

**JARM Fingerprinting: A Lucrative Approach to Bug Bounty Hunting & Cybercrime**

J. Harbor Higginbotham

Department of Cybersecurity, Metropolitan State University of Denver

CIS 4990-001: Capstone Semester in Cybersecurity

Professor Megg Bertoni

December 14th, 2023

## INTRODUCTION

JARM fingerprints are an active system fingerprinting method to identify assets on the public-facing internet. The fingerprinting tool was created by the Salesforce Security Team, An American-based customer relationship management software and cloud service provider, which made the tool open source on October 12th, 2021.

The Transport Layer Security (TLS) based fingerprinting method, written in Python, has many applications. However, deep down at its core, all JARM fingerprinting does is "Quickly verify that all servers in a group have the same TLS configuration." (Althouse, 2021) Simply put, servers with the same TLS configurations will likely have the same owners. Often, server administrators/owners will configure something once and replicate those configurations across all future and current assets (Servers, Laptops, Routers, etc.), commonly referred to as a "golden image" approach to configuration. It is because of this methodology of mass replication of configurations that JARM can proactively discover assets across the broader internet. Which in turn has skyrocketed JARM fingerprinting use within the cybersecurity community. For Example, many open-source intelligence (OSINT) tools, including VirusTotal and ShodanIO, have integrated JARM into their triage platforms.

## USE CASES & APPLICATIONS

One hypothetical use case for JARM would be, say, a malware analyst finds a command and control Domain (Example of a Domain: Google.com) or Internet Protocol (IP) address within the malware samples behavior or code. The analyst can now take the JARM fingerprint of the Domain or IP and scan the broader internet for other malicious attacker-owned assets that match that previously taken fingerprint. With this information, a cyber defender would have

many more indicators of compromise (IoCs) to look for and defend against. In short, JARM fingerprinting allows for robust cyber defense as defenders no longer need to wait for actionable information to come to them in the form of an attack. This is known in the cybersecurity community as "proactive defense."

Despite JARMS' proactive defense capabilities and theoretical use cases, like all modern technology, JARM is a double-edged sword. If one finds attacker-associated assets using JARM fingerprints, it follows that JARM could be used to identify any entity's assets as well. However, Any entity's internet-facing assets are already known; for example, to name a few methods, anyone's assets can be found via Domain Name System (DNS) queries or even on publicly available ledgers of IP address block assignments maintained by the various arms and region-specific versions of the Internet Corporation for Assigned Names and Numbers (ICANN).

It should be noted that no one method of asset discovery, also known as enumeration/reconnaissance, can ever truly reveal the whole picture, and JARM fingerprinting is no exception. JARM is limited to only public-facing assets that use the TLS protocol to encrypt communications between endpoints.

However, this limitation is also what is unique about JARM; TLS is a standard suite for securing website traffic communication over HTTPS (HyperText Transfer Protocol Secure); this means the tool can discover a plethora of web-based assets. With the speed and reliability of JARM, the tool should be able to find web-facing TLS-based resources on assets shortly after they are allocated and may even find forgotten or hidden assets/resources if one were to guess the right target. From an attacker's perspective, this means the surface of the attack has increased, which creates more opportunities for exploitation, which subsequently implies profit generation. This principle does not necessarily need to be malicious.

"A bug bounty is a monetary reward given to ethical hackers for successfully discovering and reporting a vulnerability or bug to the application's developer. Bug bounty programs allow companies to leverage the hacker community to continuously improve their systems' security posture over time." (HackOne, 2021) Bug bounty hunters are white hat hackers who participate in these bug bounty programs. These hackers only get paid for the responsible reporting of vulnerabilities; however, if they are good enough, white hat hackers can make an exorbitant amount of money.

One subset of vulnerability types that bug hunters look for are just simple misconfigurations: "Misconfigurations can lead to unauthorized access, data breaches, service disruptions, or other security incidents." (Chaudhary, 2023) These vulnerability types are most often seen on recently configured assets or on forgotten assets whose configurations are now dated and no longer secure. Given that the severity of these types of vulnerabilities can range from catastrophic to minor, any findings that result in a responsible disclosure could lead to a payout to the bug hunter who finds it first.

There are a plethora of different styles that bug hunters use, but there is one method worth highlighting due to its relevance, and it involves reconnaissance. Some hunters may look at the directly available surface with the scope of the assessment using various sufficient methods that will not be mentioned for the sake of brevity. This is a standard method and one that can result in success. However, others may go deeper with their recon and look for obscure entry points or forgotten assets that would be missed if one were to focus only on the main scope of the assessment, otherwise known as a more in depth recon approach.

## THESIS & LITERATURE/CODE REVIEW

JARM Fingerprinting is an overlooked approach of finding these rare needles in a haystack that are obscure entry points or forgotten assets. As established, JARM is fast, reliable, and built for asset discovery according to the tools GitHub repository. Because of this, using JARM may result in a lucrative opportunity for those with the skill sets to capitalize on the information JARM can provide.

The tool itself works "by actively sending 10 TLS Client Hello packets to a target TLS server and capturing specific attributes of the TLS Server Hello responses. The aggregated TLS server responses are then hashed in a specific way to produce the JARM fingerprint." (Althouse, 2021)

### The 10 Hello Packets:

```
# 10 Hello Packets #
# Array format = [destination_host, destination_port, version, cipher_list, cipher_order, GREASE, RARE_APLN, 1.3_SUPPORT, extension_orders]
tls1_2_forward = [destination_host, destination_port, "TLS_1.2", "ALL", "FORWARD", "NO_GREASE", "APLN", "1.2_SUPPORT", "REVERSE"]
tls1_2_reverse = [destination_host, destination_port, "TLS_1.2", "ALL", "REVERSE", "NO_GREASE", "APLN", "1.2_SUPPORT", "FORWARD"]
tls1_2_top_half = [destination_host, destination_port, "TLS_1.2", "ALL", "TOP_HALF", "NO_GREASE", "APLN", "NO_SUPPORT", "FORWARD"]
tls1_2_bottom_half = [destination_host, destination_port, "TLS_1.2", "ALL", "BOTTOM_HALF", "NO_GREASE", "RARE_APLN", "NO_SUPPORT", "FORWARD"]
tls1_2_middle_out = [destination_host, destination_port, "TLS_1.2", "ALL", "MIDDLE_OUT", "GREASE", "RARE_APLN", "NO_SUPPORT", "REVERSE"]
tls1_1_middle_out = [destination_host, destination_port, "TLS_1.1", "ALL", "FORWARD", "NO_GREASE", "APLN", "NO_SUPPORT", "FORWARD"]
tls1_3_forward = [destination_host, destination_port, "TLS_1.3", "ALL", "FORWARD", "NO_GREASE", "APLN", "1.3_SUPPORT", "REVERSE"]
tls1_3_reverse = [destination_host, destination_port, "TLS_1.3", "ALL", "REVERSE", "NO_GREASE", "APLN", "1.3_SUPPORT", "FORWARD"]
tls1_3_invalid = [destination_host, destination_port, "TLS_1.3", "NO1.3", "FORWARD", "NO_GREASE", "APLN", "1.3_SUPPORT", "FORWARD"]
tls1_3_middle_out = [destination_host, destination_port, "TLS_1.3", "ALL", "MIDDLE_OUT", "GREASE", "APLN", "1.3_SUPPORT", "REVERSE"]
#Possible versions: SSLv3, TLS_1, TLS_1.1, TLS_1.2, TLS_1.3
#Possible cipher lists: ALL, NO1.3
#GREASE: either NO_GREASE or GREASE
#APLN: either APLN or RARE_APLN
#Supported Versions extension: 1.2_SUPPORT, NO_SUPPORT, or 1.3_SUPPORT
#Possible Extension order: FORWARD, REVERSE
```

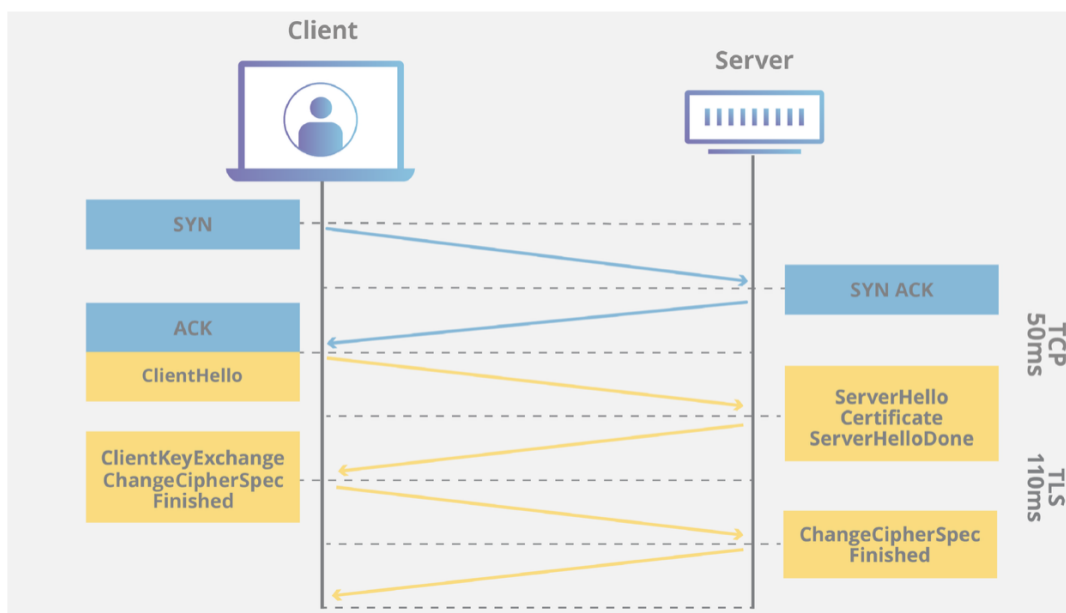
TLS is an end-to-end Transmission Control Protocol (TCP) designed to deliver messages from one endpoint (a computer or server) to one or more other endpoint(s). Its purpose is to secure communication between endpoints using both asymmetric & symmetric encryption. TLS does this by initiating and completing a TCP 3-way handshake, which is standard for TCP-based protocols. Then, the client requesting resources sends a TLS Client Hello packet to the server. This packet consists of "which TLS version the client supports, the cipher suites supported, and a string of random bytes known as the "client random" (Cloudflare). The server then responds with

a TLS Server Hello packet, "containing the server's SSL certificate, the server's chosen cipher suite, and the "server random," another random string of bytes that are generated by the server."

(Cloudflare). It is this interaction that the JARM tool uses to fingerprint the system. This is where the ten server questions previously mentioned come into play. Each of those ten questions is comprised of the following:

- The Destination Host (The target)
- The Destination Port (Where the TLS resources is on the target)
- Possible versions of TLS (SSLv3, TLS\_1, TLS\_1.1, TLS\_1.2, TLS\_1.3)
  - Note: SSL is just an older version of TLS
- Possible cipher lists (How the communication will be secured)
- The GREASE (Generate Random Extensions And Sustain Extensibility) which is "a mechanism to prevent extensibility failures in the TLS ecosystem. It reserves a set of TLS protocol values that may be advertised to ensure peers correctly handle unknown values." (Benjamin, 2020)
- The APLN (Application-Layer Protocol Negotiation) which negotiates which protocol should be used for communication
- The Supported Versions extension
- The Possible Extension Order

The TLS handshake (Cloudflare):



Once the 10 TLS Server Hellos with their various questions are sent to the server, each response from the server to each of the clients TLS Hellos individual questions is then mapped to

a hexadecimal (hex) value. These hex values are placed in order and run through a hashing algorithm. A hashing algorithm is a mathematical method to change the input irreversibly; the same input will always have the same output. This algorithm is explicitly a “hybrid fuzzy hash, it uses the combination of a reversible and non-reversible hash algorithm to produce a 62 character fingerprint”. (Althouse, 2021) Finally, the tool then responds with the FIN flag, which tells the server it no longer wishes to connect or receive resources, ending the connection before any key exchange occurs in the TLS handshake.

## METHODS

In Order to JARM fingerprint one must do the following... **NOTE:** If on Windows please download and install Python first this can be done easily by following the tutorial below.

- Python Install Tutorial: [https://phoenixnap.com/kb/how-to-install-python-3-windows\](https://phoenixnap.com/kb/how-to-install-python-3-windows/)

### 1) Open a command line terminal

- a) For Windows: type [CTRL + ALT + W]
- b) For Linux: type [SHIFT + CTRL + T]
- c) For OSX (Mac): press [F4]

### 2) Create a directory and move to it:

- a) For Windows: in the command line type “mkdir Jarm” then type “cd Jarm”
- b) For OSX & Linux: in the command line type “mkdir Jarm && cd Jarm”

### 3) Download the JARM Tool:

- a) For Windows: in the command line type the following bellow:

```
wget.exe https://raw.githubusercontent.com/salesforce/jarm/master/jarm.py >  
Jarm.py
```

b) For Linux: in the command line type the following bellow:

```
curl https://raw.githubusercontent.com/salesforce/jarm/master/jarm.py >  
Jarm.py
```

#### 4) Run the JARM Tool:

a) In the command line type “python3 Jarm.py [Target]”

### SHOWCASE OF CAPABILITIES & SUBSEQUENT IMPLICATIONS

According to Althouse, the FuzzyHash Result, also known as a JARMFingerprint, is comprised of two parts:

- 1) The first 30 characters are made up of the cipher and TLS version chosen by the server for each of the 10 client hello's sent.
- 2) The remaining 32 characters are a truncated SHA256 hash of the cumulative extensions sent by the server, ignoring x509 certificate data.

Example of running the JARM Tool on Facebook.com taken on 12/03/2023:

```
harborhigginbotham@Harbors-MacBook-Pro og % python3 jarm.py facbook.com  
Domain: facbook.com  
Resolved IP: 157.240.28.18  
JARM: 27d27d27d00000000041d43d00041dcd947229d467ddf1b9b05cf29440ee27  
Time: 0.46748132200000003 Seconds
```

“When comparing JARM fingerprints, if the first 30 characters are the same but the last 32 are different, this would mean that the servers have very similar configurations, accepting the same versions and ciphers, though not exactly the same given the extensions are different.” (Althouse,



2021) This can be best shown in the fingerprints for Instagram and Oculus which were acquired by Facebook within the last few years.

JARM for Instagram.com taken on 12/03/2023:

```
harborhigginbotham@Harbors-MacBook-Pro og % python3 jarm.py instagram.com
Domain: instagram.com
Resolved IP: 157.240.28.174
JARM: 27d27d27d0000001dc41d43d00041d1c5ac8aa552261ba8fd1aa9757c06fa5
Time: 0.4115602220000003 Seconds
```

JARM for Oculus.com taken on 12/03/2023:

```
harborhigginbotham@Harbors-MacBook-Pro og % python3 jarm.py oculus.com
Domain: oculus.com
Resolved IP: 157.240.28.52
JARM: 29d29d00000000000041d43d00041daf0a91d29a1ea7b1caa01c572cf37a99
Time: 0.430402684 Seconds
```

As we can see in the fingerprints, the first 30 characters match both for Instagram and Facebook's fingerprints, meaning they have similar configurations, but the TLS extensions are different (The last 32 characters). However, this is not the case for Oculus, which did not match either entity's fingerprints. However, when this same data was pulled prior, both Instagram and Facebook matched, and Oculus again did not match. As we can see, this means that the fingerprints tend to drift.

Looking at the data on the original JARM Github repository, we can see that at one point, these same domains once had identical JARM fingerprints.

JARMs pulled on the Git Repo Page from 2021 (Althouse):

facebook.com	27d27d27d29d27d1dc41d43d00041d741011a7be03d7498e0df05581db08a9
instagram.com	27d27d27d29d27d1dc41d43d00041d741011a7be03d7498e0df05581db08a9
oculus.com	29d29d20d29d29d21c41d43d00041d741011a7be03d7498e0df05581db08a9

This shows a significant flaw within the JARM fingerprinting process; while the fingerprints are accurate, they are susceptible to change. The data indicated that in 2021, all three companies shared a golden image for TLS configurations that were replicated across all three companies. However, this does not appear to be the case anymore. The changes can be explained in a number of ways, but what it boils down to is these companies are now separate entities. For instance, Facebook owns all three companies; however, with the recent transition of Facebook to Meta in October of 2021, which purchased Oculus to build the Metaverse (A virtual world) as the next frontier in modern technology, the JARMs all matched. However, with the recent advancements in artificial intelligence, Meta (previously Facebook) has shifted its gears from the Metavers to artificial intelligence. (Curry, 2022)

The JARM for Meta.com does not match Instagram or Facebook's fingerprints. But it does match Oculus's JARM's fingerprint, which makes sense as Meta acquired Oculus around the time of the Facebook-to-Meta transition.

JARM Fingerprint for Meta.com taken on 12/03/2023:

```
harborhigginbotham@Harbors-MacBook-Pro og % python3 jarm.py meta.com  
Domain: meta.com  
Resolved IP: 157.240.28.18  
JARM: 29d29d0000000000000041d43d00041daf0a91d29a1ea7b1caa01c572cf37a99  
Time: 0.501564108 Seconds
```

Additionally, Meta, a now parent company to Facebook & Instagram, launched another company called Threads in July of 2023 to compete with Twitter, a company within a similar social media space. The JARM fingerprint for Threads.net matches exactly Meta.com & Oculus.com's fingerprints.

JARM Fingerprint for Theads.net taken on 12/03/2023:

```
harborhigginbotham@Harbors-MacBook-Pro og % python3 jarm.py threads.net
Domain: threads.net
Resolved IP: 157.240.28.63
JARM: 29d29d0000000000000041d43d00041daf0a91d29a1ea7b1caa01c572cf37a99
Time: 0.531497223 Seconds
```

Logic would follow that once Meta launches an artificial intelligence company, that too would have the same JARM fingerprint. Additionally, as the tech conglomerate slowly phases out its failed metaverse project (Curry, 2022), it is possible Oculus will no longer have the same Golden TLS configuration replicated across Meta's public-facing systems.

While it is fascinating to showcase the power of JARM fingerprinting by witnessing a massive conglomerate shift its focus and business goals through the lens of TLS configuration replication, we still need to cover our topic's original use case. Which again was to scan the broader internet multiple times in search of new and forgotten servers. But the showcase wasn't entirely off-topic, as we know, JARM fingerprints are subject to configuration changes and contingent on TLS configuration replication (Golden image replication). With the drifts in Fingerprints witnessed due to the golden images changing, any long-lost forgotten servers that no longer get updates would stick out like a sore thumb as they would likely not match any other fingerprints. Additionally, the concept still stands that over time, scanning a targeted range of IPs associated with a company, JARM would still be able to witness new assets being allocated in real-time.

However, "in real time" is where JARMS' cracks in its usefulness begin to show. As we can see in the previous data showcased, the time it took to scan a single IP address was about .5 seconds. By default, the Jarm tool does not have a scan time counter; I added one to the code.

While half a second may seem fast, it is actually incredibly slow. The IPv4 address space consists of four 8-bit octets, meaning the total number of IPs in IPv4 alone would be  $256 \times 256 \times 256 \times 256$ , which comes out to about 4 Billion IPs rounding down. Doing the math would mean scanning the entire IPv4 internet would take approximately 63.41 years.

- 4 billion x .5 seconds a Ip = 2 billion seconds total
- 2 billion total seconds / 60 seconds in a minute = 33333333.3333 minutes
- 33333333.3333 minutes / 60 minutes in an hour = 555555.555556 hours
- 555555.555556 hours / 24 hours in a day = 23148.1481 days
- 23148.1481 days / 365 days in a year = 63.41 years.

This would mean that our malware analyst from our earlier hypothetical, who was JARM fingerprinting the internet for other attacker-owned devices, would be 64 years older (give or take). They haven't even hit IPv6 addresses yet, which is much larger than IPv4 and was created because IPv4 is too small to sustain the ever-growing population of the human race.

### **IMPROVEMENT NEEDED TO MITIGATE FOUND IMPLICATIONS**

Now that we have established that the Salesforce JARM Fingerprinting tool out of the box does not solve any of our cyber defender/bug bounty hypotheticals, let's shift focus to how the tool can be improved to better fit our thesis that JARM can be an effective enumeration tool.

1. As bug bounty programs are restricted to only relevant assets associated with the target, we need a method to get a target's asset lists without leaving the scope of the engagement, essentially narrowing the focus of the scan. A reduction in total assets to scan means our workload is reduced, subsequently affecting our total data collection time.

2. We need a database to track the drifts in the Jarm fingerprints over time, to see if new servers were allocated since the last scan, and to query for assets with identical or similar fingerprints. This is a crucial component for the scalability, which the tool currently lacks. Additionally, a program also needs to be written to output the types of findings we are looking for, essentially automating the process.
3. As the fingerprinting tool is pretty slow, we must reduce our scan time per IP for both use cases.

The easiest way to implement this would be to adapt the current JARM fingerprinting tool written in Python and upgrade it with more Python code to implement the features required to scale this for our use cases. To effectively do this, our requirements can be grouped into two problems: Data Ingesting and fingerprinting/Scanning.

Both issues require optimization of the Salesforce tool to be faster and utilize a backend database. Out of the box, the Salesforce JARM fingerprinting tool allows for scanning a single IP/Domain or a list of IPs/Domains read from a user-provided file. There is also proxy functionality to route all traffic over TOR anonymously, but this requires some prior setup.

## **SOLUTION INTRODUCTION**

Introducing the new JARM tool: JuicyJarms. It was promptly named as our goal is to squeeze JARM fingerprints from a target company or the greater internet. Currently, as of the publication of this paper, the tool itself has yet to be released as it is being adapted for public use. More on that later.



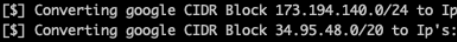
```
Smart Mode Options:
-c COMPANY, --Company COMPANY
    Specify a company name for smart-mode to enumerate
-C COMPANIES, --Companies COMPANIES
    Specify a file containing a list of target companies for smart-mode to enumerate
-t TARGET, --Target TARGET
    Enter a Target to Fingerprint
-T TARGETS, --Targets TARGETS
    Enter a file path containing a list of Target to Fingerprint
```

In addition to the previous features of scanning a single target [-t] or a list of targets [-T], the tool uses the [-c] and [-C] command line flags for the command "Python3 Juicy.py Enum -Smart-mode" to web scrape CIDR blocks (Public Ip address ranges) associated with a company. All the user needs to do is input the company name.

## SOLUTION IMPLEMENTATION

### Screenshot Showing IPv6 Address Range Sizes:

```
harborhigginbotham@Harbors-MacBook-Pro Jarm % python3 Juice.py Enum -s -c google
```



```
By: N0tHarbor
```

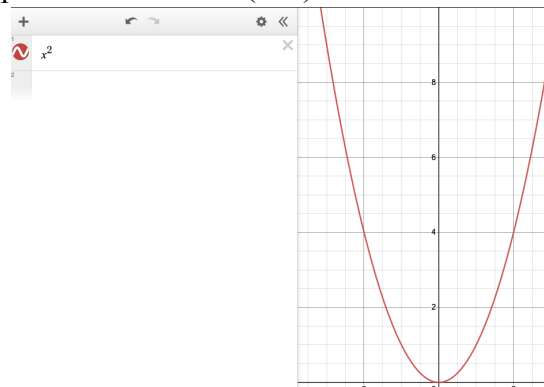
```
[ $\$$ ] Converting google CIDR Block 35.187.80.0/20 to Ip's: 0%| | 0/4096 [00:00<?, ?it/s]
[ $\$$ ] Converting google CIDR Block 34.84.208.0/20 to Ip's: 0%| | 0/4096 [00:00<?, ?it/s]
[ $\$$ ] Converting google CIDR Block 173.194.140.0/24 to Ip's: 0%| | 0/256 [00:00<?, ?it/s]
[ $\$$ ] Converting google CIDR Block 34.95.48.0/20 to Ip's: 0%| | 0/4096 [00:00<?, ?it/s]
[ $\$$ ] Converting google CIDR Block 34.185.0.0/16 to Ip's: 0%| | 0/65536 [00:02<?, ?it/s]
[ $\$$ ] Converting google CIDR Block 136.37.192.0/20 to Ip's: 0%| | 0/4096 [00:00<?, ?it/s]
[ $\$$ ] Converting google CIDR Block 2605:ef80:8095::/48 to Ip's: 0%| | 0/1208925819614629174706176 [00:00<?, ?it/s]
[ $\$$ ] Converting google CIDR Block 2605:ef80:8095::/48 to Ip's: 0%| | 0/1208925819614629174706176 [00:30<, ?it/s]
```

In this verbose output created above, we can see that the tool takes the CIDR blocks (The addresses with the `/[number]` at the end) and converts them to individual IP's that are then written to the database. However, as we can see in the IPv6 CIDR (the last two lines in the screenshot above), trillions of IPs exist in those address ranges. This would significantly reduce the scan time, consume a lot of resources, and is not feasible to scan with a single host. Again, no one wants to wait for 63 years. So, the tools API pull (the web scrape) was adapted to cut out IPv6 from its findings.

For a company that uses IPv6, and most do, this means the tools sacrifice potentially valuable information for the sake of a timely scan. This alone is merit enough to disprove that JARM fingerprinting is fast and reliable, as the tool is just too slow for the vastness that is IPv6, resulting in a lot of missed data.

In addition to IPv6 limitations, there is one fundamental issue that still needs to be addressed, and that is the algorithm. “An algorithm’s performance depends on the number of steps it takes. Computer Scientists have borrowed the term ‘Big-O Notation’ from the world of mathematics to accurately describe an algorithm’s efficiency.” (Koen, 2020) In simple terms, an algorithm is just an implementation of code. In all instances of the JuicyJarm tools algorithms, we use what is called nested for loops or a loop inside of a loop; this implementation has a Big-O of  $O(n^2)$ . This means that our data collection is positively hyperbolic, meaning if our X axis on a graph was our data points (IP addresses scanned or written to a Database) and our Y axis on a graph was time, the larger the dataset, the more our time to process this data will positively approach infinity.

Screenshot of the Desmos Graph Visualiser for  $O(n^2)$ :

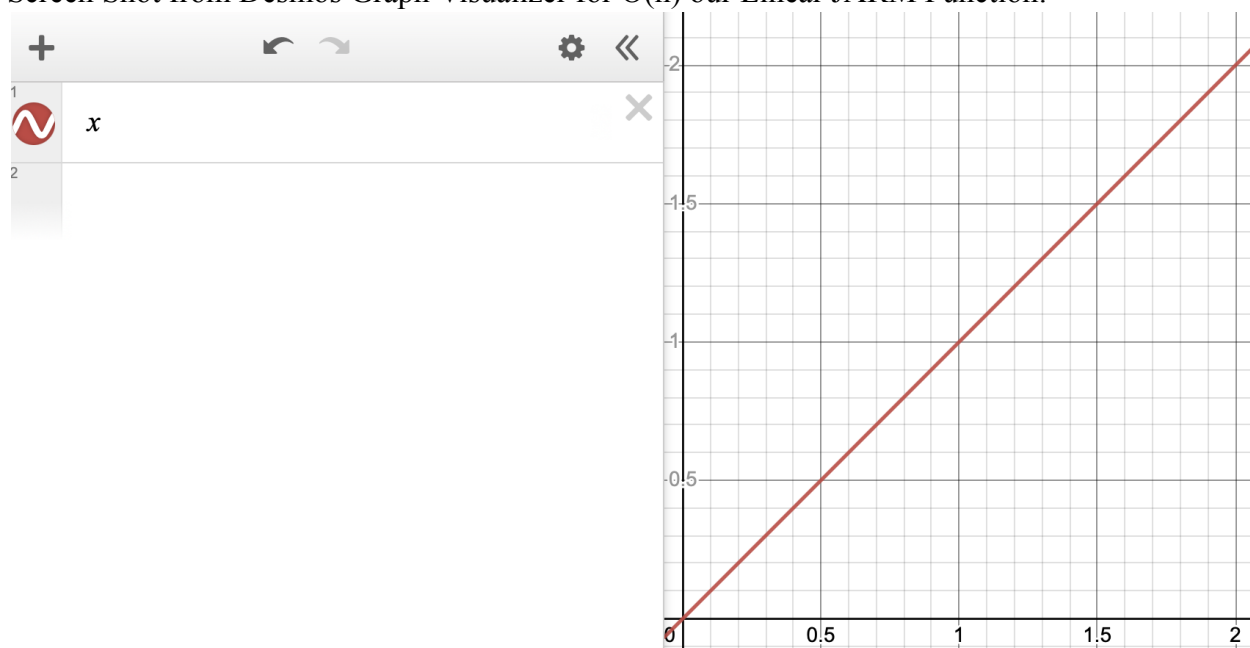


In the instance of our function to pull CIDR addresses, this takes the found CIDR blocks and loops through them one by one, while a second “nested” loop found inside the CIDR block loop takes each individual IP in that CIDR block range and writes them to a Database for later querying: Two loops or  $O(n^2)$ .



Our function to scan for JARM fingerprints uses one loop to loop over the IPs or Domains and another to loop over the possible ports (IE possible locations for a TLS encrypted resource) on the target. For the JARM function, we can make the Big-O's time complexity linear ( $O(n)$ ) by selecting only one port. The port the JuicyJarm tool sets by default is 443, the common port for HTTPS and the port that the Salesforce team used in their original use case of the tool.

Screen Shot from Desmos Graph Visualizer for  $O(n)$  our Linear JARM Function:



With iteration (Ips we scan) on the X axis and time on the Y axis, the algorithm's time complexity can be reduced by decreasing the slope of the now linear function. The decrease in slope can be achieved by threading, a method to run multiple iterations at once by utilizing all resources on a computer's central processing unit (CPU) instead of just one CPU core. Doing this unlocks the full computing power of any computer the tool is running on. By design, Python runs any program on a single CPU core (a fancy term for a section of a computer's logical resources) and locks all programs to a single core for all operations. This throttling is done in Python to reduce any Python program's resource complexity, allowing Python code to run without affecting any other programs a computer might be running, such as a web browser or the operating system

itself. (Andreassen, 2021) However, there are expectations to this, and fortunately, Input/Output (I/O) bound tasks like web requests and writing data to a file/database is one of those exceptions. This means we can do multiple operations at once rather than one at a time at the cost of exhausting more resources. (Andreassen, 2021)

## FINDINGS

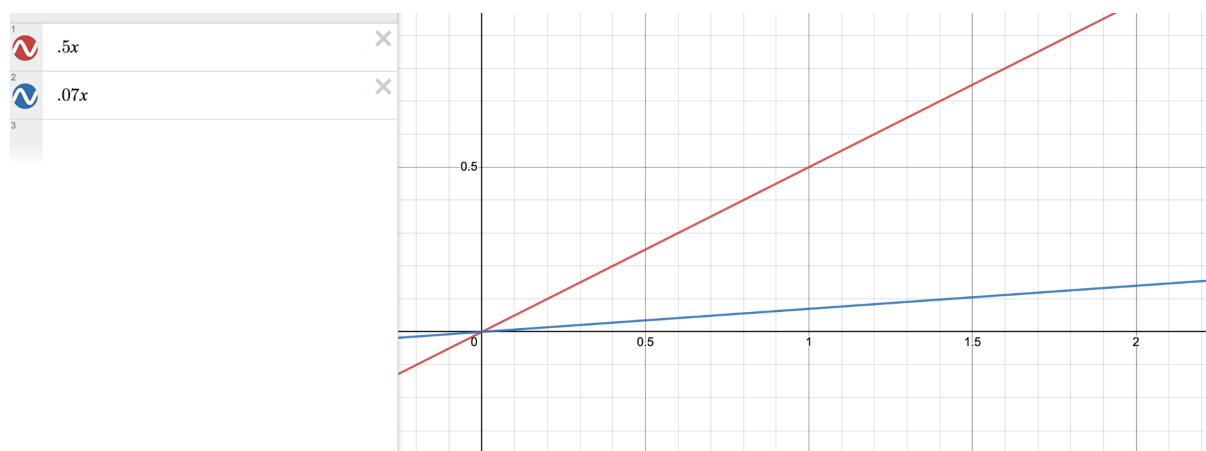
To implement this to our scanner for sending and receiving the 20 packets required to JARM fingerprint: ten total requests from our code and ten complete responses from our target. There is some logistics and code implementation overhead, but the time reduction is worth it. Our previous scan time was half a second; however, with threading implemented into the original code, we see an average nearly 10x reduction in scan time.

Threaded JARM Function Call taken on 12/03/2023:

```
Pre Hash Jarm: c02b|0303|h2|ff01-0001-000b-0010-0017,c02b|0303|h2|ff01-0001-000
||,|||,1301|0303||002b-0033,1303|0303||002b-0033,|||,1301|0303||002b-0033
Time: 0.079 Seconds
Post Hash Jarm: 27d27d27d00000000041d43d00041dcd947229d467ddf1b9b05cf29440ee27
Destination: facebook.com
```

Returning to our original time to scan a single host, we found this to be about half a second. This new scan shows a time of .079 seconds. Many other variables in this scan time can change the outcome of the overall time; one example would be network connection speeds. However, for the most part, the range of time it takes to JARM fingerprint a single target with the newly implemented threading is between .09 seconds and .02 seconds. Below is another screenshot that visualizes the difference between the threaded and non-threaded JARM functions based on the two scan times displayed thus far in this paper

Visualization of the change in slope (Red original, Blue threaded)



## INTERNET SCAN FINDINGS

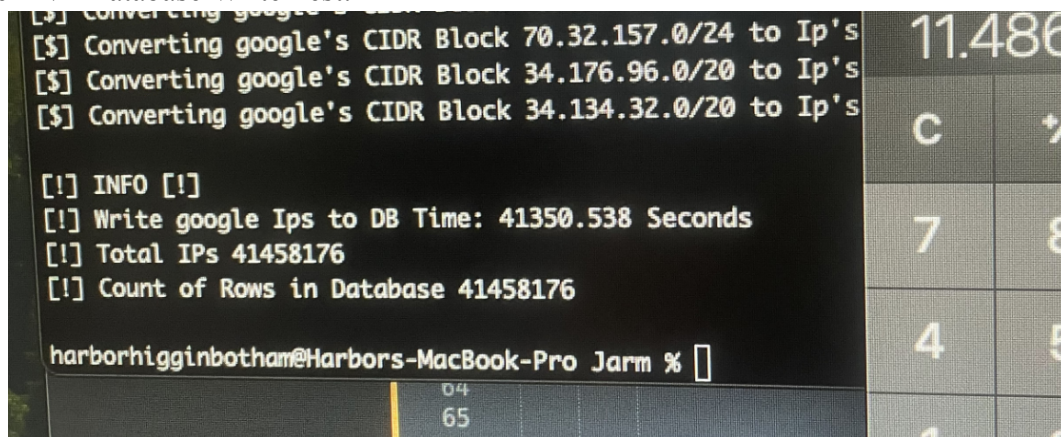
Even at this speed, all of IPv4 is about 4 billion addresses, and when multiplied by .07 seconds, we get a total estimated time of 280000000 Seconds. It would take about 8.87 years to scan IPv4 in its entirety. The last time we calculated the scan time for IPv4, we discovered that it would take about 64 years; this means that our new threaded function implemented into the JucyJarms tool would take about 13% of the total time to scan without threading.

## FINDINGS APPLICABLE TO BUG BOUNTY

Shifting the focus to a smaller target comparatively, Google.com, their IPv4 address space consists of 41,458,176 addresses, which took 11.48 hours to write just the IPs to the database with threading and took up 1.47 gigabytes of space. (It took this long because I forgot to turn off my 2019 Macbook's sleep functionality) With our current scan function of .07 seconds per scan, it would take about 33 days to scan Google's IPv4 address space. Subsequently, it would take about 66 days to run two scans when looking for new assets allocated. For a bug bounty hunter waiting on this information, other bugs would likely be found in that time frame, and a hunter would have made off with a profit before our JARM fingerprinting tool ever finished scanning. This shows that Google is a large company with ample IPv4 address space.

Shifting our focus back to Facebook, we see that Facebook consists of 1,78,432 IP addresses. This equates to about a 3.4 hour scan time of the company's IPv4 address space at .07 seconds per IP. It is manageable and proves that JARM can work on a smaller data set with its current implementation.

Google IPv4 Database Write Test:



```
[!] Converting google's CIDR Block 70.32.157.0/24 to Ip's
[$] Converting google's CIDR Block 34.176.96.0/20 to Ip's
[$] Converting google's CIDR Block 34.134.32.0/20 to Ip's

[!] INFO [!]
[!] Write google Ips to DB Time: 41350.538 Seconds
[!] Total IPs 41458176
[!] Count of Rows in Database 41458176

harborhigginbotham@Harbors-MacBook-Pro Jarm %
```

## DISCUSSION

To recap, the original JARM fingerprinting tool created by the sales force in 2020 is not a cost-effective/lucrative approach to bug hunting because the tool is too slow, nor is it an accurate one as large-medium size businesses, which are typically going to be the companies that offer bounty programs, are too big to obtain actionable information from effectively. Additionally, what was also proven is that the original tool is not an effective proactive cyber defense tool. Despite these findings, the original Jarm fingerprinting blog post by Althouse does state that the Salesforce team was able to scan the entire internet successfully. However, the time frame was not given.

Currently, the adapted tool created for this research is only effective for bug bounty hunting within the IPv4 range on small to medium-sized companies with a relatively small IPv4

surface. Since the adapted tool was designed to ignore IPv6 address ranges, it only provides a portion of the picture regarding recently allocated servers or forgotten servers. However, like most enumeration methods, they only sometimes give all the necessary information. Often, enumeration blends various techniques to get as much information as possible, which means for a small to medium-sized company running a bug bounty program; The adapted tool could be helpful in a hunt for bugs, but only in the IPv4 space, as it was proven that it is fast enough to provide actionable information. Finally, the adapted tool needed to be faster to provide a proactive cyber defense. However, it was still a success as the estimated scan time was reduced from about 64 years to about nine years.

As stated in the original JARM code, the tool was adapted to Python and posted to GitHub publicly; likely, the original tool that Salesforce used to collect their data was written in something much faster, like C++ or Go. On top of this, the Salesforce team likely used the elasticity of cloud computing to significantly reduce the scan time, meaning they probably split the monumental task of scanning the entire internet into thousands or even millions of parts and had thousands or even millions of individual instances of the tool running at once to get reasonable scan times. The benefit of this approach is time reduction, but the price of cloud computing may vary and would eat into any profits in terms of bug bounty hunting.

## CONCLUSIONS

In conclusion, JARM fingerprinting has been overlooked as an enumeration method because it is simply too slow out of the box. The overhead to bring these scan times down is why there is currently no publicly available database of JARM fingerprints and no mass

implementation of it as an enumeration method. Despite these findings, I have roped a group of colleges into attempting to adapt this tool into the cloud to bring the information JARM fingerprinting can provide to the cybersecurity community. This is why the JuicyJarm tool has yet to be released to the public, as the only feasible solution to the current tool implementation issues is to use a mass amount of cloud computing which depending on our finding may result in the monetization of the tool to cover the costs of our compute.

## References

Althouse, J. (2020, November 17). *Easily Identify Malicious Servers on the Internet with JARM*.

*Salesforce Engineering*.

<https://engineering.salesforce.com/easily-identify-malicious-servers-on-the-internet-with-jarm-e095edac525a/>

Althouse, J. (2020, November 17). *Jarm/README.Md*. *GitHub*.

<https://github.com/salesforce/jarm/blob/master/README.md>

Andreassen, T. (2022, August 1). *Visualizing Threading vs Multiprocessing in Python*. *Medium*.

<https://medium.com/@trym.synnevag/threading-vs-multiprocessing-in-python-f309ba50a9e1>

Benjamin, D. (2020, January 1). RFC 8701 *Applying Generate Random Extensions And Sustain*

*Extensibility (GREASE) to TLS Extensibility*. *RFC-Editor*.

<https://www.rfc-editor.org/rfc/rfc8701.html#:~:text=Abstract,peers%20correctly%20handle%20unknown%20values.>

Chaudhary, A. (2023, August 4). *Managing Cloud Misconfigurations Risks*. *Cloud Security*

*Alliance.*

<https://cloudsecurityalliance.org/blog/2023/08/14/managing-cloud-misconfigurations-risks/>

Cloudflare. (n.d.). *What happens in a TLS handshake? | SSL handshake.* Cloudflare.

<https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/#:~:text=The%20%27server%20hello%27%20message%3A,that%27s%20generated%20by%20the%20server.>

Curry, B. (2023, September 22). *Facebook Changes Ticker To META From FB.* Forbes.

<https://www.forbes.com/advisor/investing/facebook-ticker-change-meta-fb/#:~:text=The%20name%20change%2C%20first%20announced,the%20Facebook%20social%20media%20platform.>

H. (2021, July 16). What Are Bug Bounties? How Do They Work? [With Examples].

*HackerOne.*

<https://www.hackerone.com/vulnerability-management/what-are-bug-bounties-how-do-they-work-examples>

Internet Society. (n.d.). TLS Basics. *Internet Society.*

<https://www.internetsociety.org/deploy360/tls/basics/#:~:text=For%20this%20reason%2C%20TLS%20uses,the%20session%20key%20is%20discarded.>



Koen, S. (2020, July 9). *The Big O Notation Algorithmic Complexity Made Simple — This Is*

*NOT An Oxymoron! Towards Data Science.*

<https://towardsdatascience.com/the-big-o-notation-d35d52f38134>

Woollacott, E. (2020, November 24). *JARM fingerprinting tool helps network defenders identify*

*malicious servers, malware C2 infrastructure. The Daily Swig.*

<https://portswigger.net/daily-swig/jarm-fingerprinting-tool-helps-network-defenders-identify-malicious-servers-malware-c2-infrastructure>