An alternative method of data retrieval with Domino technique

(Alternatywna metoda pozyskiwania danych używając metody Domino)

Jan Góra

Praca inżynierska

Promotor: dr Marek Materzok

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

22 lutego 2020

Abstract

The subject of the thesis is a novel technique of effective information retrieval. By splitting a message into smaller fragments of the same length, it is possible to recover the original message without knowing the original order of the fragments. In the thesis, an assembly algorithm generating all possible reconstructions for a given set of fragments has been designed together with its implementation in the form of a small program and a module, possible optimizations, the complexity and performance analysis, examples of its applications in cybersecurity, and the comparison to other popular techniques. The effectiveness of the method was estimated based on over 15,000 conducted experiments, where the effectiveness factor is the number of unique solutions generated for chosen strings and the size of a single fragment. The experiments have shown that the effectiveness of the technique drops with the increase of the size of the original message. The advantages of the technique over other known techniques is its independence to previous states and because of which it can be effectively parallelized. In the thesis, an example application of the technique was presented, based on situations where other known techniques did not provide the exploitation opportunity of a vulnerable server.

Streszczenie

Przedmiotem badań jest nowatorska metoda efektywnego wydobywania oraz odzyskiwania informacji. Poprzez rozłożenie wiadomości na mniejsze, równej długości fragmenty, możliwe jest jej późniejsze odzyskanie bez znajomości kolejności, w jakiej fragmenty oryginalnie występowały. W pracy zaprojektowany został algorytm znajdujący wszystkie rozwiązania dla danego zbioru fragmentów wraz z jego implementacją w postaci programu użytkowego oraz modułu, możliwymi optymalizacjami, analizą złożoności oraz wydajności, przykładami zastosowania w cyberbezpieczeństwie oraz porównaniem do innych popularnych metod. Efektywność metody oszacowana została na podstawie ponad 15 000 przeprowadzonych eksperymentów, gdzie wyznacznikiem efektywności jest liczba znalezionych rozwiązań dla wybranych ciągów znaków oraz długości pojedynczego fragmentu. Eksperymenty pokazały, że efektywność metody spada wraz ze zwiększającą się długością oryginalnej wiadomości. Przewagą metody nad innymi znanymi metodami jest jej niezależność od poprzednich stanów, dzięki czemu, może być efektywnie zrównoleglona. W pracy zostało również pokazane przykładowe zastosowanie metody w sytuacjach, w których inne znane metody nie umożliwiały eksploitacji podatnego serwera.

Contents

Chapter 1

Introduction

In the rapidly evolving world, the safety and confidentiality of information became vital to societies and individuals. In the era of the Internet, keeping the information secure was proven to be challenging. With the increasing value of personal information such as passwords or identities, it became a target for attackers.

Because of arising threats, over the years many protection mechanisms have been implemented to keep the Internet users secure and which made exploitation a lot harder from the attackers' perspective.

In the ideal scenario, the attacker wants to ask a vulnerable server a question *What is the password of the user123?* to which the server responds with *The user123's password is ababa.* That's an example of a **direct information leak**.

In the scenario where the vulnerable server responds with *Yes* and *No* only, the direct data exfiltration[1] cannot be performed. To exploit that scenario, multiple questions must be sent to the server in order to retrieve the information. That technique is often referred to as **boolean-based exfiltration** or **blind exfiltration**. It asks a series of questions about the secret that narrow down the search space. A common approach is to start with an empty string and successively extend it with the confirmed characters. Table 1.1 illustrates how the technique may work for the secret *ababa* from an alphabet consisting of two letters *ab*. The last *Yes* answer from the table is the answer to a question about the searched secret. That is because the searched word *ababa* is the longest match and extending it by one character fails.

Note that in the example illustrated in Table 1.1, to retrieve the secret, 9 questions were required. However, if the length of the secret was known in advance, the same secret can be potentially retrieved with a smaller number of questions. Indeed, with the information from Table 1.2 that only *ab* and *ba* strings are found in the secret of length 5, the only possible combinations for the secret are *ababa*

[1]Data exfiltration is the unauthorized copying, transfer or retrieval of data from a computer or server. Data exfiltration is a malicious activity performed through various different techniques, typically by cybercriminals over the Internet or another network. [17]

No	Question	Answer
1	Does the secret contain ’a’?	Yes
2	Does the secret contain ’aa’?	No
3	Does the secret contain ’ab’?	Yes
4	Does the secret contain ’aba’?	Yes
5	Does the secret contain ’abaa’?	No
6	Does the secret contain ’abab’?	Yes
7	Does the secret contain ’ababa’?	Yes
8	Does the secret contain ’ababaa’?	No
9	Does the secret contain ’ababab’?	No

Table 1.1: A common technique for retrieving secrets

and *babab*. By asking one or two questions, it can be confirmed that *ababa* is the searched secret, which makes it only 5-6 questions in total.

No	Question	Answer
1	Does the secret contain ’aa’?	No
2	Does the secret contain ’ab’?	Yes
3	Does the secret contain ’ba’?	Yes
4	Does the secret contain ’bb’?	No

Table 1.2: Domino technique for retrieving secrets

The latter technique is the subject of this thesis with the proposed assembly algorithm, its applications in information security, the efficiency of the technique based on randomly generated, from the uniform distribution, secrets of different sizes and alphabets, and its advantages over other commonly used techniques.

Chapter 2

The Domino problem

A general assembly problem can be phrased as:

> *With a given set of contiguous subsequences of a sequence, is it possible to recover the original sequence? If so, what are the the possible assemblings?*

The assembly problem together with recovering techniques play a vital role in other areas such as bioinformatics, where a similar problem known as *sequence assembly* occurs. It is used to recover a full DNA code from the smaller parts. From Wikipedia page [1]:

> It refers to aligning and merging fragments from a longer DNA sequence in order to reconstruct the original sequence. This is needed as DNA sequencing technology cannot read whole genomes in one go, but rather reads small pieces of between 20 and 30,000 bases, depending on the technology used. Typically the short fragments, called reads, result from shotgun sequencing genomic DNA, or gene transcript (ESTs).

The *Domino problem* is a simplified version of the general assembly problem and its definition ensures that it is always possible to recover the original, finite sequence from a given set of contiguous subsequences. The definition of the Domino problem, along with the constraints that the problem must satisfy, will be presented in this chapter.

It's worth mentioning that the Domino problem differs from the sequence assembly and does not refer to Domino Problem from *Wang Tiles* [3] although the idea for the name is shared between both problems. Both terms *Domino problem* and *Domino technique* were invented independently during writing this thesis and will refer to the constructions presented in this thesis.

2.1 The Domino problem – definition

The **Domino problem** can be defined as:

> For a given length of the *secret*[1] and a set of *puzzles*[2] where:
>
> - each puzzle in the set has a fixed length K,
> - the set is finite and consists of at least one puzzle,
> - if the set consists of at least two puzzles, each puzzle from the set is *chained*[3] with at least one different puzzle from the set,
> - every distinct contiguous subsequence (puzzle) of length K of the secret occurs in the set,
>
> find all *assemblies*[4] satisfying the constraints:
>
> 1. Only puzzles from the Domino set occur in the searched assembly.
> 2. Each puzzle from the Domino set occurs in the searched assembly at least once.
> 3. There are exactly $N-K+1$ puzzles in the searched assembly where N is the length of the secret.
> 4. Every two adjacent puzzles in the searched assembly are chained at $K-1$ positions.

Assemblies satisfying the above constraints will later be called *solutions. Partial solution* will refer to assemblies satisfying the constraints with the exception of 2 and 3. A given set in the Domino problem satisfying the conditions from the definition will be later referenced as *Domino set.* Notice that the definition of the Domino problem ensures that at least one solution exists.

Just like in the game of Dominoes [2], the goal is to find a correct line (solution) of tiles (puzzles) in which values of adjacent pairs of tile ends (chains) must match. From there, the original sequence can be recovered by ignoring repetitions of the overlapping parts (chains).

[1]The term *secret* refers to the finite, original sequence to be recovered.

[2]The term *puzzle* refers to a contiguous subsequence of the original sequence (secret).

[3]The term *chain* refers to a sequence that connects two puzzles. For a given *chain length* L and two puzzles x and y, y is *chained* to x if the suffix of the length L of x matches the prefix of the same length of y; two puzzles x and y *are chained* if either x is chained to y or y is chained to x; if L was not provided it is assumed that both puzzles have the same length K and the *chain length* is $K-1$. E.g. 'bcd' is chained to 'abc' by a chain of length 2, but 'abc' can never be chained to 'bcd'.

[4]The term *assembly* refers to a sequence of a given number of puzzles where every two adjacent puzzles are connected in some manner. That connection will be later called a *chain*, and its definition can be found in another footnote.

2.2 Domino technique

The **Domino technique**[5] refers to a method of a secret retrieval through identifying which tuples (puzzles) from a chosen *charset*[6] exist in the secret. These tuples form a set in the Domino problem from which the secret is to be recovered.

2.3 Examples

To provide more intuition about definitions that were introduced, the Domino technique will be presented on descriptive examples.

In the thesis, the effectiveness of the technique was measured basing on secrets generated from most popular charsets presented in Table 2.1.

Charset name	Alphabet
base64	*0123456789abcdefghijklmnopqrstuvwxyz* *ABCDEFGHIJKLMNOPQRSTUVWXYZ* $-$ _
hex	*0123456789abcdef*
letters	*abcdefghijklmnopqrstuvwxyz*
lowercase	*0123456789abcdefghijklmnopqrstuvwxyz*

Table 2.1: Example of popular charsets

Here, for simplicity, the goal is to retrieve the *secret* 'example' by identifying which pairs from all 36 possible puzzles of size 2 created from a charset 'aelmpx' exist in the secret. Table 2.2 illustrates which pairs will be found in word 'example'.

Puzzle	ee	ex	ea	em	ep	el	xe	xx	xa	xm	xp	xl
In secret?	No	Yes	No	No	No	No	No	No	Yes	No	No	No
Puzzle	ae	ax	aa	am	ap	al	me	mx	ma	mm	mp	ml
In secret?	No	No	No	Yes	No	No	No	No	No	No	Yes	No
Puzzle	pe	px	pa	pm	pp	pl	le	lx	la	lm	lp	ll
In secret?	No	No	No	No	No	Yes	Yes	No	No	No	No	No

Table 2.2: All 36 pairs from charset 'aelmpx' tested against the secret 'example'

[5]The term "Domino" connected to a secret retrieval was first mentioned in "justCTF 2019 write-ups by @terjanq" article [5].

[6]The *charset* refers to an alphabet from which the secret is constructed. Table 2.1 presents popular charsets used in the thesis.

In a mathematical sense, the string 'example' is treated as a finite sequence of single characters: $(''e'', ''x'', ''a'', ''m'', ''p'', ''l'', ''e'')$.

Puzzles of size 2, by definition, are contiguous subsequences of the secret. Table 2.2 shows that puzzles: $(''e'', ''x'')$, $(''x'', ''a'')$, $(''a'', ''m'')$, $(''m'', ''p'')$, $(''p'', ''l'')$, $(''l'', ''e'')$ exist in the secret. To simplify, it will be later written as ex, xa, am, mp, pl, le.

An assembly, by definition, is a sequence of puzzles where every adjacent two are connected in some manner. For example, (ex, ex, ex) could be an instance of an assembly where every two adjacent puzzles are chained at 2 positions. However, in the thesis, it will be assumed that the connection is usually a *chain* of *chain length* $K - 1$ where K is the size of the puzzle.

A chain, by definition, is a sequence that connects two puzzles in a way that the suffix of one puzzle matches the prefix of another. As for an example, puzzles xa and am are chained by a chain of size 1: $(''a'')$, therefore am is chained to xa.

Puzzles ex, xa, am, mp, pl, le form a *Domino set* from Domino problem definition, and it can be presented as a set: $\{ex, xa, am, mp, pl, le\}$

The assembly (ex, ex, ex) is not a *partial solution* because not every two adjacent puzzles are chained at 1 position. An assembly (ex, xa, am) is an example of a partial solution, because every two adjacent puzzles are chained and other constraints for an assembly being a partial solution hold.

For a length of the secret $N = 7$ given in the Domino Problem, an assembly (le, ex, xa, am, mp, pl) is an example of a *solution* because every two adjacent puzzles are chained and there are $N - K + 1 = 7 - 2 + 1 = 6$ puzzles in the assembly. The solution can be converted back to the string, by ignoring repetitions of the chains which would be: 'lexampl'. There are exactly 6 valid solutions that can be assembled from puzzles ex, xa, am, mp, pl, le:

- example
- xamplex
- amplexa
- mplexam
- plexamp
- lexampl

Chapter 3

The assembly algorithm

One way to approach the Domino problem could be to apply *dynamic programming*. In order to apply dynamic programming to a problem, the problem must be first transformed to its equivalent recursive definition which the Domino problem can be transformed to. A brief reasoning along with a proposed implementation of the dynamic approach will be presented in this chapter.

3.1 DP Function

Domino problem can be transformed into a problem of finding all partial solutions of the length $N-K+1$ to which constraint 2 from the Domino problem must be yet applied. A function that returns a set of all possible partial solutions consisting of k puzzles for a given number $k > 0$ and a Domino set S can be defined as $DP(k, S)$. Then:

1. $DP(1, S) = S$ because every puzzle from the Domino set S satisfies the constraints for a partial solution and, at the same time, S is the largest possible set of assemblies consisting of only one puzzle.
2. A set $DP(N - K + 1, S)$ for a Domino set S, a secret length N and a length K of every puzzle from S, returns all solutions from the Domino problem after applying the constraint 2 from the Domino problem definition.
3. $DP(0, S) = \emptyset$ because there are no partial solutions consisting of 0 puzzles.

3.1.1 Recursive definition

A vital observation is that the DP function can be recursively defined as:

$$DP(k, S) = DP(k - 1, S) \times S$$

where $\times$ operator stands for a slightly modified Cartesian product[4] defined as:

$A \times B := \{(a_1, \ldots, a_m, b_1, \ldots, b_n) : (a_1, \ldots, a_m) \in A, (b_1, \ldots, b_n) \in B,$
b_1 *is chained to* $a_m\}$[1]

for two non-empty sets of assemblies A and B, where every assembly from these sets consists of n and m puzzles respectively. In other words, the formula defines a set consisting of assemblies that are a product of chaining every two assemblies from two different sets. In addition, an empty set $\emptyset$ will be a neutral element of the $\times$ operator, i.e. $A \times \emptyset = A$, $\emptyset \times B = B$ and $\emptyset \times \emptyset = \emptyset$.

3.1.2 Proof of the recursive definition

The reason why the DP function can be defined recursively in the presented way can be proven by induction.

Inductive hypothesis: Suppose the definition $DP(k, S) = DP(k-1, S) \times S$ is true, i.e. $DP(k-1, S) \times S$ returns all possible partial solutions consisting of k puzzles.

Inductive base case: The *inductive base case* $DP(1, S) = DP(0, S) \times S = S \times S$ holds because the product $\emptyset \times S = S$ and it was already shown that $DP(1, S) = S$.

Inductive step: If $DP(k, S)$ is true, then $DP(k+1, S) = DP(k, S) \times S$ returns all possible partial solutions consisting of $k+1$ puzzles from the Domino set S each.

Proof of the inductive step:

Let $(d_1, d_2, \ldots, d_k, s)$ be any assembly from the product $DP(k, S) \times S$. Then, from the *inductive hypothesis*, $(d_1, d_2, \ldots, d_k)$ must be a partial solution, and from the definition of the operator $\times$, s must be chained to d_k; thus $(d_1, d_2, \ldots, d_k, s)$ is also a partial solution because all constraints for being a partial solution hold.

Let assume that a partial solution $(e_1, e_2, \ldots, e_k, e_{k+1})$ that is contained in the set $DP(k+1, S)$ but not in the product $DP(k, S) \times S$ exists. Then, the assembly $(e_1, e_2, \ldots, e_k)$ must be also a partial solution because it satisfies all the required constraints. It creates a contradiction, because $DP(k, S)$ returns all possible partial solutions consisting of k puzzles; hence $(e_1, e_2, \ldots, e_k)$ must be contained in the set $DP(k, S)$. This proves by contradiction that the partial solution $(e_1, e_2, \ldots, e_k, e_{k+1})$ must be contained in the product $DP(k, S) \times S$ because $(e_1, e_2, \ldots, e_k)$ must be contained in $DP(k, S)$ and e_{k+1} must be contained in S by constraint 1 from the Domino problem definition.

Because any partial solution $(e_1, e_2, \ldots, e_k, e_{k+1})$ must be contained in the product $DP(k, S) \times S$ and every assembly $(d_1, d_2, \ldots, d_k, s)$ from the product $DP(k, S) \times S$ is a partial solution, it proves that the *inductive hypothesis* indeed holds.

[1] $(a_1, \ldots, a_m)$ is an assembly (sequence of puzzles) from the set A, consisting of m puzzles. Similarly $(b_1, \ldots, b_n)$ consists of n puzzles and $(a_1, \ldots, a_m, b_1, \ldots, b_n)$ consists $n + m$ puzzles and is a result of concatenating the two previous sequences (assemblies).

3.1.3 Extended recursive definition

Directly from the recursive definition of the *DP* function, it can be also defined as:

$$DP(k,S) = \underbrace{S \times \cdots \times S}_{k\ times}$$

In consequence, the recursive definition can be extended to:

$$DP(k+l,S) = DP(k,S) \times DP(l,S)$$

because $DP(k+l,S)$ can be written as

$$\underbrace{S \times \cdots \times S}_{k\ times} \times \underbrace{S \times \cdots \times S}_{l\ times}$$

and in contrast to Cartesian product, the operator $\times$ is associative[2].

3.2 Dynamic approach

Because of the defined recursive function *DP*, the Domino problem can be solved using *Dynamic Programming* where the *DP state* at times i is a structure containing all partial solutions from the set $DP(i,S)$ for a given Domino set S from the Domino problem. A *DP state* $i+1$ is created from the state i by extending each partial solution from the state with all puzzles from the Domino set (*state 1*) chained to the last puzzle in the assembly that is being extended.

Notice that every *DP state* only consists of unique assemblies. This is because the *state* 1 is created from the Domino set which consists of unique assemblies from the definition of a set and every other state is created by extending a unique assembly with the unique puzzles from the Domino set. Hence, on every *DP state* the technique is duplicate-free by design, or in other words, none of the *DP states* contain any duplicates.

Another observation is that every contiguous subsequence of a secret split into k puzzles is contained in the *DP state* k. Directly from the definition of the Domino set, all puzzles from that subsequence must exist in the Domino set, but also, every two adjacent puzzles are already chained. Because an assembly created from chaining these puzzles satisfies constraints for being a partial solution, it must be contained in the result of $DP(k,S)$.

3.3 Pseudocode

In Listing 2, the pseudocode of the described algorithm written in *Python* is presented, where each step is also briefly described in the comments.

[2]It can be proven that $(A\times B)\times C = A\times(B\times C)$, because every assembly from both sides will be in the form of $(a_1,\dots,a_k,b_1,\dots,b_l,c_1,\dots,c_m)$ for $(a_1,\dots,a_k)\in A,(b_1,\dots,b_l)\in B,(c_1,\dots,c_m)\in C$.

The algorithm takes three arguments as input:

- `N` – the length of the secret to be recovered,
- `K` – the length of every puzzle in the Domino set,
- `S` – the Domino set (list of distinct puzzles).

The algorithm first fills a *chain graph* with puzzles from the Domino set `S`, which happens in lines $6-9$ in the pseudocode. A *chain graph* is an optimization that helps to compose a product of $DP(I-1, S) \times S$ more effectively. It's a data structure that represents chains between puzzles – for each puzzle `p` from the Domino set `S`, `G[p]` returns a list of puzzles chained to `p`. In the line 9, the algorithm adds a chain between puzzles `p1` and `p2` to the graph if `p2` is chained to `p1`.

In lines 12–14, the algorithm creates the *DP state 1* (`DP[1]`) from all puzzles contained in the Domino set S.

Then, in lines 18–22, it fills the `DP[I]` (which corresponds to the *DP state I*), in a loop where I is in a range from 2 to $N - K + 1$ inclusively, with assemblies created from chaining all assemblies from `DP[I-1]` with puzzles from the chain graph `G` that returns assemblies from `DP[1]` chained to the last puzzle in each assembly being extended.

After these steps, `DP[N-K+1]` will return all partial solutions consisting of $N - K + 1$ puzzles. To generate the searched solutions, only partial solutions satisfying constraint 2 must be returned.

3.4 Implemented optimizations

In the included to the thesis application, two more optimizations have been implemented, Repetitions and Less knowledge, that improve the memory usage and the performance.

3.4.1 Repetitions

From the Domino problem constraints, if a solution is of the same length as the Domino set, it implies that all puzzles in the solution are distinct; otherwise, some puzzles occur in the solution more than once. Moreover, it can be precisely calculated how many repetitions will occur in the solution, i.e. $\textit{repetitions} = N - K - \#S + 1$, where N is the size of the secret, $\#S$ is the size of the Domino set and K is the size of a single puzzle from the set. Assemblies that exceed the allowed number of repetitions on a state i will not be extended to the state $i + 1$, which ensures that `DP[NN]` in the code from Listing 2 will only contain valid solutions, but also saves the overall computing time and memory usage by ignoring on early stage partial solutions that cannot produce a valid solution after applying the constraint 2 from the Domino problem definition.

With a function `repetitions` that returns the number of total repetitions in an assembly, the *repetitions* optimization of the algorithm from Listing 2 can be implemented as shown in Listing 1.

```
for puzzle in G[assembly[-1]]:
    new_assembly = assembly + [puzzle]
    if repetitions(new_assembly) > N - K - len(S) + 1:
        continue
    else:
        DP[I].push(new_assembly)
```

Listing 1: Repetitions optimization

3.4.2 Less knowledge

It can be noticed that in the presented assembly algorithm, to calculate the state i only two other states have to be stored in memory: the *DP state 1* and the *DP state* $i - 1$. With that observation, memory usage can be significantly decreased at the cost of losing information about partial solutions.

3.4.3 An example

To illustrate how the algorithm works, Table 3.1 shows how partial solutions at every DP state are calculated, starting with puzzles exa, xam, amp, mpl, ple. At *DP state 4*, only one solution $(exa, xam, amp, mpl, ple)$ is present which after conversion to string gives the secret *example*.

DP state	Partial solutions
0	$\{exa, xam, amp, mpl, ple\}$
1	$\{(exa, xam), (xam, amp), (amp, mpl), (mpl, ple)\}$
2	$\{(exa, xam, amp), (xam, amp, mpl), (amp, mpl, ple)\}$
3	$\{(exa, xam, amp, mpl), (xam, amp, mpl, ple)\}$
4	$\{(exa, xam, amp, mpl, ple)\}$

Table 3.1: Partial solutions at each DP state

```python
NN = N - K + 1 # Number of puzzles in a solution
G = Graph() # Custom data structure Graph

# For every pair (p1, p2) of puzzles from the Domino set S,
# check if p2 is chained to p1. If so, add it to the chain graph G[p1]
for p1 in S:
    for p2 in S:
        if p1[1:] == p2[:-1]:
            G[p1].add(p2)

# DP state 1 is created directly from puzzles from the Domino set
DP[1] = []
for puzzle in S:
    DP[1].append([puzzle])

# For every state I, try to append an assembly to DP[I] created through
# extending every assembly from state I-1 with chained puzzles from S
for I in range(2, NN+1):
    DP[I] = []
    for assembly in DP[I-1]:
        for puzzle in G[assembly[-1]]:
            DP[I].push(assembly + [puzzle])

# DP[NN] contains partial solutions consisting of N - K + 1 puzzles,
# that after filtering ones satisfying constraint 6,
# will only consist of valid solutions
```

Listing 2: Pseudocode to the Assembly algorithm written in Python

Chapter 4

The assembly algorithm performance

The conducted experiments have shown that the implementation of the assembly algorithm, even with the optimization applied, experiences performance issues, both computing time and memory.

The expected time complexity is not easy to predict but can be well visualized on Graphs 4.1, 4.2, 4.3, 4.4 and 4.6 where the painted area under the graph represents a summary number of partial solutions from all *DP states*. At the same time, it also represents the estimated number of the performed calculations (multiplied by some constant).

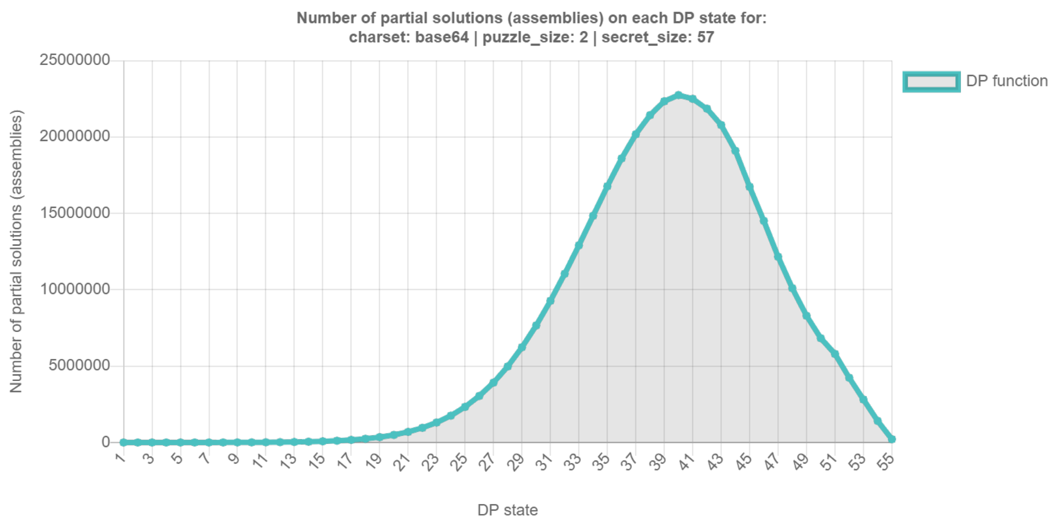

Figure 4.1: Growth of the number of partial solutions at different DP states for a secret of length 57, from *base64* charset, split into puzzles of size 2

From Graph 4.6 it is clear that the time complexity does not necessarily depend on the number of valid solutions produced but rather on the distribution of partial

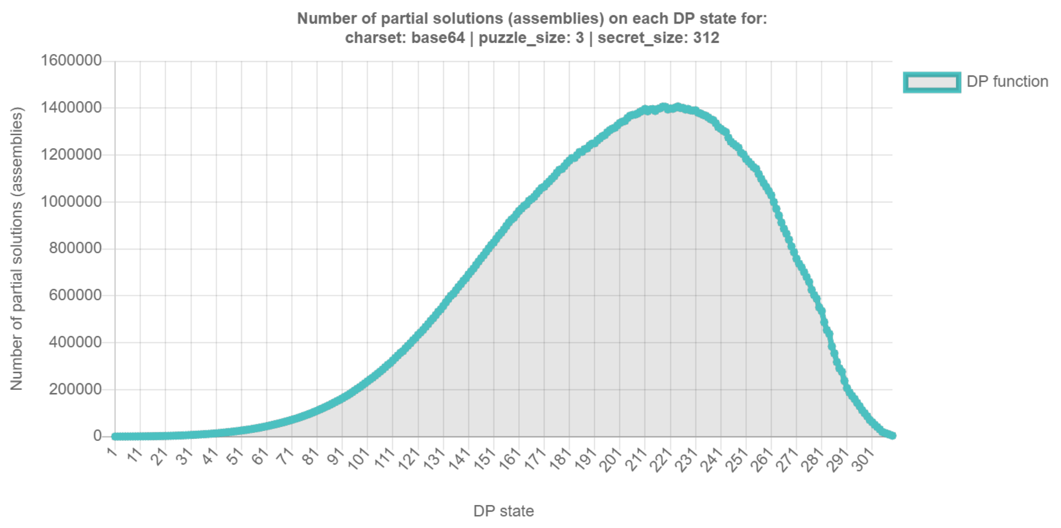

Figure 4.2: Growth of the number of partial solutions at different DP states for a secret of length 312, from *base64* charset, split into puzzles of size 3

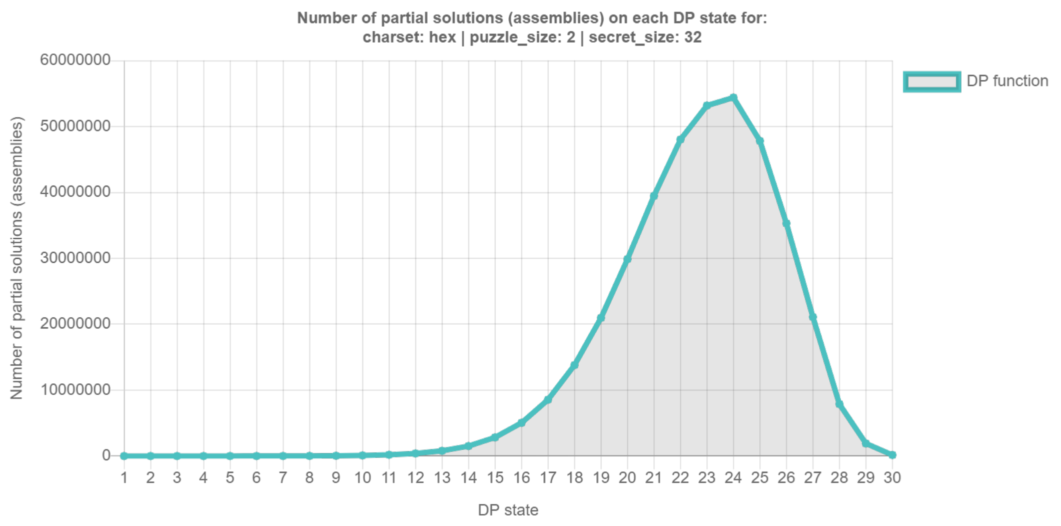

Figure 4.3: Growth of the number of partial solutions at different DP states for a secret of length 32, from *hex* charset, split into puzzles of size 2

solutions, which seems to be related to the length of the secret. From Table 4.1 we can read that for the presented case, for the secret of length 202, the number of solutions was only 17,408 while its time performance can be estimated to 20,000,000,000 ($2 \cdot 10^{10}$) operations. The area under the graph can be estimated as 200,000,000 ($2 \cdot 10^{8}$), which is the minimal number of operations that were performed, multiplied by the constant 100 (linear operations such as searching in an array, chained puzzles, etc.) gives the former estimation.

For the contrast, from Graph 4.1 the number of solutions for the secret of size 57 is 213,696 while the time performance can be estimated to 8,000,000,000 operations ($8 \cdot 10^{9}$) – the area under graph estimated as 400,000,000 ($4 \cdot 10^{8}$) multiplied by the

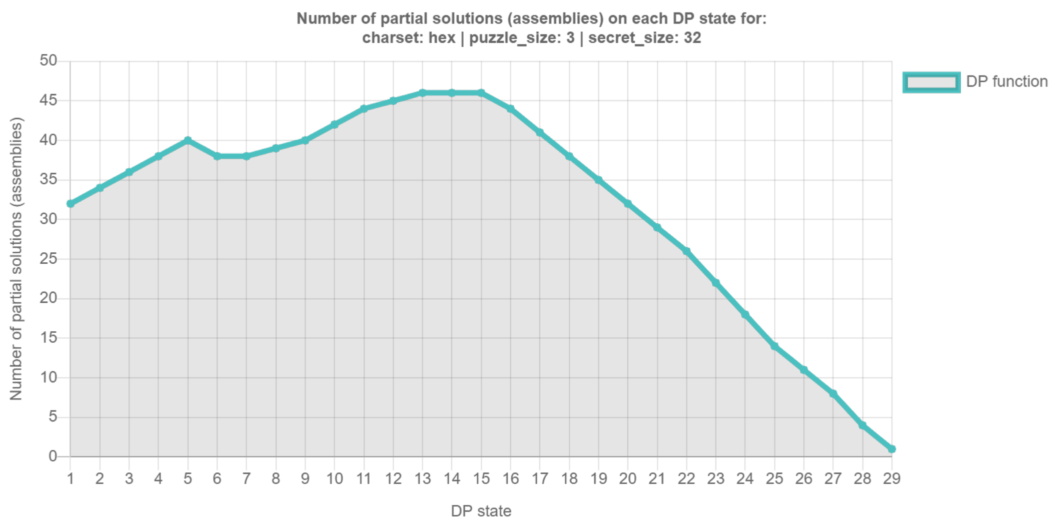

Figure 4.4: Growth of the number of partial solutions at different DP states for a secret of length 32, from *hex* charset, split into puzzles of size 3

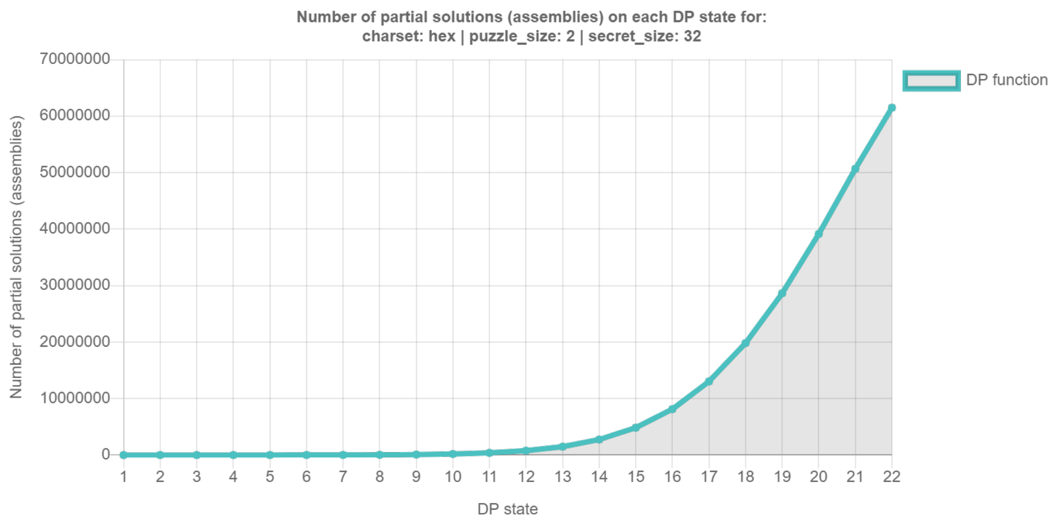

Figure 4.5: Plot illustrating the memory exhaustion

constant 20.

Memory usage related to the graph can be estimated as the maximum number of partial solutions multiplied by 3. That is because at least two states must be stored in the memory and the garbage collector [7] in *Node.js* must be taken into account. In the experiments, the maximum memory usage for the process was 30GB which can be used as a reference to these numbers.

The growth of the partial solutions can also be read in Tables 4.2 and 4.3. For each DP state i, the right column represents the difference between two states i and $i-1$. Table 4.2 presents an example of the memory exhaustion. It can be noticed, that from the state 21 to the state 22, the DP function started growing slightly

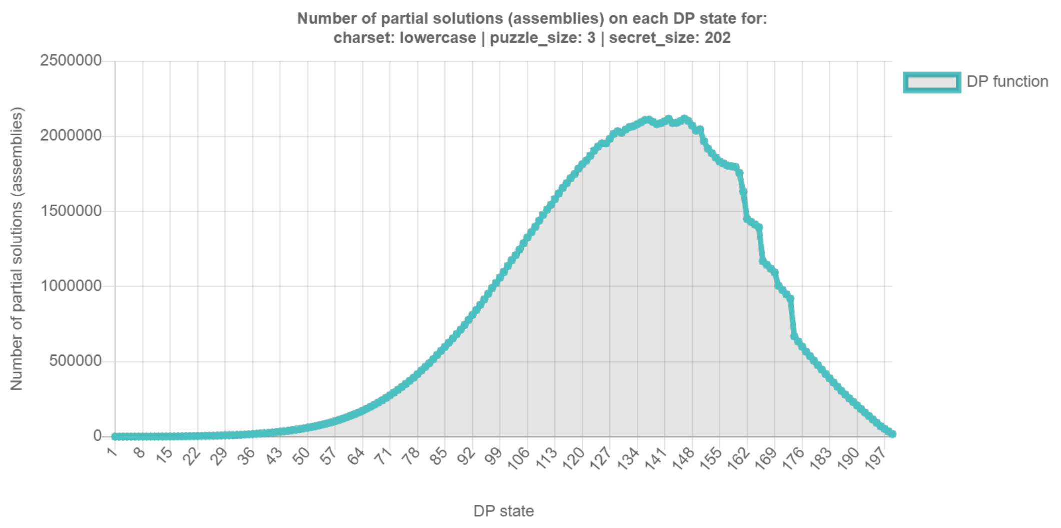

Figure 4.6: Plot illustrating high computing time for a secret of length 202, from *lowercase* charset, split into puzzles of size 2

Charset	Puzzle size	Secret length	Number of solutions	Figure
base64	2	57	213,696	4.1
base64	3	312	4,012	4.2
hex	2	32	159,072	4.3
hex	3	32	1	4.4
hex	2	32	N/A	4.5
lowercase	3	202	17,408	4.6

Table 4.1: Number of solutions for performance plots

slower, which could indicate that the DP function was near the highest peak, from where the solution would be recovered. The similarity with Table 4.3 can be spotted, where the DP function also slowed its growth on the state 21.

DP state i	Partial solutions difference
1	0
2	105
3	271
4	679
5	1,651
6	4,002
7	9,418
8	21,502
9	47,125
10	99,146
11	201,246
12	386,456
13	716,730
14	1,260,014
15	2,095,204
16	3,286,630
17	4,903,522
18	6,788,298
19	8,815,054
20	10,501,251
21	11,543,401
22	10,808,052

Table 4.2: Differences between states i and $i-1$ for Figure 4.5

DP state i	Partial solutions difference
1	0
2	78
3	179
4	402
5	919
6	2,111
7	4,769
8	10,697
9	22,923
10	49,815
11	99,269
12	204,007
13	396,092
14	725,180
15	1,311,234
16	2,209,297
17	3,513,620
18	5,246,463
19	7,167,757
20	8,917,165
21	9,597,627
22	8,558,207
23	5,152,269
24	1,210,915
25	-6,578,890
26	-12,516,990
27	-14,206,644
28	-13,205,399
29	-5,984,840
30	-1,749,220

Table 4.3: Differences between states i and $i-1$ for Figure 4.3

Chapter 5

Further optimizations

The assembly algorithm performance chapter has shown that the assembly algorithm faces performance issues, specifically high memory usage that stops the algorithm from finding the solutions. Figure 4.3 shows that the number of final solutions might be relative small [1] compared to the memory used.

In this chapter, a *better assemblings* method that attempts to improve the performance will be designed but which has not been implemented in the application attached to this thesis. To improve the overall computing time on multi-core machines, the assembly algorithm can be also parallelized[2].

5.1 Better assemblings

In the current implementation of the algorithm, the *DP state* $i+1$ is created through extensions of the state i with chained puzzles from the Domino set. That creates performance and memory issues as described in the section *The assembly algorithm performance* and shown in Figures 4.5 and 4.6.

5.1.1 The idea

Instead, from the extended recursive definition of the *DP* function, *DP state* $i+j$ can be composed as a product of states i and j, $DP(i+j, S) = DP(i, S) \times DP(j, S)$, by chaining all assemblies from states i and j. For example, the secret 'babab' of length 6 could be assembled as a result of merging two assemblies 'bab' and 'bab' where the chaining length is 1. For a chain of length 2, the same could be factorized

[1]From Table 4.1, the number of solutions was 159,072 while at *DP state 24* over 50,000,000 partial solutions had to be stored in memory.

[2]$DP(k, S)$ can be split into two disjoint subsets $A, B \in DP(k, S)$ that $DP(k, S) = A \cup B$. Then the $DP(k + 1, S) = (A \times S) \cup (B \times S)$ which can be calculated in parallel and which can be generalized to any number of subsets.

as a connection of two assemblies: 'baba' and 'bab'.

Based on experiments illustrated in Figures 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6, a pattern of how the number of partial solutions (assemblies) changes over different DP states was discovered. The distribution of partial solutions appears to behave like the natural distribution curve. That information can be used to optimize the algorithm for certain use-cases or improve the overall performance. The goal of the optimization is to omit high peaks[3] that often exceed available RAM (as presented on the Figure 4.5), even at the cost of a slightly worse computing time performance if necessary.

With the information about the shape of the curve, it's possible to design an efficient algorithm that will effectively omit the mentioned high peaks if possible. The crucial observed property of the *DP* function is that it monotonically increases until a certain point and then monotonically decreases until the *final DP state*[4] is reached. Thanks to that, more efficient algorithms can be designed.

5.1.2 The algorithm

Assuming the existence of a function that effectively calculates the product $A \times B$ of sets consisting of partial solutions, where the operator $\times$ is the same operator defined in *Recursive definition* section, the idea for omitting a high peak is:

1. Try to calculate a set of partial solutions C (which will be also referred as *DP state* C) satisfying the condition $\langle C \times C \rangle = \langle \textit{final state} \rangle$, where $\langle \cdot \rangle$ is the length of every assembly from the set.

2. If the state C cannot be calculated because of the memory limit, calculate a state C' that is the closest to the state C by the means of the DP index and does not exceed the memory.

3. If a state created as a product of $C' \times C'$ exceeds the memory, it means that omitting the high peak will be impossible. Otherwise, it is guaranteed that the state $C' \times C'$ is on the decreasing curve, from where it is possible to calculate the final state because less memory will be used at the next steps.

The proof of the statement 3: If the state $C' \times C'$ wasn't on the decreasing part of the curve, that would mean that it was on the increasing part instead. But C' is the furthest state that did not exceed the memory, therefore, it must be on the increasing part. That creates a contradiction with C' being the furthest state on the increasing part because the state $C' \times C'$ is further than the state C'.

[3]High peak refers to DP states that have the biggest number of partial solution and which could exceed the memory limit. For example, in Figure 4.3 high peak could refer to DP states in range $[18, 27]$ if the memory limit allowed to only store 3,000,000 assemblies.

[4]*Final state* refers to the *DP state* consisting of solutions.

5.1.3 The product of partial solutions

The question is how to effectively implement a function that returns a product $A \times B$ of two sets of partial solutions. The naive algorithm could iterate over all pairs (a, b) where a and b are assemblies from states A and B respectively, attempt to chain them and, if successful, append to the new bigger state. However, that approach might not be sufficient enough since its time complexity can be written as $\mathcal{O}(\#A \cdot \#B)$ where $\#A, \#B$ are sizes of the sets A, B and which can be large.

A better approach to merge two states A and B could be to use a data structure that efficiently returns list of assemblies satisfying the query:

Return list of all assemblies starting with a sequence P from a set A.

Assuming that this data structure returns the requested list of assemblies in constant time $\mathcal{O}(1)$, the chain size is $K - 1$ where K is the size of a single puzzle and that assemblies in both sets A and B are uniformly distributed, the performance of the algorithm could be estimated as $\mathcal{O}(\frac{\#A \cdot \#B}{\#S})$, where $\#S$ is the size of the Domino set. The reason why $\#A \cdot \#B$ is divided by $\#S$ comes from the assumption about uniform distribution. For a given prefix of a size of almost one puzzle from the Domino set S and the state B, the data structure should return around $\frac{\#B}{\#S}$ distinct assemblies.

Notice that in the definition of the product $A \times B$ it was assumed that both partial solutions $(a_1, \ldots, a_m)$ and $(b_1, \ldots, b_n)$ are merged to $(a_1, \ldots, a_m, b_1, \ldots, b_n)$ if b_1 is chained to a_m. This definition can be slightly modified to:

$$A \star B := \{(a_1, \ldots, a_m, b_{i+1}, \ldots, b_n) : (a_1, \ldots, a_m) \in A, (b_1, \ldots, b_n) \in B, \\ (a_{m-i+1}, \ldots, a_m) = (b_1, \ldots, b_i)\}$$

for a given number i which stands for a number of puzzles that both partial solutions must be chained by. Because b_{i+1} is chained to b_i from the definition of a partial solution, it implies that b_{i+1} is chained to a_m because $a_m = b_i$. Hence, the product $A \star B$ of two sets consists of partial solutions of the size $m + n - i$ where m and n are the sizes of every partial solutions in A and B respectively. Furthermore, the product $A \star B$ corresponds to the *DP state* $m + n - i$.

With the new definition of the product and the same assumptions as before, time complexity of calculating $A \star B$ can be estimated as $\mathcal{O}(\frac{\#A \cdot \#B}{\#DP[i]})$ where $\#DP[i]$ is the number of partial solutions in the *DP state* i. This is because, *DP state* i consists of all partial solutions of size i which is the number of puzzles in the chain.

5.1.4 Total repetitions

When merging two states A and B, of the total repetitions numbers of X and Y respectively, with the chain consisting of i puzzles, the minimum number of repetitions in the new state $A \star B$ equals to $X + Y - i$. The total number of repetitions might be higher than the formula, but cannot be lower which is a vital observation for choosing the best candidates to merge, given the optimization described in the section *Repetitions*. With that observation, the data structure can be improved to satisfy the rule:

> *Return a list of all assemblies starting with a sequence P from a set A where a total number of repetition is less than L.*

which will ignore assemblies that cannot produce a valid partial solution when merged.

5.1.5 Jump search

In Figure 4.2, it can be noticed that for longer secrets, the calculations of the solutions might take a long time while memory required to perform these calculations might be relatively low. With the function that effectively calculates product of two states, and with the information how the *DP* function behaves, a *jump search*[6] algorithm can be performed to calculate the searched state faster.

The goal of the *jump search* is to find either state C or C' from the algorithm presented in this chapter without exceeding the memory. A *DP state k* can be calculated as $DP(k,S) = \underbrace{DP(n,S) \times \cdots \times DP(n,S)}_{l\ times} \times DP(r,S)$ where $l = \lfloor \frac{k}{n} \rfloor$ and $r = k \bmod n$ for some n, because k can be factorized as $k = l \cdot n + r$s. By choosing the n as $\sqrt{L}$ where L is the index of the *final DP state*, both C and C' can be found in $\mathcal{O}(\sqrt{L})$ steps.

The reason why *jump search* will work is the increasing behaviour of the *DP function* until a certain point. If the algorithm at *DP state i* will not be able to calculate $DP(i + \sqrt{L}, S)$, then from that state the algorithm can check all states between states i and $i + \sqrt{L}$ linearly. The last successfully calculated state will be searched C' if calculating C has failed.

5.1.6 Summary

Although no specific algorithm nor data structure has been implemented to test the designed optimizations, these optimizations should significantly improve the overall performance, i.e. *jump search* should improve the computing time and *better assemblings* should at least improve usage of memory. It's worth to mention that with the proposed optimizations, information about partial solutions will be lost.

Chapter 6

Technique overview

In this chapter, effectiveness of the Domino technique will be tested based on the conducted experiments.

6.1 Time and memory complexity

The pessimistic time and memory complexity of the Domino technique is $\mathcal{O}(C^N)$ where C is the size of the charset and N is the size of the secret. The inefficient complexity comes from the fact that if all tuples from a given charset occur in the secret, then each combination of elements from the charset is a valid solution. For example, for the charset *01* and the Domino set consisting of *00*, *01*, *10*, *11* puzzles, every binary number can be assembled from the puzzles. Nonetheless, conducted experiments have estimated that for randomly[1] generated secrets, relatively small compared to the chosen charsets[2], the Domino technique works effectively – the number of unique solutions is satisfactory.

6.2 The experiments

To prove the effectiveness of the Domain technique over 15,000 experiments have been conducted for different charsets, secret sizes and puzzle sizes. Experiments have been computed for 36 hours on 11 cores within 3 different cloud instances with the limit of 35GB RAM per core.

The results are presented in Figures 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 6.10, where:

- *Min* function represents the minimum number of solutions within the experi-

[1]It is assumed that randomness comes from the uniform distribution if not specified otherwise.

[2]Relaction between secrets, charsets and puzzle sizes has been presented in Figures attached to this chapter summarized in Table 6.1.

ments per a single secret size,

- *Max* function represents the maximum number of solutions within the experiments,
- *Median* function represents the median value of the solutions within the experiments,
- X-axis represents the size of the secret,
- Y-axis represents the number of solutions per secret size in the logarithmic scale for better readability.

If the experiment exceeded the allowed memory usage the result was treated as the maximum value from the group multiplied by 10.

Table 6.2 represents the number of conducted experiments for different characteristics. For every secret from the presented ranges, 30, 20 or 10 experiments were performed depending on the chosen puzzle size.

6.3 The experiments summary

The experiments have been summarized in the form of Table 6.1. The table illustrates the optimal size of the puzzle for a given secret size in order to achieve the expected number of solutions, respectively:

- exactly one solution
- less than 10 solutions
- less than 100 solutions
- less than 1,000 solutions
- less than 100,000 solutions
- less than 1,000,000 solutions

For example, for a hexadecimal secret of size 32 and the expected number of solutions less than 19, the optimal size of the puzzle is 3.

The number of validations indicates how many tuples should be validated in order to recover the secret. In other words, it is a number of all tuples of size *puzzle_size* created from the charset. The formula can be written as $\#charset^{puzzle\_size}$. For example, 30 experiments were performed for the secret of size 32, *hex* charset and the puzzle size 2 and for each experiment, 4,096 validations would have to be made to recover the secret.

Depending on the use-cases, the optimal puzzle size can vary. Tests have indicated that by repeating the experiment multiple times, better results can be obtained. Despite the fact that the expected number of solutions for the secret size of 31 and the puzzle of size 2 is 394,128 in Figure 4.3, the minimum obtained value is only 6,048. The same can be applied oppositely. In the scenario where exactly one request can be made this becomes problematic. From the same figure, for a secret of size 5, the maximum obtained value is 80 which, therefore, gives only 1.25% chances of a successful recovery.

Charset	Puzzle size	Number of validations to recover the secret	1	10	100	$1k$	$100k$	$1M$
base64	2	4,096	20	30	40	46	54	–
base64	3	262,144	100	230	280	312	–	–
hex	2	256	10	15	20	23	30	31
hex	3	4,096	40	60	75	90	–	–
hex	4	65,536	150	230	280	320	–	–
letters	2	676	12	19	25	30	36	–
letters	3	17,576	60	95	125	145	–	–
letters	4	456,976	300	450	–	–	–	–
lowercase	2	1,296	13	23	28	34	41	45
lowercase	3	46,656	100	128	164	190	–	–

Table 6.1: Table shows how an expected number of solutions changes with the increase of the secret size for different characteristics

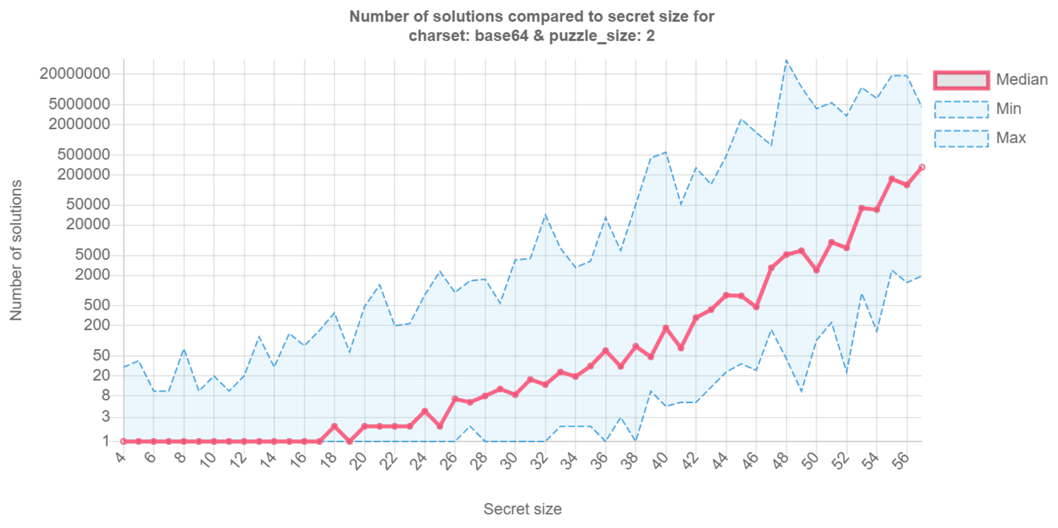

Figure 6.1: Number of solutions compared to secret length for secrets from *base64* charset

Charset	Puzzle size	Size range for a secret	Number of experiments per secret size
base64	2	4 – 57	30
base64	3	200 – 313	20
hex	2	4 – 32	30
hex	3	3 – 102	20
hex	4	200 – 323	10
letters	2	4 – 38	30
letters	3	30 – 148	20
letters	4	200 – 500	10
lowercase	2	3 – 47	30
lowercase	3	100 – 202	20

Table 6.2: Conducted experiments

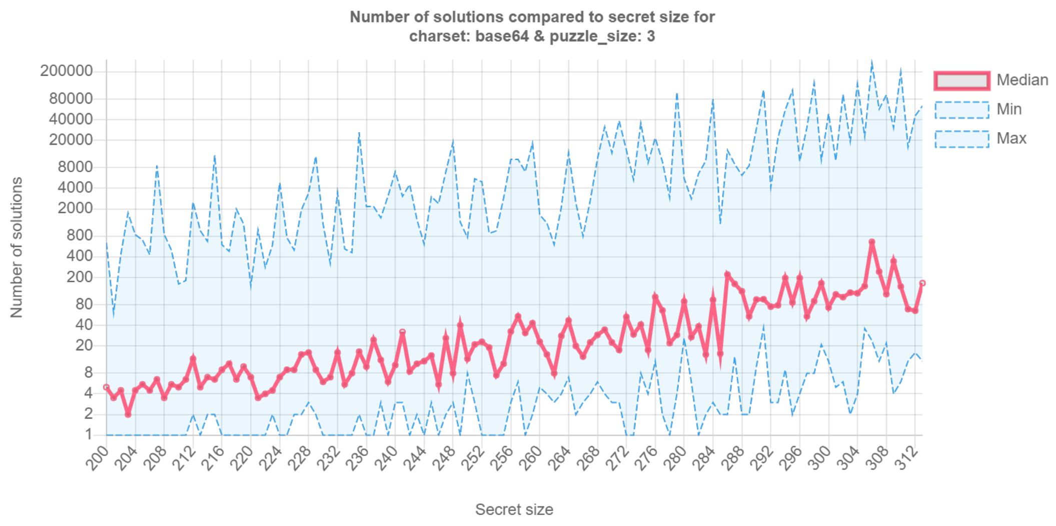

Figure 6.2: Number of solutions compared to secret length for secrets from *base64* charset

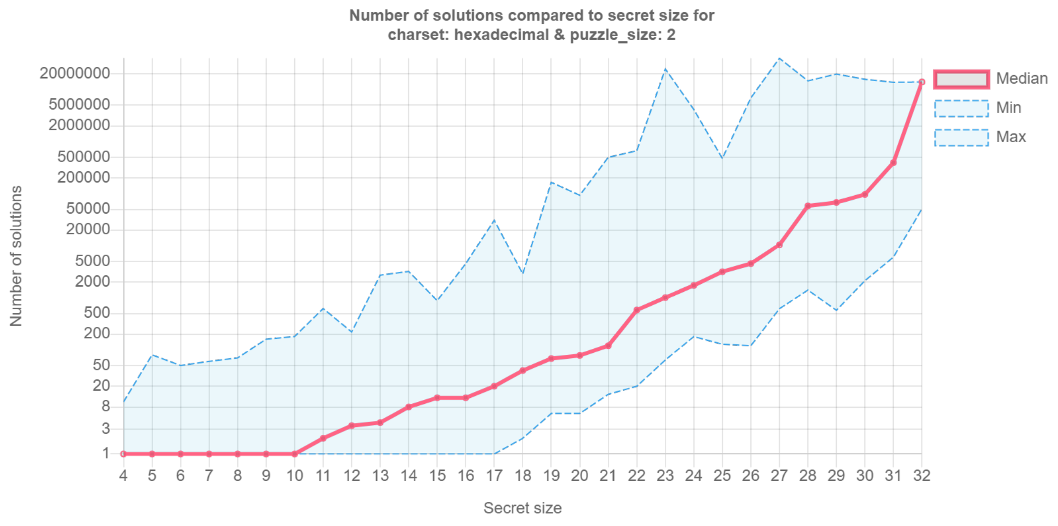

Figure 6.3: Number of solutions compared to secret length for hexadecimal secrets

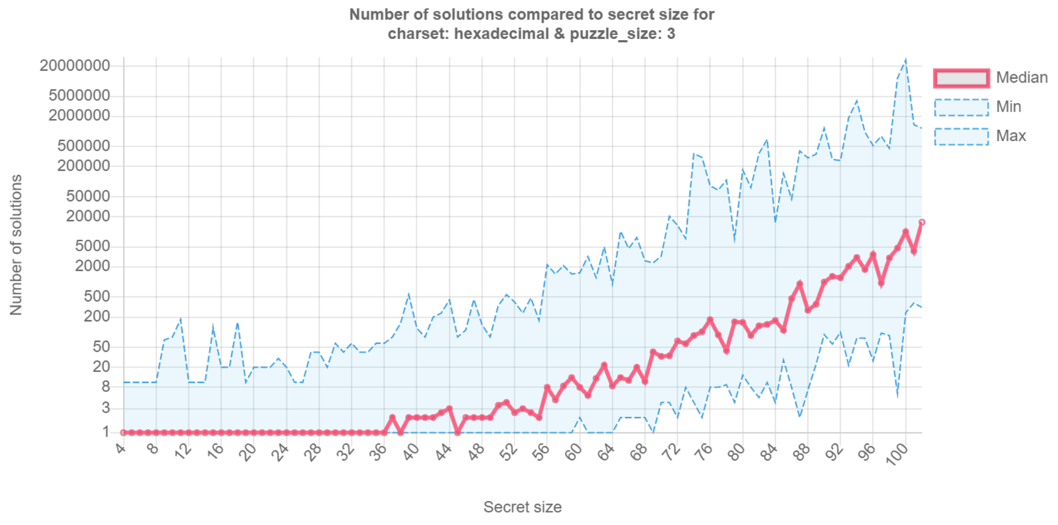

Figure 6.4: Number of solutions compared to secret length for hexadecimal secrets

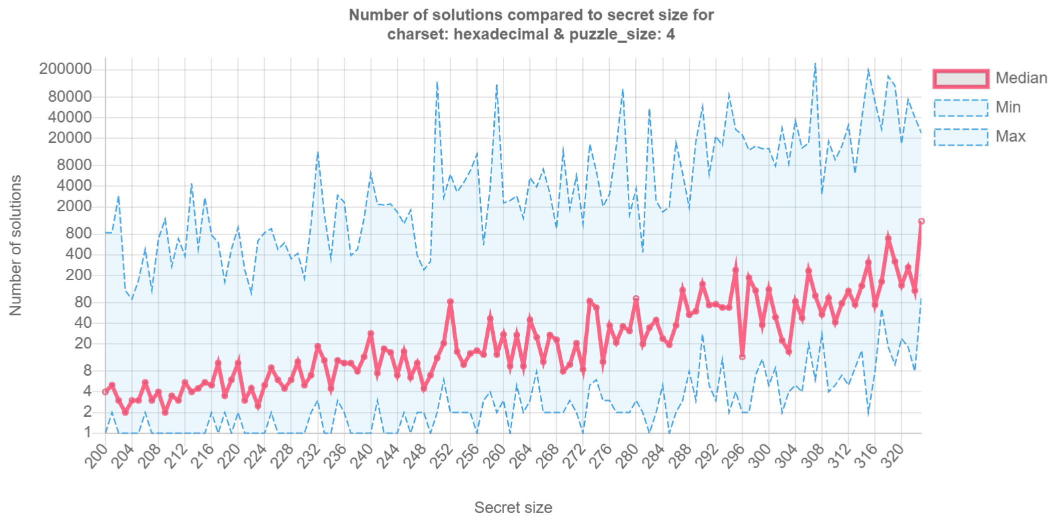

Figure 6.5: Number of solutions compared to secret length for hexadecimal secrets

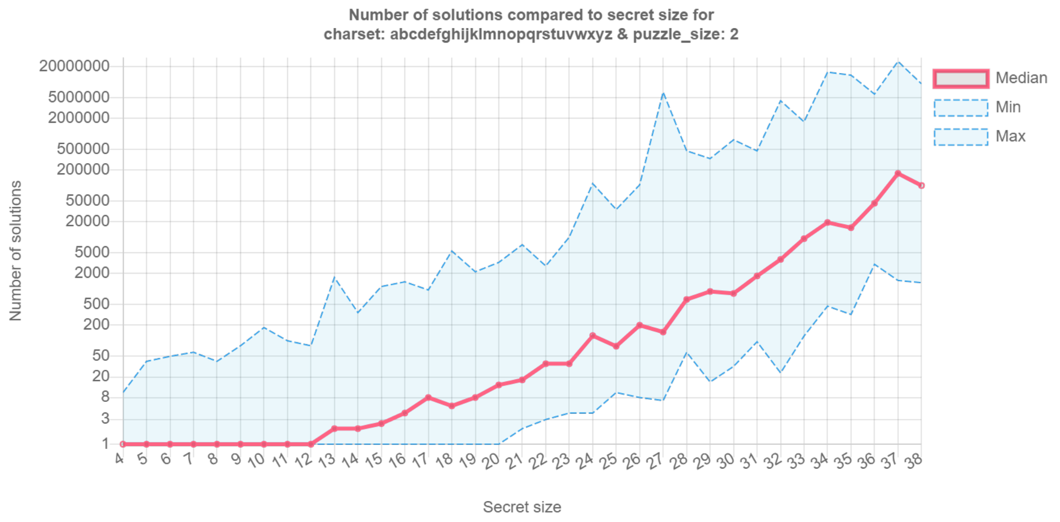

Figure 6.6: Number of solutions compared to secret length for secrets from *letters* charset

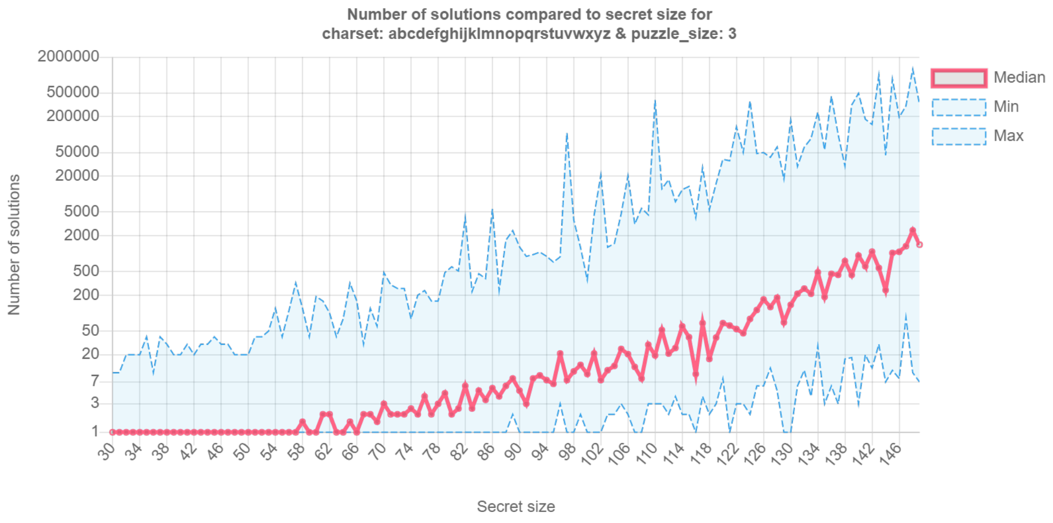

Figure 6.7: Number of solutions compared to secret length for secrets from *letters* charset

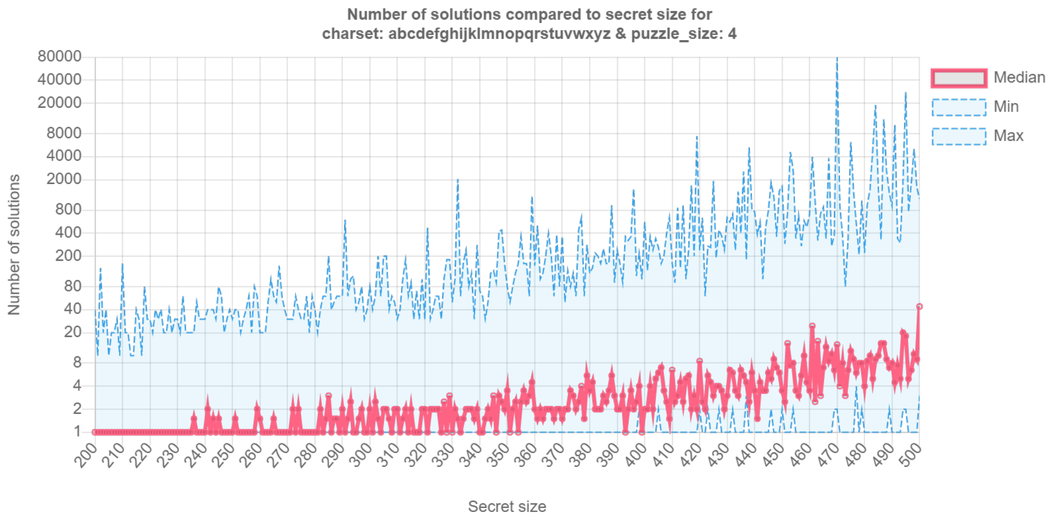

Figure 6.8: Number of solutions compared to secret length for secrets from *letters* charset

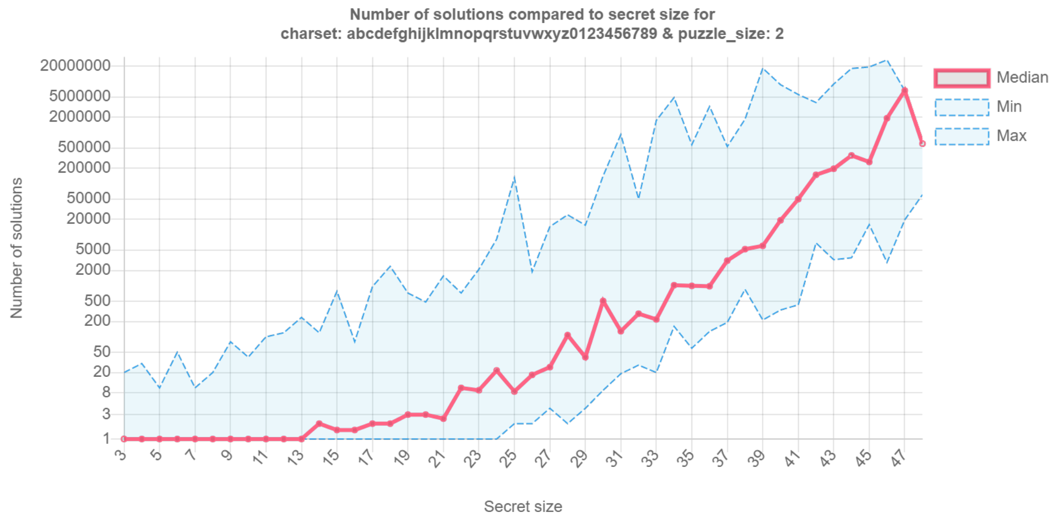

Figure 6.9: Number of solutions compared to secret length for secrets from *lowercase* charset

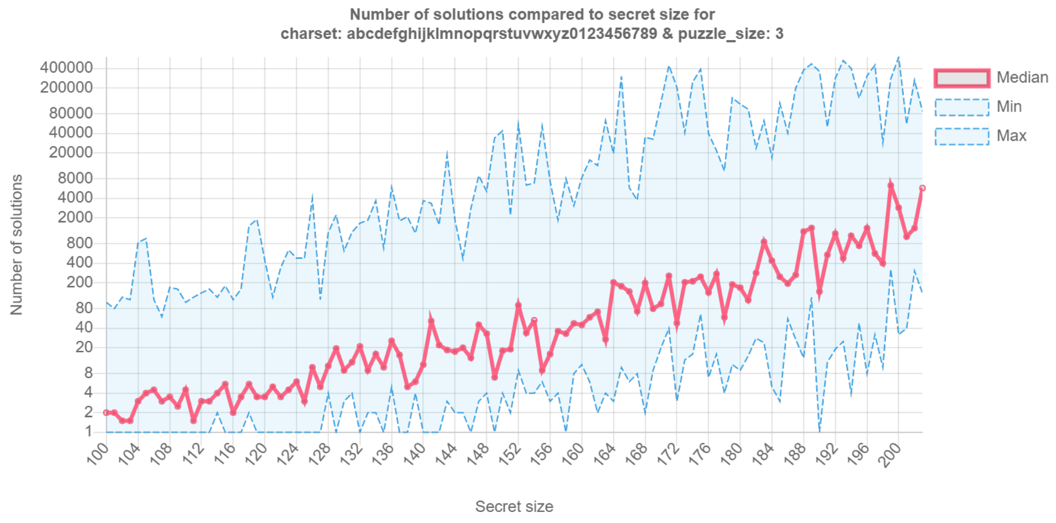

Figure 6.10: Number of solutions compared to secret length for secrets from *lowercase* charset

Chapter 7

Advantages of the Domino technique

There are a few techniques used to exfiltrate information from vulnerable software. The choice of the technique often relies upon what types of questions the vulnerable application accepts. In this chapter, only indirect exfiltration is considered. More specifically, the vulnerable application can only answer with Yes and No responses.

7.1 Common techniques

Common techniques that can be found on the Internet rely on multiple-step questions – to ask a new question the technique may need an answer to the previous. A few known techniques rely on that behavior.

7.1.1 Binary Search

Binary search is considered the most efficient technique but can be only applied in specific situations. The vulnerable application must usually either accept complex questions or perform any kind of comparison between objects. A great example of a situation when an application accepts complex questions is Blind SQL Injection [8] From Wikipedia:

> Blind SQL injection is used when a web application is vulnerable to an SQL injection but the results of the injection are not visible to the attacker. The page with the vulnerability may not be one that displays data but will display differently depending on the results of a logical statement injected into the legitimate SQL statement called for that page.

In SQL injection, every bit of the information can be exfiltrated through the binary search algorithm. This can be achieved by crafting questions in the form of: *Is the first character of the secret less than 'e'?* which in SQL language translates to `SELECT SUBSTRING(secret, 1, 1) < "e";`.

7.1.2 Prefix and suffix extension

Prefix and suffix extension can be often seen in the situation where a vulnerable application responds to questions like *Does the secret start/end with "e"?*. The example of that case are CSS-Leaks[1]. The principle of the attack is to take advantage of CSS Selectors [9] that allow styling elements containing certain phrases. For example, a question *If the value of the secret element starts with 'e', set the background image to external resource* translates to: `input[name=secret][value^=e]{background: url(external)}` in CSS language. The external resource can be used to send the retrieved information to the attacker. As in the *Binary Search*, requesting a new character requires information about the previous. To get the second character of the secret, a selector `[value^=ex]` must be inserted.

7.1.3 Expansion technique

If an application responds to questions in the form of *Does secret include 'e'?*, it is possible to successively extend the searched phrase by the newly discovered characters, just as was presented in Table 1.1 in the *Introduction* chapter and in Prefix and suffix extension but starting from a random character. The technique is especially useful in vulnerabilities known as XS-Search [2] that often search for information in the mentioned way.

7.2 Advantages of the Domino technique

An advantage of the Domino technique over other described techniques is that the technique does not rely on the previous results. Therefore, it can be run in parallel. The technique is especially useful in mentioned *CSS-Leaks*, where multiple tries cannot be performed or are too inefficient.

A popular technique that imitates a single step attack was described in the article *Better Exfiltration via HTML Injection* [10]. It connects *Prefix and suffix*

[1]The term CSS-Leaks refers to the class of vulnerabilities called CSS Injection. *CSS injection vulnerabilities arise when an application imports a style sheet from a user-supplied URL, or embeds user input in CSS blocks without adequate escaping. They are closely related to cross-site scripting (XSS) vulnerabilities but often trickier to exploit.* [15] [16]

[2]Cross-site search (XS-Search) refers to the class of attacks that circumvent the same-origin policy [11] in the browsers through numerous applicable side-channels attacks called cross-site leaks (XS-Leaks). [13] [12]

extension technique with the recursive imports in CSS via `@import` rule. The application is stalled until the attacker receives the information about the searched character. After that, the server extends the generated styles by the newly discovered characters and imports them as a stalled import rule by simply returning the response. The process is repeated until the full secret is retrieved. The disadvantages of the technique over the Domino technique are:

- external stylesheets are often blocked by Content Security Policy [3]
- because of the nature of the attack, if the searched secret occurs after the injected stylesheets in the code, the information about that element cannot be retrieved

In both cases, the Domino technique should work if only the size of the crafted exploit is accepted. Because the technique often requires many validations to be performed as shown in Table 4.1, it may result in a significant exploit size.

[3]Content Security Policy *CSP* is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting *XSS* and data injection attacks. These attacks are used for everything from data theft to site defacement to distribution of malware. [14]

Chapter 8

Applications of the Domino technique

8.1 Vulnerable websites

As part of the research, two websites were found vulnerable to CSS-Leaks that in combination with Domino technique might result in personal information exposure or successful theft of the online accounts. Both websites provided the functionality of custom styles uploads used to alter the default appearance of the website. This, however, allowed to request all pairs constructed from base64 charset and expose a unique *CSRF token*[1] that is used to protect against *Cross-site request forgery*[2] attacks and therefore, take full control over the user's account. In the other case, personal information such as email addresses, full name, email titles could be retrieved by attackers.

Both vulnerabilities were reported to vendors which at the time of writing the thesis have not been yet fixed.

8.2 Recreated example

While the permission for the disclosure of the discovered vulnerabilities has not yet been granted, a similar scenario has been recreated and included to the thesis

[1]Synchronizer token pattern (STP) is a technique where a token, secret and unique value for each request, is embedded by the web application in all HTML forms and verified on the server side. The token may be generated by any method that ensures unpredictability and uniqueness (e.g. using a hash chain of random seed). The attacker is thus unable to place a correct token in their requests to authenticate them. [18]

[2]Cross-site request forgery, also known as one-click attack or session riding and abbreviated as CSRF (sometimes pronounced sea-surf) or XSRF, is a type of malicious exploit of a website where unauthorized commands are transmitted from a user that the web application trusts. [18]

files[3]. A simple page imitating the behavior of the vulnerable application is shown in the Listing 3. In the real scenario, the attacker would control the contents of the `exploit.css` stylesheet. The goal of the attack is to retrieve the value of the `_csrf` input element.

By combining CSS Selectors with the Domino technique, it is possible to retrieve all pairs of character from the base64 charset and then recover the original token of length 36.

In Table 6.1 it can be read that for a puzzle of size 2 and a secret of length 36 from *base64* charset the expected number of solutions is between 10-100. Table 8.1 shows that there are exactly 4 possible solutions that can be recovered from 34 pairs[4] included in the *ZiKWm3RS_tbhXUVneRXT8cTwNYpN5yCeRv-f* token, which is less than expected number. In Figure 6.1 it can be noticed that the minimum number of solutions in the conducted 30 experiments for a secret of size 36 was as low as 1 unique solution produced, hence the actual number of the solution can be lower than the expected value.

In real case scenario, that recovered token could be used to perform actions on behalf of the user in the introduced *CSRF* attack and which could lead to account theft, also called *account takeover*[19].

No	Solution
1	ZiKWm3RS_tbhXUVneRXT8cTwNYpN5yCeRv-f
2	ZiKWm3RS_tbhXT8cTwNYpN5yCeRXUVneRv-f
3	ZiKWm3RXUVneRS_tbhXT8cTwNYpN5yCeRv-f
4	ZiKWm3RXT8cTwNYpN5yCeRS_tbhXUVneRv-f

Table 8.1: Number of solutions for the CSRF token

[3]The files can be found in the `./usecase/css-leaks/` directory.

[4]List of the pairs used to recover the token: *Zi*, *iK*, *KW*, *Wm*, *m3*, *3R*, *RS*, *S_*, *_t*, *tb*, *bh*, *hX*, *XU*, *UV*, *Vn*, *ne*, *eR*, *RX*, *XT*, *T8*, *8c*, *cT*, *Tw*, *wN*, *NY*, *Yp*, *pN*, *N5*, *5y*, *yC*, *Ce*, *Rv*, *v-*, *-f*. It can be noticed that the pair *eR* occurs in the token twice but is used only once in the algorithm.

```html
<!DOCTYPE html>
<html>
<head>
    <title>Vulnerable application</title>
    <!-- The file exploit.css is controlled by an attacker -->
    <link rel=stylesheet href="exploit.css"/>
</head>
<body>
<h1>Vulnerable to CSS-Leaks application</h1>
<input name="_csrf" value="ZiKWm3RS_tbhXUVneRXT8cTwNYpN5yCeRv-f"/>
</body>
</html>
```

Listing 3: A simple HTML page imitating the vulnerable application

Chapter 9

Conclusion

The goal of this thesis was to determine whether a Domino technique is an effective method of a secret retrieval. The conducted experiments have confirmed that in real-case scenarios the technique should work sufficiently, i.e. produce a relative low number of unique solutions within which the searched secret is included. The chapter *Technique overview* summarized in what scenarios it could be optimal to use the technique. Nevertheless, as the length of the secret increases, the effectiveness of the technique drops, i.e. the number of unique solutions increases. The intuition behind that behavior is that a percentage of all tuples from a charset included in the Domino set grows; hence more sequences can be assembled.

Although the presented implementation of the Assembly algorithm shown in the chapter *The assembly algorithm* has its flaws (mostly high memory usage), it does not impact the effectiveness of the technique itself. The idea for improvements to the performance of the implementation has been presented in the chapter *Further optimizations* along with a brief reasoning why the proposed optimizations should work.

What is special about the Domino technique and what makes the technique unique, is the property of independence to the previous states, i.e. to retrieve the secret, prior information about the results is not required. Therefore, the technique can be effectively parallelized.

The technique is a good fit to *CSS-Leaks* introduced in the *Applications of the Domino technique* chapter. Since the technique does not rely on the previous results, whole secrets can be retrieved without reloading the page, which other known techniques could not achieve.

In the files to the thesis, the implementation of the Assembly algorithm in the form of a *Javascript* module and *command-line client* has been included along with the recreated example of a vulnerable application and a small application to draw graphs from the data collected from over 15,000 experiments. A brief manual how to use the included programs has been described in Appendix A.

Appendix A

Brief user manual

A.1 Domino solver

In the included files, the assembly algorithm can be found in the `./solver` directory.

A.1.1 Requirements

In order to use the `domino-solver`, Node.js[1] in version 10.0.0 or later must be installed.

A.1.2 Useful commands

- add `--max-old-space-size=8000` after `node` command to increase the allocated memory for the heap to 8,000 MB. The default value is only 512 MB.

A.1.3 Command-line client

To run the command-line client of the Domino solver, execute command `node domino-solver-cli.js --help` from the `./solver` directory. This will provide a detailed instruction on how to use the program. For example, `node domino-solver-cli.js recover ex xa am mp pl le -S 7 -t json` will produce all the solutions in the JSON format.

```
> node domino-solver-cli.js recover ex xa am mp pl le -S 7 -t json
[
  "example",
  "xamplex",
  "amplexa",
```

[1]<https://nodejs.org/en/>

```
  "mplexam",
  "plexamp",
  "lexampl"
]
```

A.1.4 Javascript module

Usage

To use the `domino-solver` module, it can be included in the Javascript code via:

```
const solver = require('domino-solver')
```

Methods

Three methods are implemented in the module:

- `solve()` – returns a list of solutions for a given set of puzzles and the size of the secret
- `make_puzzle()` – returns a list of puzzles for a given secret and the size of a single puzzle
- `test_solve()` – splits the given secret into a list of puzzles and executes `solve()` method on them.

Each method is described in the snippet below:

```
// tries to solve Domino problem for the domino set as puzzles and the
// size of the secret as secret_size, and returns a list of the solutions
solver.solve(
    puzzles: Array<string>,
    secret_size: Number,
    settings?: JSON
) : Array<string>

// returns the secret split into puzzles of the size puzzle_size
solver.make_puzzles(
    secret: String,
    puzzle_size: Number,
    settings?: JSON
) : Array<string>

// splits the secret into puzzles of the size of puzzle_size,
// attempts the recovery and returns list of the solutions
```

```
solver.test_solve(
    secret: String,
    puzzle_size: Number,
    settings?: JSON
) : Array<string>
```

Options

Each method can take an optional argument `settings` described as the snippet below:

```
settings:
  withDuplicates  don't remove duplicates from the puzzles
                                          [boolean] [default: false]
  maxRepetitions  number of maximum repetitions for all puzzles.
                  Default value is calculated from secret_size
                                                            [number]
  hints           list of hints that must occur in each solution
                                         [array<hint>] [default: []]
  best_match      return the best match if no solutions were found
                                          [boolean] [default: false]
  shuffle         randomize the order of the puzzles
                                          [boolean] [default: false]
  onlyDebug       only return debug info, without solutions
                                          [boolean] [default: false]
  debugFile       file path where to save the debug info      [string]
  debug           display debug logs into the console
                                          [boolean] [default: false]
  fullKnowledge   keep all DP states in the memory
                                           [boolean] [default false]
type hint:
    string | RegExp | Array<hint>

    string:      hint must occur in a solution
    RegExp:      regular expression that a solution must match
    Array<hint>: list of hints from where at least one must match

    Examples:
        "xyz":                    must occur in solution
        /abc/:                    regular expression that must
                                  match the solution
        ["xyz", /abc/]:           at least one rule from the
                                  list must be satisfied
        [
            "xyz",
            /abc/,
           ["xyz2", ["ijk", /abc2/]]
        ]:                           more complex construction
```

Example usage

The example usage of the `domino-solver` module, also located in the `./prod/solver/example.js` file, is shown below.

```
const solver = require('domino-solver')

const puzzles = solver.make_puzzles('example', 2);
const test_solutions = solver.test_solve('example', 2);
const solutions1 = solver.solve(puzzles, 7);

const solutions2 = solver.solve(
  solver.make_puzzles('example', 3),
  20
);
const solutions3 = solver.solve(
  solver.make_puzzles('example', 3),
  20, {
    bestMatch: true
  }
);

const solutions4 = solver.solve(puzzles, 7, {
    hints: [
        /^e/
    ]
});

console.log(puzzles);
console.log(test_solutions);
console.log(solutions1);
console.log(solutions2);
console.log(solutions3);
console.log(solutions4);

/*
 *   Expected output:
```

```
*
*   [ 'ex', 'xa', 'am', 'mp', 'pl', 'le' ]
*   [ 'example', 'xamplex', 'amplexa', 'mplexam', 'plexamp', 'lexampl' ]
*   [ 'example', 'xamplex', 'amplexa', 'mplexam', 'plexamp', 'lexampl' ]
*   []
*   [ 'example' ]
*   [ 'example' ]
*/
```

A.2 Experiments

In the included files, all results from the conducted experiments have been placed in the `./experiments/` directory, where:

- `./data` folder includes results of all conducted experiments.
- `./metrics` folder includes metrics used in the thesis from the experiment data.
- `./plots` folder includes graphs used in the thesis.

A.2.1 Plot generator

All plots were generated using chart.js[2] library. To use the included the single-page application, any HTTP server will be sufficient. The main file is located in the following location: `./experiments/plot.html`.

To generate graphs from the available data:

1. Start a simple HTTP server through the command:

   node simple-server.js

2. Visit http://localhost:8888/plot.html[3].

3. You should see pregenerated metrics from data from the `./metrics` folder.

4. To generate a custom graph, the form at the top of the page can be used. For the input `hex` | `2` | `32` | `4`, clicking the `generate chart` button will generate a chart from the file located in `./data/hex/2/32/test_4.json`.

Step 1 can be omitted if the HTTP server was already set up. Otherwise, `Node.js` *will be required.*

[2]<https://www.chartjs.org/>

[3]<http://localhost:8888/plot.html>

A.3 Vulnerable application

The recreated vulnerable app from the previous chapter is included in the `./usecase/css-leaks` directory. To run the program, `Node.js` is required.

Run the exploit

1. In the console, execute the `node server/receive.js` command from the same folder.
2. Visit http://localhost:1337/leak.html[4] to see the process of secret retrieval in action.

Experiment explanation

1. Generated by the script `rules.js` file `server/public/exploit.css` is included as stylesheets.
2. Upon visiting the URL `leak.html`, a random base64 secret is generated.
3. The secret is inserted into the document in the form of `<input name="_csrf" value="[secret]" hidden>`.
4. The stylesheet `exploit.css` leaks CSRF token split into pairs of two characters from the inserted `<input>` element.
5. The server `server/receive.js` receives all puzzles and stores them in the memory.
6. Upon requesting http://localhost:1337/solutions?secret_size=36[5] the assembly algorithm is run which returns all solutions.
7. The solutions are inserted into the document.

[4] `<http://localhost:1337/leak.html>`

[5] `<http://localhost:1337/solutions?secret_size=36>`

List of Figures

List of Tables

Bibliography

[1] ”Sequence assembly” *Wikipedia*, last modified November 20, 2019, https://en.wikipedia.org/wiki/Sequence_assembly

[2] ”Dominoes” *Wikipedia*, last modified January 26, 2020 , https://en.wikipedia.org/wiki/Dominoes

[3] ”Wang tiles” *Wikipedia*, last modified November 29, 2019, https://en.wikipedia.org/wiki/Wang_tile

[4] ”Cartesian Product” *Wikipedia*, last modified January 1, 2020, https://en.wikipedia.org/wiki/Cartesian_product

[5] ”justCTF 2019 write-ups by @terjanq” *terjanq*, last updated Jan 23, 2020, https://hackmd.io/@terjanq/justctf_writeups

[6] ”Jump search” *Wikipedia*, last modified January 28 2020, https://en.wikipedia.org/wiki/Jump_search

[7] ”Node.js Garbage Collection Explained” *Gergely Nemeth*, last updated Nov 15, 2016, https://blog.risingstack.com/node-js-at-scale-node-js-garbage-collection/

[8] ”SQL injection” *Wikipedia*, last updated: January 14, 2020, https://en.wikipedia.org/wiki/SQL_injection#Blind_SQL_injection

[9] ”CSS selectors” *MDN web docs*, Mozilla, Last modified: Oct 17, 2019, https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors

[10] ”Better Exfiltration via HTML Injection”, *d0nut*, Apr 11, 2019, https://medium.com/@d0nut/better-exfiltration-via-html-injection-31c72a2dae8b

[11] ”Same-origin policy” *MDN web docs*, Mozilla, Last modified: Nov 19, 2019, https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[12] ”Massive XS-Search over multiple Google products”, *terjanq*, Nov 12, 2019, https://medium.com/@terjanq/massive-xs-search-over-multiple-google-products-416e50dd2ec6

[13] "Browser Side Channels.", *E. Vela Nava, L. Herrera, R. Masas, K. Kotowicz, A. Saftnes, terjanq, Stephen*, 2020, `https://github.com/xsleaks/xsleaks/wiki/Browser-Side-Channels`

[14] "Content Security Policy (CSP)", *MDN web docs*, Mozilla, Last modified: Jan 23, 2020, `https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP`

[15] "Stealing Data in Great style – How to Use CSS to Attack Web Application." *Michał Bentkowski*, September 25, 2017, `https://research.securitum.com/stealing-data-in-great-style-how-to-use-css-to-attack-web-application/`

[16] "CSS injection (reflected)" *PortSwigger*, 2020, `https://portswigger.net/kb/issues/00501300_css-injection-reflected`

[17] "Data Exfiltration" *techopedia*, `https://www.techopedia.com/definition/14682/data-exfiltration`

[18] "Cross-site request forgery" *Wikipedia*, `https://en.wikipedia.org/wiki/Cross-site_request_forgery`

[19] "What is Account Takeover?", `https://www.shieldsquare.com/what-is-account-takeover/`