

## Honeysweeper

*Towards stealthy Honeytoken fingerprinting techniques*

Msaad, Mohamed; Srinivasa, Shreyas; Møller Andersen, Mikkel; Audran, David; Orji, Charity U.; Vasilomanolakis, Emmanouil

*Published in:*  
Secure IT Systems

*DOI (link to publication from Publisher):*  
[10.1007/978-3-031-22295-5\\_6](https://doi.org/10.1007/978-3-031-22295-5_6)

*Publication date:*  
2023

*Document Version*  
Publisher's PDF, also known as Version of record

[Link to publication from Aalborg University](#)

*Citation for published version (APA):*  
Msaad, M., Srinivasa, S., Møller Andersen, M., Audran, D., Orji, C. U., & Vasilomanolakis, E. (2023). Honeysweeper: Towards stealthy Honeytoken fingerprinting techniques. In H. P. Reiser, & M. Kyas (Eds.), Secure IT Systems: 27th Nordic Conference, NordSec 2022, Reykjavic, Iceland, November 30–December 2, 2022, Proceedings (pp. 101-119). Springer. [https://doi.org/10.1007/978-3-031-22295-5\\_6](https://doi.org/10.1007/978-3-031-22295-5_6)

### General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal -

### Take down policy

If you believe that this document breaches copyright please contact us at [vbn@aub.aau.dk](mailto:vbn@aub.aau.dk) providing details, and we will remove access to the work immediately and investigate your claim.



# *Honeysweeper*: Towards Stealthy Honeytoken Fingerprinting Techniques

Mohamed Msaad<sup>1</sup> , Shreyas Srinivasa<sup>1</sup> , Mikkel M. Andersen<sup>1</sup> ,  
David H. Audran<sup>1</sup> , Charity U. Orji<sup>1</sup> , and Emmanouil Vasilomanolakis<sup>2</sup>

<sup>1</sup> Aalborg University, Copenhagen, Denmark

{mmsaad18,mman21,daudra21,corji21}@student.aau.dk, shsr@es.aau.dk

<sup>2</sup> Technical University of Denmark, Kgs. Lyngby, Denmark

emmva@dtu.dk

**Abstract.** The increased number of data breaches and sophisticated attacks have created a need for early detection mechanisms. Reports indicate that it may take up to 200 days to identify a data breach and entail average costs of up to \$4.85 million. To cope with cyber-deception approaches like honeypots have been used for proactive attack detection and as a source of data for threat analysis. Honeytokens are a subset of honeypots that aim at creating deceptive layers for digital entities in the form of files and folders. Honeytokens are an important tool in the proactive identification of data breaches and intrusion detection as they raise an alert the moment a deceptive entity is accessed. In such deception-based defensive tools, it is key that the adversary does not detect the presence of deception. However, recent research shows that honeypots and honeytokens may be fingerprinted by adversaries. Honeytoken fingerprinting is the process of detecting the presence of honeytokens in a system without triggering an alert. In this work, we explore potential fingerprinting attacks against the most common open-source honeytokens. Our findings suggest that an advanced attacker can identify the majority of honeytokens without triggering an alert. Furthermore, we propose methods that help in improving the deception layer, the information received from the alerts, and the design of honeytokens.

**Keywords:** Honeytokens · Fingerprinting · Counter-deception

## 1 Introduction

Cyber attacks have reached a record level in 2021, making it the highest in 17 years with a 10% increase from the previous year [14]. A \$1.07 million cost increase is related to the spike in remote work due to the COVID-19 pandemic [15] in addition to the continuous growth of IoT devices [8, 23]. Further, the time needed to identify and contain a security breach may take up to 287 days [13]. To combat this, the cyber-defense community is moving toward more active lines of defense that leverage deception-based techniques. Deception techniques confuse and divert attackers from real assets by placing fake data and

vulnerable systems across an organization’s network. Any interaction with a deceptive entity may be considered an attack. In practice, there are two leading deception technologies: *honeypots* and *honeytokens*.

*Honeypots* are deceptive systems that emulate a vulnerable program [16, 17, 20, 24], for instance, a vulnerable version of the Linux operating system (OS), an HTTP server, or an IoT device. They lure attackers and deflect them from real assets while gathering information about the techniques and tools used during the interaction. Honeypots differ by their low, medium or high-interaction level [9, 25, 26]. As the name implies, interaction refers to how much capabilities are offered to the adversary. The process of discovering the existence of a honeypot in a system is known as honeypot fingerprinting [22, 26]. The drawback of many honeypots is that their emulation of systems/protocols exposes some artifacts that attackers can detect.

*Honeytokens* are digital entities that contain synthetic/fabricated data. They are usually stored in a system under attractive names as a trap for intruders, and any interaction with them is considered an attack. Honeytokens can be files such as PDFs, SQL database entries, URLs, or DNS records that embed a token. Once accessed they trigger and alert the system about the breach [3]. Additionally, honeytokens are less complex and easier to maintain when compared to honeypots.

The honeytokens’ efficiency resides in their indistinguishability; hence, identifying that an entity is a honeytoken (known as fingerprinting), diminishes its value. In this paper, we explore and extend the research on honeytoken fingerprinting techniques and demonstrate a fingerprinting tool that can successfully fingerprint 14 out of 20 honeytokens offered by the most popular open-source honeytoken service. Our contributions in this work are as follows:

- We analyze the design of open-source honeytokens to identify potential gaps for fingerprinting purposes.
- We introduce additional techniques to detect open-source honeytokens without triggering alerts.
- We propose techniques to improve the deceptive capabilities of honeytokens and introduce features that can enhance the use of information received from alerts triggered by intrusions.

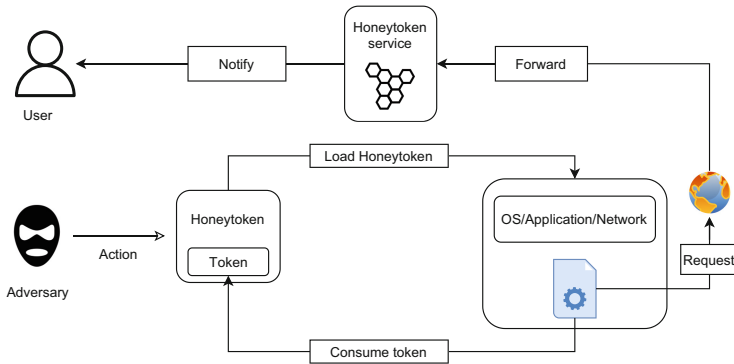
The rest of this paper is structured as follows. In Sect. 2, we discuss the background of the working mechanism and the fingerprinting mechanism of honeytokens. Section 3 summarizes the related work of honeytoken fingerprinting. Section 4 presents our proposed stealthy techniques for honeytoken fingerprinting. Moreover, in Sect. 5 we present a proof of concept for honeytoken fingerprinting. In Sect. 6, we discuss countermeasures against honeytoken fingerprinting. We conclude our work in Sect. 7.

## 2 Background

Cyber-deception is an emerging proactive cyber defense methodology. When well crafted, deception-based tools can be leveraged as source of threat intelli-

gence data. Deception techniques have two correlated defense strategies: first, to diverge the attacker from tangible assets by simulating vulnerable systems to lure attackers and attract attention, protecting tangible assets from being attacked. Second, to notify about ongoing suspicious activities, which can minimize the impact of an attack.

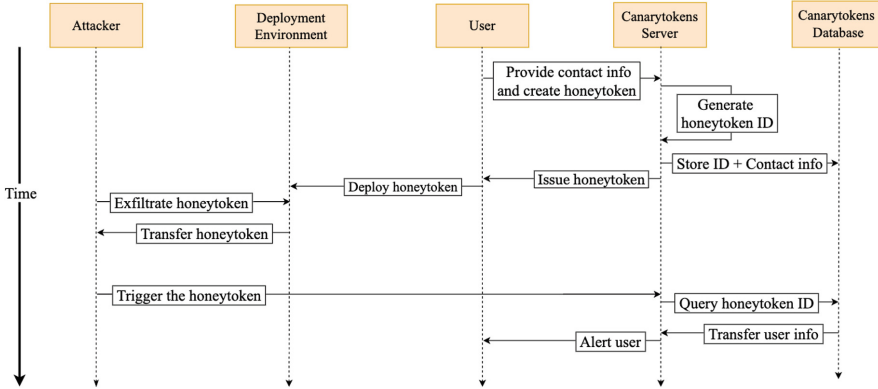
Honeytokens are deceptive entities that work by essentially triggering a notification when the user initiates an action on them. The actions can vary depending on the honeytoken type, such as read, write, query and others. The concept is to embed a token in the deceptive entity and rely on the deceptive layer to consume the token and trigger the alert. Figure 1 shows the conceptual flow of a honeytoken. The honeytoken is deployed on a user’s system at either OS, application, or network levels. On any attempt of access, the honeytoken triggers an alert to the user through the notification mechanism. The recipient’s information is obtained by placing a request to the honeytoken service. The honeytoken service acts as an endpoint and provides a back-end for managing the honeytokens and the metadata of the deployed honeytokens. Upon obtaining recipient information, a notification is sent either as an email or a text message.



**Fig. 1.** Honeytoken concept and alert mechanism

To explain the honeytoken mechanism in detail, we use the Canarytokens (honeytokens service) as a case study to provide concrete examples. Canarytokens is an open-source honeytoken provider that offers 20 different honeytoken types. All the honeytokens provided share the same deployment life-cycle as illustrated in Fig. 2.

To explain the deceptive layer and trigger mechanism, we use the PDF honeytoken from the Canarytoken service. The Adobe Acrobat Reader (AAR) offers a range of functionality for the PDF format to increase the document’s interaction. One of these functionalities is the URI function, which allows linking a local URI to the world wide web via the AAR plugin Weblink [1]. The weblink plugin exposes its functionalities to other applications through the Host-Function-Table API. Once the honeytoken is accessed with AAR, the URL is loaded by the



**Fig. 2.** Canarytokens life-cycle

weblink plugin, which on its turn will start a DNS request to resolve the domain name. This DNS request will alert the owner of the PDF honeytoken.

Unlike honeypots, honeytokens are accessible only if the attacker is within the system where the honeytokens reside. The attacker can gain access through an attack or be an insider. In both cases, honeytokens are very useful as an early alarm against successful data exfiltration if triggered.

### 3 Related Work

Since the invention of deception techniques, much research has been proposed for fingerprinting the deceptive entities [2, 4, 7, 26]. These fingerprinting techniques fall into two categories: passive and active fingerprinting. Passive techniques do not require interaction with the deceptive entity and focus on monitoring. However, active fingerprinting can be either stealthy or noisy. We define stealthy fingerprinting as the process of revealing a deceptive mechanism without triggering any alarm.

#### 3.1 Honeypot Fingerprinting

Holz et al. list some artifacts produced by the honeypot simulation to detect a honeypot [12]. For instance, by verifying the User-Mode-Linux (UML). UML is a way of having a Linux kernel running on another Linux. The initial Linux kernel is the host OS, and the other is the guest OS. By default, the UML executes in Tracing Thread mode (TT) and is not designed to be hidden and can be used to check for all the processes started by the host OS main thread. By executing the command: “*ps a*”, one can retrieve a list of processes and identify UML usage’s existence. Another sign of UML is the usage of the TUN/TAP back-end for the network, which is not common on a real system and can identify UML usage. Another place to look for artifacts is at the file *proc/self/maps* that contains

the current mapped memory regions on a Linux system. On a real OS, the end of the stack is usually `0xc0000000`, which is not the case on a guest OS. These artifacts can be used against honeypots, rendering them visible to the attacker.

Other fingerprinting techniques, such as the network latency comparison, focus on the network layer. For instance, by calculating the differences between an HTTP server and a honeypot HTTP server. Mukkamala et al. utilized timing analysis to reveal if a program is a honeypot. Comparing the timing analysis of ICMP echo requests, they showcased that an HTTP-server honeypot will respond slower than a real HTTP-server [18]. In another work by Srinivasa et al., a framework for fingerprinting different honeypots is proposed. The utilized techniques include so-called probe-based fingerprinting (such as port-scans or banner-checks), and metascan-based fingerprinting (e.g., using data from the Shodan API) [22].

### 3.2 Honeytoken Fingerprinting

Honeytokens can take the form of different data types, such as files, database entries, and URL/DNS records. The first step of fingerprinting is to classify honeytokens to build a standard fingerprint method for each type. Fraunholz et al. have classified honeytokens based on the entity type it emulates [6]. For instance, so-called honeypatches are classified as server-based honeytokens as they emulate a vulnerable decoy. The decoy may host monitoring software that collects important attack information and deceptive files that misinform the attackers. The attacker is redirected to a decoy once the system detects an exploit. Similarly, the database, authentication, and file honeytokens emulate data records and authentication credentials, such as passwords and documents. Similarly, Han et al. proposed a multi-dimensional classification of deception techniques based on the goal, unit, layer, and deployment of the deception [11]. The majority of the surveyed honeytokens are classified based on the detection goal. However, they differ in the four deception layers—the network, system, application, and data layer. In another work, Zhang et al. proposed a two-dimensional taxonomy, which eases the systematic review of representative approaches in a threat-oriented mode, namely from the domains of honeypots, honeytokens, and MTD techniques. They classify deception techniques depending on which phase of the Cyber Kill-Chain they can deceive an attacker. Honeytokens can be used in eight out of twelve phases to deceive attackers [27].

To the best of our knowledge, the only work that examines honeytoken-specific fingerprinting to date is by Srinivasa et al. [21]. The work showcases a proof of concept regarding fingerprinting a public honeytoken provider as a case study. Additionally, they suggest a honeytoken classification based on the four levels of operation and their fingerprinting technique, respectively:

- **Network level:** The honeytokens operating on this level emulate a network entity or use the network as the channel for delivering the alerts. The respective fingerprinting technique for this deceptive layer relies on sniffing the network traffic to detect such calls. In their example with the PDF honeytoken,

Srinivasa et al. observed the usage of DNS queries. However, this fingerprinting method remains passive and not stealthy as it leads to triggering the alert.

- **Application/File-Level:** These honeytokens take the format of a specific file, e.g., PDF or DOCX, and obfuscate an alert mechanism within the file. The alert is triggered if specific applications like Adobe Reader or Microsoft Word opens the honeytoken. The fingerprinting techniques relies on file decompression and obtaining the file honeytoken metadata.
- **System-Level:** These honeytokens utilize operating systems' features such as event logs and *inotify* calls as alert mechanisms. For fingerprinting these, Srinivasa et al. suggest monitoring background-running processes to check for the *inotify* call and to look out for changes in the file or the directory path.
- **Data-Level:** These honeytokens emulate data and can be hard to distinguish from actual data. The technique for fingerprinting honeytokens operating on the data level could vary depending on the data emulated and its alert mechanism. However, as mentioned by Srinivasa et al., viewing the file's meta-data can help an attacker determine whether the file is a possible honeytoken. For instance, Honeyaccount [5] creates fake user-accounts for a system to deceive attackers in using them and hence trigger the alert. On a compromised Windows machine, adversaries can list the user accounts to verify the last known activity. Additionally, adversaries can use Windows PowerShell scripts to recover meta-data about the accounts in Active Directory. This can assist in identifying fake user accounts.

Srinivasa et al. also present different fingerprinting techniques for each honeytoken type. For instance, to fingerprint a PDF honeytoken and determine its trigger channel, they monitored the network traffic when interacting with the file. This fingerprinting technique is noisy as the honeytoken triggers after the interaction. However, a stealthier fingerprinting approach for the same honeytoken was also applied. They used a PDF parser<sup>1</sup> to extract information from the PDF stream. The information consisted of a URL where the domain name belonged to the honeytoken provider. All their proposed fingerprinting techniques relied only on black box testing (i.e., triggering the honeytoken to find the deceptive layer and the alerting mechanism). Lastly, the authors did not consider multiple honeytokens but focused only on a few as a base for their proof of concept.

## 4 Methodology

To build the fingerprinting techniques, we used different methods to extract information from the honeytoken implementation. The methods include white box and black box testing.

---

<sup>1</sup> <https://github.com/DidierStevens/DidierStevensSuite/blob/master/pdf-parser.py>.

## 4.1 Honeytoken Analysis

To analyze the honeytokens, we started by building a classification to help us create fingerprinting techniques for each honeytoken class. Srinivasa et al. have established a Canarytoken honeytoken classification, and we use it as a building block for our extended version [21].

In particular, we extend the previous classification and propose a new one that maps all the publicly offered honeytokens from Canarytokens, as shown in Table 1. We added the dependency layer as a category of classification. The dependency can be at the application or the OS layer. The PDF, *.docx* honeytokens can only trigger when used with a specific application. For instance, *.docx* will only trigger with the application Microsoft Word and would not if opened with the online version Microsoft 365, concluding that it is an application-dependent honeytoken. In contrast, other honeytokens, such as the SQL-DUMP, will trigger with any query from an SQL-capable application. This classification also relates to the privileges needed to stop the triggering mechanism (e.g., the OS-dependent honeytokens will require higher privileges to interrupt the trigger process than the application-dependent ones).

The first analysis step is to classify the honeytokens based on their underlying operation. We leverage the syntax form of the token as the base for the classification. From all the 20 available honeytokens, we find four base usages: DNS, URL, SMTP, IP, and access keys base.

The second step is to classify the honeytokens based on the location of the honeytoken identifier in the token. After analyzing all the URL/DNS-based honeytokens, we observed that the token is a subdomain or a path identifier in the URL. This brought us to conclude the trigger channel based on the location. Subdomain honeytokens will use DNS as a trigger channel, while the URL honeytokens will use the HTTP protocol.

With the classification as a base, we focus on developing fingerprinting techniques that target the dependency layer and the trigger channel. We use white and black box testing in our methodology to identify the gap in the implementation of the honeytokens that can be leveraged for developing fingerprinting techniques.

**White Box Testing.** The Canarytokens (honeytoken provider) service is open source, and we used white box testing to investigate the implementation to find artifacts. In particular, we utilized manual static analysis to check the honeytokens' generation code for any predicted output or patterns that can be used as a fingerprinting base. From our testing, we discover the following:

- ID length: We identify the usage of a fixed length in the honeytoken ID.
- Hardcoded data: We analyzed the source code to search for hardcoded data in the honeytoken's generation process. For instance, upon analyzing the code for the *.exe* file honeytoken, we discover the usage of hardcoded data used to generate a certificate.

**Table 1.** Extended Canarytokens classification

Honeytoken base	Honeytoken name	Trigger channel	Alerting entity	Dependency layer
DNS Subdomain Based	Acrobat Reader PDF	DNS	Adobe Acrobat Reader & Others	Application
	Custom .exe/ Binary		Windows	OS
	MySQL Dump		User Access Control	None
	SQL Server		SQL Server	None
	SQL Server		SQL Server	None
	DNS		DNS Server	None
	Windows Folder		Windows	OS
URL Based	File Explorer	HTTP	File Explorer	None
	SVN Server		SVN Server	None
	Windows Word Document		Microsoft Word Desktop Application	Application
	Windows Excel Document		Microsoft Excel Desktop Application	Application
	QR Code		Web Browsers, Curl & others	None
	Fast Redirect			
	Slow Redirect			
SMTP Based	URL	HTTP	Web Browsers, Curl & others	None
	Custom Image Web Bug			
	Cloned Website			
IP Based	Email Address	SMTP	SMTP Server	None
	Kubernetes Config File	TLS	Kubernetes Application	
	Wireguard Config File	Wireguard Protocol	Wireguard Application	Application
	AWS Key	CloudWatch	CloudWatch	
Access Key Based				

- Template file usage: Canarytokens use a template file to generate the PDF, .docx and .xlsx honeytokens. This template is not changed and leads to static metadata that can be fingerprinted.
- File size: This is a result of the template file usage and constant file size. We consider this an additional artifact to the template to enhance the probability of accurate fingerprinting.

**Black Box Testing.** The black box testing did not focus on testing the system’s internals. Instead, we used it to extract additional information that is only available after the honeytoken generation and validate our findings. The black box included creating and interacting with the honeytoken to reveal the trig-

ger channel and the entity responsible for triggering the alert. The implemented techniques are as follows:

- Extracting metadata from the honeytokens to inspect if there are any static metadata present.
- Monitoring the network traffic when triggering a honeytoken to discover the trigger channel and confirm the white box testing findings.
- Monitoring what sub-processes were started by the application or the OS that triggers the honeytoken. This gives us an idea of how to circumvent the trigger mechanism and stop the honeytoken alert if possible.

With the knowledge gained from the black box, we classify the honeytokens into three categories depending on the token base: URL/DNS, IP, and access key based. The URL/DNS-based honeytokens have a URL or a DNS subdomain directly in the data or the file’s metadata. Regardless of the honeytoken type, they all have the same domain name, `canarytokens.com`, or the equivalent IP address. The access key is a simple AWS access key with an identifier to link the user information with the honeytoken.

## 4.2 Honeytoken Fingerprinting

The first step is to be able to fingerprint honeytokens generated from the official website of Canarytokens<sup>2</sup>. We create and download all possible honeytokens to familiarize ourselves and gain information about all the different honeytokens offered by the Canarytokens service. In particular, we are interested on the underlying trigger mechanism, the trigger channels, and the honeytoken dependency.

To begin, the fingerprinting technique was a simple keyword search in the honeytoken data. The keyword is usually related to the honeytoken provider or publicly known information. We searched for the “canarytokens” keyword in the data or the metadata of all the URL/DNS base honeytokens. Regarding the IP-based honeytokens, our initial fingerprinting method was to perform a reverse DNS lookup of the “canarytokens.com” domain name and compare it to the one in the honeytoken. Finally, we did not discover any fingerprinting strategy for the access key-based honeytokens since all the information related to the access key, since the all the information is saved at the server of the access key provider, except for a repeated pattern in the AWS key ID as displayed in Listing 1.1. The identifier has 12 constant characters `AKIAYVP4CIPP`, which can be used to fingerprint all the AWS keys originating from Canarytokens.

```
1 # 1st key
2 [default]
3 aws_access_key_id = [AKIAYVP4CIPP]G6FXFYHS
4 aws_secret_access_key = UDxJeQftE3ekx+
    KS7skayD8MuN6CVVx0uemuxBSB
```

<sup>2</sup> <https://canarytokens.org/generate>.

```

5  output = json
6  region = us-east-2
7
8  # 2nd-key
9  [default]
10 aws_access_key_id = [AKIAYPV4CIPP]CF45DQPM
11 aws_secret_access_key = 8iTskHJBDDnYpUt1a2KY/
    hTlbScFoAS51cJl4n05
12 output = json
13 region = us-east-2
14
15 # 3rd-key
16 [default]
17 aws_access_key_id = [AKIAYPV4CIPP]A3TB575H
18 aws_secret_access_key = mb8HpotCq27p4rCsQGwYpXo0xx+
    oQcIMpjdT+q0J
19 output = json
20 region = us-east-2

```

**Listing 1.1.** Canarytokens AWS access key repeated characters

The second major milestone is fingerprinting the honeytokens regardless of the domain name. We use the Canarytokens source code to set up the honeytokens service on our private honeytokens server. The keyword search or the IP address comparison approach is ineffective with this setup. However, the keyword search is still valid for the *.exe/.dll* honeytokens files due to the hardcoded data found in the certificate generation source code.

As mentioned before, the white box testing revealed that the URL/DNS-based honeytokens follow a specific pattern. The DNS/URL contains a 25-character alphanumeric identifier (ID) as displayed in Table 2, which is used to link the honeytokens with the user’s contact information. The ID is the subdomain for the DNS-based honeytokens and is the path for the URL-based ones. The placement of the URL/DNS value in the honeytokens is known to us. However, there are other URLs/DNS in some honeytokens. For instance, the URL in the *.docx* honeytokens resides in the metadata, which already includes other URLs to microsoft.com. In order to determine the existence of a honeytokens URL, we loop through each URL and see if they have a 25-character alphanumeric string in the DNS/URL. If they do, we label it as a possible honeytokens URL.

**Table 2.** URL/DNS Honeytokens followed pattern

Identifier	uq3501pu9mo56obz6kn5auhpq
URL	<a href="http://domain.name/url/path/uq3501pu9mo56obz6kn5auhpq/contact.php">http://domain.name/url/path/uq3501pu9mo56obz6kn5auhpq/contact.php</a>
DNS	uq3501pu9mo56obz6kn5auhpq.domain.name

Our analysis suggests that the file type honeytokens use a static template to generate the PDF, *.docx*, and *.xlsx* files. For instance, the *template.pdf* file in the source code leads to constant metadata in the PDF honeytoken. Normally, some metadata attributes, such as the Document UUID, should be unique for each file. A constant UUID will make it easy to identify any PDF file from Canarytokens, even if the domain name is private. Additionally, other data can make the attacker more confident that this is a honeytoken file (e.g., created and modified dates). However, the file creation and modification dates are old (7 years), and any data in it might not be valid anymore from the attacker's point of view. See Appendix Listing 1.2 for more details.

The Canarytokens implementation uses template files to generate all the file type honeytokens, which results in fixed file sizes. We observe that all the PDF, *.docx*, and *.xlsx* have the same size of 5KB, 15KB, and 7.7KB respectively. This additional artifact can be used with the template static metadata to raise the confidence of our fingerprinting method. Additionally, this constant small file size indicates that the file is empty and may not lure the attacker into interacting with it.

## 5 Proof of Concept: *Honeysweeper*

This section demonstrates the applicability of our honeytokens' fingerprinting techniques based on the Canarytoken implementation [19]. The fingerprinting tool's, namely *honeysweeper*, source code is available at our GitHub repository<sup>3</sup>.

### 5.1 Overview

From all the information gained from the black/white box testing, we built an OS-independent tool that can successfully fingerprint 14 out of the 20 honeytokens offered by Canarytokens. The tool relies on a primary fingerprinting technique matching the 25-character string identifier. However, this fingerprint method introduces the problem of false positives. As we discussed earlier, some honeytokens (i.e., file-type ones) contain more than one URL/DNS. If by any chance, another link contains a 25 characters string, the tool will label it as a possible honeytoken. Nevertheless, from an attacker's perspective, we argue that false negatives are more critical since they would raise an alarm.

*Honeysweeper* begins by revealing the honeytoken extension for the file-type ones and then extracting the DNS/URL. URL/DNS/Email honeytokens can be added in a text file and passed to the tool. As in the case of PDF, *.docx* and *.xlsx* files, the tool needs to decompress the file as shown in Appendix Listings 1.3 – 1.4, and loops through each file to extract all the tokens. Once obtained, *honeysweeper* runs the `__find_canarytoken(string)` to match any pattern that matches the 25-character string in the honeytoken content. The PDF, *.docx*, and *.exe/.dll* honeytokens have higher confidence due to the earlier additional

<sup>3</sup> [https://github.com/aau-network-security/canarytokens\\_finger\\_printer](https://github.com/aau-network-security/canarytokens_finger_printer).

artifacts, i.e., the static template as shown in Appendix Listing 1.2 and the small file size as shown in Fig. 3. The tool includes checks for the PDF template as a proof of concept and can easily be enhanced to detect other files such as *.docx* and *.xlsx*.















Name	Size	Kind
 9y2jr13ve8xjwg292euv36kiu.pdf	5 KB	PDF Document
 79a4ust07x55z4knhjz4xofri.pdf	5 KB	PDF Document
 jqntearfpyxonyk432xrdy4ut.pdf	5 KB	PDF Document
 kkavzpz3ezlanse4dsumelz24.pdf	5 KB	PDF Document
 ranmd5nmfnoij6ibbdxs1use.pdf	5 KB	PDF Document
 uekkk4dxseegb89l5hxo2ezak.pdf	5 KB	PDF Document
 ulmmft7fw5wlji429w2c5sexj.pdf	5 KB	PDF Document
 y0jkqbkr210dni9ivqo3eofel.pdf	5 KB	PDF Document
 anht4ag0x24ob21artvndffeo.xlsx	8 KB	Spreadsheet
 dwld1o5p9sjmdlk0ezg14orw4.xlsx	8 KB	Spreadsheet
 ed23csngpg0qsnss5yfpeinaa.xlsx	8 KB	Spreadsheet
 6hjjqoeaselut4wb2qhsgjxxw.docx	16 KB	Micros...(docx)
 cbyam7ftvi7wbnadni9ptnoik.docx	16 KB	Micros...(docx)
 s6b8enlmn9o3v27vg292qyxer.docx	16 KB	Micros...(docx)

Fig. 3. Honeytokens file-type constant size artifact

## 5.2 Limitations

The Wireguard and Kubernetes honeytokens are not included in *honeysweeper* as we found no possible way of fingerprinting them when deployed with a private IP. All the data in the honeytokens are randomly generated, e.g., the public and private keys. However, this technique remains effective if the honeytokens are deployed with a known honeytoken provider IP address. The fingerprinting techniques for SVN and SQL-server are not included in the fingerprinting tool since both honeytokens are not directly accessible to the attacker. A possible fingerprinting method for the SQL server can be to check the size of the table where the honeytoken resides. If the table is empty, it may not deceive the attacker for any further interaction. The other honeytokens e.g., PDF, *.docx*, and SQL-dump are available directly on the system and the fingerprinting methods are covered in *honeysweeper*.

## 6 Countermeasures Against Fingerprinting

The fixed ID length is the primary artifact shared among the studied honeytokens. We propose that the honeytoken identifier should be randomized in length

or set in a range. For instance, the ID length could be between 25 and 32 characters, making the fingerprinting process harder and removing the 25-character ID artifact. This mitigation is valid for all the honeytokens containing a URL/DNS with 25 character identifiers. However, this only solves one problem.

The following recommendations are valid for all the template-dependent honeytokens. The PDF honeytokens should have random metadata. In the case of PDF, the attacker can generate a PDF Canarytokens and compare it to any PDF exfiltrated. Even if the honeytoken administrator changes the domain name and removes the 25-character ID artifact, the metadata alone is enough to raise suspicion. To address this, we propose to randomize the PDF XMP metadata. There are a few rules to keep the metadata consistent and not leave a metadata-modification footprint [10]. We present our solution in Appendix Listing 1.5.

Moreover, the honeytoken administrator should modify the content of the *.docx*, *.xlsx*, and PDF files before deployment to change the document size which are *.docx* files are always 15 KB, the *.xlsx* files with 7.7 KB, and the PDF files with 5 KB. Once modified, the honeytokens will resemble an actual file with data and lure the attacker into opening it. Otherwise, the attacker can combine the honeytoken file size with other artifacts to ensure the existence of a trap.

The signing process for the *.exe/binary* honeytokens should be with certificates unrelated to any honeytoken provider. As seen in the Canarytokens source code, a new certificate is generated to sign the *.exe/.dll* files. We generate an executable honeytoken using the source code locally to investigate the generation process. We see that a private key and a certificate is generated to sign the honeytoken and are removed after the process is complete. Nevertheless, the information included in the signature is hard-coded. Figure [4] shows the hard-coded information in the certificate. This hard-coded information will be the same for all the *.exe/binary* honeytokens and can be an artifact.

```
[ ca_dn ]
countryName           = "ZA"
organizationName      = "Thinkst Applied Research"
organizationalUnitName = "Thinkst Applied Research CA"
commonName            = "Thinkst Root CA"
```

**Fig. 4.** Certificate hardcoded data

When deploying the stored procedure for a table on the SQL server, the administrator can set explicit permissions on the stored procedure by denying the public users from viewing the stored procedure's definition. The same approach applies for the SQL functions as a honeytoken. The function permission can be fragmented. For example, allow the public to select the functions and views but disallow viewing the definitions (syntax). Additionally, the trap table should be populated with random fake data to lure the attacker into interacting with it.

The Wireguard and Kubernetes honeytokens should use an IP address not linked with a honeytokens domain name. If no domain name is available and there is no alternative but to use the Canarytokens servers due to development and maintenance costs, an administrator can use a local server IP and redirect the traffic to Canarytokens servers.

## 7 Conclusion

Deception techniques like honeytokens are an essential extra layer of defense, and deploying them is becoming more and more common. However, for the deception technique to achieve its goal, it should be well crafted to deceive and should not include easy to exploit fingerprinting artifacts. This paper proposes fingerprinting techniques against most existing Canarytokens' honeytokens proposals and implementations. We analyze all the publicly offered honeytokens and propose countermeasures against the suggested techniques. As ethical disclosure, we informed Canarytokens of our findings. For future work, we plan on exploring other fingerprinting methods. For instance, the signature verification of the *.exe/.dll* files and other techniques. Additionally, we consider improving the honeytokens ID generation process by including a non-repudiation concept.

## Appendix

### Static Data on PDF Canarytoken

Listing 1.2 shows the static data identified on parsing the composite XML file of the PDF Canarytoken. We can observe static data on the modify date, create date and metadata date.

```

1  <x:xmpmeta xmlns:x="adobe:ns:meta/" x:xmptk="Adobe XMP
   Core 5.6-c015 81.157285, 2014/12/12-00:43:15">
2  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-
   syntax-ns#">
3  <rdf:Description rdf:about=""
4  xmlns:xmp="http://ns.adobe.com/xap/1.0/"
5  xmlns:dc="http://purl.org/dc/elements/1.1/"
6  xmlns:xmpMM="http://ns.adobe.com/xap/1.0/mm/"
7  xmlns:pdf="http://ns.adobe.com/pdf/1.3/">
8  <xmp:ModifyDate>2015-07-22T16:41:31+02:00</
   xmp:ModifyDate>
9  <xmp:CreateDate>2015-07-22T16:38:51+02:00</
   xmp:CreateDate>
10 <xmp:MetadataDate>2015-07-22T16:41:31+02:00</
   xmp:MetadataDate>
11 <xmp:CreatorTool>Acrobat Pro 15.8.20082</
   xmp:CreatorTool>
12 <dc:format>application/pdf</dc:format>

```

```

13         <xmpMM:DocumentID>uuid:a2364080-b5a8-1b46-b156-
           ea05c4972d03</xmpMM:DocumentID>=
14         <xmpMM:InstanceID>uuid:7656c56e-b1e6-f444-801f
           -06e28a50831f</xmpMM:InstanceID>
15         <pdf:Producer>Acrobat Pro 15.8.20082</
           pdf:Producer>
16     </rdf:Description>
17 </rdf:RDF>
18 </x:xmpmeta>

```

Listing 1.2. PDF honeytoken static metadata

## Fingerprinting of PDF Canarytoken

Listing 1.3 shows the pseudo code for fingerprinting of PDF Canarytoken. The method checks for URLs embedded in the PDF and against a list of known honeytoken service URLs.

```

1  def find_token_in_pdf(file_location):
2      check_template(file_location) # check for template
           artifact
3      # List for URLs found
4      list_of_urls = []
5      pdf = open(file_location, "rb").read()
6      stream = re.compile(b'.*?FlateDecode.*?stream(.*)
           endstream', re.S)
7      for s in re.findall(stream, pdf):
8          s = s.strip(b'\r\n')
9          line = ""
10         try:
11             line = zlib.decompress(s).decode('latin-1')
12             # changed this from UTF-8 to latin-1 as
13             # it throws errors. We
14             # want the app to be silent :)
15         except Exception as e:
16             print(e)
17             token = Tokenfinder.find_tokens_in_string(line)
18             if token:
19                 list_of_urls.extend(token)
20             if len(list_of_urls) == 0:
21                 print("No canaries detected")
22                 return None
23             else:
24                 print(str(len(list_of_urls)) + " canary URLs
25                     detected in the file")
26                 for url in list_of_urls:
27                     print("Canary detected!: ", url)
28                 print()

```

Listing 1.3. PDF fingerprinting

## Fingerprinting of .docx and .xlsx Canarytokens

Listing 1.4 shows the pseudo code for fingerprinting of .docx and .xlsx Canarytokens. The technique unzips the composite file formats to check for URLs embedded in the files.

```

1  def check_office_files(file_location):
2      list_of_urls = [] # List to hold all urls in the
                           file
3      try:
4          # Unzip the office file without saving to
                           folder
5          unzipped_file = zipfile.ZipFile(file_location, "
                           r")
6          # List of all the content of the zip
7          namelist = unzipped_file.namelist()
8          # Reads every file in the zip file and looks if
                           it contains the string you wish to search
                           for
9          for item in namelist:
10             content = str(unzipped_file.read(item))
11             token = Tokenfinder.find_tokens_in_string(
                           content)
12             if token:
13                 list_of_urls.extend(token)
14     except OSError as e:
15         print(f"Exception: {e}")
16     # If no results of the search
17     if len(list_of_urls) == 0:
18         return None
19     else:
20         print(str(len(list_of_urls)) + " canary URLs
                           detected in the file")
21         for url in list_of_urls:
22             print("Canary detected: ", url)
23             print()

```

Listing 1.4. .docx and .xlsx fingerprinting

## Mitigation of Metadata in Canarytoken

Listing 1.5 shows the mitigation by randomization of the file creation date and time. The randomness avoids static creation dates that is implemented by Canarytokens.

```

1  from pikepdf import Pdf
2  import uuid, random, datetime, os
3
4  # make creation date with random Time-Zone [+1 to +3]

```

```

5  def creation_date():
6      time = datetime.datetime.now()
7      rand_region = str(random.randint(1, 3))
8      stamp = time.strftime('2022-%m-%d')+ 'T' + time.
          strftime('%H:%M:%S') + '+0'+ rand_region+ ':00'
9      return stamp
10
11
12 def modification_date():
13     time = datetime.datetime.now()
14     return time.strftime('%Y-%m-%d')+ 'T' + time.strftime
          ('%H:%M:%S')
15
16 def add_metadata(source_pdf, out_dir):
17     mod_date = modification_date()
18     with Pdf.open(source_pdf) as pdf:
19         with pdf.open_metadata(set_pikepdf_as_editor=
             False) as meta:
20             meta['xmp:CreatorTool'] = 'Acrobat Pro
                22.001.20112'
21             meta['xmpMM:DocumentID'] = str(uuid.uuid4()
                )
22             meta['xmpMM:InstanceID'] = str(uuid.uuid4()
                )
23             meta['xmp:CreateDate'] = creation_date()
24             meta['xmp:ModifyDate'] = mod_date
25             meta['xmp:MetadataDate'] = mod_date
26             meta['pdf:Producer'] = 'Acrobat Pro
                22.001.20112'
27     pdf.save(os.path.join(out_dir, os.path.basename(
        source_pdf)))
28     print('Done!')
29
30 source_pdf = "/Users/mm/Downloads/pdftoken.pdf"
31 out_dir = '/Users/mm/Desktop/'
32 add_metadata(source_pdf, out_dir)

```

Listing 1.5. Metadata mitigation

## References

1. Acrobat: Acrobat API reference (2021). [https://opensource.adobe.com/dc-acrobat-sdk-docs/acrobat-sdk/html2015/Acro12\\_MasterBook/API\\_References\\_SectionPage/API\\_References/Acrobat\\_API\\_Reference/AV\\_Layer/Weblink.html](https://opensource.adobe.com/dc-acrobat-sdk-docs/acrobat-sdk/html2015/Acro12_MasterBook/API_References_SectionPage/API_References/Acrobat_API_Reference/AV_Layer/Weblink.html)
2. Aguirre-Anaya, E., Gallegos-Garcia, G., Luna, N.S., Vargas, L.A.V.: A new procedure to detect low interaction honeypots. *Int. J. Electr. Comput. Eng. (IJECE)* 4(6), 848–857 (2014)
3. Čenys, A., Rainys, D., Radvilavicius, L., Goranin, N.: Database level honeytoken modules for active DBMS protection. In: Nilsson, A.G., Gustas, R., Wojtkowski,

- W., Wojtkowski, W.G., Wrycza, S., Zupančič, J. (eds.) *Adv. Inf. Syst. Dev.*, pp. 449–457. Springer, US, Boston, MA (2006)
4. Dahbul, R.N., Lim, C., Purnama, J.: Enhancing honeypot deception capability through network service fingerprinting. *J. Phys: Conf. Ser.* **801**, 012057 (2017). <https://doi.org/10.1088/1742-6596/801/1/012057>
5. Faveri, C.D., Moreira, A.: Visual modeling of cyber deception. In: 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 205–209 (2018). <https://doi.org/10.1109/VLHCC.2018.8506515>
6. Fraunholz, D., et al.: Demystifying deception technology: a survey. *CoRR* abs/1804.06196 (2018). <https://arxiv.org/abs/1804.06196>
7. Fu, X., Yu, W., Cheng, D., Tan, X., Streff, K., Graham, S.: On recognizing virtual honeypots and countermeasures. In: 2006 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing, pp. 211–218 (2006). <https://doi.org/10.1109/DASC.2006.36>
8. Ghirardello, K., Maple, C., Ng, D., Kearney, P.: Cyber security of smart homes: development of a reference architecture for attack surface analysis. In: *Living in the Internet of Things: Cybersecurity of the IoT - 2018*, pp. 1–10 (2018). <https://doi.org/10.1049/cp.2018.0045>
9. Guarnizo, J.D., et al.: Siphon: towards scalable high-interaction physical honeypots. In: *Proceedings of the 3rd ACM Workshop on Cyber-Physical System Security*, pp. 57–68. CPSS 2017, Association for Computing Machinery, New York, NY, USA (2017). <https://doi.org/10.1145/3055186.3055192>
10. Gungor, A.: Pdf forensic analysis and XMP metadata streams (2017). <https://www.meridianiscovery.com/articles/pdf-forensic-analysis-xmp-metadata/>
11. Han, X., Kheir, N., Balzarotti, D.: Deception techniques in computer security: a research perspective. *ACM Comput. Surv.* **51**(4), 1–36 (2018). <https://doi.org/10.1145/3214305>
12. Holz, T., Raynal, F.: Detecting honeypots and other suspicious environments. In: *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, pp. 29–36 (2005). <https://doi.org/10.1109/IAW.2005.1495930>
13. IBM: how much does a data breach cost? (2021). <https://www.ibm.com/security/data-breach>
14. IBM: Insights into what drives data breach costs (2021). <https://www.ibm.com/account/reg/uk-en/signup?formid=urx-51643>
15. IBM: key findings (2021). <https://www.ibm.com/downloads/cas/OJDVQGRY>
16. La, Q.D., Quek, T.Q.S., Lee, J., Jin, S., Zhu, H.: Deceptive attack and defense game in honeypot-enabled networks for the internet of things. *IEEE Internet Things J.* **3**(6), 1025–1035 (2016). <https://doi.org/10.1109/JIOT.2016.2547994>
17. Mokube, I., Adams, M.: Honeypots: Concepts, approaches, and challenges. In: *Proceedings of the 45th Annual Southeast Regional Conference*. p. 321–326. ACM-SE 45, Association for Computing Machinery, New York, NY, USA (2007). <https://doi.org/10.1145/1233341.1233399>
18. Mukkamala, S., Yendrapalli, K., Basnet, R., Shankarapani, M.K., Sung, A.H.: Detection of virtual environments and low interaction honeypots. In: 2007 IEEE SMC Information Assurance and Security Workshop, pp. 92–98 (2007). <https://doi.org/10.1109/IAW.2007.381919>
19. Research, T.A.: Canarytokens. <https://github.com/thinkst/canarytokens>
20. Sethia, V., Jeyasekar, A.: Malware capturing and analysis using dionaea honeypot. In: 2019 International Carnahan Conference on Security Technology (ICCST), pp. 1–4 (2019). <https://doi.org/10.1109/CCST.2019.8888409>

21. Srinivasa, S., Pedersen, J.M., Vasilomanolakis, E.: Towards systematic honeytoken fingerprinting. In: 13th International Conference on Security of Information and Networks. SIN 2020, Association for Computing Machinery, New York, NY, USA (2020). <https://doi.org/10.1145/3433174.3433599>
22. Srinivasa, S., Pedersen, J.M., Vasilomanolakis, E.: Gotta catch'em all: a multistage framework for honeypot fingerprinting. arXiv preprint [arXiv:2109.10652](https://arxiv.org/abs/2109.10652) (2021)
23. Srinivasa, S., Pedersen, J.M., Vasilomanolakis, E.: Open for hire: attack trends and misconfiguration pitfalls of iot devices. In: Proceedings of the 21st ACM Internet Measurement Conference, pp. 195–215. IMC 2021, Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3487552.3487833>, <https://doi.org/10.1145/3487552.3487833>
24. Vasilomanolakis, E., et al.: This network is infected: hostage - a low-interaction honeypot for mobile devices. In: Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, pp. 43–48. SPSM 2013, Association for Computing Machinery, New York, NY, USA (2013). <https://doi.org/10.1145/2516760.2516763>
25. Vasilomanolakis, E., Karuppayah, S., Mühlhäuser, M., Fischer, M.: Hostage: a mobile honeypot for collaborative defense. In: Proceedings of the 7th International Conference on security of information and networks. SIN 2014, vol. 2014, pp. 330–333. ACM (2014)
26. Vetterl, A., Clayton, R.: Bitter harvest: systematically fingerprinting low- and medium-interaction honeypots at internet scale. In: 12th USENIX Workshop on Offensive Technologies (WOOT 18). USENIX Association, Baltimore, MD (2018). <https://www.usenix.org/conference/woot18/presentation/vetterl>
27. Zhang, L., Thing, V.L.: Three decades of deception techniques in active cyber defense-retrospect and outlook. *Comput. Secur.* **106**, 102288 (2021). <https://arxiv.org/abs/2104.03594>