Contrasting Crypto and Exfiltration Ransomware with Shamir's Secret Sharing Data Flooding

Daniele D'Ugo
Alma Mater Studiorum
Università di Bologna, Italy
Email: daniele.dugo@studio.unibo.it
ORCID: 0009-0005-2640-0883

Saverio Giallorenzo
Alma Mater Studiorum
Università di Bologna, Italy
INRIA, France
Email: saverio.giallorenzo@unibo.it
ORCID: 0000-0002-3658-6395

Simone Melloni
ARPAE Emilia-Romagna, Italy
Email: smelloni@arpae.it
ORCID: 0000-0002-9535-8747

***Abstract* – Ransomware poses a significant threat to both individuals and organizations, with crypto-ransomware and exfiltration attacks causing widespread damage, financial loss, and operational disruption. Ranflood is a ransomware attack mitigation tool that confuses and overwhelms attackers by flooding the system with decoy files, thereby slowing down the attack and providing a critical window for intervention. In this paper, we extend the coverage provided by Ranflood with a new, advanced flooding strategy based on Shamir's Secret Sharing (SSS) to counteract both crypto- and exfiltration ransomware. Our SSS-based strategy confounds ransomware by generating many shards from each user's file, which we tune for high resilience when contrasting crypto-ransomware (so that the user can regain access to lost data from a few shards) and secrecy against exfiltration (so that the attacker needs many shards to recover the victim's data). We explore the theoretical and practical challenges of applying SSS in this context, present the design and implementation details of our flooder, and empirically evaluate and profile its performance.**

***Index Terms* – Ransomware, Shamir's Secret Sharing, Data Flooding**

I. Introduction

Ransomware [1], [2] is one of the most pressing cyber-security threats worldwide. In its most general definition, ransomware extorts victims through their data.

Crypto-ransomware encrypts the data of the user, who needs to pay a ransom to gain back access. While individuals represent a target for this kind of malware – who could agree with paying the ransom to get back their personal data, like photos and videos – the main targets of crypto-ransomware attacks are enterprises and organisations, which, without access to their data, cannot carry out their business and end up blocking physical processes and organisations.

Another form of ransomware attack is that of *exfiltration* ransomware (which can also go in combination with the crypto kind, usually leading to cases of "double extortion"). In this case, the ransomware sends the data of a user to a storage location under the control of the attacker. The threat regards the divulgation of sensitive information such as private images, intellectual property, financial records, and clients/suppliers data either to the public or to buying competitors.

While technical solutions such as improved backup systems, network segmentation, and advanced threat detection are important mitigation techniques against ransomware attacks, they alone cannot address the full scope of the problem. These challenges demand new, advanced, and diverse software tools that (working in unison) can mitigate the problem, either detecting and contrasting attacks or helping the user in restoring a compromised system.

Our contribution builds upon the defensive technique of *Data Flooding against Ransomware* (DFaR) [3], designed to counteract ransomware attacks. The basic principle of DFaR involves overwhelming attackers (ransomware) by flooding the system under attack with a large volume of decoy files. The goal is to counteract malware attacks by both confounding the valuable files of the user with the decoy ones and also contending IO resources with the attacker to slow down (or potentially crash) its encryption routine, buying crucial time for response measures to kick in – DFaR techniques mainly represent last-line-of-defence solutions that act when the user cannot immediately intervene, e.g., on an unresponsive remote server. The delay provided by DFaR tools offers a valuable time for security teams to recognise the attack and properly respond (e.g., via on-site manual interventions).

Ranflood [4] is an open-source project that implements the DFaR mitigation principles. Ranflood implements three flooding strategies: *Random*, which floods the system with random-content decoy files, and *On-the-fly* and *Shadow*, whose decoy files are copies of existing user's files – the former uses the files available during an attack while the latter uses archival copies. Thanks to Ranflood's companion tool, *FileChecker*, users can restore a flooded system, removing decoy files and recovering lost data through their flooding copies.

a) Contribution: We introduce a new interpretation of the DFaR approach in the form of a flooding strategy based on Shamir's Secret Sharing [5] (SSS).

Briefly, SSS is a security technique for splitting a *secret* into *parts* such that one can reconstruct the secret by combining enough of the parts. In our new copy-based flooding strategy, the secrets are the files of the user, so that Ranflood can use their parts to flood the system under attack. The flooder leverages SSS's ability to parametrise the number of parts needed for reconstructing the secret to contrast both crypto- and exfiltration ransomware attacks. In crypto-ransomware

contrast mode, the flooder generates many parts from the same file to maximise the contention/confounding effect against ransomware. This mode also requires the smallest number of parts to reconstruct the secret, boosting the chances of recovering the latter. In exfiltration contrast mode, the flooder balances between confusion and secrecy by raising the needed parts' threshold, lowering the chances that an attacker who exfiltrated the parts can use them to access the user files.

In Section II, we present preliminary information on Ranflood and introduce the general technique of Shamir's Secret Sharing. In Section III, we discuss the theoretical challenges linked to using SSS as a flooding technique – in particular, we illustrate the security concerns of using SSS against exfiltration, formally quantifying them – and present our flooder's design, intended to mitigate the analysed threats. Then, we move on to the relevant implementation details of the flooder, including both the contrast and restoration phases. Since our solution has a prominent practical component, we develop it as part of the Ranflood open-source ransomware contrast tool [6] and use the implementation to empirically test its usage in Section IV. This evaluation is useful to understand the factors that determine the performance of our new strategy and tune them w.r.t. ransomware attacks. We conclude by positioning our contribution w.r.t. related work in Section V and drawing final remarks and future steps in Section VI.

II. Background

We now present the tools and concepts behind our work: *Ranflood* and *FileChecker* and *Shamir's Secret Sharing*.

A. Ranflood

Ranflood is an open-source drop-in solution built to contrast the action of ransomware. Ranflood implements *Data Flooding against Ransomware* (*DFaR*), an approach whereby anti-ransomware software floods the file system (at targeted locations) with files, mitigating/contrasting the attack of ransomware by slowing down its execution in two ways: it lures malware into reading and encrypting decoy files, and it competes with the attacker for IO access, hindering its actions on storage. At the moment, Ranflood supports three flooding strategies: *Random*, which uses randomly-generated decoys, and *On-the-fly* and *Shadow*, which duplicate a user's files.

While DFaR techniques can cover all phases of ransomware contrast – *detection*, *mitigation*, and *restoration* – in this work, we focus on a) attack mitigation, consisting in the *flooding* itself, and b) post-attack *restoration*, which essentially regards recovering lost files through copies generated during the flooding and removing unnecessary decoy files. For the flooding, the copy-based strategies require preliminary *snapshot*s such as checksums (On-the-fly) of the original files or copies thereof (Shadow) to decide which content to duplicate (e.g., to avoid creating copies of corrupted files which the ransomware would skip). Besides directing copy-based flooding, snapshots support restoration by identifying both corrupted files and duplicates.

Ranflood is implemented in Java, allowing any system supporting the Java Virtual Machine to run it – the project offers native binaries for Linux, Windows, and macOS thanks to the GraalVM compiler.

Architecturally, Ranflood follows the *client-daemon* pattern, where an always-on *daemon* accepts commands from clients. This architecture is highly modular, so that clients can start, stop, and monitor *flooding* sessions and configure them independently. At the heart of the daemon, we find the *engine* and the *task manager*.

The engine implements the core functionalities of *flooding* and *snapshooting* and manages the execution on multiple threads, following the *Proactor* pattern: any operation of writing on (or copying) a file from a *flooder* generates a task, added to the task manager scheduler, which multiplexes their execution on different threads to achieve the maximal degree of concurrency afforded by the host machine.

The *FileChecker* [3] is a companion tool to Ranflood that operates in the restoration phase. Essentially, the FileChecker can generate checksums of the files present in a given location and use checksums to discriminate between pristine and corrupted files. The FileChecker supports restoring the system to its original state, discriminating between safe-to-delete decoy files and the ones useful to restore lost original files.

B. Shamir's Secret Sharing

Shamir's Secret Sharing (SSS) is a technique for dividing some information, the *secret*, into n parts – dubbed *shards* – so that, given a *threshold* $k \leq n$ number of shards, one can recover the secret. SSS enjoys *information-theoretic security*, i.e., an actor who steals fewer than k shards cannot reconstruct the secret, frustrating their effort.

SSS exploits the uniqueness of the Lagrange interpolating polynomial such that, given k (distinct) coordinate pairs $a_1, a_2, ..., a_k$, there is only one polynomial, $f(x)$, of degree $k-1$ passing through them (i.e., having $k-2$ turning points). Moreover, once fixed, one can choose other $n-k$ (distinct) points (with $n \geq k$) on the formed curve, $a_{k+1}, a_{k+2}, ..., a_n$, such that any subset of k out of the n points $a_1, a_2, ..., a_n$ allows one to obtain the polynomial via interpolation.

Algorithmically, given $s \in \mathbb{N}$ secret to be split/encrypted, and $n, k \in \mathbb{N} : k \leq n$, resp. the number of parts and the threshold to reconstruct s, the steps of SSS involve:

1) generating the k coefficients of the polynomial f of degree $k-1$ $(a_0, a_1, ..., a_{k-1})$, setting $a_0 = s$ and randomly choosing a_i, $i \in [1, k-1]$;
2) calculating the n points $p_i = (i, f(i))$, $i \in [1, n]$.

Note that f passes through the *secret* – point $(0, s)$ – since we set the term (a_0) to s. Thanks to interpolation, since f has degree $k-1$, we need k out of the n points to retrieve it.

Technically, SSS uses *finite field*s (like for elliptic curves) because interpolation requires divisions, which can lead to under/overflow errors when operating in $\mathbb{Q}$ or $\mathbb{R}$, and finite fields guarantee *perfect secrecy* – they prevent an attacker from gaining information about s with fewer than k shards.

a) Modular Arithmetic and Finite Fields: A finite field, denoted $\mathbb{F}_{p^r}$, is an algebraic structure where $p, r \in \mathbb{N}$, p prime

and $r > 0$, determine the order (number of elements) $q = p^r$ of the finite field.

For instance, $r = 1$ defines a well-known finite field whereby $\mathbb{F}_p$ is the set of residue classes modulo p and:

- integers $0, 1, ..., p-1$ represent the elements in $\mathbb{F}_p$;
- the field supports modular addition and multiplication, with their respective identity elements 0 and 1;
- each operation has an inverse element for each $x \in \mathbb{F}_p$, i.e., $\exists y, z \in \mathbb{F}_p : x + y = 0 \wedge x \cdot z = 1$; in particular, we can define division a/b in $\mathbb{F}_{p^r}$ as $a \cdot b^{-1}$.

Putting $\mathbb{F}_p$ together with the above algorithm, we can have the elements of $\mathbb{F}_q$ represent each coefficient of the polynomial, and each coordinate as a pair of elements of $\mathbb{F}_q$.

This observation implies that, to use SSS, we need to define – as public parameter – a finite field whose order q is such that $q > s \wedge q > n$, i.e., whereby we can store our secret in integer form ($q > s$) and create enough distinct shards ($q > n$).

III. A Shamir's Secret Sharing Flooding Strategy

We now show how we apply SSS to implement flooding and recovery strategies for Ranflood. First, we present the theory behind our work, explaining the principles we followed to define a useful SSS model for our purposes. Then, we illustrate its implementation.

A. Choosing a Shamir's Secret Sharing Model

To choose a suitable SSS model for our implementation, we need to fix some assumptions behind its usage. The foremost item regards deciding which finite field to use (cf. Section II-B). Intuitively, one can use a set of remainder classes modulo a prime p, choosing p as an upper bound for both the size of the secret (s) and the number of shards one can split it into (n). However, one such solution is quite naïve: it would only work with files of a fixed dimension/number of shards. Moreover, if we consider s to be the integer representation of the sequence of bytes of the file we want to encrypt, we obtain poorer performance the larger the file since there is a direct relationship between a field's order and the complexity of modulo operations [7]. Considering these elements, we follow the technique (discussed below) of fixing an appropriately large q, divide the bytes b that make up a file into $\lceil b/q \rceil$ pieces, and obtain the shards as aggregates of separate q-sized splits.

Far from being purely theoretical speculations, we find the effect of these observations in existing and widely used open-source implementations of SSS, which impose limitations on the secret size/number of shards given a fixed q of choice.[1] On the contrary, projects like Vault by HashiCorp and codahale/shamir implement arbitrary-length inputs efficiently by using the finite field $\mathbb{F}_{2^8}$.[2]

Technically, $\mathbb{F}_{2^8}$ (also known as Galois Field 256 or $GF(256)$, from the mathematician who first introduced the concept of finite fields), is a finite field of order 256 that allows one to efficiently implement the common algebraic operations over the relatively small number of values of the field:

- unary operations of logarithm base 2 and exponentiation of 2 have 256 possible outputs, which can be pre-computed and stored in *lookup tables* – the lookup table for exponentiation has size $2 \cdot 256$ to also avoid the modulo operation in case of overflow, e.g., we can efficiently calculate $log_2(a+b)$, with $a, b \in [0, 256)$, even if $a + b \geq 256$;
- *xor* implements addition and subtraction (explained later);
- multiplication takes advantage of the mentioned lookup tables, such that $a \cdot b = 2^{log_2(ab)} = 2^{log_2(a)+log_2(b)}$;
- division derives from applying multiplication by an inverse $b^{-1} = 2^{log_2(b^{-1})} = 2^{255-log_2(b)}$.

The properties of $\mathbb{F}_{2^8}$ make it such a good candidate that it is also used in the internal operations of *AES* [8], which efficiently works on single bytes thanks to $\mathbb{F}_{2^8}$.

Visualising $\mathbb{F}_{2^8}$ is harder than $\mathbb{F}_p$ (which we can easily represent as $0, 1, \cdots, p-1$), however, we can establish an isomorphism between $\mathbb{F}_{2^8}$ and $\mathbb{F}_2[x]/p(x)$, i.e., the quotient of the ring of polynomials on $\mathbb{F}_2$ ($\sum_{i \geq 0} a_i x^i$, $a_i \in \{0, 1\}$) over an irreducible polynomial, chosen of degree 8. In this way, we represent an element in $\mathbb{F}_{2^8}$ as an element of the set of remainders of $\mathbb{F}_2[x]/p(x)$ – each element is an equivalence class of polynomials with the same remainder when divided by $p(x)$. Since $p(x)$ has degree 8, there are 2^8 such equivalence classes, and we can visualise an element of $\mathbb{F}_{2^8}$ as a polynomial of degree 8 with coefficients in $\mathbb{F}_2$, i.e., $\sum_{i=0}^{7} a_i x^i$.

Hence, we can see an element of $\mathbb{F}_{2^8}$ as a vector of coefficients, just like a byte is a vector of bits where a_0 is the least significant bit. In this way, addition and subtraction work on polynomials with the supplementary constraint that we must take the remainder modulo 2, which allows us to execute them as bit-wise xor operations – xor and modulo addition directly correspond, while $a - b \equiv a + b (mod\ 2)$, for a and b in $\{0, 1\}$.

The choice of $p(x)$ is an implementation detail, and it is orthogonal to the structure of the finite field – thus, we call all fields of this order F_{2^8}. The only requirement for $p(x)$ is that it must be irreducible (i.e., we cannot express it as the product of two non-constant polynomials with coefficients in $\mathbb{F}_2$) and the algebraic structure is a field (with element-wise inverses).

The last ingredient we need to fix is a generator g, i.e., an element of the field whose powers generate all non-zero elements of the field – for $\mathbb{F}_{2^8}$, it means that $g^1, \cdots, g^{255}$ generates all 255 non-zero elements of the field – so that we can populate the lookup tables. Similarly to $p(x)$, the choice of g is arbitrary.

Following AES [8], we choose to use the polynomial $0x11b$ ($x^8 + x^4 + x^3 + x + 1$) and the generator $0x03$ ($x + 1$).

1) Security Concerns: Since we foresee the usage of the SSS flooder for contrasting both crypto-ransomware and exfiltration ones, we analyse the consequences of using SSS for generating many shards as a flooding technique. In particular, we focus on secrecy against exfiltration, i.e., the relationship between

[1] For example, highly-starred GitHub projects like https://github.com/shea256/secret-sharing, https://github.com/dsprenkels/sss, and https://github.com/timtiemens/secretshare limit secrets to a fixed number of bytes.

[2] Resp. at https://github.com/hashicorp/vault and https://github.com/codahale/shamir.

performance, threshold levels (higher thresholds imply more shards an attacker needs to recover the secret), and duplicate files (which threaten to lower the technique's security level).

Finite fields allow SSS to achieve perfect secrecy, i.e., one cannot obtain information on the secret with fewer than k shards – technically, given k points there is only one function (the Lagrange interpolating polynomial) of degree $k-1$ that passes through them. Even having $k-1$ points, the attacker does not obtain any information on the k-th one, leaving them the only option of trying to guess it – obtaining for each k-th value a different interpolating polynomial, making it theoretically impossible to guess the secret.

However, using the same SSS model repeatedly can weaken its security guarantees. Indeed, if we have two secrets s_1 and s_2 corresponding to the same value (e.g., in our case, different files with the same content) and we split them using the same polynomial f, their shards would all lie on f – they could even be the same. This fact implies that an attacker has more possible shards, e.g., $2n$ considering the same n for both secrets, they could use to retrieve the secret value ($s1 = s2$), i.e., the union of $s1$'s and $s2$'s shards.

To avoid this problem, one can randomly change the polynomial for each secret, which raises the question: *how likely is it to obtain the same curve multiple times*? The question is critical because collisions – i.e., reusing the same polynomial – could inadvertently leak information, especially when the same content appears across different files.

To quantify this likelihood, we analyse the probability of generating the same polynomial twice over the finite field $\mathbb{F}_{2^8}$. Formally, we look at the probability of picking a polynomial f of degree $k-1$, equivalent to picking a set of k coefficients $a_0, \cdots, a_{k-1} \in \mathbb{F}_{2^8}$. Thanks to the Lagrange interpolating polynomial, we can calculate f by picking k coordinates $c_1, \cdots, c_k$ which, following Section II-B, have the form $c_i = (x_i, y_i)$, with y_i random element of $\mathbb{F}_{2^8}$; corresponding to picking a specific sequence of k elements of $\mathbb{F}_{2^8}$.

Assuming a uniform distribution of the random number generator used to obtain the elements, we can pick any byte (i.e., an element of $\mathbb{F}_{2^8}$) with probability $1/256$. Then, the probability of picking a specific polynomial f at random is $P(f) = (1/256)^k$. In general, this formula estimates how hard guessing f is by considering a part of the k elements fixed, i.e., given k' shards in $f_{k'}$ with $0 < k' \leq k$ we have $P(f_{k'}) = (1/256)^{k-k'}$. Since $P(f_{k'})$ grows exponentially relatively to k' (we consider k fixed since the user sets its value), one can largely reduce the probability by each unitary decrease of k'.

Summarising, even partial knowledge of k' coefficients (or, equivalently, k' shards) dramatically reduces the uncertainty of guessing f, which undermines the secrecy guarantees against exfiltration attacks provided by SSS-based flooding. One can increase k to reduce the chance of collisions, but increasing k translates into performance costs. Thus, we need a practical way to estimate how large k shall be to obtain good performance while keeping collision probabilities acceptably low.

A foundational insight for estimating the possibility of collisions comes from the birthday problem [9]. Brink generalised the birthday problem [10] considering a random variable $n(d)$ uniformly distributed over d values and calculated the number of extractions one needed to obtain the same value twice with a probability of at least 50% as

$$n(d) = \left\lceil \sqrt{2d \ln 2} + \frac{3 - 2\ln 2}{6} + \frac{9 - 4\ln 2^2}{72\sqrt{2d \ln 2}} - \frac{2\ln 2^2}{135d} \right\rceil$$

for all $d \leq 10^{18}$ (Brink conjectured holding for all $d \in \mathbb{N}$). Applying the formula to $d = 256^k$ gives us useful approximations for choosing appropriate k values for f, reported in Table I.

TABLE I
TABLE RELATING DIFFERENT k VALUES WITH THE NUMBER OF BYTES (AND MB) NECESSARY TO HAVE AT LEAST A 50% PROBABILITY OF COLLISION (SAME POLYNOMIAL) – ASSUMING A UNIFORM DISTRIBUTION $d(256^k)$ FOR A BYTE AND $d(256^k)/10^6$ FOR A MB.

k	$d(256^k)$ (B)	$d(256^k)/10^6$ (MB)
7	3.16×10^8	3.16×10^2
9	8.09×10^{10}	8.09×10^4
11	2.07×10^{13}	2.07×10^7

The results show that, with $k = 7$, after splitting 316 MB of data we have a 50% collision probability, which becomes more and more likely the more files the flooding splits. Using $k = 9$ raises the required threshold to 80.911 GB and $k = 11$ brings it to 20.7 TB.

We weigh in on these statistical results concerning the configuration of the flooder in Section III-B. Here, we draw some high-level remarks regarding the usage of SSS in the context of exfiltration-based attacks, since crypto-ransomware would in any case need to encrypt the shards to make their content inaccessible, irrespective of the strength of their secrecy guarantees. Contrarily, under exfiltration attacks we want strong secrecy guarantees of SSS so that the attacker needs as many shards as possible to reconstruct the original file, striving to minimise the data gathered from the victim. Since Ranflood's strategy relies on confounding legit user files with a deluge of decoy ones and contending resources with the ransomware, we need to balance fast shard-production performance with secrecy guarantees. To practically estimate this trade-off, we benchmark the performance of the implementation in Section IV under different polynomial configurations.

In general, we see interesting future work in investigating the feasibility of these attacks to SSS, e.g., assuming that an attacker retrieved different sets of shards, estimate how hard it is to detect they are related and use them to perform a same-polynomial kind of attack (on the same bytes).

B. Implementing an SSS Flooding Strategy

The core elements of a suitable SSS flooder implementation are: a) a model of F_{2^8} (cf. Section III-A), b) efficiency of operations (e.g., using lookup tables and xor-based arithmetic) on the model's elements, and c) functionalities for splitting/merging a file into/from a set of shards.

Since the open-source Java library codahale/shamir[2] correctly and efficiently provides these functionalities, we use it in

TABLE II
SCHEMA OF THE SHARD FILE FORMAT.

content	size (bytes)
fixed header signature ($0x123456789ABCDEF0$)	8
n	4
k	4
generation	8
original file checksum	20
the shard content checksum	20
length of the original file path	4
original file path	variable
the shard content	variable

our implementation. For completeness, we briefly discuss the (simplified) logic of its `split` and `merge` functionalities.

`split` takes as input a sequence of bytes (the secret) and implements the byte-wise encryption schema from Section III-A. For each byte b, `split` generates a random polynomial f of degree $k-1$ that passes through the point $(0, b)$, so that any k distinct points on it allow one to perform the Lagrange interpolation and obtain back b. Then, it generates the n shards for that byte by applying $f(x)$ for $1 \leq x \leq n$. Finally, it assembles the obtained byte-wise shards into n byte sequences so that the first sequence contains all the bytes for $x = 1$, the second for $x = 2$, until the n-th for $x = n$.

Complementarily, `join` takes a set σ, $|\sigma| \geq k$, of sequences of shard-bytes (from the related shard files) produced by `split` and proceeds to byte-wise reconstruct the original file via the Lagrange interpolation of the bytes at the i^{th} position of the sequences, to obtain the polynomial and find the secret's i^{th} byte at position 0.

Since the flooding generates new (shard) files, we define an ad-hoc file format useful to store the shards' metadata and support restoration. We schematise the file format in Table II and discuss it below. While we treat this metadata as clear text, one can embed it in shards using standard encryption to further complicate pursuing reconstruction attacks.

The first 8 bytes of the file encode the format's signature. Then, we find the SSS parameters n and k, followed by an 8-byte number called "generation" which carries information on the processing of the original file; in a nutshell, we use the generation metadata to distinguish between shards of the same content but belonging to different split sessions. The mechanism of generations allows, in the restoration phase, to correctly associate shards not only pertaining to the same file but also having been generated with the same polynomial – only assembling the shards pertaining to the same generation allow the correct identification of the original secret. Next, we have the original file checksum (used to check its status during restoration, if reachable), the shard checksum (to verify the shard content), the original file path (for retrieving the file) – and the path length, since it is variable – and the shard content (as generated with the `split` routine).

C. Overview of the Implementation

From an implementation standpoint, the intervention to integrate an SSS-based strategy in Ranflood requires the implementation of a dedicated *flooder*. However, following the implementation of the On-the-fly flooder [3], we integrate the usage of a *snapshooter*, which runs before the flooding phase (e.g., when users create new files or save their work) to create a list of the valid file checksums. During flooding, the list allows the flooder to copy only pristine files and avoid duplicating corrupted ones. We reuse the existing On-the-fly snapshooter since it meets our needs.

Algorithm 1 SSS Data Flooding

```
input: n, k, targetPath, snapshot, exfil
while keepFlooding do
    for filePath in GETFILES(targetPath, snapshot) do
        fileBytes ← READFILE(filePath)
        shards ← SPLITCONTENT(n, k, filePath, fileBytes)
        for shardBytes in shards do
            WRITEFILE(shardBytes)
        if exfil then
            DELETEFILE(filePath)
```

Attack victims can use the FileChecker to restore their files after an attack. Notably, since we save the path and checksum of the valid files in the shards (cf. Table II), we increase the resiliency of the FileChecker restoration routine by complementing the checksum list of valid files with the metadata found in the shards – which constitute a sort of distributed version of the list of valid files.

We present the main aspects of the implementation of the SSS flooder in Section III-D. Since one can use shards to restore a user's files, we describe in Section III-E how we extend the FileChecker to support this functionality.

D. SSS in Ranflood: Mitigation

The design of the SSS Ranflood flooder follows several traits of the On-the-fly one (including its snapshooter).

We start discussing the pseudocode in Algorithm 1, which reports a simplified, sequential form of the logic of the SSS flooder – the actual implementation launches tasks run in parallel and coordinated by Runflood's proactor. The flooder requires the provision of a snapshot to discriminate between corrupted and valid files.

In Algorithm 1, the flooder takes as inputs the number of shards per file to produce n, the threshold of shards necessary to recover the secret's content k (restricted within the allowed limits, i.e., $2 \leq k \leq n \leq 255$), the flooding's $targetPath$, and the $snapshot$ – we discuss the $exfil$ parameter later. Then, for all valid files in the $targetPath$ (including the ones found in subpaths) the flooder reads the content ($fileBytes$) of the file and generates the shards, calling the SPLITCONTENT procedure. That procedure generates the content of each shard file in the expected format (cf. Table II), managing the mechanism of generations to discriminate among shards of the same file created at different times and storing the metadata (file checksum and path, shard checksum) along with the binary content of the shard generated following the logic in Sections II-B, III-A and III-B (using the `split` function). For each generated shard, the flooder writes its content on disk.

After having generated the shards, and before passing to the next file, if the $exfil$ parameter is set to true – i.e., the user called the flooder as a countermeasure against an exfiltration attack – the flooder deletes the original file, which the user can later rebuild in the restoration phase. Note that enabling the deletion of files in Algorithm 1 would prevent further flooding from happening once the files in the $targetPath$ have terminated – the while loop surrounding the flooder's logic. In the actual implementation, losing user files does not prevent the flooder from generating new shards. Indeed, the flooder saves the valid copies of the user's files in memory and cyclically processes them as if read from disk.

a) A Note on Caching Files and Shards: The implementation of Ranflood's On-the-fly strategy [3] caches user files after their first read, to reduce both I/O read overheads and the risk of data loss due to ransomware encryption. The same reasons justify refining SSS's flooding implementation by caching user files after their first read. Following this line of reasoning, one could consider also caching the shards themselves to skip the split phase – which has high computational complexity, cf. Section IV-A – and write them on disk multiple times. However, this optimisation would be both impractical and detrimental to the effectiveness of contrasting exfiltration attacks. Memory-wise, if caching a file takes space equal to its disk usage, caching n shards of that file would take n times that amount. Security-wise, when considering exfiltration attacks, one shall avoid reusing (cached) shards because copies of the same shard would reduce the secrecy of shard encryption (cf. Section III-A1).

E. SSS in FileChecker: Restoration

We implement the restoration phase by extending the FileChecker to perform the rebuilding of the file shards.

The shards generated by the SSS flooder include the checksum of their original file (cf. Table I). Hence, our extension gathers the shards that belong to the same file (e.g., checksum-wise), checks their validity, and uses them to retrieve the original content and restore it – if the original file is missing and the reconstructed content corresponds to the original's checksum. The procedure uses, in particular, the `join` function described in Section III-B to perform the byte-wise reconstruction of the content from the shards (only if there are at least k valid gathered shards).

IV. Evaluation

Since the performance of our flooder depends on different parameters (n, k, etc.) we start our evaluation, in Section IV-A, by analysing how these parameters influence the complexity of the tasks handled by the flooder. This analysis allows us to define informed values for these parameters, which we benchmark in Section IV-B.

A. Computational Complexity Analysis

To analyse the complexity of our flooder's logic, we abstract away from IO access (both the ransomware and flooder access the files) and focus on the parts related to the SSS technique.

Algorithm 2 Polynomial generation

```
function GENERATE(k, secret)
    f ← byte[k + 1]    ▷ k+1-array to get a k-degree polynomial
    repeat
        populateWithRandomBytes(f)
    until f[k] ≠ 0      ▷ the coefficient of x^k must be non-zero
    f[0] ← secret
    return f
```

The `split` functionality (cf. Section III-B) features a loop on each byte of the secret, where we call a procedure that GENERATEs a random polynomial of degree $k-1$ (reported in Algorithm 2), and an inner loop on n to retrieve the bytes of each shard by applying Horner's method [11] to evaluate the polynomial at the given position.

In Algorithm 2, we obtain the polynomial of degree k by populating a $k+1$-size array of bytes that stores the coefficients of the terms. We generate the coefficients using the function $populateWithRandomBytes$, which fills all the elements of the array with random bytes. Since we want to obtain a k-degree polynomial, we re-apply the function (within the **repeat-until** loop) as long as the coefficient of the leading term is non-zero, i.e., we make sure we have a polynomial of degree k. Finally, we set the constant coefficient of the polynomial to $secret$ – the value of the polynomial at the intersection with the y-axis as per SSS. Using mainstream implementations[3] the complexity of the $populateWithRandomBytes$ function is linear in the size of array f (where the complexity of random byte generation is constant time) and we essentially run one time the **repeat-until** loop because, given that the function uses a uniform random distribution, we have $1/256$ possibilities that the byte of the leading term is zero. Hence, Algorithm 2 has complexity $O(k)$.

Applying Horner's method on $\mathbb{F}_{2^8}$ makes the complexity of the calculation linear in the degree of the polynomial, since additions and multiplications are constant-time, hence, the complexity of evaluating the polynomial on n values is $O(nk)$.

Then, given b number of bytes of the secret, the complexity of the `split` procedure is $O(b(k+kn)) \sim O(bkn)$, linearly depending on the file size, the threshold, and the number of shards ($2 \leq k \leq n$). Since the size of files is an external parameter which we do not control, we focus our analysis on k and n. As presented in Section IV-B, we empirically test different combinations of k and n that guarantee good performance while keeping the product as small as possible.

B. Performance of the Split Routine

We put into practice what we learned through computational analysis and investigate these results through experiments.

We run our benchmarks on an AMD Ryzen 7 1700 (3GHz) 8-core, 16-thread CPU, 16GB DDR4 3200MHz RAM, NVMe M.2 (writing speed of 5000MB/s), Debian 12 64bit, GNU/Linux 6.6.13, and OpenJDK 21.0.3 (3GB of maximum heap).

[3]https://docs.oracle.com/javase/8/docs/api/java/util/Random.html#nextBytes-byte:A-

n	k	split (ms/KB)	n	k	split (ms/KB)
255	2	1.666	50	2	0.334
255	10	8.551	100	2	0.57
255	50	61.171	200	2	1.069
255	100	124.589	200	3	1.65
255	255	328.976	250	10	8.074
2	2	0.119	50	50	12.098
10	2	0.164	10	10	0.547

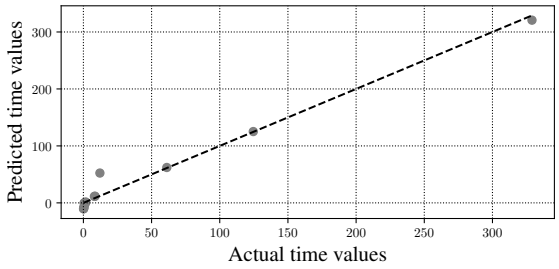

Fig. 1. Recorded `split` times (ms/KB) with varying n and k (top), linear regression plot (bottom).

We report in Fig. 1 the speed of the `split` procedure (in ms/KB) at varying (valid) combinations of n and k, on inputs counting 1000 bytes, averaged over 1000 tests each.

We shed further light on the relationship between n and k in determining `split`'s performance by conducting a linear regression analysis on the data from Fig. 1, reported at its bottom. Visually, the alignment fits most data points, confirmed by an R^2 value of 0.981. The resulting regression coefficients are 0.048 for n and 1.262 k, indicating that the latter exerts the strongest influence on the speed of `split`.

Intuitively, we conjecture that k has a stronger impact than n due to operations in the GENERATE procedure (cf. Algorithm 2) where we allocate a k-size byte array for the polynomial and perform on it the related operations for its population.

Summarising, one shall avoid increasing both n and k, since similar values impact quadratically on performance, in particular, k has the strongest impact. Looking at the application scenarios, when contrasting crypto-ransomware, we can afford low k values and, thus, relatively large n ones; in exfiltration attack scenarios, we want a high threshold for recovering the original file content, and we need to carefully balance k and n to trade speed off of secrecy strength. Quantitatively, using small n values leads to detrimental performance losses, presumably due to some overhead (e.g., $n = 10$ and $k = 2$ requires 0.05 ms/KB more than $n = 2$ and $k = 2$). In practice, $n \geq 10$ and $k \leq 10$ seem a good compromise between secrecy strength and performance.

C. Performance of the SSS Flooder

We now empirically evaluate the performance of the SSS flooder, looking at the speed it affords when creating the shards. In the previous section, we focussed on the frequency of completing a single splitting. Here, we focus on the average number of bytes (the shards) written, no matter the frequency, equally rewarding larger parameters values which require longer split execution times, but, at the same time, produce more files. Note that, differently from Fig. 1, we now consider MB/s (equivalent to KB/ms) instead of ms/KB, to highlight byte production rather than single-execution duration.

We report the results of our experiments in Fig. 2. Each plot shows on the x-axis varying values of either n (two bottommost) or k (two topmost) for a resp. k or n fixed, while we have on the y-axis the splitting speed. The plots include the sample points with their relative split speed in MB/s (i.e., the reciprocal of the data in Fig. 1).

Looking at the plots, when we fix n, we obtain a stronger speed reduction the higher the k. When we fix k, we notice a hyperbolic-like trend where higher values of n only slightly increase (if not slightly decrease) the splitting speed. We explain this behaviour from the observations made in Section IV-B, i.e., that the split speed (ms/KB) is a quadratic function (having complexity $O(nk)$) and since in benchmarking the flooder we track the reciprocal of the split speed multiplied by n (the number of files we produce), we observe a behaviour that approximates a hyperbolic function.

In general, we notice that, with fixed k, increasing n results in more bytes written over time – although we find a "peak" at 200 for $k = 2$ and 100 for $k = 10$. A possible explanation is the aforementioned overhead, which is balanced out by specific n values (possibly also due to hardware idiosyncrasies).

We can use these results to tune the SSS flooder for peak performance. In particular, we recall that n is mainly related to the flooding (how many shards we generate from a file), with k/n indicating the redundancy of a file's shards. Hence, against crypto-ransomware, low k values increase redundancy and restoration chances while, against exfiltration, one must minimise the attacker's ability to reassemble the victim's secrets using high k values.

From the data, we consider 200 a good value for n, picking $k = 2$, when contrasting crypto-ransomware. When contrasting exfiltration, we find $n = 150$ as peak performance for the fairly large value of $k = 10$. However, one can obtain better performance with good secrecy levels by ranging k between 5 and 8 (cf. Fig. 2). Given these results, we set the default parameters for contrasting crypto-ransomware to $n = 200$ and $k = 2$, reaching a splitting speed of 187.1 MB/s, whilst we set the parameters for contrasting exfiltration to $n = 150$ and $k = 6$, reaching a splitting speed of 53.76 MB/s.

V. RELATED WORK

We position our contribution within the existing literature on ransomware mitigation and Shamir's Secret Sharing (SSS) applications. Ali et al. [12] present the most closely related work through their Decentralised Ransomware Recovery Network (DRRN), which distributes encrypted sensitive files across networked edge nodes using SSS principles to enable data recovery without paying ransoms. While both approaches employ SSS to counter crypto- and exfiltration ransomware attacks through data shards, our proposal fundamentally offers a dynamic, active defence system integrated within Ranflood that generates data shards during an attack, contrasting with DRRN's requirement for pre-attack deployment as a static storage solution. Beyond

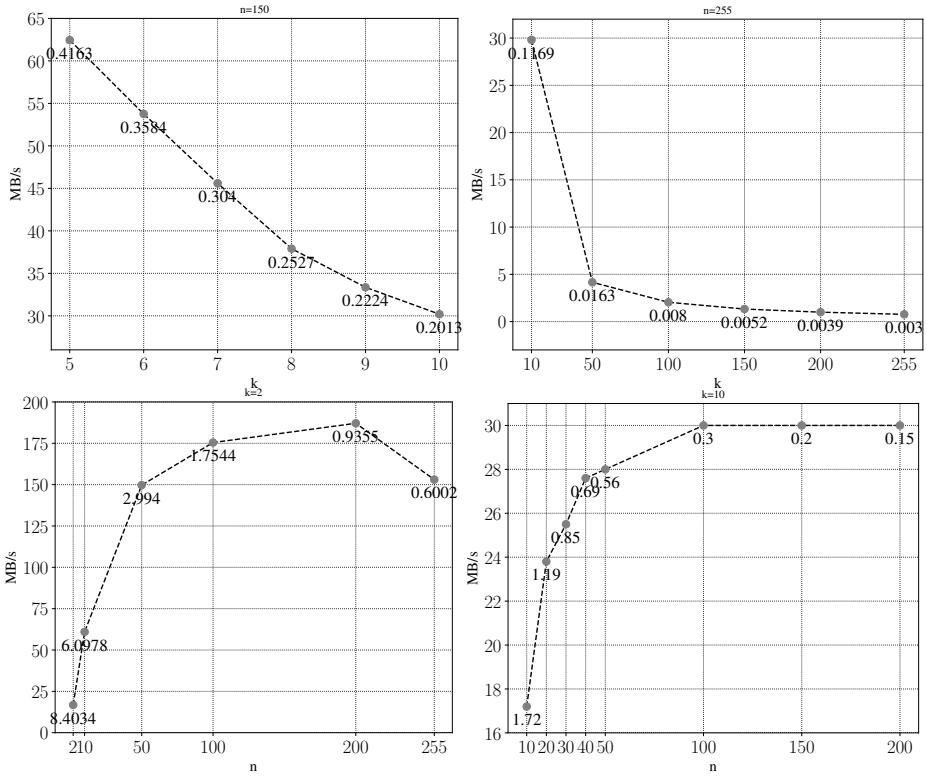

Fig. 2. Splitting speed of the SSS flooder. From left to right, varying k with $n = 150$, varying k with $n = 255$, varying n with fixed $k = 2$, and varying n with fixed $k = 10$.

ransomware mitigation, researchers have applied SSS to various security challenges. Cristòbal et al. [13] deployed SSS within peer-to-peer systems to combat pollution and free-riding attacks, wherein trusted nodes distribute secret key shards that malicious peers struggle to collect to reconstruct shared content. Goubin and Martinelli [14], alongside Coron et al. [15], use SSS as a countermeasure against side-channel attacks on AES by partitioning sensitive variables and performing cryptographic operations on the resulting shards, thereby preventing attackers from extracting information without observing multiple intermediate values. Lipton et al. [16] enhance virus detection schemes by concealing secret keys within memory as shards, enabling systems to detect malware injection through failed key reconstruction attempts. Examining Ranflood's positioning amongst general ransomware mitigation solutions, we observe distinct advantages over alternatives such as ShieldFS [17], R-Locker [18], Lee et al.'s tool [19], and Microsoft's Controlled Folder Access [20]. ShieldFS integrates a specialised file system with ransomware detection capabilities that copies critical data to secure locations upon attack detection, whereas Ranflood achieves similar restoration capabilities through copy-based strategies (enhanced by our SSS-based flooding approach) with minimal plug-in setup. R-Locker employs static honeypot files for detection and mitigation while Ranflood deploys decoy files dynamically. Lee et al.'s moving-target defence strategy deceives ransomware by altering file types and extensions; similarly Ranflood incorporates a moving-target strategy by confounding the user's files with dynamically created decoys – alongside a resource contention mechanism. Microsoft's solution restricts ransomware through location-specific permissions but can leave unchecked areas vulnerable.

VI. Discussion and Conclusion

Shamir, in the seminal paper on his secret sharing technique [5] notes "[i]n other applications the trade-off is not between secrecy and reliability, but between safety and convenience of use". We see an application of Shamir's observation in our work by noticing that it is not about the "secrecy" of hiding one's secret nor about the contrast between reliability and secrecy; it is rather about the trade-off (mathematically represented by the two parameters n and k) that allows one to find the right compromise between ease of access (but also of attack) and secrecy (and risk of losing access).

More practically, Shamir's consideration applies to our flooding tool by presenting a compromise between making a file easily recoverable (low k) and hardly attackable (high k), requiring a balancing between redundancy and difficulty of retrieval: the former is better suited for contrasting crypto-ransomware, making it possible to restore a file even if one lost many of its shards; the latter fits the exfiltration case, where we do not want the attacker to obtain the user's data.

Looking at future work, a practical aspect we abstracted away in this proposal regards the names and formats of the generated shard. Indeed, ransomware usually follow some criteria [2] to

decide which files to encrypt, e.g., depending on their location (user's folders) and extension (documents, pictures). Existing Ranflood strategies already implement these factors [3], e.g., producing random files that resemble valid documents, pictures, and audio files. By including in the shard generation such patterns, we could increase the likelihood of luring/swaying crypto-/exfiltration ransomware away from the user's files.

Another future direction regards exploiting higher powers of 2 as order $q = p^r$, instead of $r = 8$. Values multiple of 8 (i.e., a byte) could grant performance boosts, better exploiting the 64-bit operations of today's hardware by choosing fields of a higher order. Besides hardware optimisation, higher field orders would reduce the raw number of operations performed (e.g., $r = 16$ implies dividing the original files into pairs of bytes instead of individual ones, performing half the operations with the finite field, possibly halving execution times). However, this solution would require more memory (e.g., to populate the look-up tables for the logarithm) – e.g., using $r = 32$ (32-bit integer) would require 17 GB.

A third interesting path is that of focussing on contrasting crypto-ransomware with techniques adjacent to SSS: erasure codes [21] – particularly Reed-Solomon ones [22]. Using these techniques, we would trade secrecy off of both computational efficiency and higher recovery rates. In this case, the shards could further help in fending off partial-encryption ransomware attacks such as the LockBit family [23], by increasing the resiliency of files against these types of attacks.

ACKNOWLEDGEMENT

Research partly supported by project PNRR CN HPC - SPOKE 9 - Innovation Grant LEONARDO - TASI - RTMER funded by the NextGenerationEU European initiative through the MUR, Italy (CUP: J33C22001170001). We thank Matteo Cicognani for supporting the collaboration between ARPAE and Università di Bologna.

REFERENCES

[1] R. Richardson and M. North, "Ransomware: Evolution, mitigation and prevention," *International Management Review*, vol. 13, no. 1, pp. 10–21,101, 2017. [Online]. Available: https://www.proquest.com/scholarly-journals/ransomware-evolution-mitigation-prevention/docview/1881414570/se-2

[2] H. Oz, A. Aris, A. Levi, and A. S. Uluagac, "A survey on ransomware: Evolution, taxonomy, and defense solutions," *ACM Comput. Surv.*, vol. 54, no. 11s, pp. 238:1–238:37, 2022. [Online]. Available: https://doi.org/10.1145/3514229

[3] D. Berardi, S. Giallorenzo, A. Melis, S. Melloni, L. Onori, and M. Prandini, "Data flooding against ransomware: Concepts and implementations," *Computers & Security*, p. 103295, 2023.

[4] D. Berardi, S. Giallorenzo, A. Melis, S. Melloni, and M. Prandini, "Ranflood: A mitigation tool based on the principles of data flooding against ransomware," *SoftwareX*, vol. 25, p. 101605, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711023003011

[5] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[6] R. Team, "Ranflood github repository," https://github.com/Flooding-against-Ransomware/ranflood, 2025, [Accessed Aug. 2025].

[7] D. V. Chudnovsky and G. V. Chudnovsky, "Algebraic complexities and algebraic curves over finite fields," *J. Complex.*, vol. 4, no. 4, pp. 285–316, 1988. [Online]. Available: https://doi.org/10.1016/0885-064X(88)90012-X

[8] N. I. of Standards, T. (NIST), M. J. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. E. Bassham, E. Roback, and J. D. Jr., "Advanced encryption standard (aes)," 11 2001.

[9] E. H. Mckinney, "Generalized birthday problem," *The American Mathematical Monthly*, vol. 73, no. 4, pp. 385–387, 1966.

[10] D. Brink, "A (probably) exact solution to the birthday problem," *The Ramanujan Journal*, vol. 28, pp. 223–238, 2012.

[11] W. G. Horner, "A new method of solving numerical equations of all orders, by continuous approximation. [abstract]," *Abstracts of the Papers Printed in the Philosophical Transactions of the Royal Society of London*, vol. 2, pp. 117–117, 1815. [Online]. Available: http://www.jstor.org/stable/109939

[12] S. Ali, J. Wang, V. C. M. Leung, and A. Ali, "Decentralized ransomware recovery network: Enhancing resilience and security through secret sharing schemes," in *Proceedings of the 9th International Conference on Internet of Things, Big Data and Security, IoTBDS 2024, Angers, France, April 28-30, 2024*, A. Kobusinska, A. Jacobsson, and V. Chang, Eds. SCITEPRESS, 2024, pp. 294–301. [Online]. Available: https://doi.org/10.5220/0012713500003705

[13] C. Medina-López, V. González-Ruiz, and L. G. Casado, "On mitigating pollution and free-riding attacks by shamir's secret sharing in fully connected p2p systems," in *2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2017, pp. 711–716.

[14] L. Goubin and A. Martinelli, "Protecting AES with shamir's secret sharing scheme," in *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, ser. Lecture Notes in Computer Science, B. Preneel and T. Takagi, Eds., vol. 6917. Springer, 2011, pp. 79–94. [Online]. Available: https://doi.org/10.1007/978-3-642-23951-9_6

[15] J. Coron, E. Prouff, and T. Roche, "On the use of shamir's secret sharing against side-channel analysis," in *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Mangard, Ed., vol. 7771. Springer, 2012, pp. 77–90. [Online]. Available: https://doi.org/10.1007/978-3-642-37288-9_6

[16] R. J. Lipton, R. Ostrovsky, and V. Zikas, "Provably secure virus detection: Using the observer effect against malware," in *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, ser. LIPIcs, I. Chatzigiannakis, M. Mitzenmacher, Y. Rabani, and D. Sangiorgi, Eds., vol. 55. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 32:1–32:14. [Online]. Available: https://doi.org/10.4230/LIPIcs.ICALP.2016.32

[17] A. Continella, A. Guagnelli, G. Zingaro, G. D. Pasquale, A. Barenghi, S. Zanero, and F. Maggi, "Shieldfs: a self-healing, ransomware-aware filesystem," in *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, S. Schwab, W. K. Robertson, and D. Balzarotti, Eds. ACM, 2016, pp. 336–347. [Online]. Available: http://dl.acm.org/citation.cfm?id=2991110

[18] J. A. Gómez-Hernández, L. Álvarez-González, and P. García-Teodoro, "R-locker: Thwarting ransomware action through a honeyfile-based approach," *Comput. Secur.*, vol. 73, pp. 389–398, 2018. [Online]. Available: https://doi.org/10.1016/j.cose.2017.11.019

[19] S. Lee, H. K. Kim, and K. Kim, "Ransomware protection using the moving target defense perspective," *Comput. Electr. Eng.*, vol. 78, pp. 288–299, 2019. [Online]. Available: https://doi.org/10.1016/j.compeleceng.2019.07.014

[20] Microsoft. (2024) Protect important folders with controlled folder access. https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/controlled-folders?view=o365-worldwide. Accessed Sept. 2024.

[21] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *Comput. Commun. Rev.*, vol. 27, no. 2, pp. 24–36, 1997. [Online]. Available: https://doi.org/10.1145/263876.263881

[22] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960. [Online]. Available: https://doi.org/10.1137/0108018

[23] Trend Micro Research, "Ransomware Spotlight: LockBit | Trend Micro (US)," https://web.archive.org/web/20240823210045/https://www.trendmicro.com/vinfo/us/security/news/ransomware-spotlight/ransomware-spotlight-lockbit, 2024, accessed Sept. 2024].