application directory in the form of a *DocumentRoot* primitive. By extracting the value of *DocumentRoot*, we have located the web application directory.

PHP Dynamic Analysis ③. Having located the web application directory, PARALLAX can proceed with locating the database linked with the web application. The overall objective is to be able to extract the database from the *Original* host system and transfer it to the *Clone*. In order to access and extract a database (in the case of LAMP stacks, a relational MySQL database), we only need the name and system port of the database, along with the database credentials.

In order to find the necessary database information agnostically, we will leverage the fact that web applications communicate with their database to retrieve and serve stored content. Having already located the web application directory, we have access to the PHP codebase, which initiates connections with the linked database when the web application is live. We employ dynamic analysis over the PHP codebase of the web application to locate and isolate the connections that the web application initiates with the database.

Specifically, we deploy a PHP *miniserver*, which is a PHP server running in parallel with the live web application, on an arbitrary port of the *Original* host sytem. Since the *miniserver* is deployed using the original web application's codebase, we essentially deploy a version of the web application on a different port of the system. Thus, we are able to analyse the *miniserver* version of the original web application, without interfering with the live LAMP stack or disrupting its availability.

In order to dynamically analyse the web application codebase, we use the Xdebug extension on the PHP engine of the deployed *miniserver* [29]. Using Xdebug allows us to collect function traces of the deployed web application, including all function calls, parameters, and return values. We initiate the function trace process and make a single headless request to the *index.php* page of the *miniserver* web application. After the request has been served, PARALLAX has collected the relevant function trace and the *miniserver* can be shut down.

Given the function trace, we wish to find those PHP function calls that initiate database connections, and extract the arguments pertaining to those connections, i.e., the database name, user, password, host, etc. In PHP, there are two main ways to connect to a MySQL database; (i) using the MySQLi extension, and (ii) using PHP Data Objects, otherwise known as PDO (the MySQLi extension may be employed in either a procedural, or an object oriented manner; however, the function names, parameters, and syntax remain the same in both cases). While the MySQLi extension employs predefined PHP functions to connect to a database and perform operations, PDO is a database abstraction which constructs a data object representing the connection to the database, and then proceeds to issue queries and fetch data from the database through an instance of that data object. Since we know the ways that a database connection may be initiated in PHP, PARALLAX parses the Xdebug function trace for instances of the predefined MySQLi functions that are linked to database connections,(i.e., *mysqli_real_connect*, *mysqli_connect*, *mysqli_select_db*, *mysqli_stat*), as well as PDO constructors, which represent database objects.

Using Xdebug on the deployed *miniserver*, we extract these parameters. As a result, we are able to obtain the database configuration of any live LAMP stack. With this information, we can export the database MySQL dump of the database linked to the web application, to use in the honeypot clone. We underline that all of the dynamic analysis is taking place on the *miniserver*, meaning that we extract the database information without interrupting the live web application.

Export Web Application Directory & Database ④. As mentioned above, we locate the web application directory using the enabled web server virtual hosts, and we extract the database configuration and export the MySQL database dump using our PHP dynamic analysis implementation on the Xdebug-powered *miniserver* deployed in parallel to the live LAMP stack. The next step of the honeypot clone generation pipeline involves sending the entire web application directory, the database configuration (name, username, password, host, port), and the MySQL dump back to the *Controller*. In turn, the *Controller*, will send the aforementioned elements to the *Clone* host to begin the honeypot generation and deployment.

Honeypot Clone Deployment ⑤. As our primary objective, we want to deploy an exact replica of the original live LAMP stack on the *Clone* host. However, since this replica will be used as a honeypot, we have to account for the fact that attackers will be redirected to it and will engage and actively interact with its resources. Considering such system isolation security concerns, as well as the significant advantages of containerization, i.e., portability, scalability, resource efficiency, ease of replication and maintenance, we deploy the honeypot clone web application as Docker containers.

The honeypot clone is made up of two Docker containers; the web application and the database. For the web application container, we build a custom image on top of an Apache-based PHP Docker image. This custom image contains the extracted

web application directory from the *Original* host. On the other hand, the database container is a MySQL image that uses a persistent volume, which allows us to copy the extracted MySQL dump inside the container, and import it into the container's MySQL server. Given the persistent volume module, which allows us to copy data into the database container, we are able to import any database into the container. In Section IV-C, we elaborate upon the detection and replacement of the sensitive material of the original database, and we will describe how we import the synthetic database into the database container.
The honeypot clone containers are automatically deployed by the *Controller*. The images are built and deployed using custom-made *Dockerfile* and *docker-compose* files, which are automatically updated and configured for the honeypot clone generation of the given web application. By building and deploying the aforementioned containers on the *Clone* host, we have successfully generated a containerized honeypot clone version of the original web application. In Section IV-B, we elaborate on the network configuration that implements the parallel world setting of PARALLAX.

B. Network Deployment

We want to implement a parallel world network configuration, in which the honeypot clone that we have generated is deployed alongside the original web application. The new network configuration should not interefere with legitimate visitors to the web application, while it should redirect attackers to the honeypot clone. This means that the experience of benign users should be the same as before PARALLAX was deployed, while malicious actors will be engaged in deceptive defenses. Assuming that the original LAMP stack is employing an Apache web server, we modify the original web server configuration in order to enhance its capabilities and employ it as an intrusion detection mechanism, which will inform the visitor redirection scheme.
In order to implement a scalable redirection logic and dynamically handle requests at scale, we install two new software components on the *Original* host; a reverse proxy (Openresty [30]) and our in-memory key-value database (Redis [31]). The reverse proxy extends the capabilities of a web server and allows us to implement the intrusion detection schemes and redirection logic in the form of Lua scripts on top of the web server [32], while the in-memory database provides us with a lightweight solution to persistently store key-value pairs, represent our firewall rulesets.
The transition from the original network configuration to the new dynamic web server includes the following steps:

1) PARALLAX installs the aforementioned technologies on the *Original* host.
2) PARALLAX uses custom-made web server configuration file templates and updates them with the information extracted from the original web server. This includes the domain names and aliases, the web application directory paths and the path to the SSL certificates.
3) Similarly, custom-made Lua script templates that implement the intrusion detection, redirection, and persistent storage are updated with the relevant information on the databases.
4) The in-memory databases are deployed on arbitrary ports and PARALLAX configures their communication with the new web server.

In Fig. 3, we present a flow diagram for a request arriving to a PARALLAX-powered web application. The PARALLAX network configuration consists of two main components; the redirection logic and the target environments. In the following sections, we elaborate on these components.
1) Redirection Logic: At a high level, the employed redirection scheme focuses on identifying and mitigating malicious actors by monitoring failed authentication attempts on the login page of a web application. A client that submits correct credentials to the login page of the web application is considered to be a benign visitor to the web application, meaning that PARALLAX's reverse proxy directs these request to the original web application. This is exactly the same behavior as before the PARALLAX deployment. On the other hand, when a failed authentication is detected, we redirect all subsequent requests from that IP address to the honeypot clone. If a redirected client attempts to login to the honeypot clone of the web application, they will be allowed to the admin panel of the web application, whatever credentials they provide. This means that any set of credentials will allow a presumed malicious actor to have access to the back end of the honeypot clone. We allow redirected users a grace period of 10 minutes from the moment of initially being redirected, during which they can roam free and and interact fully with the honeypot clone. After the 10 minutes, we block that specific IP, we delete the honeypot clone containers, and we re-deploy them from scratch. In that way, we sanitize the honeypot clone from any changes made by a malicious visitor who had access to the admin panel of the web application.
Client Request ①. A client request to the web application first arrives at the reverse proxy. Since the new web server is installed on the *Original* host, no changes are apparent from the client's perspective. This means that all client requests arrive to the same IP address as before, with that IP address using the same domain name. Furthermore, since all of the failed authentication detection, redirection, and request manipulation is taking place at the reverse proxy level on the *Original* host, PARALLAX is completely covert to all clients. The reverse proxy also handles the domain redirection and the TLS termination. After TLS has been terminated, we proceed with the dynamic part of the redirection scheme.
Blocklist & Redirect ②. In order to keep persistent storage of previously encountered IP addresses, we employ three databases; (i) *Block*, (ii) *Redirect*, and (iii) *Request time*. We employ these databases as firewall rules and we consult them regarding the redirection of IP addresses that PARALLAX has already encountered. First, we check whether the source IP address of the client request has been already blocked by PARALLAX, by looking it up on the *Block* database. If the IP address can be found on the *Block* database, we redirect all requests from that IP to a HTTP 403 Forbidden page.

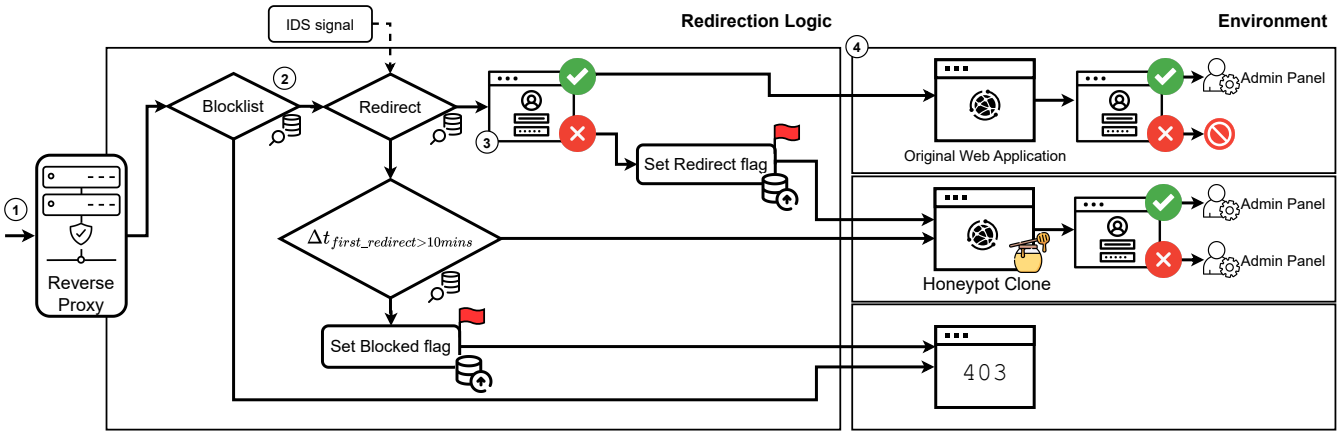

Fig. 3: PARALLAX deployment configuration

Alternatively, if the IP address is not blocked, we check if the requests should be redirected to the honeypot clone, by looking up the source IP address on the *Redirect* database. If the IP address is indeed on the *Redirect* database, we check whether the specific source IP address has already being tagged for redirection more than 10 minutes ago, by looking it up on the *Request time* database. If more than 10 minutes have passed since the first failed authentication request of the given IP address, we add this IP address to the *Block* database. If less than 10 minutes have passed, we redirect all requests from the given IP address to the honeypot clone environment. Lastly, in the case where the IP address can be found neither in the *Block*, nor in the *Redirect* databases, we proceed with the dynamic failed authentication detection of the PARALLAX web server. While for the present implementation we employ a single failed authentication attempt as the redirection trigger, any Intrusion Detection signal may be used to enable the redirection logic of PARALLAX. Considering that any signal may be employed as the trigger for PARALLAX, our system can be incorporated alongside any sophisticated rule-based or Machine Learning intrusion detection schemes, with very minimal adjustments.

Redirection Logic③. The PARALLAX web server logs all requests to the web application. If a POST request to the login page of the web application is detected, we check the validity of the credentials provided. If the correct credentials are provided, the user continues their interaction with the original web application, without any interference from PARALLAX. Alternatively, if the credentials are false, a failed authentication is detected, and all subsequent requests from that IP address are redirected to the honeypot clone. Furthermore, in order to track returning IP addresses and track the alloted 10 minuntes of interaction for each redirected user, we add the source IP address to the *Redirect* and *Request time* databases.

Redirection Environments④. PARALLAX may direct visitors toward three environments: (i) the original web application, (ii) the honeypot clone, and (iii) an HTTP 403 Forbidden page. The original web application is on the *Original* host, same as the reverse proxy, while the honeypot clone and the HTTP 403 are both parts of the *Clone* host.

Benign visitors are directed to the original web application, with no interfenrece from PARALLAX. This means that only users submitting the correct credentials may access the admin panel. On the other hand, visitors who submit incorrect credentials to the web application's authentication endpoint are redirected to the honeypot clone. Contrary to the original web application, any set of credentials submitted to the authentication page of the honeypot clone will grant the user access to the admin panel of the web application. In this way, we are engaging malicious actors in deception, allowing them full access to all the sensitive areas of the web application, while we observe and log their behavior. Lastly, we allow malicious actors an arbitrary and fully configurable grace period of 10 minutes, during which they interact with the honeypot clone. Once this grace period is over, the source IP address of the user is blocked and all requests from that IP address will receive an HTTP 403 Forbidden response. Upon blocking the user from further interacting with the PARALLAX-protected web application, we collect the access logs of the malicious user's interaction with the honeypot clone. In order to remove any changes that each malicious user may have introduced to specific instance of the honeypot clone, we automatically delete all the honeypot clone containers and spin up fresh ones from scratch. Thus, after the configurable grace period that we afford to each attacker, we automatically obtain a new, clean honeypot clone instance, where we redirect attackers that have never before interacted with the PARALLAX.

C. Sensitive Data Replacement

As discussed in Section IV-A, during the honeypot generation procedure we are able to automatically and agnostically generate and deploy a clone of the original web application. However, since we are using the cloned web application as a honeypot and we want attackers to fully interact and even compromise it, we must make sure to not include any of the sensitive data of the original web application in the honeypot clone. In order to prevent attackers from accessing this sensitive material, we detect the sensitive data included in the database of the original web application and we replace that data with synthetic, realistic, context-aware database entries.

Since we have access to the database of the original web application, the detection and replacement of the sensitive data comprises of agnostically and automatically locating the sensitive database entries, generating their synthetic replacement, and creating the new synthetic database to be used in the honeypot clone. The new synthetic database follows the exact schema of the original database, and only differs in the entries of the detected sensitive data.

1) Sensitive Data Detection: We automatically detect the sensitive data included in the database of the original web application, employing three different approaches: (i) Heuristics and entropy, (ii) Role-based interaction analysis, and (iii) Large Language Models.

a) Heuristics and entropy: In the present approach, we employ two elements to detect sensitive data; (i) heuristics rules for database column identifiers, and (ii) the entropy of the column entry. Regarding heuristics, we detect sensitive and private data, i.e., Personally Identifiable Information (PII) based on key identifiers, as described in HIPAA [33], GDPR [34], and PIPEDA [35]. For example, *social security number* is a HIPAA identifier. Thus, we parse the web application database column names, and if such a column name exists, we mark the column as sensitive. In total, we manually curated a list of more than 150 individual sensitive data identifiers, which we employ in our heuristics-based approach.

Regarding entropy, we use the content of the database entry and employ high entropy as an indicator of more sensitive data. Specifically, higher entropy indicates that the data includes a greater degree of randomness, comprises of more complex alphanumeric sequences, and is more difficult to predict. Thus, a database entry of high entropy may include sensitive information, such as passwords, credit card or social security numbers, or encryption keys. For example, it is a common security practice to store hashed user credentials on the web application database. Such an entry will have a higher entropy, and will be detected by our entropy-based mechanism.

The combination of heuristics and entropy provides a complete sensitive information detection scheme, operating both on the keys, as well as the values of the web application's database.

b) Role-based interaction analysis: The heuristics and entropy approach, focuses on the keys and values of the database, i.e., the content of the database. With our role-based approach, we operate agnostically and require no knowledge of the content of the database. The logic of this approach lies in the access that differently privileged users have on the database. Specifically, an unauthenticated user can only access the public-facing endpoints of the web application and will be accessing different parts of the database, compared to an authenticated user, who has access to the admin panel, the back-end, etc. Furthermore, we know that an authenticated user is a privileged user and can access a much greater portion of the database. The entries that only the authenticated user can access are deemed sensitive.

In order to attribute specific access on parts of the database to appropriate user roles, i.e., unauthenticated or authenticated, we deploy an Xdebug [29] miniserver, which allows us to track all the function calls to the web application. From these function calls, we extract the database queries that each user makes. To ensure sufficient interaction with the database, we employ monkey testing. Monkey testing is a software testing method, where a simulated user sends random inputs and actions (i.e., clicks, keystrokes, etc.) to the application to evaluate its behavior and uncover different execution paths. In our case, we employ monkey testing in two scenarios; an unauthenticated user, who only has access to the public-facing parts of the web application, and an authenticated user, who can access both public-facing, as well as the sensitive parts of the web application. The database columns that appear on the function call traces of the authenticated user and not in the traces of the unauthenticated user, are marked as sensitive.

c) Large Language Models: Considering the ability of Large Language Models (LLMs) to understand and interpret natural language, we employ Llama 3.1 [36] (latest open source LLM suitable for general purpose text-based tasks at deployment time), on a machine running a 32GB Nvidia Quadro P5000 GPU with 18 CPUs, to detect sensitive or confidential information based on the database column name. For a given database, we extract all the database table and column names, and we provide them in the form of $< database\_table >, < database\_column >$ to the LLM. After trial and error with prompt engineering, we provide the final format of the prompt that yielded the most consistent output in Appendix A. The output of the LLM is a list of the database columns that it deems to refer to sensitive information, and, thus, we mark those database columns as sensitive.

All three of the aforementioned sensitive information detection schemes focus on different components of the database to extract conclusions on the the sensitivity of the data. We underline that they are independent between them and can be combined to yield higher degrees of detection accuracy.

2) Synthetic Data Generation: Having detected the sensitive information included in the original web application database, we extract the sensitive columns and their content. We provide all the content of the sensitive database entries to a local Llama instance and prompt it to *"Generate realistic fake data for a database"*, making sure that the LLM does not change the original database column tables or names. In that way, the LLM generates realistic, synthetic database entries, which are coherent and consistent with the context of the original ones, while the scheme of the original database remains unchanged. Given the output of the LLM, we replace the original database entries with the synthetically generated ones and we export the synthetic database as a MySQL dump. Having deployed the honeypot clone as containers with persistent volumes, we can copy the synthetic database into the honeypot clone database container, and load it into the MySQL server of the honeypot.

V. System Evaluation

In order to evaluate both the performance, as well as the security properties of PARALLAX, we generated and deployed honeypot clones for five live LAMP stacks employing popular, open-source web applications. We deployed PARALLAX for

TABLE I: PARALLAX deployment time for five open-source web application

Application	Release	Application Directory Size [MB]	Deployment Time [s]
Wordpress	6.7.2	26	316.84
Joomla	3.10.12	9.2	230.67
Drupal	10.2.3	19	367.37
Prestashop	8.1.4	238	522.68
Owncloud	10.13.4	73	546.39
Average	-	-	396.79

WordPress, Joomla, Drupal, Prestashop, and Owncloud web applications. These applications collectively account for more than the 45% of all sites on the web [37], while they encompass distinct and diverse aspects of online activity, i.e., blogs, general purpose web applications, online shopping, file management services.

The objective of our in-the-wild experiment is to demonstrate that PARALLAX can automatically and agnostically generate and deploy honeypot clones of live web applications in a performant and resource-efficient manner. Furthermore, we evaluate the extent to which attackers will interact with a PARALLAX-protected web application. To that end, we deployed PARALLAX for live instances of each of the aforementioned web applications and we monitored the incoming traffic for variable periods of time over the course of two months. Prior to using PARALLAX, the original web applications were populated with realistic content to entice malicious actors (e.g., banking and financial services content, online store, etc.), while each instance had its proper domain and its respective SSL certificates, to ensure compatability with real-world deployments and to futher advertise our websites to malicious bots [38].

A. Performance Evaluation

PARALLAX was successful at automatically and agnostically generating and deploying a honeypot clone for all five web applications. Moreover, the honeypot generation and deployment procedure did not interfere with the availability of the live web applications. However, although the post-deployment network configuration of PARALLAX introduces no measurable overhead, PARALLAX does introduce an one-time slow-down to the request serve time during the cloning procedure.

In Table I, we present the five open-source web applications that were evaluated, along with the PARALLAX deployment time. All performance experiments were carried out on live web applications, which were deployed on hosts of the exact same characteristics (4GB RAM, 4-core CPUs), and all metrics presented are the average of three deployment runs. On average, PARALLAX needs less than 7 minutes to fully generate and deploy a honeypot clone alongside the original web application. Given the dynamic analysis involved in the clone generation procedure, we observed that the size of the web application directory, along with the complexity of the codebase of each web application have a noticeable effect on the time that PARALLAX takes to fully deploy.

In Figure 6, we present the CPU usage of the *Original* host during the deployment of PARALLAX for a live WordPress web application. Specifically, we present three scenarios; (i) the PARALLAX deployment in isolation, (ii) continuous requests sent to the web application in isolation, and (iii) deploying PARALLAX while the host is under the load of continuous requests. First, we observe that PARALLAX, on average, does not surpass the 50% CPU usage mark at any time during the clone generation and deployment procedure. Furthermore, the PARALLAX deployment may be carried out, even while the host system is under the load of continuous requests. The only requests that may potentially be dropped, are the ones that arrive exactly during the shutting down of the original web server, and while the PARALLAX web server is being deployed. Across all the PARALLAX deployments for the different web applications, we found that the average time window during which requests could possibly be dropped is 0.736 seconds. Given that this narrow window only exists during the one-time deployment operation of PARALLAX, we consider PARALLAX to not interfere with the availability of the web application.

From Figure 6, we also observe that requests that reach the web application during the PARALLAX deployment are experiencing a small delay in comparison to requests that arrive in isolation. In Table III, we present the delay introduced to a request that arrives exactly during the PARALLAX deployment (one-time process spanning 7 minutes on average). Across all examined web applications, a request that arrives during the cloning procedure will incur an average delay of less than 50 ms. Since PARALLAX can reach full deployment in less than 7 minutes, the average delay for requests arriving to the web application during those 7 minutes is less than 50 ms. On the other hand, after the cloning procedure is finished, the fully deployed system introduces no measurable overhead. We provide CPU usage plots for the rest of the evaluated web applications in Figures 7, 8, 9, and 10. **RQ1:** Given the above, we conclude that **PARALLAX is indeed able to clone LAMP stacks automatically, agnostically, and in a performant manner.**

B. Security Evaluation

Having succesfully deployed PARALLAX for all five web applications, we monitored all incoming traffic to the original web application, as well as the honeypot clone. Since the primary objective of PARALLAX is to protect the real web application, by redirecting malicious actors to the honeypot clone, for our security evaluation, we wish to examine the extent to which malicious visitors will actively engage with the deceptive defenses of PARALLAX and how their behavior differs when redirected to the honeypot clones.

1) Attacker engagement: In the span of two months, the deployed PARALLAX-protected web applications received more than 600K requests from over 20K unique IPs. Specifically, the deployed web applications received in total 683,391 requests, with 640,954 requests staying on the original web applications, while 42,864 requests were internally redirected by PARALLAX to the respective honeypot clone. Although the majority of visitors did not trigger the redirection mechanism,

we observed large discrepancies in the behavior of the redirected vs. the non-redirected visitors. Specifically, the average number of requests per non-redirected visitor was 27.84, while the average number of requests per malicious actor that was redirected to the honeypot clone was 185.56. This means that malicious actors engaged with the honeypot clone 6.25 times more than benign visitors engaged with the original web application. Thus, PARALLAX manages to protect the original web application, by actively engaging the attackers. Moreover, given their persistent and unremitting interaction with the clone, the realistic, context-aware, synthetically generated data used to populate the honeypot instances appear to have managed to deceive attackers in believing that they have managed to breach a real web application.

2) Attacker behavior: Along with actively engaging attackers in a covert manner, PARALLAX also allows defenders to observe aspects of the attackers' behavior which would otherwise be ignored. Across all deployed honeypots, we observed that the paths of the requests to the original web application were substantially different compared to the paths for the requests received on the honeypot clone. Qualitatively, we present the POST request paths that were made towards the original web application and towards the honeypot clone in Fig. 11 and 12 of Appendix C, respectively. We observe different attacker behaviors between the two hosts, with POST request paths to the honeypot clone referring to login endpoints used presumably to carry out attacks, i.e., *"wp-admin", "wp-login.php", "xmlrpc.php", "register", etc.*, while POST request paths to the original web application rest predominantly innocuous.

In order to quantify the differences of the requested paths between the original web application and the clone, we calculated the weighted Jaccard similarity [39]–[42] of the two request path sets to be 4.46 %. This means that the requests to the original web application and the requests to the honeypot clone are two very dissimilar sets, and share few common paths. We note that the frequencies of path occurence were used as the weights for the dissimilarity calculation.

Along with the measurable shift that PARALLAX instigates on the attackers' behavior, by employing a realistic honeypot clone, we are able to gather threat intelligence, which would otherwise not be revealed. In Figure 4, we present the unique GET and POST requests that appear in the original and the clone hosts. We observe that while the majority of the requests that appear on the honeypot clones are shared with the original web application, there is a significant number of requests that only appear on the clones (1613 unique GET requests and 62 unique POST requests). Considering that these requests are unique to the honeypot clone, PARALLAX allows defenders to further enhance their fingerprinting techniques, by incorporating new behavior patterns to their threat intelligence. Furthermore, considering that we allow attackers to interact with the honeypot clone for only 10 minutes, after which time we block any incoming request from their IP address, while benign visitors to the original web application are not bounded time-wise, we consider the number of unique requests to the honeypot clone to be substantial.

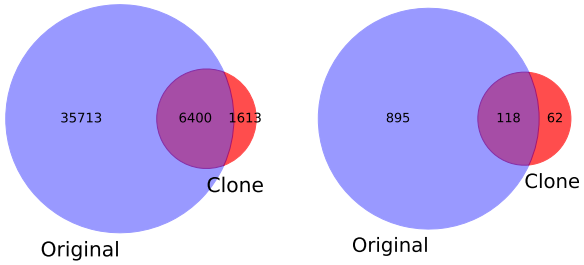

Fig. 4: Unique GET (left) and POST (right) requests distribution between *Original* and *Clone* hosts

RQ2: With more than 600K requests from over 20K unique IPs received by out honeypots, along with the aforementioned attacker behavior, we conclude that **real attackers actively engaged with our PARALLAX generated honeypots.**

3) Automated vulnerability tools: Along with malicious users and bots, we also evaluated the PARALLAX-protected web applications against automated vulnerability detection tools. In particular, we employed the brute force capability of off-the-shelf tools, such as WPScan [43] and the NMAP Joomla brute force script [44], to examine whether PARALLAX would be able to fool them. Indeed, after evaluating both these tools (loaded with arbitrary username, password pairs) against a WordPress and a Joomla PARALLAX deployment respectively, both falsely claimed to have found correct credentials, since PARALLAX redirected them to the honeypot clone. Neither of the automated vulnerability detection tools detected the deception in play.

VI. DECEPTION EVALUATION

The deceptive capabilities of PARALLAX lie in covertly and seamlessly redirecting attackers to the honeypot clone. Moreover, PARALLAX detects the sensitive data of the original web application and replaces them with synthetic data on the clone. In this manner, PARALLAX ensures that no sensitive or private data of the original web application appear in the honeypot. As mentioned in Section IV-C1, PARALLAX is able to detect the sensitive data of a web application database in three different ways; (i) heuristics and entropy, (ii) role-based interaction analysis, and (iii) LLMs. In this section, we evaluate the aforementioned sensitive data detection schemes for the five PARALLAX-protected web applications. Furthermore, in order to evaluate the deceptive capability, the realism and the context-awareness of the synthetically generated content that we employ to populate the database of the honeypot clone, we carried out a user study. In our user study, participants were presented with two sets of data, one real and one synthetic. Then, we asked them to determine which data was real and which data was synthetic. Our goal is to measure the effectiveness of the proposed synthetic information generation scheme in producing realistic data, which are indistinguishable from the real, original ones.

A. Sensitive Data Detection

In Table II, we present the results of the sensitive data detection schemes. For each PARALLAX-protected web ap-

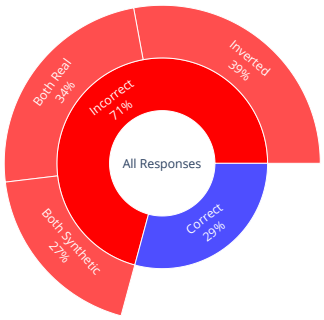

Fig. 5: User study participant responses

plication, we present the percentage of its database columns that were classified as sensitive by the different detection mechanisms (absolute count of database columns detected as sensitive in Table IV of Appendix B). We observe that, overall, the heuristics and the role-based interaction schemes classify more database columns as sensitive, while the entropy and LLM approaches detect sensitive data for less than 5% of the database columns. Considering that the heuristics-based detection scheme includes a wide range of privacy-related primitives, such as sensitive data keywords, PII identifiers, etc., this approach may be employed to detect explicitly defined sensitive database entries. On the other hand, the role-based analysis detection mechanism allows for agnostic detection of sensitive entries, since it classifies database columns as sensitive if they can only be reached by an authenticated user. While these two approaches call for larger parts of the database to be replaced with synthetic entries, the entropy-based approach and the LLM-based detection offer a more refined and targeted set of database columns. Overall, PARALLAX offers a configurable sensitive data detection suite, allowing defenders to select the appropriate scheme for their web application.

B. Synthetic Data Generation Experiment

Our study design consists of the collection and analysis of anonymous opinions regarding the two sets of data. The data presented to the participants were in the form of profiles of individuals, containing the following information; first name, last name, gender, location, relationship status, employer, email, and password. Each participant was given 10 prompts, where each prompt included two profiles and asked participants to select one of the following options: (i) Both profiles are real, (ii) Both profiles are synthetic, (iii) Profile 1 is real and Profile 2 is synthetic, or (iv) Profile 1 is synthetic and Profile 2 is real. In Table VI of Appendix D, we provide an example of the data that was presented to the participants. Unbeknownst to the participants, each prompt contains one real and one synthetic profile. The construction of the real profiles comprises of data from publicly available prior breach data [45], and passwords from the rockyou common password list [46]. The synthetic profiles are generated using Llama 3.1 and by specifying the characteristics of the profile information to accurately match the language conventions of the real data, i.e., the relationship status following social media conventions, the locations referring to real cities, the employers being real companies that match the aforementioned location. Furthermore, we removed the last three characters of the email address to avoid any coincidental matching with real email addresses. In order to avoid bias in the presentation of the prompts, both the order of profiles, as well as the order the possible answers were randomized. Lastly, for each prompt, after having made their selection from the multiple choice question, the participants were asked to provide an explanation regarding their selection. In terms of the scale of the user study, we employed 100 participants through the Amazon Mechanical Turk platform. Relying on an online platform, ensured that our data collection contains enough representative data from different users with different levels of web application expertise.

TABLE II: Sensitive information detection methods

Application	Heuristics [%]	Entropy [%]	Role-based [%]	LLM [%]
Wordpress	42.55	5.32	26.6	6.38
Joomla	10.25	2.09	5.62	0.2
Drupal	23.99	7.23	40.21	1.94
Prestashop	25.56	3.61	48.93	1.78
Owncloud	21.81	4.67	69.16	0.93
Average	24.83	4.58	38.10	2.25

C. Ethical Considerations

After the review of our user-study protocol by our IRB, we were informed that there was no requirement for approval or exemption, since we only collected anonymous opinions regarding the presented data, while no private, user-specific information was collected. The participants were invited to our user study on the Amazon Mechanical Turk and Microworkers platform, where we set up our user study as human intelligence tasks/microtasks. Upon completion of the study, the participants were compensated at a rate of $12/hour.

D. Results

In Figure 5, we present the results of the multiple choice question component of the user study. We observe that only 29% of the participants correctly identified both the real and the synthetic profiles. Conversely, 71% of participants misdentified the profiles, positing that both profiles were real, both profiles were synthetic, or inverting the real and synthetic profiles. Since participants were given four choices regarding the profiles, a random selection would result in a correct answer rate of 25%. With the correct answer was chosen at a rate of 29%, we conclude that the synthetic generation of sensitive information is able to produce realistic, context-aware sensitive information that is indistinguishable from real data. Thus, employing our synthetic data generation framework to replace the sensitive data of a web application can produce realistic data, which can further attract, entice, and engage malicious actors redirected to the honeypot clone.

Regarding the participants' explanations, we observed that they struggled significantly to reliably distinguish between real and synthetic profiles, demonstrating that the synthetic data was highly successful at appearing credible. Across all

choices, participants relied on criteria centered around plausibility and consistency, such as common names, standard email formats, credible employers, and the complexity of passwords. Crucially, the real profiles were often mistakenly deemed implausible due to unusual details, while the sophisticated design of fake profiles made them seem more realistic.

RQ3: From the user study, we conclude that **humans are not able to effectively differentiate between real sensitive information and synthetically generated ones**, such as the ones that PARALLAX employs in the honeypot clones. In Appendix D, we present summaries of the explanations for each answer category.

VII. Related Work

A. Deception-based Defenses

Deception-based defenses encompass all security techniques that protect digital assets by actively deceiving malicious actors. Unlike traditional security measures, deception-based defenses aim to lure, actively engage, misdirect or divert malicious actors. Honeypots, a cornerstone of deceptive defenses ever since their inception [47]–[51], allow defenders to collect valuable information on attacker behavior [52]. In order to address the multitude of diverse attack vectors, honeypots have been extended to large scale honeypot networks [53], while their core concept has been adapted to smaller-scale, context-specific, honey-x or decoy elements, such as honeywords [13], honeypatches [54], honeytokens [11], [55], and even files and filesystems [14], [15], [56]. Such defensive techniques have been employed in a wide array of different security settings, ranging from small-scale, host-based machines [14], [15], [57] to Industrial Control Systems [58], and from web applications [20], [21], [59], [60], to mobile phones and instant messaging [61]–[63]. In this paper, in the context of web applications, we focus on automatically and agnostically generating honeypot clones of live LAMP stacks.

B. Honeypot generation

Although honeypot generation for web applications has been proposed before [19], [64]–[67], none of the proposed solutions allow for an automatic, application-agnostic generation of a honeypot clone based on a live web application. Furthermore, while the majority of the relevant deceptive defenses literature focuses on individual components of the web application, i.e., fake HTTP parameter generation [21], deceptive form fields [68], deception-based authentication approaches [59], [60], PARALLAX incorporates a holistic, lightweight, resource-efficient deception-based solution.

C. Synthetic Data Generation

The generation and deployment of deceptive elements is an established cyber deception technique [15]–[17], [59], [69]. Most such approaches have been directed towards creating realistic fake documents and filesystems [56], [70]–[74]. Although the generation of realistic, synthetic database content has been well studied [11], [75], the automatic detection of sensitive web application data and their replacement with context-aware honeytokens, as well as their effectiveness remain an open question. In our approach, we introduce three independent sensitive data detection schemes, we replace the sensitive database entries with synthetic, context-aware data on the honeypot clones, and we evaluate their realism and effective indistinguishability with a large scale user study.

VIII. Discussion

As discussed in Section IV-C1, PARALLAX introduces three independent sensitive data detection schemes. Although these schemes may also be employed together for wider coverage, false negatives are still possible. Considering that PARALLAX is a fully automated honeypot clone generation framework, requiring no human intervention, we envision a more conservative approach to false negatives to entail a human-in-the-loop verification of the replaced sensitive data.

PARALLAX was developped under the assumption that the original LAMP stack is using an Apache web server. Given the modular nature of web server configurations, solutions that are effective for Apache can be easily adapted for other web server technologies, with little additional effort, only involving straightforward configuration adjustments.

The single failed authentication attempt is not the most robust detection scheme. However, although PARALLAX's contributions and innovation lie in the mitigation of malicious actors using honeypot clones of web applications, our system is fully configurable, and can be adapted to work with more elaborate intrusion detection mechanisms.

IX. Conclusion

In this paper, we introduce PARALLAX, a deception-based web application defense framework, which protects web applications by automatically and agnostically generating and deploying a containerized honeypot clone of any live LAMP stack application. PARALLAX deploys the honeypot clone alongside the original web application, and covertly redirects attackers to the honeypot clone, while not interfering with legitimate users. Furthermore, we propose three independent sensitive data detection mechanisms, which allow us to locate and replace the sensitive data of the original application for the honeypot clone instance. PARALLAX populates the honeypot clone database with realistic, context-aware, synthetic data, thus protecting the private web application data of legitimate web application users. We evaluate PARALLAX in three ways; (i) resource efficiency of the honeypot generation and deployment, (ii) security in terms of isolation and behavioral patterns of redirected attackers, and (iii) deceptive capabilities of the sensitive data replacement. We conclude that PARALLAX is resource-efficient and triggers different attacker behaviors, while implementing covert and seamless deceptive defenses through the use of realistic synthetic data.

Acknowledgments We thank the anonymous reviewers for their helpful feedback. This work was supported by the Army Research Office (ARO) under grant W911NF-24-1-0051 and the National Science Foundation (NSF) under grant CNS-2126654.

Availability A video demo of PARALLAX along with the source code are available at https://parallaxpaper.github.io/.

REFERENCES

[1] S. Kottler, "February 28th DDoS Incident Report," https://github.blog/news-insights/company-news/ddos-incident-report/, 2018, [Online; accessed 23-February-2025].

[2] L. Hay Newman, "GitHub Survived the Biggest DDoS Attack Ever Recorded," https://www.wired.com/story/github-ddos-memcached/, 2018, [Online; accessed 23-February-2025].

[3] B. Toulas, "Massive brute force attack uses 2.8 million IPs to target VPN devices," https://www.bleepingcomputer.com/news/security/massive-brute-force-attack-uses-28-million-ips-to-target-vpn-devices/, 2025, [Online; accessed 23-February-2025].

[4] P. Release, " 911 S5 Botnet Dismantled and Its Administrator Arrested in Coordinated International Operation ," https://www.justice.gov/archives/opa/pr/911-s5-botnet-dismantled-and-its-administrator-arrested\-coordinated-international-operation, 2024, [Online; accessed 23-February-2025].

[5] R. Pompon, "How Credential Stuffing Bots Bypass Defenses," https://www.f5.com/labs/articles/threat-intelligence/how-credential-stuffing-bots-bypass-defenses, 2020, [Online; accessed 23-February-2025].

[6] Radware, "A Game of Cat and Mouse: Dynamic IP Address and Cyber Attacks ," https://www.radware.com/security/ddos-threats-attacks/ddos-attack-types/dynamic-ip-address-cyber-attacks/, 2016, [Online; accessed 23-February-2025].

[7] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the mirai botnet," in *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017.

[8] B. A. Azad, S. Vargas, and A. Martinetti, "Using machine learning to detect bot attacks that leverage residential proxies," https://blog.cloudflare.com/residential-proxy-bot-detection-using-machine-learning/, 2024, [Online; accessed 23-February-2025].

[9] S. Sivakorn, J. Polakis, and A. D. Keromytis, "I'm not a human: Breaking the google recaptcha," *Black Hat*, vol. 14, pp. 1–12, 2016.

[10] I. Omisola, "Cloudflare JS Challenge: How It Works and How to Solve It," https://www.zenrows.com/blog/cloudflare-js-challenge-bypass, 2024, [Online; accessed 23-February-2025].

[11] M. Bercovitch, M. Renford, L. Hasson, A. Shabtai, L. Rokach, and Y. Elovici, "Honeygen: An automated honeytokens generator," in *Proceedings of 2011 IEEE International Conference on Intelligence and Security Informatics*. IEEE, 2011, pp. 131–136.

[12] A. Shabtai, M. Bercovitch, L. Rokach, Y. Gal, Y. Elovici, and E. Shmueli, "Behavioral study of users when interacting with active honeytokens," *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 3, pp. 1–21, 2016.

[13] A. Juels and R. L. Rivest, "Honeywords: Making password-cracking detectable," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 145–160.

[14] B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo, "Baiting inside attackers using decoy documents," in *Security and Privacy in Communication Networks: 5th International ICST Conference, SecureComm 2009, Athens, Greece, September 14-18, 2009, Revised Selected Papers 5*. Springer, 2009, pp. 51–70.

[15] J. Voris, J. Jermyn, N. Boggs, and S. Stolfo, "Fox in the trap: Thwarting masqueraders via automated decoy document deployment," in *Proceedings of the Eighth European Workshop on System Security*, 2015, pp. 1–7.

[16] M. B. Salem and S. J. Stolfo, "Modeling user search behavior for masquerade detection," in *Recent Advances in Intrusion Detection: 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings 14*. Springer, 2011, pp. 181–200.

[17] M. Ben Salem and S. J. Stolfo, "Decoy document deployment for effective masquerade attack detection," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 8th International Conference; DIMVA 2011, Amsterdam, The Netherlands, July 7-8, 2011. Proceedings 8*. Springer, 2011, pp. 35–54.

[18] N. Nikiforakis, M. Balduzzi, S. Van Acker, W. Joosen, and D. Balzarotti, "Exposing the lack of privacy in file hosting services," in *4th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET 11)*, 2011.

[19] M. Müter, F. Freiling, T. Holz, and J. Matthews, "A generic toolkit for converting web applications into high-interaction honeypots," *University of Mannheim*, vol. 280, pp. 6–1, 2008.

[20] M. Sahin, C. Hebert, and A. S. De Oliveira, "Lessons learned from sundew: a self defense environment for web applications," in *Proceedings of the 2020 Measurements, Attacks, and Defenses for the Web (MADWeb) Workshop in the Network and Distributed System Security Symposium (NDSS)*, 2020.

[21] M. Sahin, C. Hébert, and R. Cabrera Lozoya, "An approach to generate realistic http parameters for application layer deception," in *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022, Rome, Italy, June 20–23, 2022, Proceedings*. Springer, 2022, pp. 337–355.

[22] L. Spitzner, *Honeypots: tracking hackers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[23] "Duplicator," https://duplicator.com/, 2025, [Online; accessed 10-March-2025].

[24] "Migrate Guru," https://migrateguru.com/, 2025, [Online; accessed 10-March-2025].

[25] "JetPack," https://jetpack.com/support/backup/, 2025, [Online; accessed 10-March-2025].

[26] "Softaculous," https://www.softaculous.com/, 2025, [Online; accessed 10-March-2025].

[27] "W3Techs," https://w3techs.com/technologies/details/os-linux, 2025, [Online; accessed 1-March-2025].

[28] R. Hat, "Ansible," https://www.redhat.com/en/ansible-collaborative, 2025, [Online; accessed 6-March-2025].

[29] Xdebug, "Xdebug," https://xdebug.org/, 2025, [Online; accessed 1-March-2025].

[30] "Openresty," https://openresty.org/en/, 2025, [Online; accessed 6-March-2025].

[31] "Redis," https://redis.io/, 2025, [Online; accessed 6-March-2025].

[32] "Lua," https://www.lua.org/, 2025, [Online; accessed 6-March-2025].

[33] D. of Health Care Services, "List of HIPAA Identifiers," https://www.dhcs.ca.gov/dataandstats/data/Pages/ListofHIPAAIdentifiers.aspx, 2025, [Online; accessed 18-March-2025].

[34] GDPR, "What is considered personal data under the EU GDPR?" https://gdpr.eu/eu-gdpr-personal-data/, 2025, [Online; accessed 18-March-2025].

[35] Perforce, "PIPEDA: Understanding Canada's Data Privacy Law ," https://www.perforce.com/blog/pdx/pipeda, 2025, [Online; accessed 18-March-2025].

[36] Meta, "Llama," https://www.llama.com/, 2025, [Online; accessed 18-March-2025].

[37] W3Techs, "Usage statistics and market share of content management systems," https://w3techs.com/technologies/overview/content_management, 2025, [Online; accessed 09-April-2025].

[38] B. Kondracki, J. So, and N. Nikiforakis, "Uninvited Guests: Analyzing the Identity and Behavior of Certificate Transparency Bots," in *Proceedings of USENIX Security Symposium (USENIX Security)*, 2022.

[39] S. Ioffe, "Improved consistent sampling, weighted minhash and l1 sketching," in *2010 IEEE international conference on data mining*. IEEE, 2010, pp. 246–255.

[40] X. Li and P. Li, "Rejection sampling for weighted jaccard similarity revisited," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 5, 2021, pp. 4197–4205.

[41] L. d. F. Costa, "Further generalizations of the jaccard index," *arXiv preprint arXiv:2110.09619*, 2021.

[42] F. Chierichetti, R. Kumar, S. Pandey, and S. Vassilvitskii, "Finding the jaccard median," in *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 2010, pp. 293–311.

[43] WPScan, "WPScan," https://wpscan.com/, 2025, [Online; accessed 15-April-2025].

[44] NMAP, "Script http-joomla-brute," https://nmap.org/nsedoc/scripts/http-joomla-brute.html, 2025, [Online; accessed 15-April-2025].

[45] L. Abrams, "533 million Facebook users' phone numbers leaked on hacker forum," https://www.bleepingcomputer.com/news/security/533-million-facebook-users-phone-numbers-leaked-on-hacker-forum/, 2025, [Online; accessed 19-March-2025].

[46] Kaggle, "Rockyou common password list," https://www.kaggle.com/datasets/wjburns/common-password-list-rockyoutxt, 2025, [Online; accessed 19-March-2025].

[47] C. Stoll, "Stalking the wily hacker," *Communications of the ACM*, vol. 31, no. 5, pp. 484–497, 1988.

[48] ——, *The cuckoo's egg: tracking a spy through the maze of computer espionage*. Simon and Schuster, 2005.
[49] L. Spitzner, *Honeypots: tracking hackers*. Addison-Wesley Reading, 2003, vol. 1.
[50] M. Sink, "The use of honeypots and packet sniffers for intrusion detection," *SANS Institute 2000–2002*, vol. 15, pp. 1–6, 2001.
[51] N. Provos *et al.*, "A virtual honeypot framework." in *USENIX Security Symposium*, vol. 173, no. 2004, 2004, pp. 1–14.
[52] J. Zhuge, T. Holz, X. Han, C. Song, and W. Zou, "Collecting autonomous spreading malware using high-interaction honeypots," in *Information and Communications Security: 9th International Conference, ICICS 2007, Zhengzhou, China, December 12-15, 2007. Proceedings 9*. Springer, 2007, pp. 438–451.
[53] The Honeynet Project, https://www.honeynet.org/, 2023, [Online; accessed 7-March-2023].
[54] F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser, "From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation," in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, 2014, pp. 942–953.
[55] M. Msaad, S. Srinivasa, M. M. Andersen, D. H. Audran, C. U. Orji, and E. Vasilomanolakis, "Honeysweeper: Towards stealthy honeytoken fingerprinting techniques," in *Nordic Conference on Secure IT Systems*. Springer, 2022, pp. 101–119.
[56] T. Taylor, F. Araujo, A. Kohlbrenner, and M. P. Stoecklin, "Hidden in plain sight: Filesystem view separation for data integrity and deception," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, DIMVA 2018, Saclay, France, June 28–29, 2018, Proceedings 15*. Springer, 2018, pp. 256–278.
[57] B. M. Bowen, P. Prabhu, V. P. Kemerlis, S. Sidiroglou, A. D. Keromytis, and S. J. Stolfo, "Botswindler: Tamper resistant injection of believable decoys in vm-based hosts for crimeware detection," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2010, pp. 118–137.
[58] E. Vasilomanolakis, S. Srinivasa, C. G. Cordero, and M. Mühlhäuser, "Multi-stage attack detection and signature generation with ics honeypots," in *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2016, pp. 1227–1232.
[59] T. Barron, J. So, and N. Nikiforakis, "Click this, not that: Extending web authentication with deception," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 462–474.
[60] B. Tsouvalas and N. Nikiforakis, "Knocking on admin's door: Protecting critical web applications with deception," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2024, pp. 283–306.
[61] M. Balduzzi, P. Gupta, L. Gu, D. Gao, and M. Ahamad, "Mobipot: Understanding mobile telephony threats with honeycards," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 723–734.
[62] M. Nassar, R. State, and O. Festor, "Voip honeypot architecture," in *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*. IEEE, 2007, pp. 109–118.
[63] S. Antonatos, I. Polakis, T. Petsas, and E. P. Markatos, "A systematic characterization of im threats using honeypots," in *ISOC Network and Distributed System Security Symposium (NDSS)*, 2010.
[64] F. Araujo, W. Kevin *et al.*, "Compiler-instrumented, dynamic {Secret-Redaction} of legacy processes for attacker deception," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 145–159.
[65] A. Niakanlahiji, J. H. Jafarian, B.-T. Chu, and E. Al-Shaer, "Honeybug: Personalized cyber deception for web applications," 2020.
[66] M. Musch, M. Härterich, and M. Johns, "Towards an automatic generation of low-interaction web application honeypots," in *Proceedings of the 13th international conference on availability, reliability and security*, 2018, pp. 1–6.
[67] M. Arslan, B. Çarıkçı, and Y. M. Erten, "Deception through cloning against web site attacks," in *2021 International Conference on Information Security and Cryptology (ISCTURKEY)*. IEEE, 2021, pp. 63–68.
[68] C. Pohl, A. Zugenmaier, M. Meier, and H.-J. Hof, "B. hive: A zero configuration forms honeypot for productive web applications," in *ICT Systems Security and Privacy Protection: 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings 30*. Springer, 2015, pp. 267–280.
[69] J. Lee, J. Choi, G. Lee, S.-W. Shim, and T. Kim, "Phantomfs: File-based deception technology for thwarting malicious users," *IEEE Access*, vol. 8, pp. 32 203–32 214, 2020.
[70] T. Chakraborty, S. Jajodia, J. Katz, A. Picariello, G. Sperli, and V. Subrahmanian, "A fake online repository generation engine for cyber deception," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 2, pp. 518–533, 2019.
[71] P. Karuna, H. Purohit, R. Ganesan, and S. Jajodia, "Generating hard to comprehend fake documents for defensive cyber deception," *IEEE Intelligent Systems*, vol. 33, no. 5, pp. 16–25, 2018.
[72] F. Araujo, D. L. Schales, M. P. Stoecklin, and T. P. Taylor, "Integrity, theft protection and cyber deception using a deception-based filesystem," Jan. 7 2020, uS Patent 10,528,733.
[73] P. Karuna, H. Purohit, S. Jajodia, R. Ganesan, and O. Uzuner, "Fake document generation for cyber deception by manipulating text comprehensibility," *IEEE Systems Journal*, vol. 15, no. 1, pp. 835–845, 2020.
[74] N. C. Rowe, "Measuring the effectiveness of honeypot counter-counterdeception," in *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, vol. 6. IEEE, 2006, pp. 129c–129c.
[75] D. Reti, N. Becker, T. Angeli, A. Chattopadhyay, D. Schneider, S. Vollmer, and H. D. Schotten, "Act as a honeytoken generator! an investigation into honeytoken generation with large language models," in *Proceedings of the 11th ACM Workshop on Adaptive and Autonomous Cyber Defense*, 2024, pp. 1–12.

APPENDIX A
LLM SYNTHETIC DATA DETECTION

"The following pairs represent database tables and columns. The first element is the name of a database table and the second is a column of that table. The two elements are separated by a comma and each line represents a pair. Tell me which pairs are more likely to contain sensitive, personal, or private material, such as PII."

APPENDIX B
PARALLAX PERFORMANCE AND RESULTS

TABLE III: PARALLAX deployment response delay

Application	Original Response Time [ms]	PARALLAX Response Time [ms]	Delay [ms]
Wordpress	141.86	227.08	85.22
Joomla	94.95	133.64	38.69
Drupal	15.01	17.37	2.36
Prestashop	96.34	163.69	67.35
Owncloud	144.35	191.71	47.36
Average	98.502	146.698	48.196

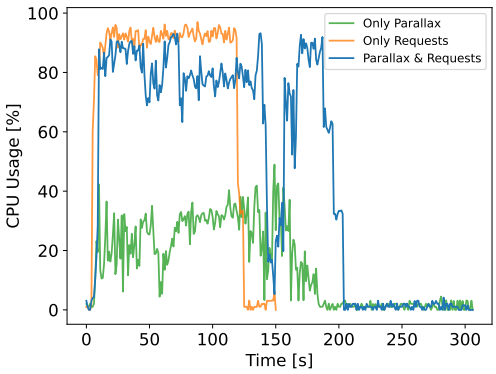

Fig. 6: Wordpress PARALLAX deployment CPU Usage during honeypot generation

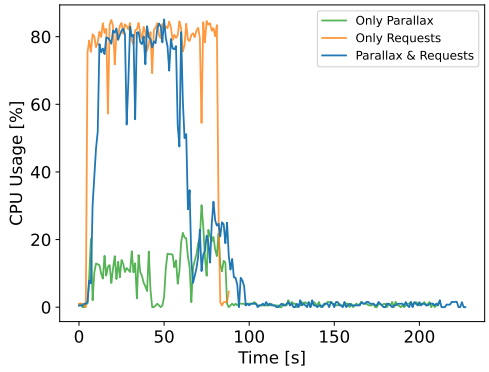

Fig. 7: Joomla PARALLAX deployment CPU Usage during honeypot generation

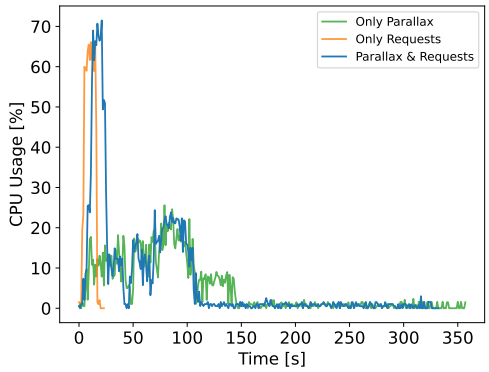

Fig. 8: Drupal PARALLAX deployment CPU Usage during honeypot generation

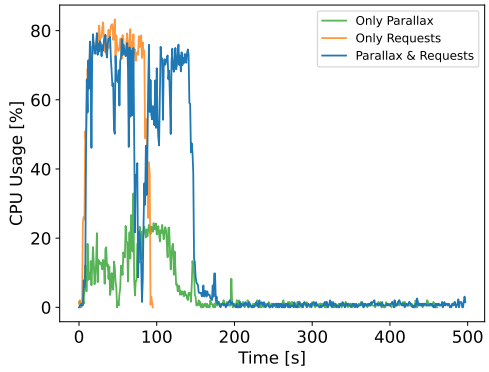

Fig. 9: Prestashop PARALLAX deployment CPU Usage during honeypot generation

TABLE IV: Sensitive information detection methods (absolute values)

Application	Column Count	Heuristics	Entropy	Role-based	LLM
Wordpress	94	40	5	25	6
Joomla	1531	157	32	86	3
Drupal	567	136	41	228	11
Prestashop	1690	432	61	827	30
Owncloud	321	70	15	222	3

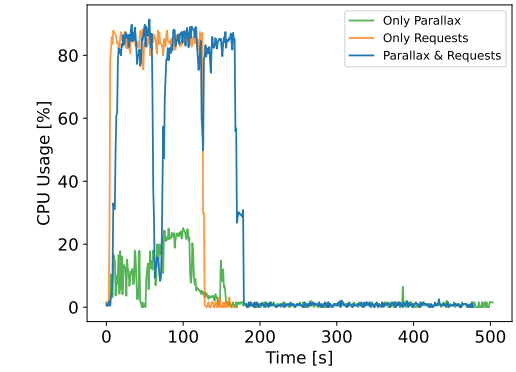

Fig. 10: Owncloud PARALLAX deployment CPU Usage during honeypot generation

APPENDIX C
PARALLAX DEPLOYMENT IN THE WILD RESULTS

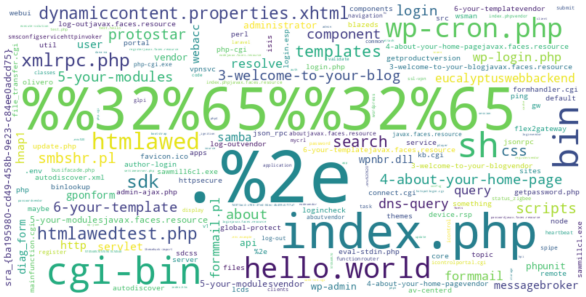

Fig. 11: POST request paths to original web application

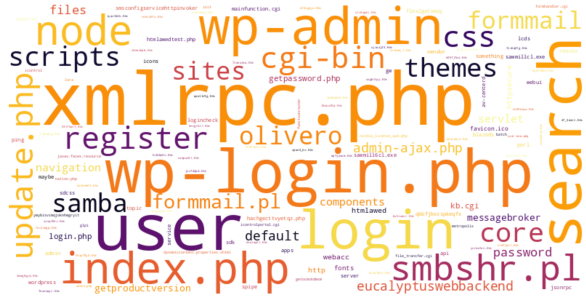

Fig. 12: POST request paths to honeypot clone

APPENDIX D
QUALITATIVE ANALYSIS OF USER STUDY PARTICIPANTS RESPONSES

A. Both Real

For submissions claiming that both profiles were real, most explanations suggest that both profiles have plausible details, such as common names, standard email formats, relatable relationship statuses, and believable employers. Furthermore, the strength of the profile passwords was mentioned by many participants as a potential indicator of synthetically generated material, while the credibility of the employer was deemed to enhance the legitimacy of the profile. In Quote 1 of Table V of Appendix D, we present one such comment.

B. Both Synthetic

Answers that claimed both profiles to be synthetic mentioned that both profiles have names and email structures that seem too generic or lack common sources of information. Furthermore, participants claimed that the use of simplistic, generic, or easily guessable passwords directed their decisions, while the presence of unrealistic or unusual details in the occupation, location, and relationship status of the profiles further established a lack of realism in both profiles. Overall, participants failed to find the real profile, and even attributed implausibility to information of real profiles, as presented in Quote 2 of Table V of Appendix D.

C. Inverted

The inverted answers refer to participant responses where the real profile was deemed to be synthetic, and vice versa. Plausibility and consistency played an important role in participants' responses, with explanations highlighting how certain name combinations and email formats deviate from typical patterns, while other participants questioned plausibility and consistency between the employer and location. As mentioned in Quote 3 of Table V of Appendix D, consistency between different profile components was a key indicator. Overall, participants emphasized the importance of password complexity, name and email plausibility, and employer and location consistency. The combination of real and synthetic data lead to synthetic data appearing to be more realistic and genuine than the real data.

D. Correct

In this case, participants correctly identified both profiles. Similarly to wrong responses, participants focused on password complexity, the legitimacy and plausibility of employers, as well as the overall consistency among the data itself, i.e., unusual name and email combinations. We underline that the aforementioned explanations are very similar to explanations provided to incorrect answers.

TABLE V: User study participant explanations

Quote 1: Both real testimonial
"Both profiles appear to follow a realistic pattern, with standard names, relationship statuses, employers, and valid email addresses. Although Profile 2's password seems slightly complex, it is not enough to make it synthetic."
Quote 2: Both synthetic testimonial
"Both profiles appear to follow a realistic pattern, with standard names, relationship statuses, employers, and valid email addresses. Although Profile 2's password seems slightly complex, it is not enough to make it synthetic."
Quote 3: Inverted testimonial
"Profile 1 has an unusual combination of password and location, and the email address seems generic, which are typical signs of a synthetic profile. Profile 2, on the other hand, seems to use a common employer (Dow Inc.), and the email address follows a standard format, suggesting it could be real."

TABLE VI: Example of user study profiles

	First Name	Last Name	Gender	Location	Relationship Status	Employer	Email	Password
Profile 1	Joshua	Johnson	male	Los Angeles, California	Married	Procter & Gamble	joshua.johnson***@gmail.com	nenes1401power
Profile 2	Lauren	Martin	female	Portland, Oregon	Single	Lockheed Martin Corporation	lauren.martin***@icloud.com	marinheira2006nice