Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Shamir's Secret Sharing and Ranflood: a Flooding Strategy against Crypto and Exfiltration Ransomware

Relatore:
Chiar.mo Prof.
Saverio Giallorenzo

Presentata da:
Daniele D'Ugo

Sessione I
Anno Accademico 2023/2024

Sommario

Con l'ampio impiego odierno di sistemi informatici per archiviare dati, preoccupa la diffusione di software malevoli con cui i cyber-criminali ne cercano ricatti ("ransom"): attraverso i *crypto ransomware*, essi possono cifrare i file della vittima e richiedere un pagamento per il ripristino; con gli *exfiltration ransomware*, possono rubare i file e minacciare di diffonderli.
Di conseguenza, esiste anche un campo di ricerca nel loro contrasto. Nessuna soluzione è perfetta, e quella analizzata in questa tesi, il "Data Flooding against Ransomware", consiste nel rallentare il malware, creando file-esca ("flooding") e competendo con esso nell'uso delle risorse computazionali, per dare più tempo a una persona di agire fisicamente sulla macchina (per esempio, spegnendola). Inoltre, usando delle copie degli stessi file dell'utente come esche, è possibile eseguire un successivo ripristino, riducendo la quantità di dati persi nell'attacco.

Il lavoro qui presentato parte dall'analisi di uno strumento sperimentale e open-source per il contrasto dei *ransomware*, *Ranflood*, per espanderlo con nuove tecniche.
Ranflood è un software che implementa tre diverse strategie di *flooding*; l'aggiunta consiste in una quarta, nonché in nuove idee per migliorarne l'efficienza: essa è basata sullo *Shamir's Secret Sharing*, un protocollo che permette di dividere un segreto in più frammenti, e di ricostruirlo a patto che se ne possiedano più di una certa soglia prestabilita. Se contro i *crypto ransomware* i frammenti possono agire da esche, contro gli *exfiltration ransomware* si riduce ulteriormente il rischio di danni alla vittima, poiché l'attaccante non può riassemblare un file originale senza aver trovato abbastanza pezzi.
Attraverso i parametri di soglia minima e numero di frammenti creati, oltre che con altri parametri, è possibile dare un indirizzo diverso all'esecuzione del *flooding*. Dovendo pensare a diversi casi d'uso, e in particolare a due diversi tipi di *ransomware*, il giusto modo di configurarli dipende dalle esigenze principali di ogni situazione, creando un copromesso più o meno bilanciato tra di esse. In tal senso, in test effettuati cercano di dare delle risposte a vari scenari diversi basandosi su delle metriche astratte dai dati per adattarsi alle proprie necessità.

La tesi, dunque, prima illustra l'architettura del software *Ranflood* e le basi matematiche dello *Shamir's secret sharing*: del primo si analizzano soprattutto la soluzione *client-daemon* e la distribuzione di task di input/output su più thread; del secondo, si passa dai polinomi, ai campi finiti, all'algoritmo. A seguire, si mettono insieme i concetti, entrando nel dettaglio con l'implementazione e le scelte effettuate: si mostrano le valutazioni iniziali che hanno portato a varie scelte, come quella dell'implementazione di *Shamir's secret sharing*, nonché quelle sulla struttura con cui integrarsi con la base preesistente di *Ranflood*. Per finire, si illustrano analisi computazionali ed esperimenti, da cui vengono poi tratte delle osservazioni - per lo più intuitive - sulla validità della strategia implementata per il suo scopo e sulla sua efficienza, pensando anche agli ambiti reali, e su possibili modi di migliorarla.

Abstract

With the large usage of data storage system, among cybercriminals also increases the employment of *crpyto* and *exfiltration ransomwares*. At the same time, scientific research is always working on solutions to contrast them, and *Data Flooding against Ransomware* is one of these.
Our work focused on one open-source project in particular implementing it, *Ranflood*, trying to expand it with a new, novel *flooding* strategy based on the *Shamir's Secret Sharing*, to address both *crypto* and *exfiltration* threats. Developed in 1979 to make a secret only available when a quorum of its parts is gathered together, *Shamir's secret sharing* is hereby applied to split a file in more parts, such that they constitute a flood against *crypto ransomwares*, and to make the original content only retrievable when enough of them are obtained, such that the *exfiltration* is less likely to succeed, while the victim can hopefully restore his data.
This thesis goes through the mathematical details, the analysis of *Ranflood*, and then the implementation of our strategy, the made choices, computational analysis, intuitive tests and following conclusions.

Contents

1 Introduction **3**
- 1.1 Ransomwares . . . 3
- 1.2 Countermeasures . . . 4
  - 1.2.1 Flooding and *Ranflood* . . . 5
- 1.3 Our contribution . . . 5

2 Background **6**
- 2.1 Ranflood . . . 6
  - 2.1.1 Architecture . . . 7
  - 2.1.2 fileChecker . . . 9
- 2.2 Shamir's Secret Sharing (theory) . . . 9
  - 2.2.1 Mathematical principle . . . 9
  - 2.2.2 Algorithm . . . 10
  - 2.2.3 Modular arithmetic . . . 11

3 Our contribution **12**
- 3.1 Theory . . . 12
  - 3.1.1 Shamir's Secret Sharing (our model) . . . 12
    - 3.1.1.1 Security . . . 14
- 3.2 Implementation . . . 16
  - 3.2.1 Shamir's secret sharing (implementation) . . . 17
    - 3.2.1.1 codahale/shamir . . . 17
    - 3.2.1.2 sssfile wrapper . . . 19
      - 3.2.1.2.1 SSSSplitter . . . 19
      - 3.2.1.2.2 SSSRestorer . . . 20
  - 3.2.2 SSS in Ranflood . . . 21
    - 3.2.2.1 OnTheFly flooder . . . 21
  - 3.2.3 SSS flooder . . . 22
    - 3.2.3.1 On structural choices . . . 24
    - 3.2.3.2 SSS snapshooter . . . 27
    - 3.2.3.3 SSSFlooder parameters . . . 27
    - 3.2.3.4 SSS in fileChecker . . . 28
- 3.3 Testing . . . 29
  - 3.3.1 Computational analysis . . . 30
  - 3.3.2 Empirical performance evaluation . . . 33
  - 3.3.3 Practical performance evaluation . . . 34

4 Conclusion **41**
4.1 Summary 41
4.2 Future works and optimizations 42
4.3 Observations 43

Chapter 1

Introduction

In a world where every individual accesses a computer daily, to store his personal data, or pictures, or files he's working on, or just his favourite applications, and where he communicates via internet with servers all over the world retaining information on hundreds, thousands, millions people, inevitably some others try to exploit such systems for a profit. Among the several categories of cyberattacks, ransomwares are certainly well-known due to the recurring news about their employment even against large institutions or firms, but also minor attempts against common users happen frequently.
Many solutions to prevent attacks are constantly being developed, but none is perfect, like in the whole IT field, being ever-evolving. What we're going to discuss in this thesis is a novel method to try and contrast ransomwares; we will explain the ideas behind it and go through an implementation of ours, and then reason about its advantages, disadvantages and best applications.

1.1 Ransomwares

To start with, by *ransomware* we intend a *malware*, i.e. a malicious software, which is, somehow, installed on a victim's machine and allows an attacker to extort a gain from him (that is, the "ransom"). In its most basic form, it starts encrypting (some or all) system's files, so that the victim is forced to pay a certain amount of money to get the encryption key and be able to restore the environment - this is called a *crypto ransomware*. More precisely, different ransomwares could behave differently: some could limit themselves to annoy the victim's personal files, so that he can still use the computer, but won't be able to access them and, probably, to launch some applications; in this case, one or more text files will probably be put in commonly used directories (e.g. the desktop) to make him know how to pay the ransom; others could make the entire system unusable, showing a custom screen, on startup, with the instructions.
How a malware can be installed there, at the moment, is not of our concern: it could exploit any system's, network's or web's vulnerability, or just the user's lack of attention, e.g. when involuntarily visiting the wrong page or clicking the wrong button (placed there maliciously by cybercriminals).

However, this isn't their only working mechanism; particularly concerning is becom-

ing the *exfiltration* type of *ransomware*, which (also) steals the encrypted data, so that the "ransom"'s payment is instead required to prevent its diffusion. This kind of attack is probably even more dangerous, seeing that, once the collected data has been sent to the remote server controlled by the attacker - hence the "exfiltration" -, there's no guarantee that it will be deleted after the payment; indeed, requests for money or something else could be going on for much longer, putting even a greater pressure than data loss. Moreover, the theft could be of high concern for big data centers, having access to many people's personal, sensitive details, or for institutions handling classified information, such as armies, governments, research centers.
These malwares could threaten the victim both on the data encryption and exfiltration sides, but cybercriminals often only focus on the latter, being easier to perpetrate and of higher impact: in fact, *crypto ransomwares* can come with difficulties such as the transmission of the encryption key, or possible problems in restoring the victim's system, costing them time, worsened reputation or other risks.

1.2 Countermeasures

As always, "prevention is better than cure" in this context. Given the large amount of means a ransoware can use to spread, there are many actions one could or should take to avoid attacks in the first place.
From the common user's or employee's point of view, being informed and conscious is the first step, allowing to mistrust dangerous interactions, and also using security protections, filters or antiviruses helps in dodging spam attempts. A company (of any size), especially if handling many people's information, should apply more sophisticated techniques, network firewalls, intrusion detection systems, etc. Other common measures, for anybody, are keeping software updated, using the least needed privilege in configurations and when performing any action; especially for ransomwares, then, an attack's impact can be largely reduced by efficient, periodical backups, especially for sensitive data.
Nonetheless, being able to act even after an infection has started can reduce damages. After a ransomware has been launched, common measures to take immediately would be to physically disconnect the machine from the network (to avoid any attempt to spread the attack), suspend, hibernate or shut it down, even though this could result in corrupted data, and the malware could also persist after a restart - at least, it would allow to access the computer or disk in an isolated environment, or to perform a restoration, if only a few, less relevant files have been encrypted, or if recent backups are available. Furthermore, in case of (data) exfiltration acting as fast as possible would be of even greater importance - even removing the access to the network.

Anyway, one of the focuses of both research and private software solutions consists in gaining time to take the action by slowing down the attack. The tool we present hereinafter, to which we contributed by applying our research, *Ranflood* [2], accomplishes this through *Data Flooding against Ransomware* (*DFaR* [9]).

1.2.1 Flooding and *Ranflood*

As the name implies, *DFaR* consists in flooding a file system - or specific folders -, by creating new files, with the intent of slowing down the ransomware both by making it encrypt useless files (instead of the sensitive ones) and by concurring with it in the resources usage, i.e. loading the CPU with tasks and increasing the input/output usage, for the disk. Note that a similar approach could be implemented on a network, to hinder data exfiltration to the remote server, although not being neither the object of *Ranflood* nor of our research.

Ranflood is an open-source project (available on GitHub [2]), written in Java to ensure high compatibility across devices and developed under a research of the University of Bologna (on which two papers exist [9][10]). It currently implements *flooding* through three different strategies, which the user can select accordingly: *Random* creates random files, *On-the-fly* and *Shadow* create both copies of existing files, with the difference that the latter uses a more sophisticated method based on *tar* archives (more on strategies and *Ranflood* later, in 2.1). The last two also allow to restore lost files using the created copies, if anyone survived the infection.

1.3 Our contribution

Our research also focuses on *flooding against ransomware*, introducing a new strategy based on *Shamir's secret sharing*, an algorithm allowing to split a *secret* in more *parts* such that, if enough of them are combined together, the *secret* can be reassembled. In our case, the *secrets* are the files to encrypt against the ransomware, so that their *parts* can be used to flood the environment.
We anticipate that, due to the encryption overhead of the algorithm, it can't be as fast as a straight copy - performed by the aforementioned strategies -; however, our work can be taken as a starting point on this not so widely known secret-sharing protocol, especially when applied to files and to *ransomware mitigation*, given the lack of other research on the topic. Moreover, besides flooding against *crypto ransomwares*, our method could be of particular interest against *exfiltration ransomwares*, which would gain less information by stealing encrypted files they can't decrypt.
Our work, as we will see throughout this thesis, consisted of a research on *ransomwares*, *flooding* and *Shamir's secret sharing* (section 2.2), and a study of the *Ranflood* software (2.1), all followed by the combination of these, made of theory and suppositions (3.1), practice, through the implementation itself (3.2), analysis and tests (3.3), and considerations on the developed solution, either presented during the implementation or in conclusion (4) chapter.

Chapter 2

Background

In this chapter we'll explain more in detail the concepts our work is based on. In particular, our development required an initial study of the *Ranflood* software (2.1) - in order to align our implementation with its purposes and philosophy - and of the *Shamir's Secret Sharing* protocol (2.2) - to understand how to use it at its best.

2.1 Ranflood

Ranflood is an open-source software, written in Java, with the purpose of addressing the action of ransomwares in a generic and configurable way. Although its features and ideas have been presented in the two papers dedicated to it [9] [10], they're not all been implemented yet: in this thesis we will also talk about what's currently missing. *Ranflood* is based on the concept of *Data Flooding against Ransomware* (*DFaR*), i.e. it tries to flood the file system (or some selected folders) with files, slowing down the execution of the malware by both luring it to read and encrypt useless, bait files and concurring with it for the IO access, hindering its actions on the disk. In *Ranflood*, flooding can be performed with three possible strategies: *Random* creates files containing random bytes; *On-the-fly* and *Shadow*, in different ways, spread copies of existing files.

The software follows the *client-daemon* pattern, and is thus composed of a *daemon*, thought to be always running, and a command which can be issued by an user to make it perform an action. This architecture allows a modular approach, making it possible for the client to start, stop or monitor a *flooding* session at run-time, and to configure it via the available parameters. Furthermore, there can clearly be more clients. Because of the *client-daemon* pattern, we can define *Ranflood* a *drop-in* solution to *ransomwares*.

The project aims at covering all phases of a threat: *detection*, *mitigation* and *restoration*. The *detection* phase is the one still missing in the implementation, but the modularity of the *client-daemon* approach makes it easy to integrate any other external detection software, making it act as a *client* to start a *flooding* when needed.
In the future, however, as presented in the papers, these strategies could also serve this purpose by applying an *honey-potting* technique: basically, a new research could try to recognize a suspicious activity by monitoring generated files, as the user wouldn't nor-

mally open them. Moreover, taking advantage of the *drop-in* implementation, *Ranflood* could avoid the common problem of *honey-potting* of being invasive in the user's system: to reduce the intrusiveness of decoy files, one could periodically flood a limited number of folders, monitor them and eventually restore the original environment.
The *mitigation* phase is the most obvious one, consisting in the *flooding* itself.
Lastly, the *restoration* phase, that can only be performed with the copy-based strategies (*On-the-fly* and *Shadow*) due to their nature, removes the generated files and only keeps one in case the original one was corrupted.

The choice of the Java language is linked with the high availability of Ranflood, easily operable on any platform supporting the Java Virtual Machine (native binaries [3] are already available for Linux, Windows, macOS thanks to the GraalVM5 compiler [12]).

2.1.1 Architecture

For a better understanding of our contribution, this section we will give a deeper insight on the software architecture.
Regarding the *flooding* strategies, and in particular the two copy-based ones (*On-the-fly* and *Shadow*), they require a preliminary *snapshooting*: a *snapshot* of a directory saves, inside a database indexed on its path, the current state of its content - *On-the-fly* saves the checksum of each file contained in it, directly or in any subdirectory, while *Shadow* saves the directory itself, compressed as a *tar* archive.
A *snapshot* of a path serves two purposes: in first place, in the middle of a *flooding* to contrast an ongoing *ransomware* attack, it allows to avoid creating copies of a file if its hash has changed in the mean time, meaning that it has already gone corrupted and it's better to focus on the others; secondly, in the *restoration* phase, it allows to identify the corrupted copies and pick an intact one to recreate the original file.
Of course the *Random* strategy would have no use for *snapshots*.

As we saw before, Ranflood is formed of a *client* and a *daemon*: since the first is a simple command-line tool (which could include a graphical interface, in the future) to issue commands for the latter, via socket (using the *ZeroMQ* library [15]), the whole logic of the program is inside the daemon.
From an abstract point of view, the *daemon* is made of two components: the *engine* and the *task manager*.
The engine, precisely, implements its core functionalities of *flooding* and *snapshooting*. In the code, these two are represented by the *Flooder* and *Snapshooter* interfaces, exposing the methods, respectively, to start or stop a flood and to take snapshots, besides removing and listing existing ones; each strategy, then, is an implementation of *Flooder*. The *daemon*, so, according to its parameters - configurable in a settings file (in the *.ini* format) -, creates an instance of each *flooder*, making it available to execute the received commands.

For optimization purposes, given the high amount of input/output access required, a *TaskManager* is used to manage the execution on multiple threads, playing the role of the *proactor* according to the *Proactor* pattern. In fact, any operation of writing (or

copying) to a file from a *flooder* is not executed immediately; instead, a *Task* is created for that single write operation, and added to the *TaskManager*'s scheduling: in this way, the *flooder* can rapidly visit each target path, while tasks are gradually launched on different IO threads.
The *Proactor* pattern also has the advantage of not being slowed down by a task's error, since one's execution is decoupled from it's dispatching.
Additionally, any task, failed or not, is repeated indefinitely until the flooding is stopped; exactly, each ongoing *flooding* is represented itself by a *Flood Task*, so that the *TaskManager* can receive from each the *write tasks* it created during its execution.
Since all tasks reside in main memory, even if an user's file has been lost, copies of it can still be created (starting from their cached content).

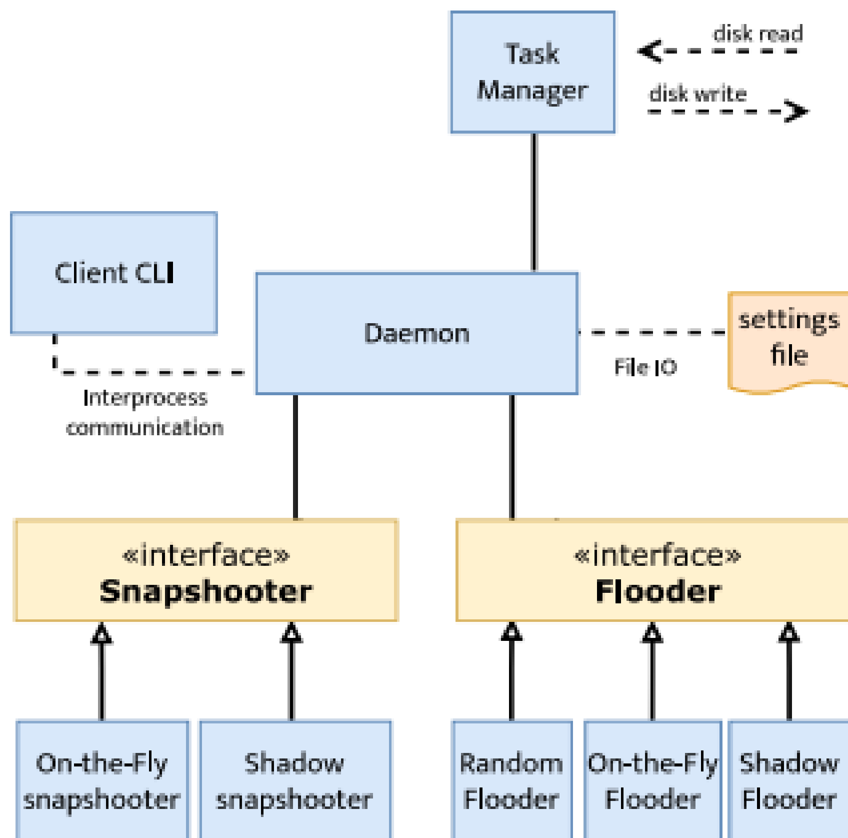

Figure 2.1: Model of Ranflood's architecture. Image from Ranflood's paper [10].

The figure above summarizes *Ranflood*'s structure: the *daemon*, core of the application, made of *snapshooters* and *flooders* (the *engine*) and the *Task Manager* (the *proactor* for IO tasks), executes commands and is configured according to a settings file and, via interprocess communication, to a *client*'s commands.

2.1.2 fileChecker

The restoration phase, after an attack, is performed by the *fileChecker* tool, actually a separate program from the core *Ranflood*, although being in the same repository.
Its functioning isn't as complex as for the *mitigation* phase. Put simply, as we already spoke of how *flooding* works and further details aren't necessary at the moment, currently *fileChecker* checks the state of a directory's content, comparing their checksums with previous *snapshots*, and produces a report, also taking in account the presence of generated copies. Since the restoration phase doesn't take place in a context of emergency like the *mitigation*, it doesn't necessarily need multithreading or tasks and other patterns seen in the *Ranflood daemon*, if not just for minor improvements.

2.2 Shamir's Secret Sharing (theory)

Shamir's secret sharing (*SSS*) is a secret sharing algorithm for dividing a private information (the *secret*) in *parts* (which we'll also call *shares* or *shards*) and distributing them (a *threshold*). The *secret* cannot be reassembled unless a sufficient number of (any of) them is collected together. It was first proposed in 1979 by Adi Shamir [11] (the *S* in *RSA*, which he co-invented). [21]
Beyond our novel use of it to contrast *ransomwares*, a "secret sharing" protocol also finds its utility in making something only accessible when enough parts are in reach: e.g. to split a password among some people and make it possible to perform a sensitive operation of some kind only when enough of them agree to it; in some terms, this is similar to how we use it against *exfiltration*, to retain *ransomwares* from obtaining user's data if they haven't collected enough pieces.

SSS has the property of information-theoretic security, meaning that even if an attacker steals some *shares*, it is impossible for the attacker to reconstruct the *secret* unless they have stolen the quorum number of *shares* [21].
For this thesis, we define n as the number of *shards* a *secret* is split into, and k the minimum number of (distinct) *shards* required in order to reassemble the *secret*.

2.2.1 Mathematical principle

Shamir's secret sharing is based on the uniqueness of the *Lagrange interpolating polynomial*: given k (distinct) coordinate pairs $a_1, a_2, ..., a_k$, there is only one polynomial, $f(x)$, of degree $k-1$ passing through all of them (consequently, having $k-2$ turning points). Moreover, once fixed, one can choose other $n-k$ (distinct) points (with $n >= k$) on the formed curve, $a_{k+1}, a_{k+2}, ..., a_n$: now, any subset of k out of the n points $a_1, a_2, ..., a_n$ allows to obtain the polynomial via interpolation. [20]

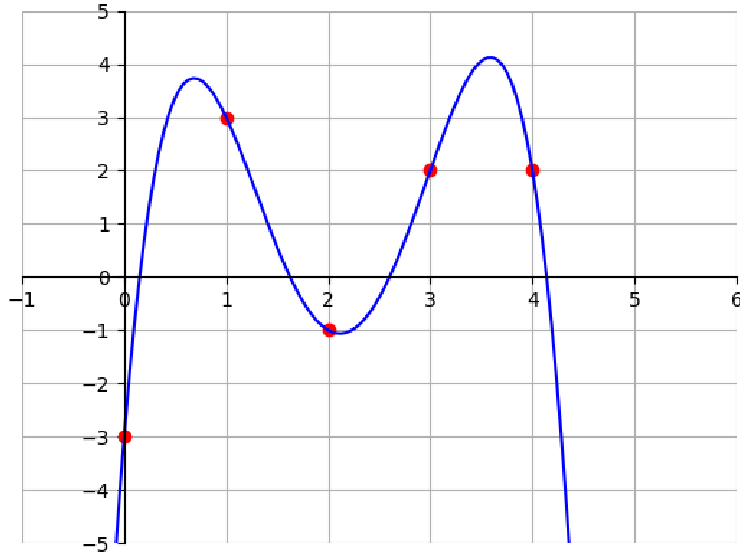

Figure 2.2: Lagrange polynomial of degree 4, with $k = 5$ points and 3 turning points

In the above example, a polynomial of degree $k - 1 = 4$ was obtained, via interpolation, starting from $k = 5$ red points; later, one could have $n = 6 > k = 5$ points by picking anyone on the curve (e.g. where $x = 5$).

2.2.2 Algorithm

There isn't a standard implementation of SSS, but the general idea follows the following steps:

1. once the parameters have been fixed, we define:
   (a) $s \in \mathbb{N}$ the secret to encrypt
   (b) $n, k \in \mathbb{N} : k <= n$, respectively, the number of shards to create and the minimum number for being able to re-obtain s
2. generate the k coefficients of the polynomial f of degree $k - 1$ $(a_0, a_1, ..., a_k)$:
   (a) $a_0 = s$
   (b) a_i: randomly chosen
3. calculate the n points $p_i = (i, f(i))$, for $i = 1, 2, ..., n$

Note that f passes through the *secret* (point $(0, s)$), since its first coefficient is s. Also, thanks to interpolation, it suffices to have k of the n points in order to reassemble it, being f of degree $k - 1$.

In reality, SSS should be implemented using a *finite field* (like for elliptic curves), for a few reasons.
First of all, performing operations in $\mathbb{Q}$ or $\mathbb{R}$, since interpolation requires the use of divisions, can lead to computation errors, due to *underflow* or *overflow*. Moreover, there wouldn't be *perfect secrecy*, because an attacker could gain some information about s with less than k shards (e.g., that s is even [22] - we omit the proof here, being unnecessary for the understanding of this thesis).

2.2.3 Modular arithmetic

A finite field, denoted as $\mathbb{F}_{p^r}$, is an algebraic structure composed by a set of $q = p^r$ elements (where $p, r \in \mathbb{N}$, p is prime and $r > 0$) and two operations with certain properties. q is called *order* of the finite field.
The most widely known example is when $r = 1$, since $\mathbb{F}_p$ is the set of residue classes modulo p, which can also be seen as follows:

- the p elements are represented by the integers $0, 1, ..., p - 1$;
- the two operations are modular addition and modular multiplication;
- each operation has its own identity element, in order, 0 and 1;
- besides the other properties (closure, associativity, identity, commutativity), there exists an inverse element for each $x \in \mathbb{F}_p$, with respect to each operation, that is, $\exists y, z \in \mathbb{F}_p : x + y = 0 \land x \cdot z = 1$; using the inverses, we can define the opposite operation of multiplication in $\mathbb{F}_{p^r}$, and express $\frac{a}{b}$ as $a \cdot b^{-1}$.

Putting it together with the aforementioned algorithm, the way modular arithmetic is used is in the fact that each x and y of a coordinate pair is actually an element of $\mathbb{F}_q$, and each coefficient of the polynomial is too.
So, among the other parameters to choose for SSS, we need a finite field of a sufficiently large order, such that

$$q > n \land q > s \tag{2.1}$$

Specifically, we require:

1. $q > s$, in order to be able to store the integer representing the secret (otherwise, we wouldn't be able to express it with the available elements)
2. $q > n$, in order to be able to create enough distinct shards (otherwise, since shard i is represented by the point with $x = i \in \mathbb{N}$, with $i = 1, 2, ..., n$, there wouldn't be enough elements in $\mathbb{F}_q$ to express all the distinct x_i)

Finally, the selected finite field is a public parameter of the protocol.

Chapter 3

Our contribution

After having taken a look at the background knowledge on the software and at some mathematical principles, in this chapter we will show how *Shamir's secret sharing* was applied to expand the *Ranflood* software. First, in section 3.1 we will reveal the theory behind our work, and explain what lead some of our decisions; then, in section 3.2 we will illustrate the implementation in deep; finally, in section 3.3, we will analyze the performance and try to find intuitive measurements.

3.1 Theory

3.1.1 Shamir's Secret Sharing (our model)

In order to choose the correct model for our implementation of Shamir's secret sharing, it's important to first of all take note of our needs, i.e. how we're going to use it. After having understood how the protocol works, in section 2.2, the main requirement to keep in mind is that, as we showed in section 2.2.3 with formula 2.1,

$$q > s \land q > n$$

An intuitive approach would be using a set of remainder classes modulo a prime q, choosing q as upper bound for both the size of a *secret* and the number of *shards* one can split it into. Unfortunately, this wouldn't be so immediate to apply to encrypt a file of arbitrary dimensions; moreover, the integer representation of a sequence of bytes isn't much compact, and performing modulo operations with it becomes slow as it increases - for reference, a 2048 bits integer, like a common size for n in *RSA*, correspond to about 257 bytes. Then one must assume that, whatever the size of q is, a file of size b bytes has to be divided into $\lceil b/q \rceil$ pieces, so that each of them can be split with SSS separately.

After formulating such observations, we started doing some research and analysis of already existing implementations of SSS, preferably in Java, to find the one best fitting our needs. As expected, many have the limitations imposed by the size of the prime number q of their choice, not allowing the encryption of larger secrets, nor the splitting in a larger number of *shards*. Some also suggest, as an alternative, to first encrypt the secret with a symmetric key encryption algorithm and then use SSS on the key only:

while this could be effective to make it accessible only in presence a quorum of shards holders, it doesn't fit our case.
To cite some examples, an implementation on GitHub in Go [6] limits secrets to $2^{127} - 1$ bits, using a prime number of 128 bits, while *ssss* [7], available as *apt* package, has a limit of 1024 bits, so could have a similar implementation (though we didn't check its code).
Of course, as we said, one could extend such implementation by repeating SSS on a single file multiple times, however what we found much more interesting were other projects which, besides implementing the mechanism to accept inputs of arbitrary length, use the finite field $GF(256)$. It is the case of the security tool *Vault* by *HashiCorp* [5] (in Go), and of the GitHub repository *codahale/shamir* [4], which, being in Java, we used directly in our work. We will now see in detail what $GF(256)$ is, and why it's a reasonable choice given it's high efficiency.

$\mathbb{F}_{2^8}$ (also known as $GF(256)$, where *Galois Field* (GF) is synonym of *finite field*, and 256 describes the same GF as 2^8) is a finite field of order 2^8. Being used in our implementation, the x and y coordinates of each point, as well as polynomials' coefficients, are elements of $\mathbb{F}_{2^8}$.

Such choice isn't trivial in computer science. The possible values of a *byte* are 256 and, being these in a small number (with respect to the possible values of an integer), their operations can be implemented in very efficient ways:

- the "unary" operations (i.e. with a fixed operand) of logarithm base 2 and exponentiation of 2 only have 256 possible outputs, which can be pre-computed and stored in *lookup tables* (actually, the *lookup table* for exponentiation has size $2 \cdot 256$, to also avoid the operation of modulo in case of *overflow* - e.g. we can more easily calculate $log_2(a + b)$, with $a, b \in [0, 256[$, even if $a + b >= 256$);

- the operations of addition and subtraction are just equivalent to a *xor* (we'll see why later);

- the multiplication takes advantage of the same *lookup tables*, as

$$a \cdot b = 2^{log_2(ab)} = 2^{log_2(a)+log_2(b)}$$

  (with a sum in $\mathbb{N}$, not a *xor*);

- dividing is the same as multiplying by an inverse, and

$$b^{-1} = 2^{log_2(b^{-1})} = 2^{255-log_2(b)}$$

  (since $-log_2(b)$ is $255 - log_2(b)$ in $GF(256)$).

It's not a coincidence that $GF(256)$ is also used (among others) in the internal operations of *AES*, which works on single bytes and can gain significantly better performances thanks to its efficiency.

Visualizing $\mathbb{F}_{2^8}$ isn't as simple as it is for a finite field with a prime number of elements -

a set of remainder classes -, though (a useful source which helped us understanding it is at [8]). An element t of $\mathbb{F}_{2^8}$ is indeed a polynomial of degree 8, whose coefficients' values are in $\mathbb{F}_2$ (that is, one of $\{0, 1\}$):

$$t = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \tag{3.1}$$

For this reason, t can be seen as a vector of coefficients, just like a *byte* is a vector of *bits*, and so can be represented in a single *byte* where a_0 is the least significant *bit*.
Hence, the operations of addition and multiplication are those defined on polynomials, with the further constraint that for each coefficient we must take the remainder modulo 2 - thus the addition is a *bit*-wise *xor*.
Finally, even the whole polynomial t is taken under a modulo, in the same way that p is the modulo in $\mathbb{F}_p$. In this case, the modulo is, of course, a polynomial m, of degree 8, such that $t \text{ mod } m$ is always of degree less than 8. m is called a *reduction polynomial*. The choice of m is only an implementation detail, since it doesn't influence the structure of the finite field - that's why all fields of order 256 are called $GF(256)$, having the same structure. The only requirement here is that m must be prime (just like the modulo p), in order for the algebraic structure to be defined a *field* (i.e. there exists an inverse for each of its elements) - for a polynomial to be prime it means that it can't be factored in lower degree ones.

A generator g must also be chosen, that is an element which, by repeatedly multiplying by itself, takes all non-zero values of the finite field, until it wraps back to itself when $g^{256} = g$. In fact, the non-zero elements of a finite field also form a *multiplicative group* [19].
In the same way as m, the choice of g is arbitrary as well. A generator is required to populate the *lookup tables*, just by exploiting its property, applying repeated multiplications on it in order to obtain all elements.

Our implementation uses the same reduction polynomial ($0x11b$, i.e. $x^8+x^4+x^3+x+1$) and same generator ($0x03$, i.e. $x + 1$) as AES.

3.1.1.1 Security

After having shown how it works, we can now make some considerations on Shamir's secret sharing's security.
We already anticipated that it has perfect secrecy, in 2.2, if modular arithmetic is used. Consequently, no information on the *secret* can be gained with less than k *shards*, and it would have to be considered lost. In fact, while given k (distinct) points there's only one function of degree $k-1$ passing through them - the Lagrange interpolating polynomial -, given only $k-1$ points a different interpolating polynomial would be yielded according to which additional k-th is chosen; thus, it would be theoretically impossible to guess the *secret*: additionally, recovering a shard's coordinate pair is equivalent to recovering that of the secret.

Another valid observation would be that, when using SSS repeatedly, supposing of knowing that two different *secrets* s_1, s_2 were split using the same polynomial f (and, consequently, have themselves the same value, or the unique interpolating polynomial wouldn't

be the same), their shards are also the same, or at least all lie on f: this implies that one could retrieve both $s_1 = s_2$ with only k parts (instead of $2k$ when reassembling each individually) and, more surprisingly, even combining a few shards from s_1 with others from s_2, provided that they're k (let's say, k_1 from s_1 and $k_2 = k - k_1$ from s_2).
Another interesting case would be when s_1, s_2 use two different polynomials f_1, f_2, but f_1, f_2 are somehow related, e.g. f_2 is f_1 plus a constant element: this equally requires knowing that they are correlated, as well as knowing what the correlation is.

To know the likeliness of obtaining the same curve twice, we can calculate the probability of such event, related to the $GF(256)$ we chose. More precisely, we are looking for the probability of picking a certain polynomial f of degree $k-1$, that is to say having k coefficients $a_0, a_1, ..., a_{k-1} \in GF(256)$; furthermore, thanks to the Lagrange interpolating polynomial, f can also be calculated after picking k points $p_1, p_2, ..., p_k$ which, like in the algorithm in 2.2.2, have the form $p_i = (i, y_i)$, with y_i being a random element of $GF(256)$: both events are equivalent to be looking for a specific, ordered, sequence of k elements of $GF(256)$.
Assuming an uniform distribution of our random number generator, any byte (i.e. an element of $GF(256)$) can be chosen randomly with a probability of $\frac{1}{256}$. It follows that the probability for a polynomial to be generated is

$$P[f] = (\frac{1}{256})^k \tag{3.2}$$

This formula can also show the number of possible polynomials one can obtain when having only $k' < k$ shards, by replacing k with $k - k'$. With k' fixed shards, in fact, $k - k'$ could have any values, thus showing how hard guessing is:

$$P[f] = (\frac{1}{256})^{k-k'}, 0 < k' \leq k \tag{3.3}$$

Since the probability grows exponentially relatively to k (which is actually the only variable), it can be largely reduced by slightly increasing it, and thus the probability of a conflict can be reduced as well. However, considering that, for our case, we need to use SSS on possibly massive amounts of bytes of several files (each individual byte of any *split* file), it could be useful to also determine at least an approximation of a good choice for k.
The *birthday (paradox) problem* [17] tells us that collisions are more likely than one could imagine: the generalized problem, specifically, for a random variable $n(d)$ uniformly distributed over d values, consists in calculating the number of events that must occur in order for two of them to have the same outcome [18]. The following formula [16] holds for all $d <= 10^{18}$, and is conjectured to hold for all $d \in \mathbb{N}$:

$$n(d) = \lceil \sqrt{2d \ln 2} + \frac{3 - 2\ln 2}{6} + \frac{9 - 4\ln 2^2}{72\sqrt{2d \ln 2}} - \frac{2\ln 2^2}{135d} \rceil \tag{3.4}$$

In the following table are some example results for possible values of $d = 256^k$:

Results of applying $d(256^k)$ and $\frac{d(256^k)}{10^6}$ for different values of k		
k	$d(256^k)$ (bytes)	$\frac{d(256^k)}{10^6}$(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 $MegaBytes$ of data there's a 50% probability of the problem to happen, which could not be so unlikely when flooding a large system, but using $k = 9$ and $k = 11$ already allows to raise the required threshold to 80.911 $GigaBytes$ and 20.7 $TeraBytes$ respectively.

Although posing ourselves such concerns during our research, they weren't directly addressed in our implementation, for a few reasons (which also apply to some other choices). First of all, security concerns, of course, only apply to exfiltration, where ideally the victim want the attacker to decrypt and retrieve as little data as possible. In second place, the main purpose of a flooding is to be faster than the ransomware, as much as possible, and even in case of exfiltration it's a relevant strategy - although further research is needed to find the best balance between flooding speed and encryption security to minimize damage (also refer to the testing section 3.3).
Finally, additional experiments are required to show the feasibility of certain attacks to SSS. Provided that an attacker, once retrieved some shards, can attempt any brute-force operation on them offline, we don't know how hard it would be to identify shards $s_1, s_2, ..., s_l$ and $s'_1, s'_2, ..., s'_m$ related, respectively, to two bytes b_1, b_2 splitted with the same polynomial f: one approach, for example, could be to find and associate all b_1's and b_2's shards when they don't have any conflicting ones, e.g.

$$s_1 = s'_1 \land s_2 = s'_2 \land \forall i = 2, 3, ..., l. \forall j = 2, 3, ..., m. (s_i.x \neq s_j.x')$$

(as a function can only have one y value for a given x).
Moreover, in favour of the attacker, statistical analysis could help him further and, despite saying that a *secret* is unrecoverable with less than k shards, it could also allow deducing some bytes (secrets), knowing some information on them, e.g. having a more likely expected value.

3.2 Implementation

With Ranflood's architecture in mind, we tried to add our new functionalities being as little intrusive as possible, keeping the project's structure. Mainly, we introduced two new strategies based on Shamir's secret sharing, for two different purposes, named *SSS-Ransomware* and *SSS-Exfiltration*. In this section, we will show the integration of SSS in *Ranflood* from the lower implementation levels up to the $Flooder$ and the *tasks* (3.2.2).

3.2.1 Shamir's secret sharing (implementation)

As we mentioned, *codahale*'s library [4] was imported directly in the repository. Furthermore, partly because we did our first experiments in a separate project, partly to separate the concerns between the flooding execution and the logic of SSS, we built an additional wrapper library for *codahale/shamir*, to make the access from the flooder more straightforward, so that it could only concentrate on IO operations and tasks.

3.2.1.1 codahale/shamir

The imported library is only made of two files: *GF256* and *Scheme*.
GF256 implements the finite field $GF(256)$ exactly as we discussed above, with the efficient operations on its elements, including lookup tables, plus other functions for polynomials over $GF(256)$. The class offers methods to:

- get the degree of a polynomial;
- evaluate it at a given x;
- generate a random polynomial, i.e. get a random array of bytes (a polynomial can be represented as an array of coefficients, and in this context each coefficient is an element of $GF(256)$);
- interpolate a set of points, using Lagrangian interpolation, to retrieve the interpolating polynomial f and return $f(0)$, i.e. the secret.

Some of these functions will be showed in the testing section, for analysis purposes (3.3).

On the other hand, *Scheme* fundamentally implements the two SSS functions of *split*() and $join()$.
split() takes in input a sequence of bytes as the *secret(s)* to encrypt and applies the SSS splitting algorithm to each byte s_t: it gets a random polynomial f_t of degree $k-1$, passing through the point $(0, s_t)$, so that any k distinct points on it can be used to reobtain it (through interpolation), and finally calculates n pairs of coordinates, obtained by evaluating $f_t(x)$ for $x = 1, ..., n$. After having done this for each input byte, the return value is a *Map* (named *parts*) of n entries, as pairs of an integer key and a sequence of bytes: the key indicates each of the $x = 1, ..., n$ coordinates used (since each byte s_t was split using the same n and k), while $parts[x]$ contains a sequence (array) of bytes, where $parts[x][y]$ is the y-th shard of s_t, i.e. $f_t(x) = y$.

Algorithm 1 SSS split function

```
function SPLIT(s[])
    parts ← [n][|s|]
    for t ← 0 to |s| do
        f ← GF256.generate(k − 1, s[t])
        for x ← 1 to n do
            parts[x][t] ← GF256.eval(f, x)
        end for
    end for
    return parts
end function
```

On the contrary, *join*() takes a *Map* of *parts* to calculate and return s. Again, this is done byte by byte, so *parts* has the same format as in *split*() and s is a sequence of bytes as well, each associated with its own interpolation function. Note that one can use *join*() even with less than n points (but it'll only work with at least k).

Algorithm 2 SSS join function

```
function JOIN(parts[][])
    s ← [length(parts)] // outer dimension size
    for t ← 0 to |s| do
        points ← [length(parts)][2]
        for x ← 1 to length(parts) do
            points[x][0] ← parts[x].key
            points[x][1] ← parts[x].value[t]
        end for
        s[t] ← GF256.interpolate(points)
    end for
    return s
end function
```

The following table shows an example of what would be the variables' structure when applying a *split* or *join* function, on a secret s made of 3 bytes, in order, s_0, s_1, s_2, and using $n = 5$ and $k = 4$. Here, in the *split* phase, for each s_t, a random function f_t is calculated (of degree $k - 1 = 3$), together with the n points $f_t(x)$, for $x = 1, ..., 5$; hence, the returned *parts* contains 5 sequences, each of 3 bytes (being the length of the secret). In the *join* phase, instead, supposing we have at least 4 parts, for example $(1, parts[1]), (2, parts[2]), (3, parts[3]), (5, parts[5])$, it's possible to retrieve, individually, each f_t via interpolation, and then re-calculate $s = \{s_0, s_1, s_2\} = \{f_0(0), f_1(0), f_2(0)\}$.

SSS with secret of 3 bytes, $n = 5$, $k = 4$						
secret	interpolating / random function	parts[1]	parts[2]	parts[3]	parts[4]	parts[5]
s_0	f_0	$f_0(1)$	$f_0(2)$	$f_0(3)$	$f_0(4)$	$f_0(5)$
s_1	f_1	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	$f_1(5)$
s_2	f_2	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	$f_2(5)$

3.2.1.2 sssfile wrapper

The wrapper library for *codahale/shamir*, called *sssfile*, revolves around two classes, *OriginalFile* and *ShardFile*. Given the invertible and symmetrical nature of SSS's *split/join* operations, they can both be obtained from each other, depending on the situation's needs.
OriginalFile can store the information required to split an existing file, namely: the SSS parameters n, k, plus an additional *generation* (that we'll see later); the file's path and hash; for its shards, a list of either *parts* - i.e. their byte contents - or paths where they were written.
ShardFile, on the other hand, makes it possible to interface with a file where a shard was written. In fact, a shard file's content is organized in precise sections, in the following order:

A shard file's sections		
	content	size
0	fixed header signature, chose arbitrarily ($0x123456789ABCDEF0$)	$8B$
1	n	*int*'s size ($4B$)
2	k	*int*, $4B$
3	generation	*long*'s size ($8B$)
4	original file's hash ($SHA-1$)	$20B$
5	this *part*'s hash ($SHA-1$)	$20B$
6	length of the original file's path	*int*, $4B$
7	original file's path	variable
8	this *part*	variable

Note that, until the length of the original file's path, all sections have fixed size. The purpose of each will be made clear in the rest of this thesis, through its concrete usage.

Original files and shard files can be obtained, from *sssfile*, via its access classes *SSSSplitter* and *SSSRestorer* (respectively for the *split* and *join* operations).

3.2.1.2.1 SSSSplitter

An *SSSSplitter* can be instantiated with fixed n and k, and later used to return the *OriginalFile* object relative to a given path, of which it immediately obtains and stores

the *shards* using the *codahale/shamir* library. The original file then allows to retrieve the content (in bytes) of each of its *shards*, so that one can write it to a file: this is done with an iterator, so all *shards*' contents don't have to be created all at once. Note that, with the said approach, changing the parameters n, k implies creating a new $SSSplitter$.

The *generation* field is used to distinguish between splitting sessions, so that the same original file can be encrypted multiple times even reusing the same n,k: it follows that it should never be repeated, so a timestamp is used at the moment (as nanoseconds since the epoch).
In fact, when applying SSS to the same file more than once - either with the same or new n,k -, each time new, random polynomials are chosen to divide it in *parts*. This would represent a problem when trying to restore it: if *shards* originated from different splitting sessions are mixed together, by *joining* them one would obtain back different polynomials than the ones used to encrypt the original bytes, and thus wrong *secrets*. While one wouldn't put together *shards* from different original files - provided that, in our model, we include the original hash inside each *shard* -, neither files *split* with different n and k - again, because we include them in each one -, the *generation* parameter is also needed for the additional flexibility of being able to reuse all combinations of these parameters. Using a timestamp as *generation* value also allows to bypass *daemon*'s restarts: timestamps will be different anyway between *daemon* lifecycles.
Finally, an original's file hash is taken as parameter in the creation of an $OriginalFile$ object, given the complexity of computing an hash, and the fact that $Ranflood$ often already does that - as we will see.

Noticeably, the $SSSplitter$ doesn't perform any unnecessary expensive operation, as, beyond receiving the hash in input, also avoids any I/O operation, requiring also the original file's bytes content. Such choice aims at optimizing performances in conjuction with the rest of the *Ranflood* software, which already takes care of such computations. The same can't be said for the $SSSRestorer$, where we care less about efficiency, not being to execute in a critical moment, but also because it works differently, more autonomously rather than in conjunction with the whole *Ranflood*'s architecture.

3.2.1.2.2 SSSRestorer

When trying to restore a directory's content, an $SSSRestorer$ can be instantiated using its path. Indeed, this phase requires an initial, thorough scan of all the files in contains, recursively, as *shard* files related to the same *splitting* operation can't be trivially found and put together, because their contents have to be checked first; at the same time, this is a more flexible method, allowing, for instance, to change a *shard*'s filename or even path (we'll discuss this in the conclusion, not being our focus).
So, a custom structure that extends a $LinkedHashMap$ (i.e. an $HashMap$ with eased sequential access) is used to sort files, saving each group of related shards in a different $OriginalFile$ object (as we said, this can contain a list of its *shards*' paths). In fact, when a *shard* is recognized - using its header signature -, it's "added" to the structure in a custom way, identifying its original file through an hash code that takes in account both the original file's hash (which was saved in each *shard*) and the generation (so that

there can be multiple $OriginalFile$s, for the same path but different splitting sessions). The said hash code is thus the same used in the (linked) hash map to index its elements (original files).

To recap, the scanning must be performed all at once initially, for a given directory; of course it could take quite some time, for large file structures. Afterwards, symmetrically to the splitting phase, one can use the $SSSRestorer$ to iterate through successfully retrieved original files and their content, so that they can be written back. Since an original file also contains references to its shards, one can also remove them, being useless after their source file has been restored.

One could also notice that it is only required to find k *shards* for each original file; unfortunately we don't have a smarter way to group *parts* than to visit and look into each one's *sections*, especially when wanting to be flexible and allow *shards* to change filename or path. Also, a thorough scan is able to report on the whole situation, showing statistics on found shards and making it possible to later remove them.

3.2.2 SSS in Ranflood

Now that we've discussed all the theory and the lower level implementation details, we can dive deep in the integration of SSS with *Ranflood*, which, as we said, is accomplished through the two new flooding strategies $SSSRansomware$ and $SSSExfiltration$.
From the client's point of view, they appear exactly like the others, as choices for a flooding strategy; the *settings.ini* file (for the *daemon*) also has two new sections. However, in the *daemon*'s implementation, they actually refer to the same $SSSFlooder$ class implementing the $Flooder$ interface, just with different parameters.
Additionally, given the property of SSS of making it possible to reassemble a *secret* (file) from its *parts*, the SSS flooders provide the option to delete a file once it's been *split*. This is particularly useful against exfiltration, so that the attacker can only retrieve it by stealing enough related *shards*, while it's only counter-acting with crypto ransomwares, against which our only *mitigation* measure is creating as many files as possible. Hence, by default this parameter is set to true and false, respectively for *exfiltration* and *ransomware* mode.

3.2.2.1 OnTheFly flooder

$SSSFlooder$ was initially branched from the $OnTheFlyFlooder$, which, given a directory path, uses a *snapshot* (specified by the *client*'s command) to recursively create a $WriteCopy$ *task* for any of its files which were not corrupted (i.e. having still the correct checksum).
The existence of a previously taken snapshot is hence a prerequisite, and it allows to skip the copy of already corrupted files in favour of the good ones, to maximize the chances of their restoration. The *snapshooting* phase, through the *daemon* parameter $ExcludeFolderNames$, can also be configured to skip some directories, so that, if a

checksum isn't saved for them, neither their files will be used for a *flooding*, not having a *snapshot* entry.
Lastly, a single $WriteCopy\ task$, as the name indicates, creates exact copies (three, to be fair) of a file, with a filename obtained from the original one by adding a random adjective out of about 1000 present in a dictionary (the *Nomen est Omen* library [13]). Considering that, like we said in 2.1.1, such tasks are repeatedly added to the $TaskManager$, after some iterations a copy with any possible adjective should be created (if no error occurs, e.g. end of memory).

3.2.3 SSS flooder

$SSSFlooder$ uses almost the same directory inspection as $OnTheFlyFlooder$, changing mainly the tasks' behaviour.
In first place, it inspects the input directory recursively, reading each file's content and using it to create a $WriteSSSFileTask$. Each task, then, is responsible for *splitting* it and writing each *shard* to a new file.

The $SSSFlooder$ main loop, differently from the $OnTheFlyFlooder$, also works on a file if a snapshooter was not specified, or if it doesn't have a checksum available for it: due to the self-contained nature of the SSS *shards*, in fact, it's not necessary, as their sections already provide the needed functionalities, even including the original file's hash (as we saw earlier). For this reason, the folders excluded from the flooding (specified in *settings.ini*), here, are checked befor starting the recursion inside a subdirectory of the current path, as the *snapshooter* can no longer be relied on for them.

Additionally, as mentioned, the $SSSFlooder$ permits to delete a file once it has been *split*. After its content has been read and passed to the newly created SSS task, a new $RemoveFileTask$ is created to handle its removal. This is a single-use task (more in 3.2.3.1), i.e. it's only executed once, in order to avoid unwanted behaviours. A normal task, in fact, would be run cyclically, so, in this case, any newly created file with the same name would be later deleted; this could interfere with any other program still running on the infected machine and creating files, as well as with our own flooders, since each strategy has some way of generating filenames which could yield the same one multiple times (especially when combining them).
A limitation of using our single-use tasks is that, in the current implementation, they aren't executed again in case of error: since one should consider many possible reasons for it to occur and different solutions as well, we chose to give priority to avoiding the said conflicts.
We actually considered the option to recreate an original file from inside a $WriteSSSFileTask$ when not enough *shards* aren't written successfully; the main problem with it is that a write task's execution will still be retried sooner or later, but a single-use task won't, meaning that the removal won't happen again on the re-created file in case of successful *split*, defying its purpose. In a more accurate scenario, actually, since the removal task is added and thus launched after the *splitting* task, in case of error of the latter the file will already be present at the moment of recreating it, but it will be deleted soon after. Finally, due to their repetition, a *splitting* task could recreate the file even if it had al-

ready been *split* successfully in a previous iteration, so a more sophisticated mechanism would be needed to track such cases anyway.
Moreover, a common case in which writing a *shard* throws an exception is when the disk is full: is such case, clearly, neither recreating the original file would be possible.

The following pseudo-code simplifies what we just described:

Algorithm 3 *SSSFlooder* main loop

```
sssSplitter ← SSSSplitter
function createTasks(path, excludedDirs)
    if path.isDirectory() then
        if path ∉ excludedDirs then
            for file ∈ path do
                createTasks(file)
            end for
        end if
    else
        bytes ← readFile(path)
        signature ← getSignature(bytes)
        if not usingSnapshooter or (snapshooter.getSignature(path) == signature) then
            createSSSTask(path, bytes, signature, sssSplitter)
            if removeOriginals then
                createRemoveTask(path)
            end if
        end if
    end if
end function
```

When run, a *WriteSSSFileTask* *splits* a file into its *parts* - with *sssfile* - and immediately writes them to new files. The writings of all shards relative to a file are thus executed inside the same task.
Shards filenames are obtained from a filenames generator singleton class, providing a static method to get (almost) unique names. Specifically, the method, given a filename with or without an extension, adds a numerical counter to the end of the filename - or before the extension, if present -, and increases it as long as the resulting filename is already associated with another file:

```
function getUniquePath(path)
    name ← path.substr(0, path.indexOf('.'))
    ext ← path.substr(path.indexOf('.'), path.length())
    repeat
        counter ← counter + 1
    until file(name + counter + ext).exists()
    return name + counter + ext
end function
```

The counter, being of type *long*, allows $2^{64} \approx 2 \times 10^{19}$ distinct filenames.
The main concern with SSS is to be certain that all shards get written to different files, or at least k, so that the *secret* is still retrievable. Using a static class for all SSS tasks ensures their names don't overlap. While some other entity could overwrite the created shards, such conflict won't presumably happen by hand of another strategy - e.g. executed in combination with SSS -, since others put a random adjective at the end of a filename, instead of a number.
We don't handle, though, the case where the counter overflows back to 0. At least, checking a personal computer of ours relieved us, since we found out it only contained, in total, about 2×10^6 files (running on *Debian 12*). When trying to understand the maximum, approximate amount of *shards* that might be created, the number of found files should fundamentally be multiplied by n - how many times each one is *split* -, which, using the maximum $n = 255$, yields 5×10^8 *shards* in total.
There are other variables though: each SSS task is executed cyclically until the flooding is stopped, always adding new files, and also *Ranflood*, being a daemon and thus thought to be run indefinitely without being restarted, might start many floods in its lifecycle. Furthermore, running more floodings in sequence or at the same time would result in continuously copying and *splitting* the generated files as well - unless *snapshots* are used, which only allow to operate on those files whose checksum was taken in advance. On the other hand, a flooding isn't so fast and won't likely work so much, plus the specified excluded directories could make ignore many files.
In short, we considered our choice adequate for our experiments, knowing anyway that, in case of necessity, the implementation could be trivially modified to extend filenames' limits, e.g. with longer counters (more numbers put in sequence).

3.2.3.1 On structural choices

While deciding on how to integrate SSS with *Ranflood*, we considered the already existing software's structure and tried to adapt ourselves to it and to change it as little as possible. After having understood the *proactor*'s (2.1.1) execution flow, our major concern was on the best place to *split* a file, write its *shards* and eventually delete it. Likely, IO operations had to handled inside tasks, to exploit its main advantage of managing multithreading.

Our first approach has been to *split* the file inside the $SSSFlooder$ loop and pass each of its *parts* content to a different $WriteFileTask$ (not $WriteSSSFileTask$). This was the most intuitive, as it imitated more the $OnTheFly$ flooder. However, while the last one

always writes the same content, by making copies, the first one creates a different content for each *shard*: considering (as we saw in 2.1.1) that a task persists in main memory together with its parameters (particularly, the content to write for a $WriteFileTask$), if *On-the-fly* already requires B bytes of main memory in order to perform a flood from copies of B bytes, this implementation of SSS would require $n \times B$ bytes. For instance, what $On-the-fly$ does with 3 GB of data could result in SSS using 300 GB of RAM, clearly leading to out-of-memory exceptions or to being forced to reduce its range.
Nonetheless, we wouldn't like to change the flooder's behaviour to overcome such limitation. In fact, as discussed in *Ranflood*'s papers [9] [10], this is a wanted trade-off between memory and mitigation effectiveness: without caching its content, one would have to read a file each time, but this - or any other already created copy - could have been encrypted by the ransomware in the mean time, so duplicating it would still cotribute to the flooding but without a possibility of restoration.

Consequently, we opted for "shifting" the whole chain of execution towards the tasks. In the current implementation - which we already explained in the previous section - each original file is assigned a single task, handling both the *splitting* and the writing of its *parts*. In this way, the memory occupancy is the same as for *On-the-fly*, i.e. equal to the total read bytes.
We also changed the handling of filenames. In the first case, each was chosen in the $SSSFlooder$ and passed as task's parameter, implying that a *shard* was always written with the given content to the given path. Considering that a task is scheduled cyclically (2.1.1) - with the same parameters -, the behaviour was similar to that of *On-the-fly*, in the fact that it continuously rewrites the same file - which is positive for restoration if it had been encrypted by the ransomware. Now, each time a file's task is executed, this is also *split* in *parts* differently - since it happens by calling $sssfile$, which generates them randomly with the given n, k - and filenames where to write them are different as well. While being the most intuitive approach here (to use the same filenames, one would have to cache them), it also has some interesting implications.
At the moment, when a file's task is repeated, new *shards* will be crated in addition to those of the previous *generation*: while, then, a file - if not encrypted in the meantime by the *malware* - was being overwritten with its same content, now new ones are created, contributing to the flood, feeding the *ransomware* with more work and creating more content useful for restoration. On the other side, a new execution requires repeating SSS operations, which are actually quite expensive (for performance, refer to 3.3.2).
However, decentralizing the *splitting* towards the tasks also allows the main loop to more quickly iterate through files and soon create their tasks, delegating their later, computationally-heavy execution to the IO threads. Reading immediately as many non-corrupted bytes as possible also grants an advantage on the ransomware, since they get safely cached in main memory.

The last question to discuss in this paragraph is about the original file's removal. As mentioned in 3.2.3, the $SSSFlooder$, right after the SSS tasks, creates a new task for it, which is single-use for the said reasons.
Ranflood didn't implement single-use tasks, so we had to add them, possibly in the less intrusive way. Here we include the main execution of the *TaskManager*, to better explain our slight modification later:

```
flooder
tasks ← newList()
function getNextTask
    tasks.addAll(flooder.getTasks())
    return tasks.remove(0)
end function
function signalExecution
    if flooder ≠ null then
        current ← getNextTask
        getNextTask.run()
    end if
end function
```

This is a simplified version, since the $TaskManager$, to handle multiple flooders at the same time, cyclically takes from each one all of its task and adds them to its schedule. In the given pseudo-code, then, there's a single flooder's instance, and a list of tasks. The $TaskManager$'s execution is started through its $signalExecution$(): at every iteration, the tasks created by the flooder (or the flooders) are retrieved, and the first one in the queue is run.

What we changed was also retrieving single-use tasks together with the "normal" ones, so that they get inserted in the same scheduling. $getNextTasks$(), so, became:

```
function getNextTask
    tasks.addAll(flooder.getTasks())
    tasks.addAll(flooder.getSingleUseTasks())
    return tasks.remove(0)
end function
```

At the same time, each flooder now has to expose two different methods, one for retrieving the old tasks and one for the single-use ones. While, for compatibility, the old strategies return an empty list, $SSSFlooder$ stores his in two different, internal lists, and the single-use one gets cleared anytime the $getSingleUseTasks$() is called:

```
tasks ← newList()
singleUseTasks ← newList()
function getSingleUseTasks
    res ← singleUseTasks
    singleUseTasks.clear()
    return res
end function
```

Additionally, $getTasks$() (and $getSingleUseTasks$()) initially had as return type a list of $WriteFileTask$, but we changed it in a list of $FileTask$: now it's more general but

didn't have any implications in the code; we did so since the $RemoveFileTask$ inherits from $FileTask$ but not from $WriteFileTask$, purely for coherence in naming reasons.

3.2.3.2 SSS snapshooter

Its snapshooter, the $SSSSnapshooter$, actually implements its methods pointing to those of the $OnTheFlySnapshooter$, and even from the point of view of client and daemon's settings it doesn't exist and they're considered to be the same. In fact, its same implementation, based on files' checksums, is what we needed, with the implied advantage of skipping operations on already corrupted files; however, the other reason the $OnTheFly$ strategy needed it, i.e. for later restoration, is not of our concern, since our shards already keep the original file's hash.
Shards actually only needed something to unequivocally group together those belonging to the same splitting session, so using the file's hash, beyond working for this purpose, also gives the benefit of making them self-contained. As pointed out in the *Ranflood* papers [9] [10], the single point of failure constituted by the snapshots database files is among the software's possible improvements, hence our decision to also address it while implementing SSS. The risk, in fact, would be that of losing the snapshots files to the ransomware, debilitating the flooding counter-action.
There are also other sections, in a *shard*, which could act as identifiers, but each could suffer from some rare cases, so keeping them all makes the model more robust - while also providing other benefits. Specifically, *generation* is currently a timestamp, but we aren't sure its value is always different, especially with multithreading; the file path, instead, isn't a very stable option, as it could create conflicts in case of file's moved around, using the same name in turn: this could also happen on behalf of the ransomware, which often also changes a file's name or parent directory after encrypting it, or of a flooder, which surely creates files (or changes them, as the SSS one does).

3.2.3.3 SSSFlooder parameters

The options available in *settings.ini* for the two strategies are identical, but can be configured separately. Currently, they are: *ShardsCreated*, that is, n; *ShardsNeeded*, k; *RemoveOriginals* for removing original files after having split them; *ExcludeFolderNames* for the directories to ignore. n and k, at initialization, are clamped to the allowed limits, i.e. $2 \leq k \leq n \leq 255$.

The reason for having two separate SSS flooders is that, although offering similar functionalities, in the critical moment of a ransomware infection it could be useful to have two different solutions ready to use, whose parameters have already been tweaked for different scenarios according to one's defense strategy (either manually or with the default settings). Crypto and exfiltration ransomwares have truthfully distinct goals and so distinct weaknesses: while the former should be slowed down by both creating as many files as possible and keeping disk and cpu busy, the latter, beyond that, should also be retained from gaining any information on what it's trying to steal.
Aside from the *RemoveOriginals* option, then, lower k is preferable against a ransomware,

so that one can restore the original file even when many shards have gone corrupted, actually collapsing to the $OnTheFly$ strategy, whereas larger k, against exfiltration, reduces attacker's chance to reassemble the stolen data.

To sum up, default options reflect our observation and thus are: for $SSSRansomware$, $n = 200$, $k = 2$ and don't remove original files; for $SSSExfiltration$, $n = 150$, $k = 6$ and remove original files. Their choice is explained in the testing section (3.3).

Since these parameters are quite technical and default ones are defined, we think that the average *Ranflood*'s user is unlikely to want to change them, especially after the default ones will be assigned through more accurate experiments. For this reason, we also modified the *IniParser* class provided in the software's repository, responsible for reading the *daemon*'s configuration from the *settings.ini*, to make sure that it allowed not to specify a parameter: this only required to make it return *null* for a missing parameter instead of throwing an error. Such change doesn't seem to have any other side effects, since every other flooder already checked not to receive *null* values, while we specifically implemented ours to use the default values in such case.

As a final point, despite our guidelines to pick optimal parameters, further research may reveal which combination n and k best fits each of the two distinct scenarios: e.g. one could investigate which is the best balance between concentrating on splitting too much the same file, with a large n, rising the risk of malware accessing many others still intact, and splitting each too carelessly, thus covering a wider area but allowing the infection to take out an original file more easily.

3.2.3.4 SSS in fileChecker

To recap, *fileChecker* is the tool, complementary to *Ranflood*, to restore an environment after a flooding and an infection. It initially featured the two commands *save* and *check*, to be used in sequence, on a folder, to first retrieve the checksums of files and then check which ones weren't corrupted or can be re-obtained thanks to the copies generated via *On-the-fly* or *Shadow*.

To implement the *restoration* phase of the SSS flooder, we added the subcommand *restore*. As parameters, similarly to the other two, it requires paths where to store the produced logs, and can use a checksums file (created by *save*), although it's not necessary, since shards are self-contained and already include their original hash, as we saw. Additionally, one can optionally specify to remove shards files once they've been used, and to receive more (debugging) logs, produced directly by *sssfile*, optionally printable to another logfile.

restore doesn't perform anything relevant, besides handling and collecting logs. Most of the logic is inside *sssfile*, exactly as we showed in 3.2.1.2.2. The command, in fact, first instantiates an $SSSRestorer$ and calls its scanning method on the input directory, then iterates on the $OriginalFile$ objects it retrieved: for each, writes it to a file and, in case of success and if the user requested it, deletes its *shards*. Like for the *splitting*

phase, *sssfile* doesn't perform any IO operation itself, but only returns the content of a file; unlike the former, though, it reads files in the scanning phase.

The most relevant part in the *restore* code is that which handles the writing to files, avoiding conflicts and first checking if the retrieved content is correct.
While the reassembled file's hash is immediately checked inside $sssfile$ against the one saved in its *shards*, *restore* also compares it with the one in the checksum file, if available. A case in which this could be helpful is when files have been changed - not by the ransomware - after the snapshot, or when they were deleted but new ones with the same names were created later. On the contrary, the same happens when a file has been encrypted, and *SSSFlooded split* it anyway because a *snapshooter* wasn't used and it wasn't able to know whether it was still valid or not, in that moment. This is a double-edged perk of $SSSFlooder$, and the user will have to find out himself if some file has become useless because encrypted.
The restored file is created with a different name in some cases, specifically: if its signature doesn't match the one in the checksum file, but it does with the one saved in the shards, indicating the problem with just described; if a file with such name already exists, as it could have just been created by the user - after the attack - or be relevant anyway. However, if the same-name, already existing file has the same signature of the restored one, the latter is just ignored (and, in case, its shards deleted), meaning that it had already been restored or was never lost at first.
The creation of an unique file is handled through the same $getUniquePath()$ function (3.2.3). In this case, the risk of making its counter overflow appears much reduced: $fileChecker$ is a single-execution command and not a *daemon*, and original files are only in proportion of $1 : (n \times generations)$ with *shards* (even though *ransomwares* could mess them up) - where *generations* indicates the number of generations in which the specific file was *split*.

Finally, every relevant information is collected and printed in a report file, in the json format.

3.3 Testing

The strategy we developed is quite strange, compared to the already present ones: it adds some bytes originating from original files unlike *Random*, but they aren't proper copies of them, unlike *On-the-fly* and *Shadow*. *SSS* actually performs some kind of encryption before writing contents, differently from straight copies, so we doubted since the beginning of our research its performance would be any similar to other strategies.

And in fact, tests reveal they aren't at all. They were run on our personal machines, allocating 6 GB for the Java Virtual Machine. Our simple testing session consisted in using the same *Random* flooder to create random files in a folder, then try and run the following flooders, each four times:

- $SSSRansomware$ using snapshots;

- $SSSRansomware$ without snapshots;
- $SSSExfiltration$ with snapshots;
- $SSSExfiltration$ without snapshots;
- $OnTheFly$ (of course with snapshots);

The SSS ones used the default parameters. *Random* was only run for 2000 ms and each flooder for 500 ms, as our machine didn't have enough memory to perform larger scale tests.
In such conditions, while $OnTheFly$ produced a few thousands new files $(2000 - 4000)$ from a starting set of $6000 - 8000$ (variable numbers through tests), SSS could only run a few iterations (sometimes only 1) and generate a few hundreds $(200 - 600)$: a single *split*, in fact, could require about 1 second, for files of more than 500 Kilobytes.
Actually, we weren't using yet the parameters found out through the experiments we are going to show hereby, but, as said, *splitting* a file still requires some time additionally to the IO overhead, while copy-based strategies don't have to perform any additional operations.

Even without running proper tests, in case of success, their reliability could still have been supported by the more formal ones run for *Ranflood* [9].
However, being SSS much inferior in flooding efficiency, we can't apply them; instead, also with the limited time we had, we concentrated our efforts on calculating the absolute efficiency in terms of bytes created per second, and on trying to understand if our strategy could still be applied to some cases.
In the following subsections, then, we go even deeper into the implementation to analyze the computational efficiency of our code; then, we use the acquired knowledge to tweak SSS parameters and finally compare them to possibly real scenarios.

3.3.1 Computational analysis

Any flooding strategy, after performing some operations to manipulate data, has to request the IO access to write files, and also *ransomwares* need it in order to read their contents and then write it, after having encrypted it - or, in case of a pure *exfiltration ransomware*, it only reads it; thus, to compare our flooder most of all with *ransomwares*, we can leave out the constant overhead of IO and focus on comparing their respective "some operations" parts.

In order to do that, we need to recall the implementation of the *codahale/shamir* library, so we can calculate its computational efficiency.

The *split* algorithm (1) features a loop on each byte of the secret, inside which two operations are performed: a call to $GF256.generate()$, to generate a random polynomial of degree $k - 1$, and another loop on n, to evaluate it on each *shard*'s x coordinate, for the current *secret*'s byte.
Hence, we hereby show the pseudo-code for $GF256.generate()$ and $GF256.evaluate()$ (which uses *Horner*'s method):

Algorithm 4 polynomial generation in $GF(256)$

function GENERATE($degree, secret$)
  $f \leftarrow byte[degree + 1]$
  repeat
    $f \leftarrow randomBytes()$
  until $degree(f)! = degree$
  $f[0] \leftarrow secret$
  return f
end function

As function to get a sequence of random bytes ($randomBytes()$), the one in *java.security.SecureRandom* was used. It is linear in the bytes array size, since (according to Java's documentation [1]) it gets a random number for each element, using the following constant function:

$$(seed \oplus 0x5DEECE66DL) \quad \& \quad ((1L << 48) - 1)$$

As we can see, this is called again until the fetched random polynomial is of the correct degree; since it suffices that the most significant byte is not 0, "in most cases" (assuming the random function has uniform distribution) the loop will only be run once (precisely, with a probability of 255/256). Hence, $generate()$ has complexity

$$O(degree)$$

For clarification, before returning, *generate* sets the least significant byte to *secret*, indicating that the polynomial has value *secret* in correspondence of $x = 0$ (intersecating the y axis), as we saw through the SSS algorithm.

Algorithm 5 polynomial evaluation in $GF(256)$

function EVAL($p[], x$)
  $result \leftarrow 0$
  for $coeff \leftarrow p.size$ TO 1 **do**
    $result \leftarrow (result * x) + coeff$
  end for
  return $result$
end function

$eval()$ is also linear, since it only iterates through constant mathematical operations: its complexity is

$$O(p.size)$$

We recall that mathematical operations in $GF(256)$ are very efficient, and actually all constant.

To put it all together, in *split* (1) the $generate()$ function is invoked with degree k, so its complexity is $O(k)$, and $eval()$ is $O(k)$ too, since the generated polynomial has degree k (and so is the corresponding representation in bytes coefficients), meaning that

the inner for loop, run n times, has complexity $O(n \times k)$. Iterating $|s|$ times (size of the secret s, in bytes), the complexity of *split* is

$$O((n \times k + k) \times |s|) \tag{3.5}$$

For simplification sake, however, we will usually only consider the complexity of a single byte:

$$O(n \times k) \tag{3.6}$$

The $+k$ addition was also simplified, being irrelevant in the *big-O* notation - although it was worth mentioning, as it may be significative in a later calculation.

For completeness, we also analyzed the *join*() function (2), which reassembles a set of *parts*. It has a for loop too, running, for each of the *secret*'s bytes, a linear inner loop on the *part*'s array's size, followed by the call to $GF256.interpolate()$; we omit the implementation of the latter, being the Lagrangian interpolation, and affirm that it takes in input a list of points and has a complexity of $O(points.size^2)$, due to its nested double loop.
In *join*(), *interpolate*() is invoked using a *secret*'s byte's *shards* - for each byte in *secret*. Considering that the number of parts, for a successful reconstruction, needs to be between k and n, the complexity, for a single byte, is

$$o(k^2) \cap O(n^2) \tag{3.7}$$

This calculation immediately indicates that a very simple optimization could significantly increase $fileChecker$'s *joining* speed: cutting the array of retrieved *shard* files to only k elements (instead of using all of them, which could be n, in an ideal case) could significantly impact the performance, especially with an high imbalance between the two parameters.
Although not being as important as the mitigation phase, where being faster than the *ransomware* is the priority, we included this improvement in our implementation when noticed it.

For what concerns the *split* phase, it can be said that its complexity depends on k and n as well: the main intuition, enforced by our tests - shown in the next section (3.3.2) - is that their product has to be kept as low as possible, so when preferring to increase a parameter one should try to decrease the other one.

Finally, this analysis underlines the difference between using $GF(256)$ or another finite field of order not multiple of a byte's size (2^8). The latter's modulo operations, indeed, unable to be naturally blended with the computer's architecture, would add a non-constant complexity to each loop's iteration, and increase the general complexity: addition/subtraction wouldn't be simple *xors*, and, most of all, logarithm and exponentiation would require the use of algorithms, rather than tables *lookups* - usually proportional to the finite field's order.

3.3.2 Empirical performance evaluation

In this section we put in practice what we learnt through computational analysis and validate its results through experiments.
Here, we show some sample executions of *split/join*, where we monitored the elapsed time and reported the average speed as the milliseconds required to *split/join* a Kilobyte of data (being time over space, lower is better) - also equivalent to s/MB -; the *joins* were performed using all n *shards*; the last column shows the $\frac{split}{n*k+k}$ ratio (multiplied by 1000 for more readibility), used to find a correlation between the previous analysis and the empirical results. Experiments were conducted trying different combinations of values for n and k, on inputs counting 1000 bytes (1 KB) and averaging each test on 1000 attempts:

Comparison of *join* and *split* in ms/KB using different n,k				
n	k	split (ms/KB)	join (ms/KB)	$\frac{split \times 1000}{n*k+k}$ ratio
255	2	1.666	367.177	3.254
255	10	8.551	367.413	3.340
255	50	61.171	368.945	4.779
255	100	124.589	362.571	4.867
255	255	328.976	368.248	5.039
2	2	0.119	0.195	19.833
10	2	0.164	1.03	7.455
50	2	0.334	15.968	3.275
100	2	0.57	58.951	2.822
200	2	1.069	229.666	2.659
200	3	1.65	231.59	2.736
250	10	8.074	349.274	3.217
50	50	12.098	16.459	4.744
10	10	0.547	1.031	4.973

The *split* operation is the one we focused more on for the said reasons, and the first one we'll discuss. The observed results somehow resemble the computational cost of $O(n \times k)$ (formula 3.6): the first five rows, showing the impact of k with a fixed $n = 255$, highlight an approximately linear dependence on it (e.g. $124.589 \approx 61.171 \times (100/50)$, with $k = 100$ and $k = 50$ respectively, while $328.976 \approx 124.589 \times (255/100)$, with $k = 255$ and $k = 100$), and the same linearity appears in the following five rows, with varying n and fixed $k = 2$, with respect to n. Of course, needing $k \leq n$, the imbalance is "monodirectional", and we can only try to fix one parameter and lift the other until $n = k \vee n = 255$.
Moreover, an equal $n \times k$ product indicates not too dissimilar results: comparing $10, 10$ with $50, 2$ (for n, k) - having the same product 100 - and $50, 50$ with $250, 10$ - $n \times k = 2500$ - shows *split* times not too distant from each other - considering the scale of the whole parameters space, where a *split* could take from 0.119 ms/KB, using $2, 2$, up to 428.976, using $255, 255$ -, although their differences aren't exactly negligible (respectively, 0.547 against 0.334 for $n \times k = 100$ and 12.098 against 8.074 for $n \times k = 2500$).
We didn't dig deeper for the meaning of such small differences, except by trying to

understand the correlation between actual time and expected complexity through the ratio, in the fifth column, calculated as $\frac{split}{n * k + k}$ (multiplied by 1000 for readability purposes), following the more precise cost of formula 3.5 (leaving out the number of bytes, since it's constantly 1 KB).
Intuition suggests these common rules: the ratio is around 3 for not-too-small values of n ($n \geq 10$) and relatively small for k ($k \leq \frac{n}{25}$), around 5 for $n \geq 10 \wedge k \geq \frac{n}{25}$ (e.g. $n = k$) and significantly higher for low n's. The first conclusion, then, (supposing that the computational analysis is correct) are that smaller parameters, implicating less iterations, render the algorithm more subject to minor implementation overheads, whose impact becomes attenuated when considering a larger scale scenario. Indeed, the *big-O* notation itself isn't applicable to too small values, as its own definition is related to then mathematical concept of limits:

$$f(x) = O(g(x)) \iff \exists x_0, M \in \mathbb{R} | \forall x \geq x_0 . f(x) \leq Mg(x)$$

The incoherences of higher k's, instead, are less clear, but presumably also linked to some implementation detail (e.g. arrays allocations, for variable sizes).

The two lessons we can learn here, then, are that increasing both n and k at the same time, of the same amount, quadratically impacts performance, while only increasing n does it linearly, and that using too small values brings unnecessary losses (e.g. $n, k = 10, 2$ only requires 42% more time, i.e. 0.05 more ms/KB, than $2, 2$, with n increased of 500%, whereas $200, 2$ has a fairer increase of 100% with respect to $100, 2$, with n also increasing of 100%). In practice, $n \geq 10$ and $k \leq 10$ seems and optimal choice, not bringing too much implementation overhead an not increasing the costs quadratically with a large k.
Regarding the *join*, instead, we limit to confirm what previously said in 3.3.1. Since we performed *joins* using all n *parts* of a file instead of the only minimum k required, their times are visibly higher than those of *splits*; however only using k would make a significant difference: if you take the example of $10, 2$ (n, k), the join time is 1.03 (much higher than the relative *split* time of 0.164) because 10 *shards* were used, but using only 2, being $k = 2$, would yield the same time observed with $n, k = 2, 2$, i.e. 0.195. In fact, the analytical complexity here is quadratic with respect to the number of used *parts*, and k instead of n constitutes a clearer improvement as far away their values are from each other.

We also cite the official *codahale/shamir*'s results, present in its repository's *Readme* [4]. Using $n = 4$, $k = 3$ and *joining* in the optimal way, with only k *parts*, they report about 0.2 ms to *split* 1 KiB (1024 bytes) and 0.4 ms to *join* 1 KiB. We didn't include our tests with such parameters in the table, but we obtained similar values - anyway, $4, 3$ is about twice $2, 2$ (which we reported), and their times are too.

3.3.3 Practical performance evaluation

In this last section dedicated to tests, we plot the previously showed values, in addition to other ones, to enforce our intuitions more clearly, after defining a new comparison metric.

After evaluating SSS's algorithm performance, the question is how fast are *ransomwares* instead. Well, an *AES* encryption - likely to be used by a *crypto ransomware* - has a linear complexity with respect to the number of input bytes, and our tests in *CBC* mode and with *PKCS5 padding* truthfully revealed a speed around 0.003 ms/KB - taking the average on 1000 results for 100 KB inputs.
An *exfiltration ransomware*, on the other hand, might be limited by the network bandwidth - which, nowadays, can also be of hundreds or thousands of Megabit, or much slower for the unluckier ones.

Despite being scary, *ransomwares* performances can be seen from a different perspective, when evaluated in the context of a flooding. If between the encryption (for a *crypto* one) or exfiltration (for the other one) of one file and another some baits also have to be processed, the effective *ransomware*'s speed, i.e. the speed at which it progresses towards its goal of covering the whole file system, reduces: for instance, if a *crypto ransomware* can process 100 MB/s and the *flooder* can write 90 MB/s - no matter what's its frequency, i.e. if it writes many files once in 10 seconds or, constantly, a file every 0.1 s -, it results that the former can only encrypt 10 MB/s of useful data (among all the copies), so he still makes progress, but at a slower pace; on the other hand, if the *flooder* produced 200 MB/s, the *malware* would be overwhelmed my baits, as more and more of them would be created while he couldn't keep the pace and would only catch a real file by chance once in a while.
The example is clearly quite naive, assuming that the *ransomware* would soon be baited by the generated files, while one would have to consider the probability that it picks a real file or not, in a large file system containing both types; on the other hand, further research could find a way to push the *ransomware* towards precise files, e.g. by giving them specific names, extensions, headers, contents.
Nonetheless, we still deemed this intuition relevant in order to measure our *flooder*'s performance, not having enough time to run more accurate tests for this work: it's still true, in fact, that creating copies faster than the *malware* processes files will continuously reduce its probability of picking real files, as well as flooding more slowly than it will only feed it a bait once in a while. The difference between flooding and encryption (or exfiltration) speed - be it positive or negative - can likely determine a positive or negative acceleration in the *ransomware*'s progression rate.

For the SSS *flooder*, we can define the speed at which *shards* are created - and thus bytes written - as

$$absolute_split_speed = (\frac{1}{split_speed} \times n)\text{MB/s} \tag{3.8}$$

While the previous *split speed* (in the previous section, in the table) only indicates the frequency at which a single *splitting* is completed, the so-defined *absolute split speed* focuses on the average number of bytes written (while creating *shards*), no matter the frequency, equally rewarding also larger parameters values which obviously require longer executions, but, at the same time, produce more files all at once. We precise that, while the previous *split* speed was expressed as ms/KB (to highlight the single execution duration), we now take the reciprocal, in order to use the speed in terms of MB/s (equivalent

to KB/ms). The intuition of multiplying the speed by n comes from the fact that, for 1 KB of processed data (the secret), n KB are produced (one for each part): in fact, each part is of its same size - besides the constant *shard*'s header, which we didn't count, for simplicity and also because its impact is attenuated with larger files.

The following plots, finally, show how the *absolute split speed* varies with different parameters, by either fixing n or k (as indicated in each caption). The x axis indicates the varying parameter (n or k) and the y indicates the absolute splitting speed; the sample points are those in red, of which are highlighted the x coordinates, and over each one a label indicates its own *relative split speed* in MB/s (i.e. the reciprocal of the *split speed* in the table).

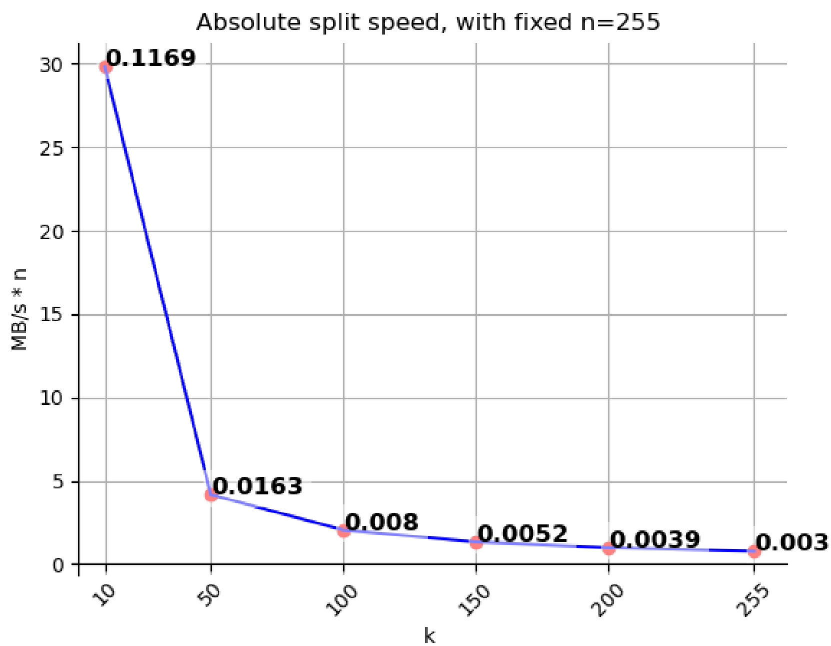

Figure 3.1: Absolute splitting speed, with fixed n=255.

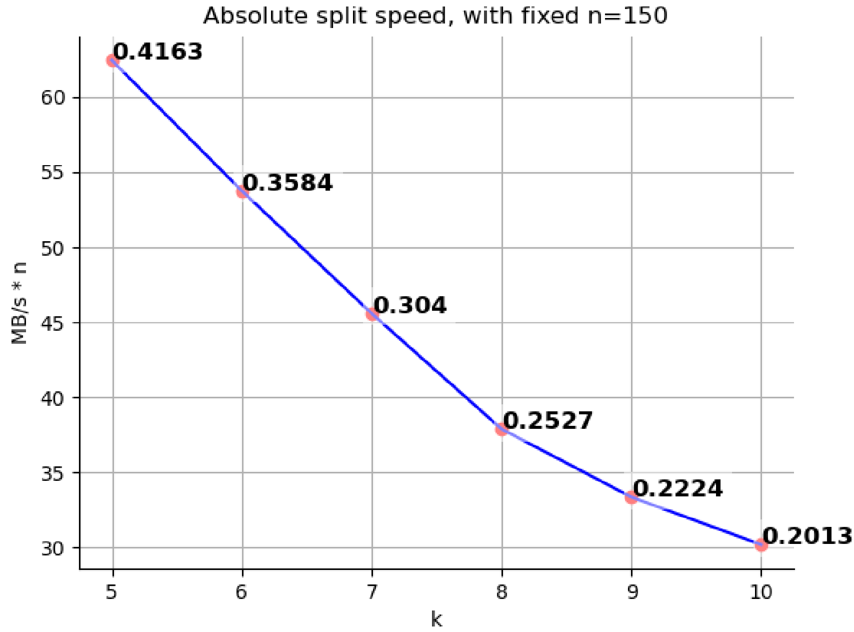

Figure 3.2: Absolute splitting speed, with fixed n=150.

As we saw with the table, while n is fixed, increasing k reduces the *relative split speed* - being the reciprocal of the previous *split speed* -, whereas the displayed labels show decreasing values (in MB/s) as k increases. Moreover, the two plots, testing many values from 5 up to 255, seem to underline an hyperbolic decreasing. A possible explanation deriving from what we saw is that the *split speed* (ms/KB) is a quadratic function (having complexity $O(nk)$), and taking its reciprocal (for the *relative split speed* we consider now) multiplied by n (for the *absolute speed*) yields a function in the form of $\frac{1}{k}$ (hyperbola):

$$\frac{1}{nk}n = \frac{1}{k}$$

From such comparison, it seems that choosing a k as low as possible is better.

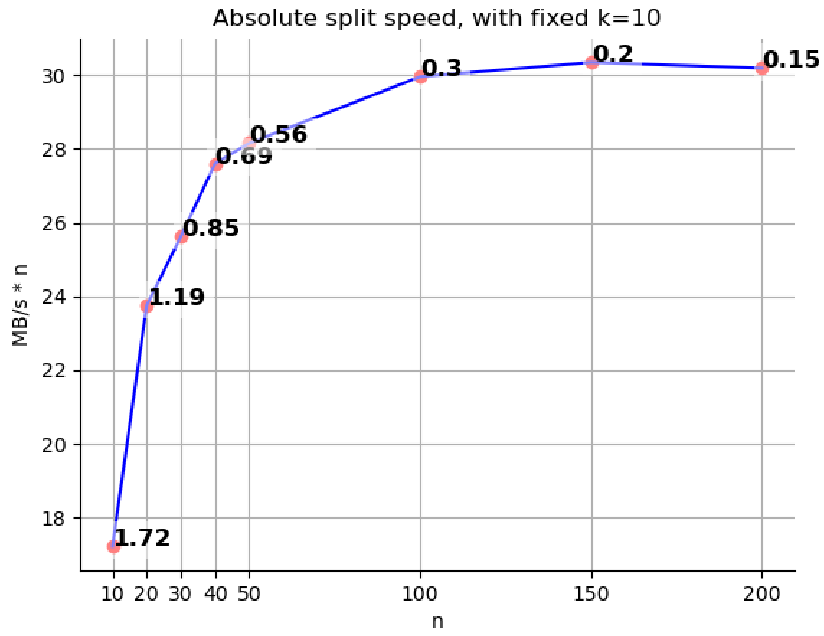

Figure 3.3: Absolute splitting speed, with fixed k=10.

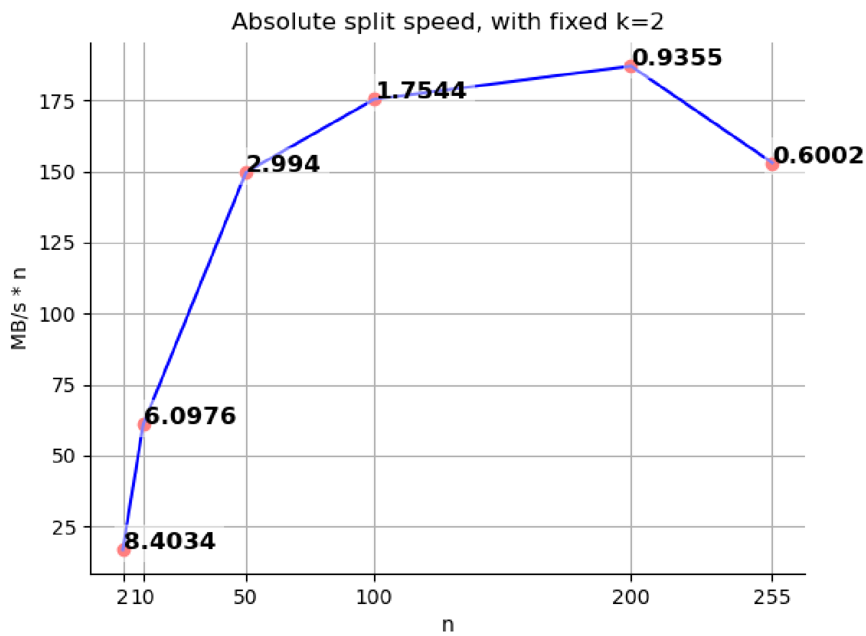

Figure 3.4: Absolute splitting speed, with fixed k=2.

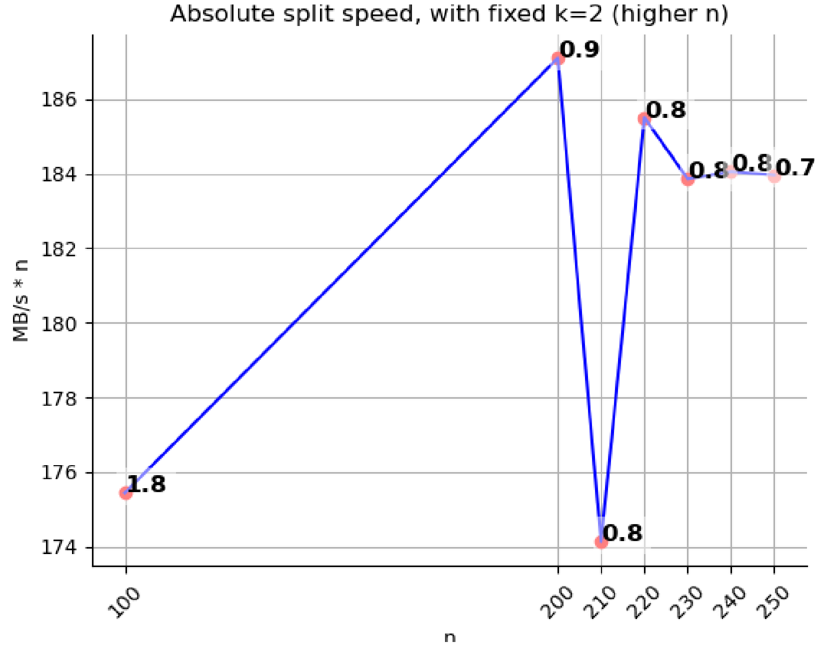

Figure 3.5: Absolute splitting speed, with fixed k=2 (higher n values).

On the other hand, it's impressive that, with fixed k, increasing n results in more bytes written over time, although the peak performance is for n around $150 - 200$ with the tested $k = 2$ and $k = 10$, while the *absolute speed* starts decreasing again soon after. A possible explanation is the aforementioned implementation overhead, which seemingly disappears at the found values of n, while the reasons for the decreasing performance after the peak are unclear.

The order in which plots appear almost reflects the progression of our experiments to find the optimal values for n and k. It's worth mentioning that, while looking for the best performance, we also require suitable parameters for our needs: while n itself is mostly related to the flooding impact, and thus can be more easily tweaked to improve performance, the relation $\frac{k}{n}$ indicates the redundancy of a file's copies. Against a *crypto ransomware*, then, low k increases redundancy and restoration chances, while against *exfiltration* one must also try to reduce the attacker's ability to reassemble victim's *secrets*, using higher values.

After what learnt through the previous section (3.3.2), we wanted to start our experiments by understanding the general impact of varying n with fixed k and k with fixed n. Thanks to the first plot, with $n = 255$ (3.3.3), we soon discovered how badly higher k impacted performance; however, given the previous reasoning, while using $k = 2$ is surely feasible and actually preferable against *crypto* attacks, we also needed to test higher values in order to contrast *exfiltration*.
The measuring of n was hence performed with the two different fixed values $k = 2$ (3.3.3) and $k = 10$ (3.3.3). After having observed the said peak for n around $150 - 200$, we first focused on contrasting *crypto*, by picking $k = 2$ and looking for the exact peak in

absolute speed using steps of 10 for n (3.3.3), but the initial value of 200 was actually the one seemingly yielding better results.
Regarding the *exfiltration* threat, we individuated $n = 150$ as peak and, finally, looked more deeply to find a better k for it. Given that $k > 10$ resulted in excessive performance losses, and $k < 5$ seems to low to contrast it, we tested all the values in between; specifically, the decrease is approximately linear in the range $[5, 8]$, so we also discarded $k = 9$ and $k = 10$. Among the remaining values, we arbitrarily chose $k = 6$ as default value in our implementation: while increasing n seems quite inexpensive and usually convening, such low values for k are anyhow far away from those we expected from *Shamir's secret sharing* at the beginning of our research, but this analysis shows how much rising it impacts performances, and one must find the best possible balance between flooding speed and restoration ease to best reduce an attacker's capacity of obtaining sensitive information on the victim.

To summarize, the default parameters for $SSSRansomware$ are $n = 200$ and $k = 2$, with an *absolute split speed* of 187.1 MB/s, whilst for $SSSExfiltration$ they are $n = 150$ and $k = 6$, with an *absolute speed* of 53.76 MB/s. Of course, given the high impact of k, $SSSRansomware$ is much faster.

Having defined our metrics, retrieved empirical data and found some almost-optimal parameters, we can finally answer the question whether the SSS *flooder* can make any difference.
First, a *crypto ransomware* with an encryption time of 0.003 ms/KB - by calculating its reciprocal - has an encryption speed of 333 MB/s. $SSSFlooder$ could thus, at most (assuming again that generated files have high probabilities of being picked), about half its speed (since $333 \approx 187.1 \times 2$), for an intuitive calculation - or slow it down by 44%, more precisely. Anyway, these evaluations are only for research purposes, since copy-based strategies, not having to perform any additional operations on data rather than writing them, are significantly more effective against *crypto*. Even other improvements for *Ranflood* introduced in this project, such as making the *shards* self-contained, could still be included in the other strategies in the future.
The *exfiltration* speed, instead, would much depend on a network's speed, which is probably very high for a server, while, for a common user, it can be only a few Megabytes, or some tens or even hundreds, for the lucky ones (not so much in this context): the efficacy of $SSSExfiltration$, with a *flooding* speed of 53.76 MB/s, would highly depend on the case. Anyway against *exfiltration*, despite being much slower than other strategies at flooding, the fact that the attacker can't reassemble a secret without enough parts could constitute an advantage in some cases.
For both *ransomware* types, we leave to the future proper tests in a real environment, to verify the correctness of our choices.

Chapter 4

Conclusion

4.1 Summary

To summarize, in this thesis we gave an insight on a the anti-*ransomware* solution of *Data Flooding against Ransomware*, we presented our comprehension of the *Shamir's secret sharing* and of how it can be applied to it, we illustrated the *Ranflood* tool and, finally, we went through the development of our implementation of *SSS* for *Ranflood*.

In the "Introduction" chapter (1), we gave the initial definitions for *ransomwares*, *exfiltration ransomwares*, looked at a few options to contrast them, and specifically *Data Flooding against Ransomware*, and at the open-source implementation of it in *Ranflood*.
In the "Background" chapter (2), we gave our perspective on the objects of our first study to approach this topic: in 2.1 we analyzed more deeply the architecture of *Ranflood* (2.1.1), with its client-daemon structure and *proactor* pattern to manage multithreading, and its complementary tool *fileChecker* (2.1.2), needed in the restoration phase; in 2.2 we gradually introduced *Shamir's secret sharing*, from the basic concept and goals (2.2.1), the algorithms (2.2.2) and then explaining why and how to add modular arithmetic (2.2.3), with finite fields.
In the "Our contribution" chapter (3), we started from the background knowledge to build our model of flooder based on *SSS*: in 3.1 we depicted the precise scheme of *SSS* we used to handle files (3.1.1), based on the finite field $GF(256)$ for its high efficiency on bytes, and listed some security concerns of ours regarding it (3.1.1.1).
In 3.2 we put all in practice: in 3.2.1 we considered a few options on *SSS*'s implementations and showed our classes implementing it, divided in the external repository *codahale/shamir* (3.2.1.1) for the "rawer" parts and our *sssfile* wrapper for handling more complex files structures (3.2.1.2), both for the *splitting* (3.2.1.2.1) and the *restoration* phase (3.2.1.2.2); in 3.2.2 we started from the provided *On-the-fly* copy-based flooding strategy (3.2.2.1) to implement our $SSSFlooder$ (3.2.3), analyzing its optimal structure (3.2.3.1), its limited usage of *snapshots* (3.2.3.2) and the parameters through which it can be configured and how they impact the execution (3.2.3.3), and its complementary *restore* command for the *fileChecker* (3.2.3.4).
Our implementation was concluded by some informal tests and observations, in 3.3: in 3.3.1 we performed some computational analysis on the underlying SSS operations; in 3.3.2 we validated it through some sample executions; in 3.3.3 we combined analysis and

practice to find optimal parameters, and reasoned about the possible performance of the $SSSFlooder$ in real scenarios.

4.2 Future works and optimizations

Computer science and software security are ever-evolving themselves, and on the combination *Shamir's secret sharing* and *Data Flooding against Ransomware* especially there isn't any research yet. Hence, any study around it might be of interest, but also improvements to our own solution, *Ranflood*, now with *SSS*, would be welcome.
Some fields where additional research is needed were already pointed out throughout this thesis. Some others had already been made in the papers on *Ranflood* [9] [10].

We didn't question too much the names of the *shard* files generated, but *ransomwares* usually have some criteria to prefer some to encrypt first, e.g. their path, name, extension, headers, looking mostly for personal data and ignoring common programs.
Something is already done from the existing strategies, which append commonly targeted extensions to files, while we didn't give it much relevance in our work, for the SSS flooder, not being our focus.
By understanding such patterns, one could apply more complex plans, by luring *malwares* towards useless, copied files and keeping them away from the most relevant ones.

Relatively to our $SSSFlooder$, we already mentioned, in the context of testing, how interesting a proper research to find the optimal parameters would be, possibly validating out choices.

Moreover, $SSSFlooder$ could be "merged" with $ShadowFlooder$, exploiting its compressed caching together with SSS to get the best of both aspects.

Our *Shamir's secret sharing* model could also benefit from exploring different implementations, specifically different finite fields, measuring and comparing their performances.
Remaining close to the computers architecture, an idea which attracted us but we didn't have enough time to explore was to exploit higher powers of 2 as order $q = p^r$, instead of $r = 8$. Values multiple of 8 (i.e. resembling a byte) could possibly grant the same performance boosts, but we suppose that they could also better exploit the full 64 bits of today's processors. Moreover, this would reduce the raw number of operations performed (e.g. $r = 16$ would imply dividing the original files in pairs of bytes instead of individual ones, performing half the operations with the finite field and presumably halving execution times). The only thing one would have to do is (as we saw in 3.1.1) to pick a generator to populate the *look-up tables*, and the sole drawback would be the memory requirements:

$$size = 2^r \times \frac{r}{8} \times 2$$

With $r = 16$, each of the 2 *look-up tables* would be populated by 2^{16} values, each of $dfrac168 = 2$ bytes (i.e. *short* type integers), resulting in a required size of 262.144 KB, still acceptable - to also get the described improvement on the second table, we should

multiply by 3 instead of 2, as it would double its size.
However, $r = 32$ (type *int*) would already require 17 GB.
We don't know, then, if there exists some compromise, allowing to only compute, e.g. half of each table (by performing only one operation on each *lookup*).

To conclude, we state that further improving the $SSSFlooder$'s performance doesn't seem so easy. In fact, it forcefully requires *shards* to be the same size as the original path, otherwise it wouldn't be possible to retrieve it with any k of them: in other words, each *shard* must keep some information about each single part of the original file, otherwise they would be unbalanced. A consequence is that there will hardly be an optimization, for instance, on *splitting* large files, unless specific research on its mathematical aspects are conducted.

4.3 Observations

Throughout this thesis we posed ourselves many questions concerning security, optimizations, implementation details, common or rare use cases for our tool.
While for implementation doubts we could at least provide the best solutions according to the information in our hands, our work is also crossing a new ground, with the unusual combination of *Shamir's secret sharing* and *Data Flooding against Ransomware*, and there are many directions in which further research may improve our findings through experimentation. On the other hand, the large number of contrasting requirements makes such that the goal becomes to only find the best compromises between them; this particularly applies in real scenarios, when we'd need to foresee any possible use case, outcome, concurring process or external factor for our software.

We instead mainly focused on understanding the potential of *SSS* and providing a new mean to fight *crypto* and *exfiltration* attacks.
By starting from an open-source project, we were able to take the good from it to merge our efforts (starting from the available application design, studies and the existing tools such as flooders, snapshooters etc.), and by contributing to it with our research, we also shared what we learnt, in the hope to stimulate others to purse this work.

The same Adi Shamir, in his first paper on *Shamir's secret sharing* (1979) [11], said relatively to it:

> "In other applications the tradeoff is not between secrecy and reliability, but between safety and convenience of use."

Shamir's secret sharing, in fact, isn't about the "secrecy" of hiding one's secret instead of making it attack-proof ("security by obscurity"), nor about the high "reliability" at the cost of security (in his terms, think about distributing your keys to everyone so you never lose them, but also allowing everyone to enter you house); instead, through two parameters (a minimum threshold k and a maximum amount n) it allows to find the best compromise between an easy to access (but also to attack) and a safe (but also easy to lose access to) system.

This also applies to our flooding tool, where the compromise between making a file easily recoverable (low k) - by both the user and the cybercriminals - and hardly attackable (high k required) transposes to the balance between redundancy and difficulty of retrieval: the former is better suited for contrasting *ransomwares*, making it possible to restore a file even if many of its *shards* were encrypted; the latter fits the *exfiltration* case, where we don't want the *attacker* to obtain user's data.
Furthermore, such observations need to find an additional compromise with performance, where increasing k for safety comes at a cost in terms of *flooding* speed.

On the memory and disk consumption compromises were already present in *Ranflood* and are still being made: caching files contents and checksums grants better performances in some cases, more resilience in others.
Indeed, despite the flexibility of self-contained *shards*, the advantages of cached information in *snapshots* can hardly be overcome. Thus, the implementation of our *shards*, with their *sections*, should mostly be seen as a more resilient alternative to them, capable of working even in more critical situations, but, for example, one can't recognize if a file has already been encrypted by the ransomware without a snapshot, at the moment.

In reality, when facing the urgent threat of a malware, even other security concerns not to loose a few files become secondary, compared to the major urge to beat it in speed and flood the file system before too much damage is suffered, and to give the user the time to physically reach the machine: in practice, a few, rare execution errors or files lost, despite not being irrelevant, can be justified if this significantly increases the flooder's performance.

Bibliography

[1] Java documentation, SecureRandom, next int: https://docs.oracle.com/javase/8/docs/api/java/util/Random.html#next-int-

[2] Davide Berardi, Saverio Giallorenzo, Andrea Melis, Simone Melloni, Marco Prandini - Ranflood, permanent link to the repository for the paper: https://github.com/ElsevierSoftwareX/SOFTX-D-23-00396

[3] Davide Berardi, Saverio Giallorenzo, Andrea Melis, Simone Melloni, Marco Prandini - Ranflood, binaries: https://github.com/Flooding-against-Ransomware/ranflood/releases/tag/0.5.9-beta

[4] Implementation of SSS, by github.com/codahale: https://github.com/codahale/shamir.

[5] Implementation of SSS, in HashiCorp Vault: https://github.com/hashicorp/vault/blob/main/shamir/.

[6] Implementation of SSS, by github.com/posener: https://github.com/posener/sharedsecret/blob/master/sharedsecret.go.

[7] Implementation of SSS, ssss, available as apt package: http://point-at-infinity.org/ssss/ssss.1.html.

[8] Explanation of Finite Fields and GF(256), by Russ Cox: https://research.swtch.com/field.

[9] Davide Berardi, Saverio Giallorenzo, Andrea Melis, Simone Melloni, Marco Prandini - "Data Flooding against Ransomware: Concepts and Implementations": https://doi.org/10.1016/j.cose.2023.103295

[10] Davide Berardi, Saverio Giallorenzo, Andrea Melis, Simone Melloni, Marco Prandini - "Ranflood: A mitigation tool based on the principles of data flooding against ransomware": https://doi.org/10.1016/j.softx.2023.101605

[11] Adi Shamir - "How to share a secret", 1979: https://dl.acm.org/doi/pdf/10.1145/359168.359176

[12] graalVM: https://www.graalvm.org/.

[13] Nomen est Omen library: https://github.com/igr/nomen-est-omen.

[14] RxJava library: https://github.com/ReactiveX/RxJava.

[15] ZeroMQ library: https://zeromq.org/.

[16] D. Brink, A (probably) exact solution to the Birthday Problem, Ramanujan Journal, 2012, [1].: https://link.springer.com/article/10.1007/s11139-011-9343-9.

[17] Wikipedia, birthday paradox: https://en.wikipedia.org/wiki/Birthday_problem#.

[18] Wikipedia, birthday paradox generalization formula: https://en.wikipedia.org/wiki/Birthday_problem#Arbitrary_number_of_days.

[19] Wikipedia, Finite field properties: https://en.wikipedia.org/wiki/Finite_field#Properties.

[20] Wikipedia, Lagrange polynomial: https://en.wikipedia.org/wiki/Lagrange_polynomial.

[21] Wikipedia, Shamir's Secret Sharing: https://en.wikipedia.org/wiki/Shamir%27s_secret_sharing.

[22] Wikipedia, Shamir's Secret Sharing: https://en.wikipedia.org/wiki/Shamir%27s_secret_sharing#Problem_of_using_integer_arithmetic.