Understanding the fundamentals of attacks

What is happening when someone writes an exploit?

Halvar Flake / Thomas Dullien

November 3, 2016

The /home owners association

Table of contents

The need to define “exploit”

What, exactly, is an ‘exploit’?

What, exactly, is an exploit?

Central concept in computer security.

Valuable currency: Used to be barter-traded, now is traded for $$$.

Could you define it so that an old-school computer scientist understands?

What, exactly, is an ‘exploit’?

In spite of centrality of the concept, it is ill-defined.

Informal definitions abound, from overly abstract to overly concrete:

“A program that allows you to do something that you are not supposed to be able to do.”

“I recognize an exploit when I see it.”

“Magic”

“When you overwrite EIP”

Why is the lack of a definition a problem?

Problem 1:

Since we do not have clear definitions, we fall into scholastic arguments:

"Adding a cookie to the heap here will stop exploits - it breaks the technique described in Phrack!"

"It is just a single bit error, and hence not exploitable."

Since we do not have clear definitions, these statements cannot be reasoned about properly.

Why is the lack of a definition a problem?

Computer security is full of non-scientific statements, assertions, and unquantifiable handwaving.

Mitigations are supposed to "raise the bar", but:

Bar-raising

1. How much will the bar be raised?
2. Will it cost the attacker time **once**, or will it cost him time on each exploit?
3. Is the cost (performance, complexity) worth the gain?

Our discussions are similar to "how many angels can balance on the tip of a needle".

Why is the lack of a definition a problem?

Most defenders have no first-hand experience with a modern exploit.

With security jobs doubling every 4 years, scholasticism is a real problem.

Incentive structures are harmful:

Happy defender

1. Come up with overspecific countermeasures
2. Show progress to superiors & get promoted

Happy attacker

1. Bypass overspecific countermeasures
2. Hack all the things
3. Show progress to superiors & get promoted

Why is the lack of a definition a problem?

Problem 2:

We fail at teaching it properly, and instead teach a "bag of tricks"

The broken exploitation curriculum:

1. Learn about stack smashing
2. Learn about heap implementation X, Y, and Z
3. Learn about SQL injection
4. Learn about ASLR and other mitigations
5. Learn about XSS
6. Learn about other overly special cases

Disjoint, disparate, and terrible at transmitting principles. Too many people "miss the forest for the trees" as a result.

Why is there no good theoretical body on exploits?

"Philosophy of science is about as useful to scientists as ornithology is to birds" (Feynman)

Exploit practitioners know the intuitions they need, and have no incentive to formalize or disseminate their mental models.

This talk

Explains the "right" way to think about exploits and exploit writing[1].

Introduces an attacker model for memory corruptions that allows us to reason about large classes of attacks.

Discusses a few theoretical results & their consequences.

Omits proofs and extreme formalism.

... but remains a theoretical CS talk, so don't fall asleep!

[1] "Right" as subjectively judged by me, but I have impeccable taste in exploits.

Credit where it is due

Intuitions behind this talk are known to many exploit practitioners.

"Folk theorems" - results are known, but not formally written down anywhere.

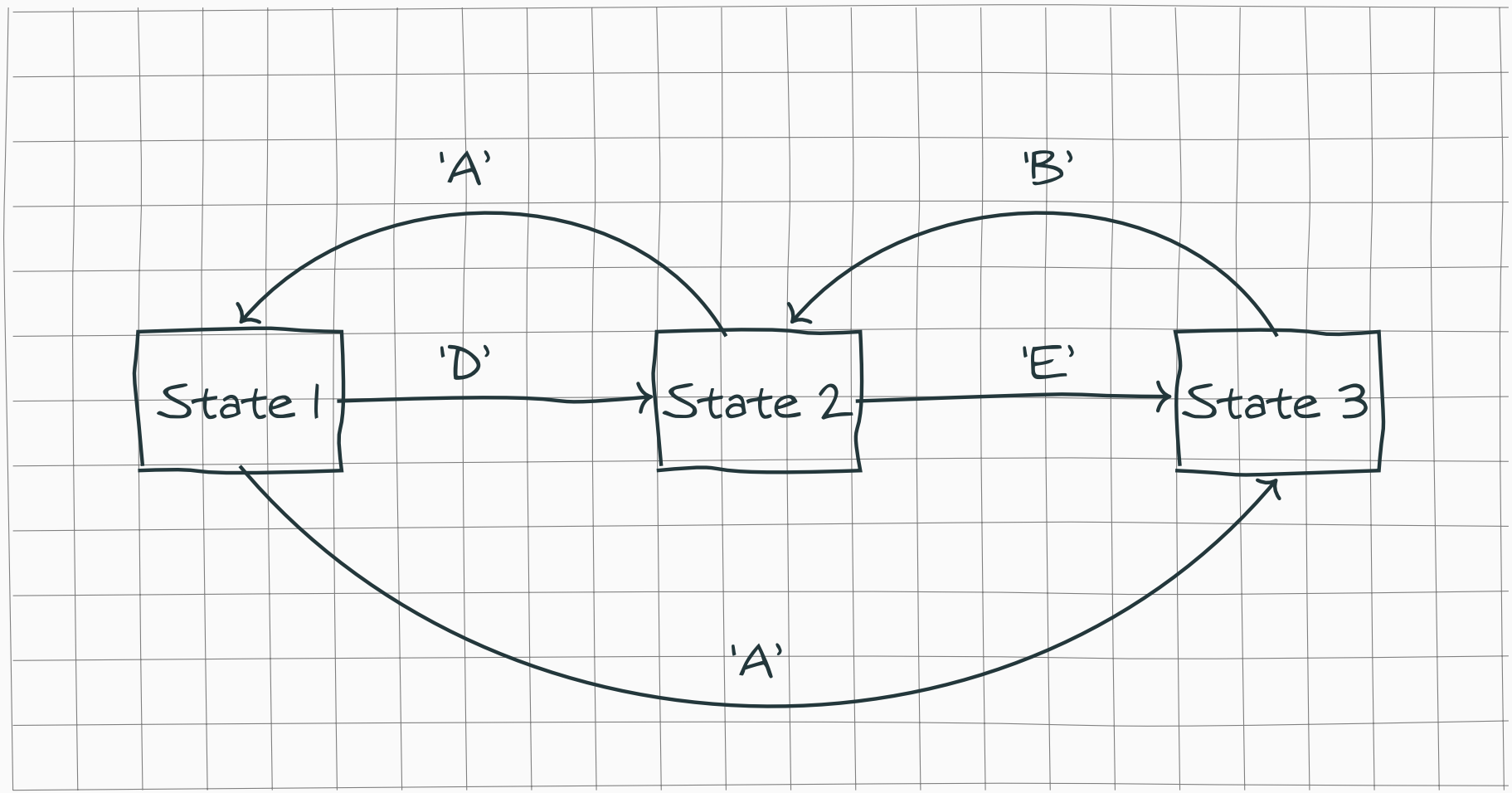
The LANGSEC[2] community have contributed strongly to the discussion and evolution of ideas.

[2] L. Sassaman, M. L. Patterson, S. Bratus, J. Vanegue etc. ...

Defining bug-free and buggy software

Recap: Finite state machines

A graph of states, and input drives the machine along edges ("transitions") to other states.



If you are unfamiliar with finite-state machines, think of such diagrams:

The intended (finite state) machine

Why write software? Because you wish to address a problem.

You want to have a machine that does something specific for you.

Ideally, you want to have a specific finite-state machine that addresses your problem.

The intended (finite state) machine

Unfortunately, you do not have this machine: All you have is a general-purpose computer.

So you build an **emulator** that emulates the machine you would like to have on a general-purpose computer.

Programming is, at its heart, construction of finite-state-machine emulators. [3]

[3] Some people may disagree, but it is true for finite-RAM (e.g. real) machines

The intended (finite state) machine

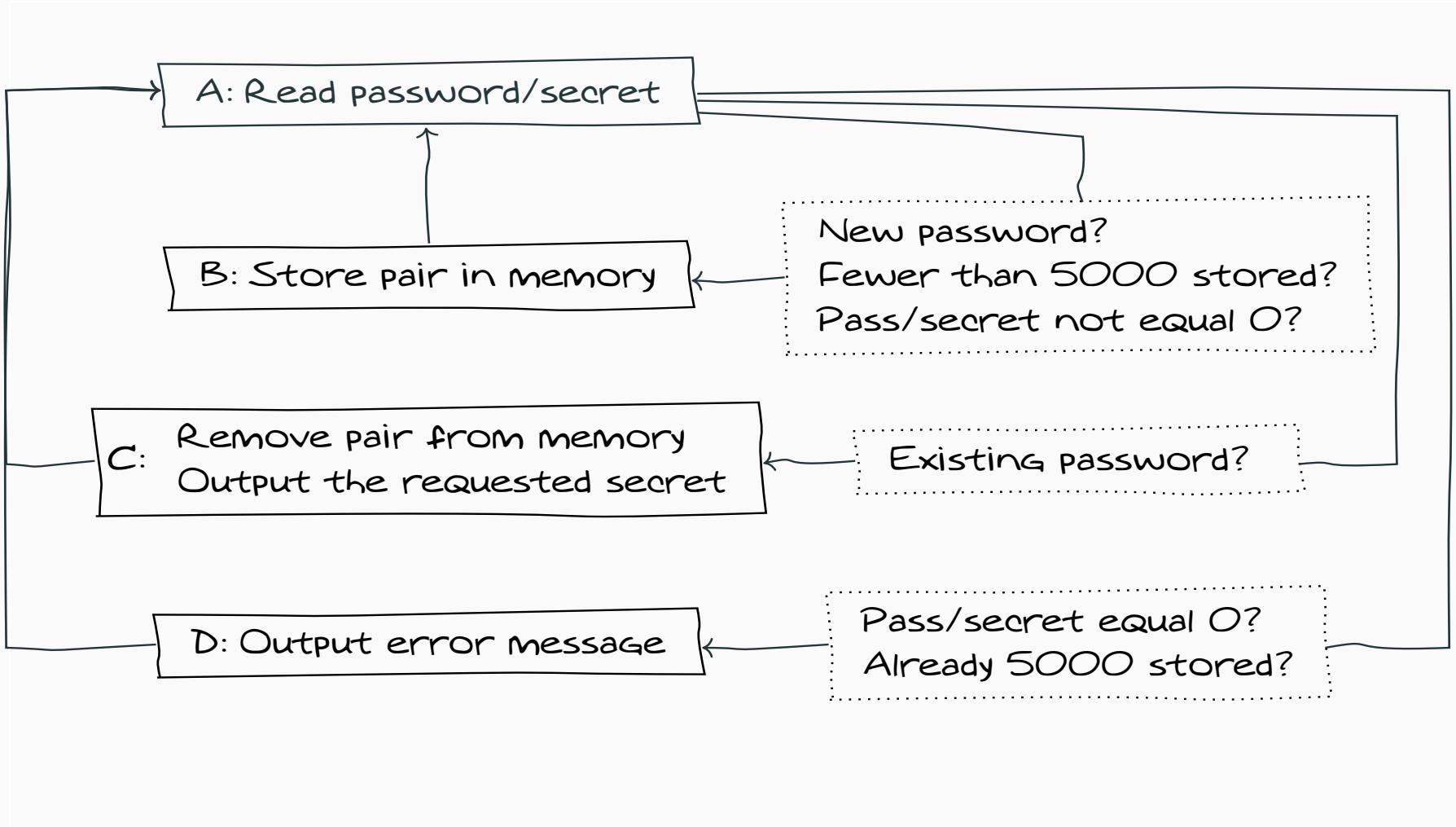
The machine that addresses the problem is the **intended finite state machine**: The machine you wanted to build.

By definition, it is bug-free: It does exactly what you want it to do, and nothing else.

All theory is grey, so let's define a small IFSM: A system that stores 5000 password / secret-value combinations

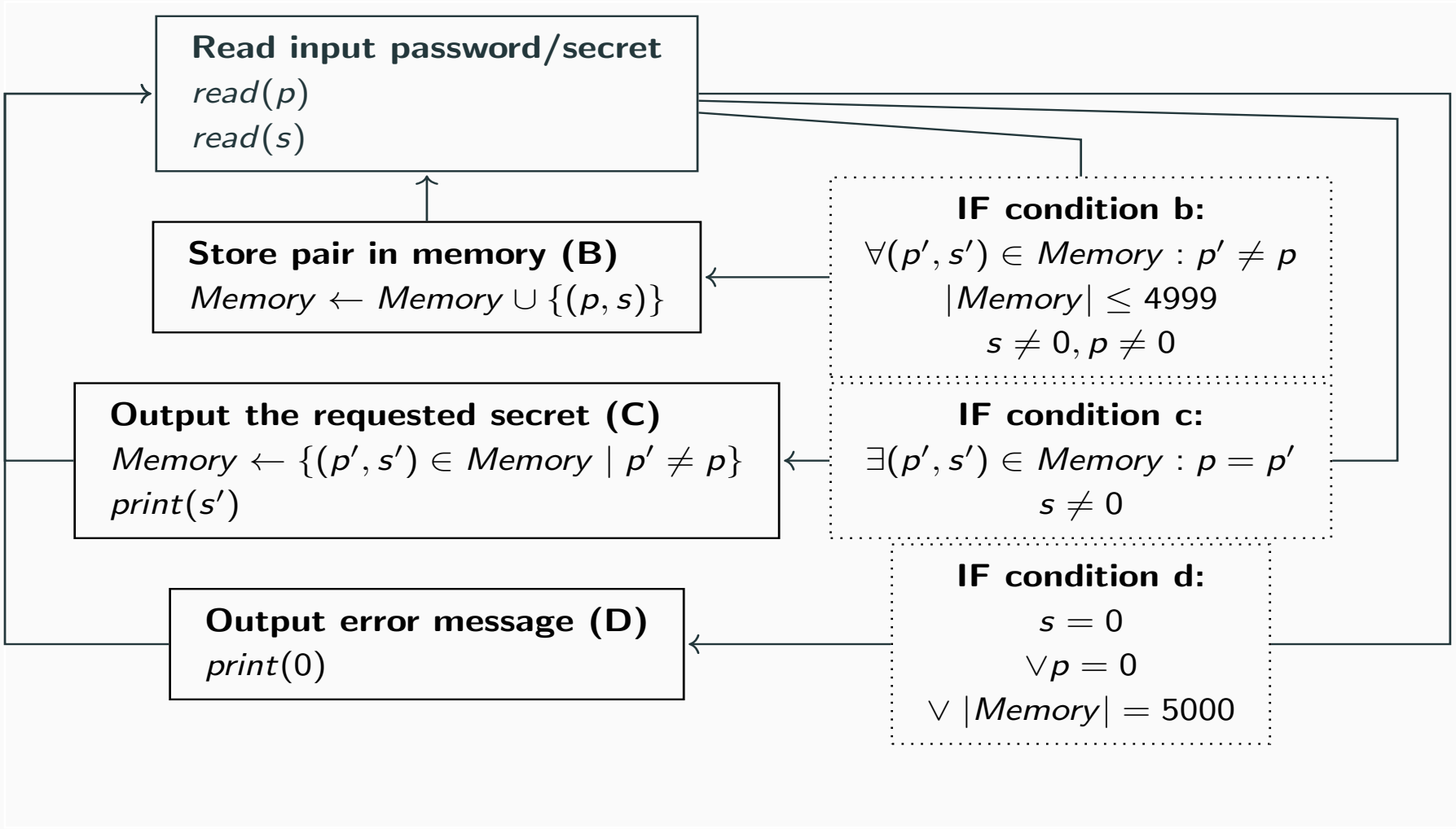
The IFSM

A simple secret-keeping machine:



The IFSM, slightly more formal

A simple secret-keeping machine:



Implementing the IFSM

This machine can be implemented in many ways.

We will examine 2 different implementations.

Memory in the IFSM is a set. Computers do not know sets.

Our two implementations will emulate *Memory* via (1) a flat array and via (2) a linked list.

Flat-array implementation: Writing

Divides memory into pairs of memory cells.

Pairs are either empty (and both 0), or represent secret/password combo.

Insertion is linear scan for empty pair and writing there.

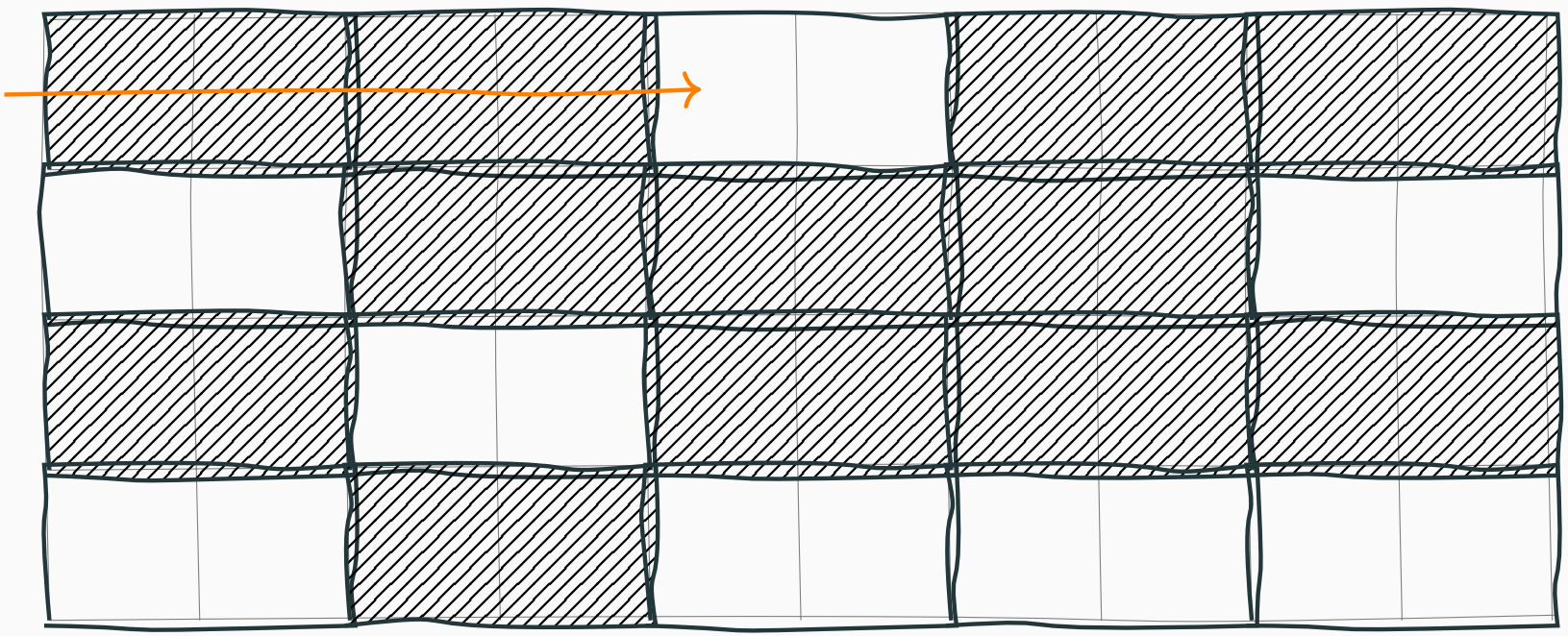
Removal is linear scan for correct pair and removing it.

We refer to such a pair as (p, s).

Flat-array implementation: Writing

Transition (B): $\mathit{Memory} \leftarrow \mathit{Memory} \cup \{(p, s)\}$.

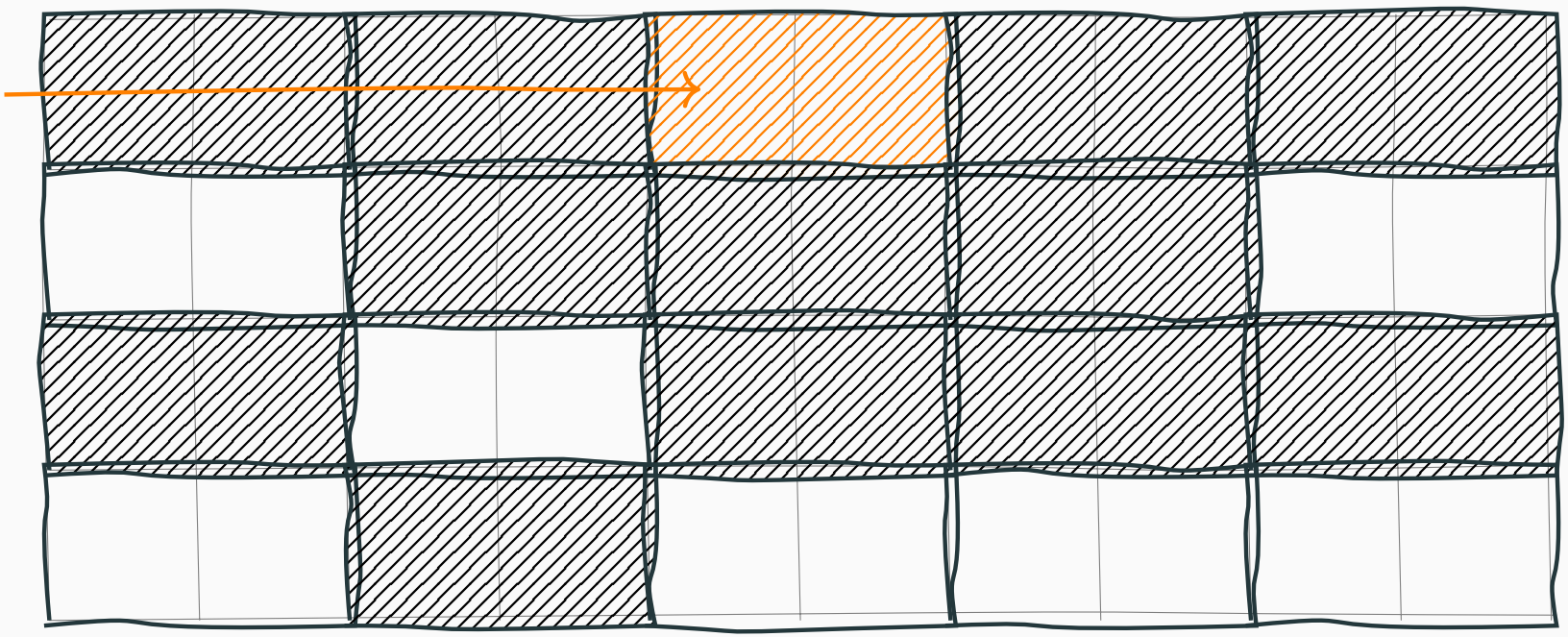
Search for first pair with $p = 0$, write data there.



Flat-array implementation: Writing

Transition (B): $\mathit{Memory} \leftarrow \mathit{Memory} \cup \{(p, s)\}$.

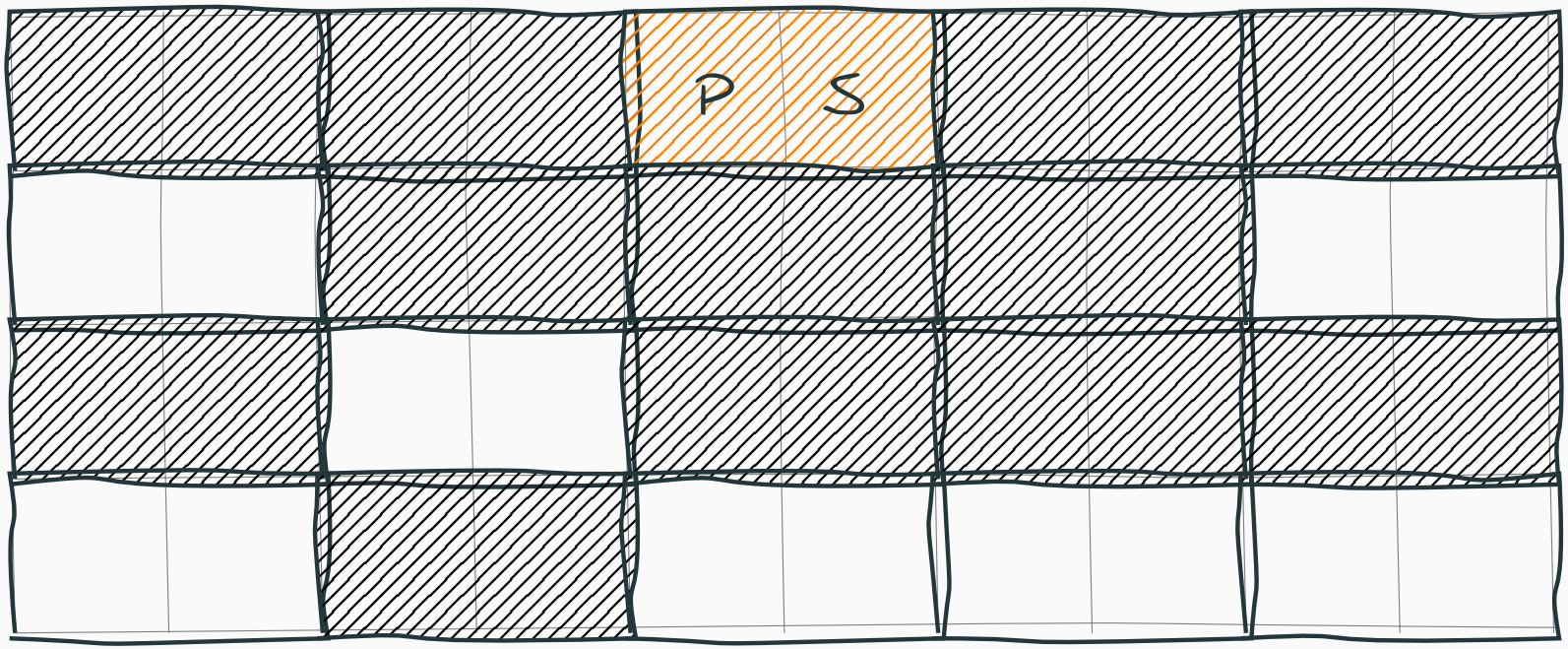
Search for first pair with $p = 0$, write data there.



Flat-array implementation: Writing

Transition (B): $\mathit{Memory} \leftarrow \mathit{Memory} \cup \{(p, s)\}$.

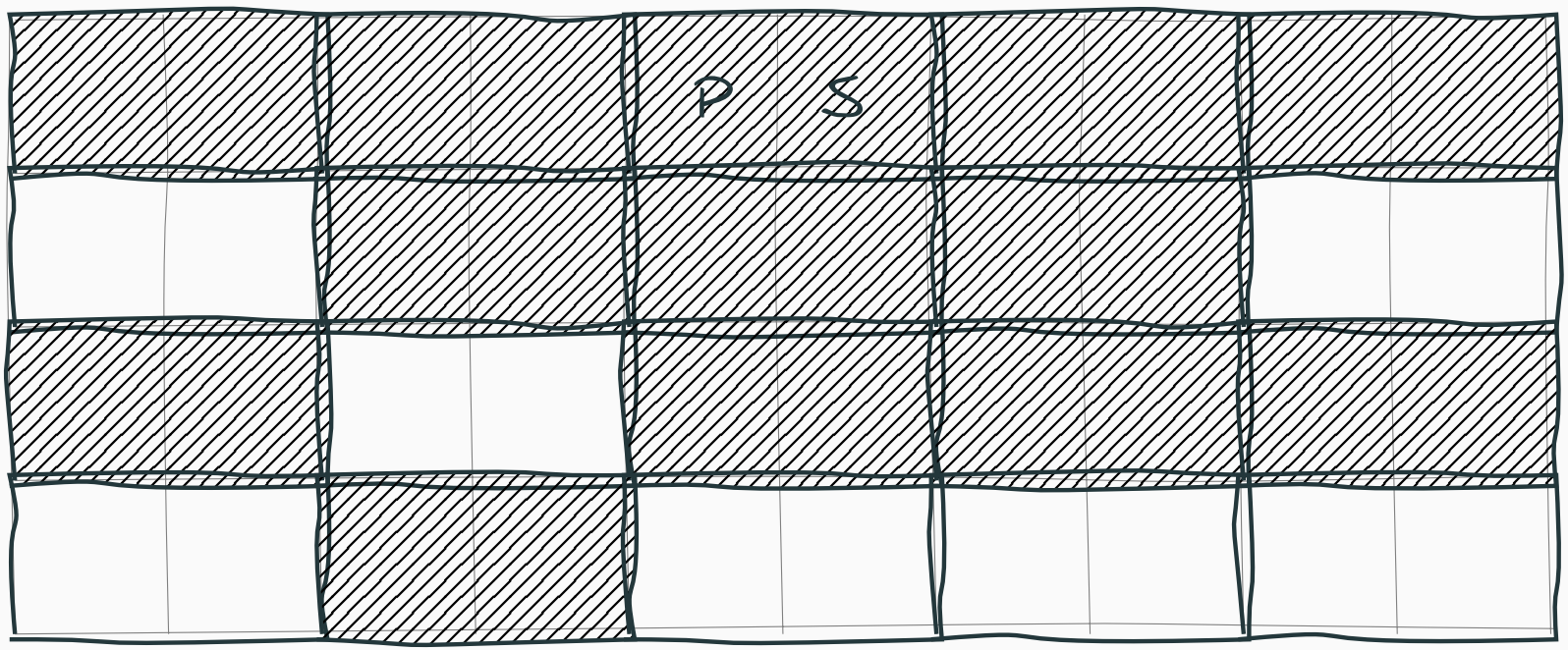
Search for first pair with $p = 0$, write data there.



Flat-array implementation: Writing

Transition (B): $\mathit{Memory} \leftarrow \mathit{Memory} \cup \{(p, s)\}$.

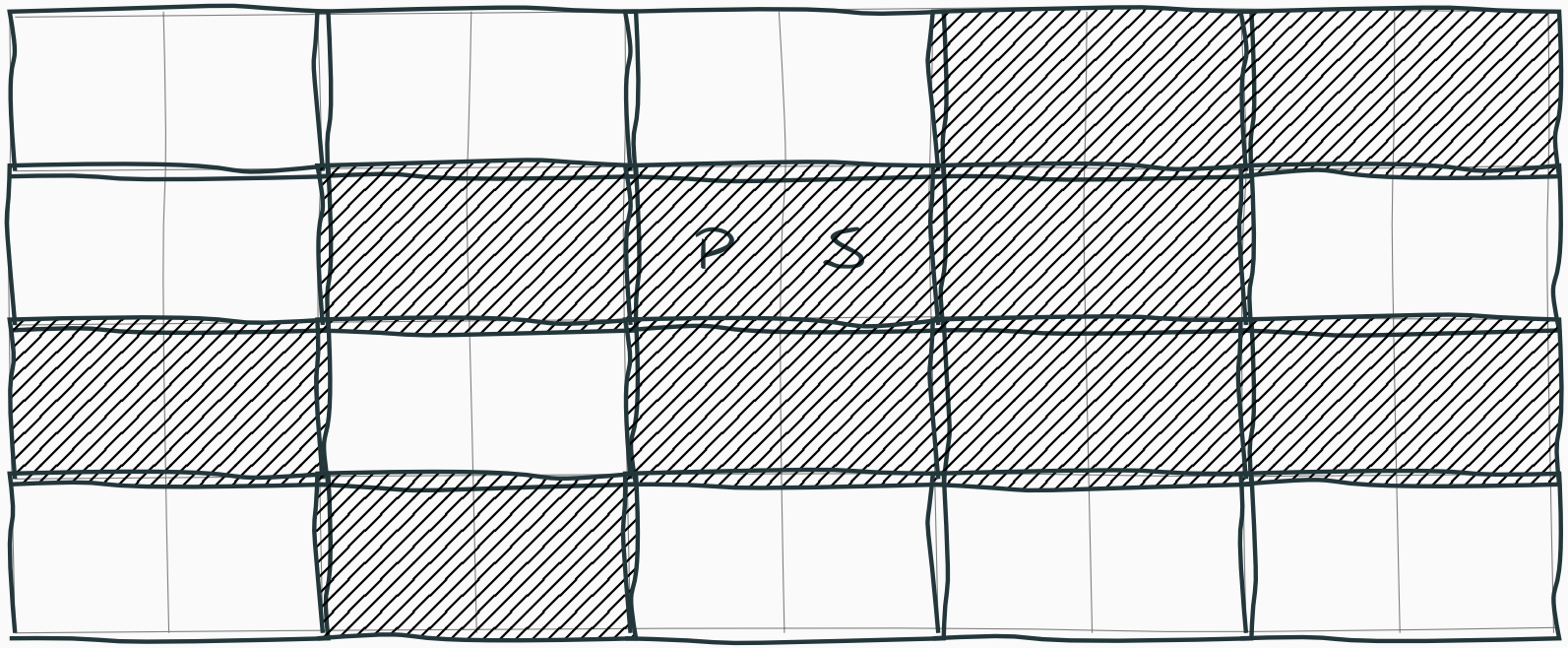
Search for first pair with $p = 0$, write data there.



Flat-array implementation: Reading

Transition C: $Memory \leftarrow \{(p', s') \in Memory | p' \neq p\}$.

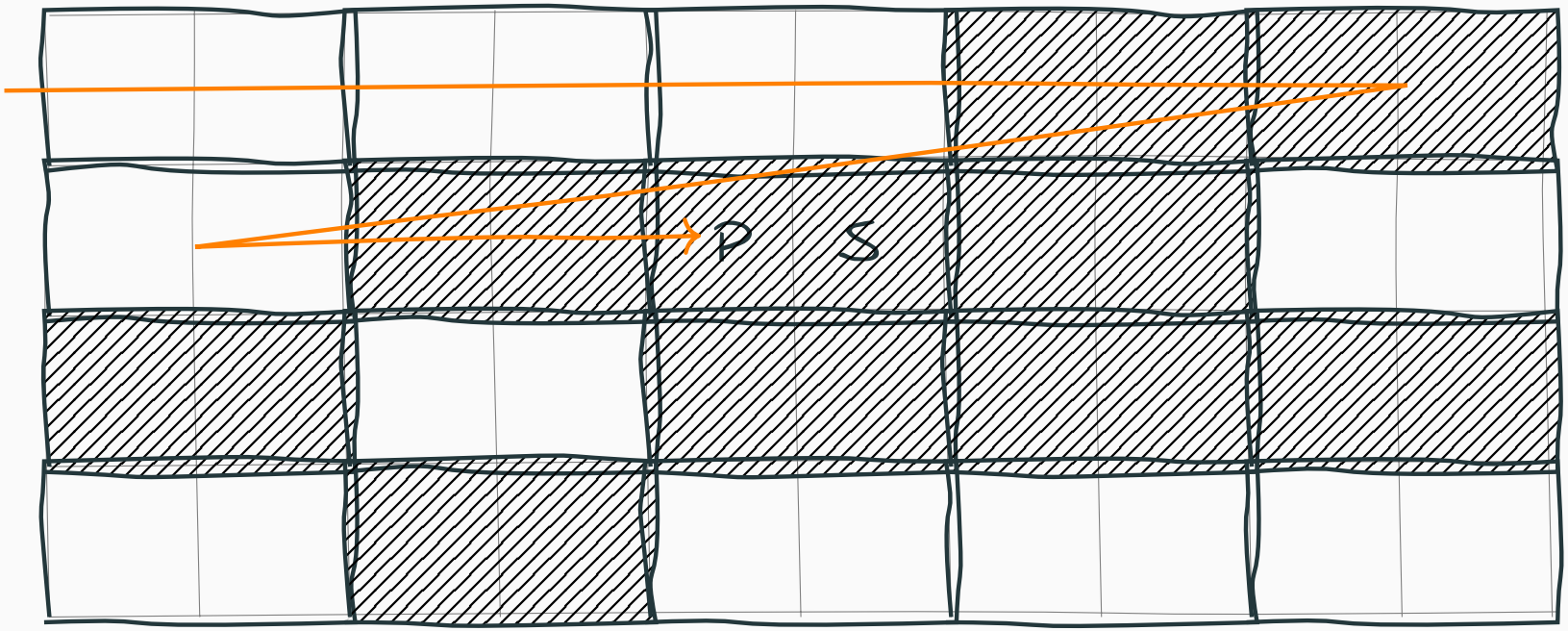
Searching for a specific p and removing the pair.



Flat-array implementation: Reading

Transition C: $Memory \leftarrow \{(p', s') \in Memory | p' \neq p\}$.

Searching for a specific p and removing the pair.

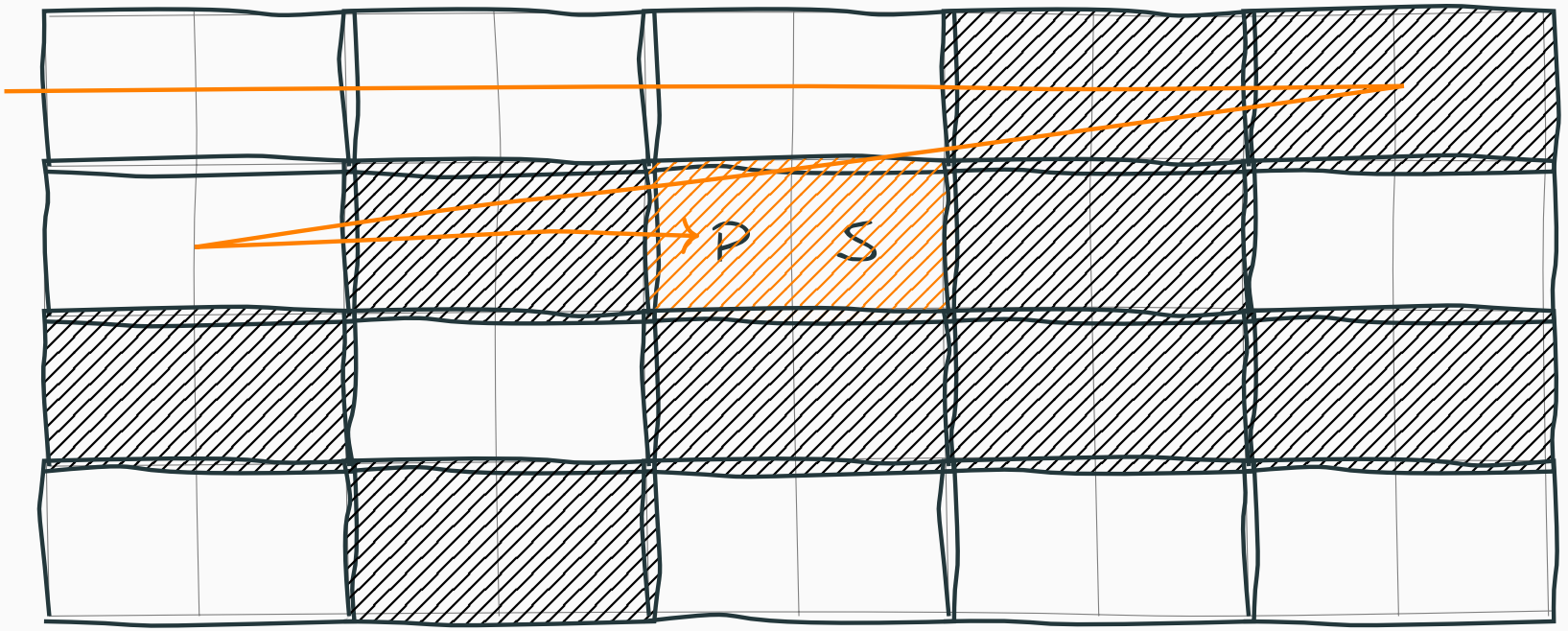


Linear sweep until right p is found.

Flat-array implementation: Reading

Transition C: $Memory \leftarrow \{(p', s') \in Memory | p' \neq p\}$.

Searching for a specific p and removing the pair.

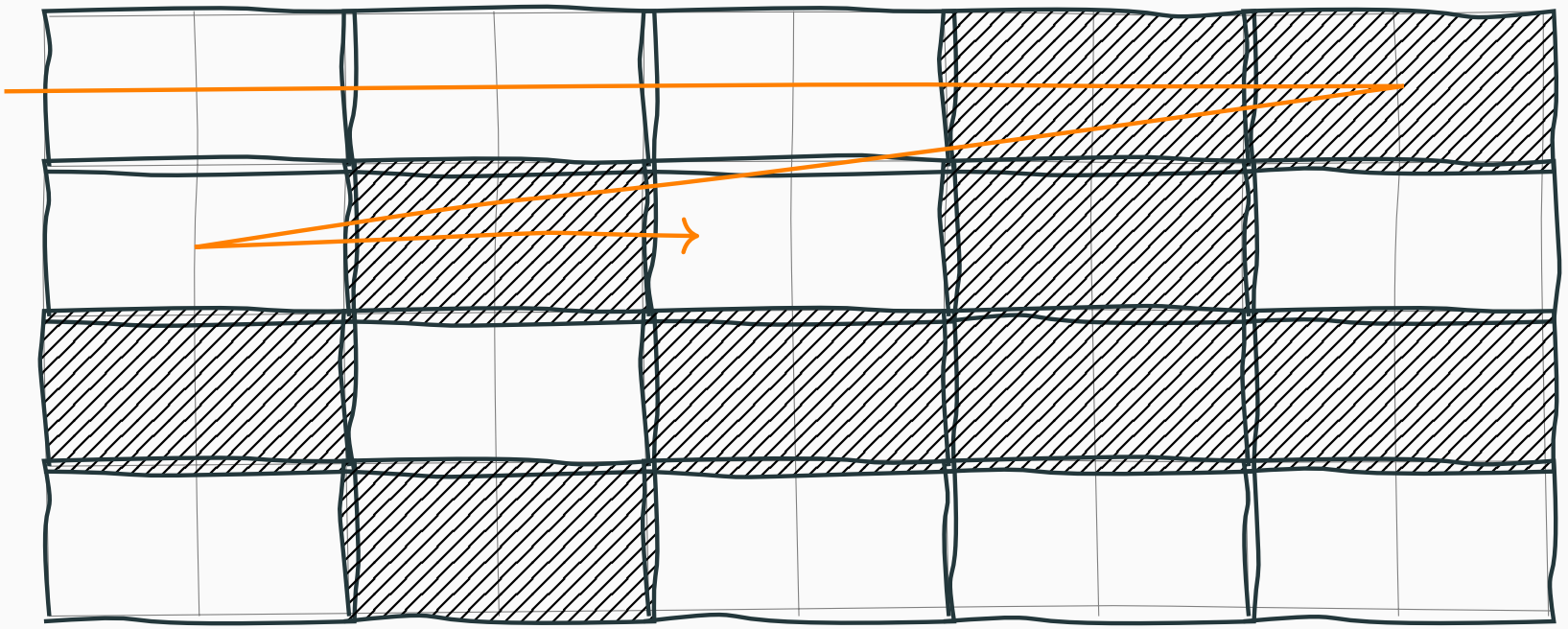


Linearly sweep until right p is found. Output corresponding s.

Flat-array implementation: Reading

Transition C: $\mathit{Memory} \leftarrow \{(p', s') \in \mathit{Memory} | p' \neq p\}$.

Searching for a specific p and removing the pair.

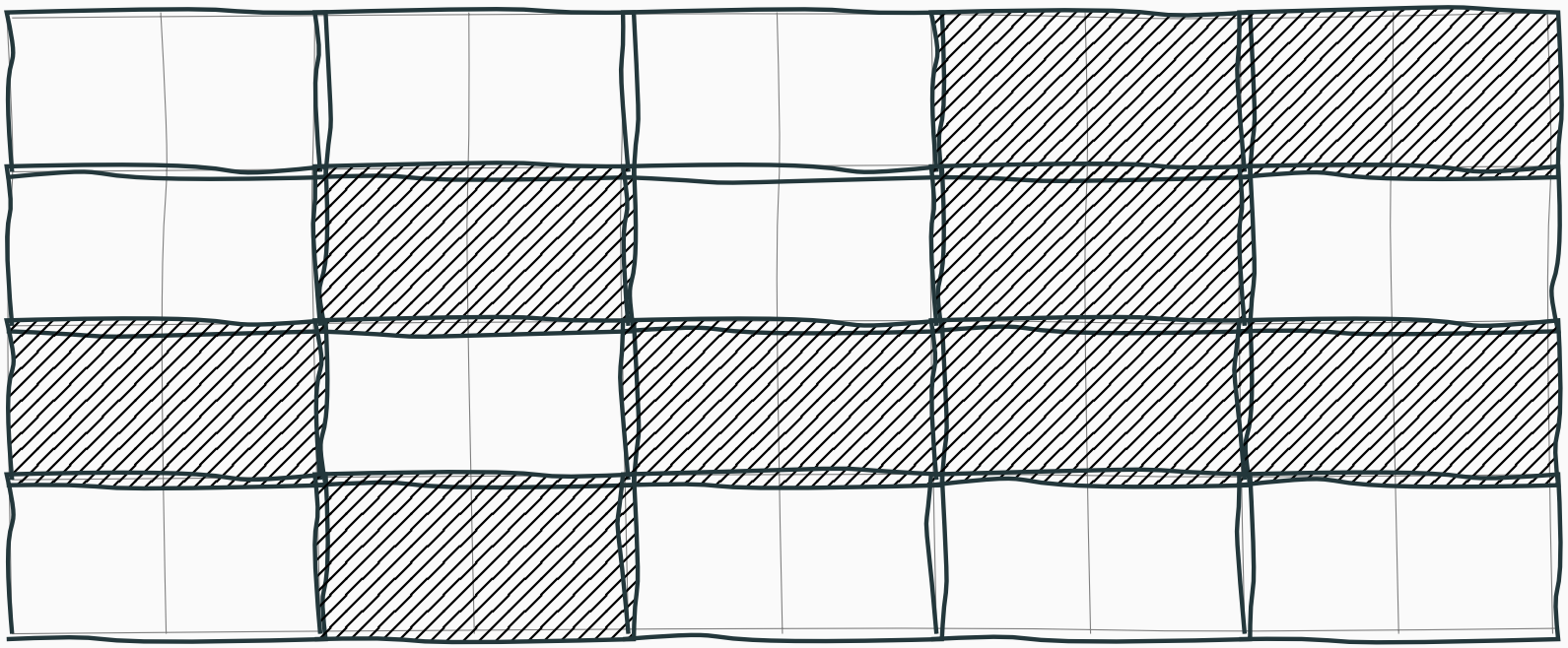


Linearly sweep until right p is found. Output corresponding s. Zero memory cell.

Flat-array implementation: Reading

Transition C: $Memory \leftarrow \{(p', s') \in Memory | p' \neq p\}$.

Searching for a specific p and removing the pair.



Linearly sweep until right p is found. Output corresponding s. Zero memory cell.

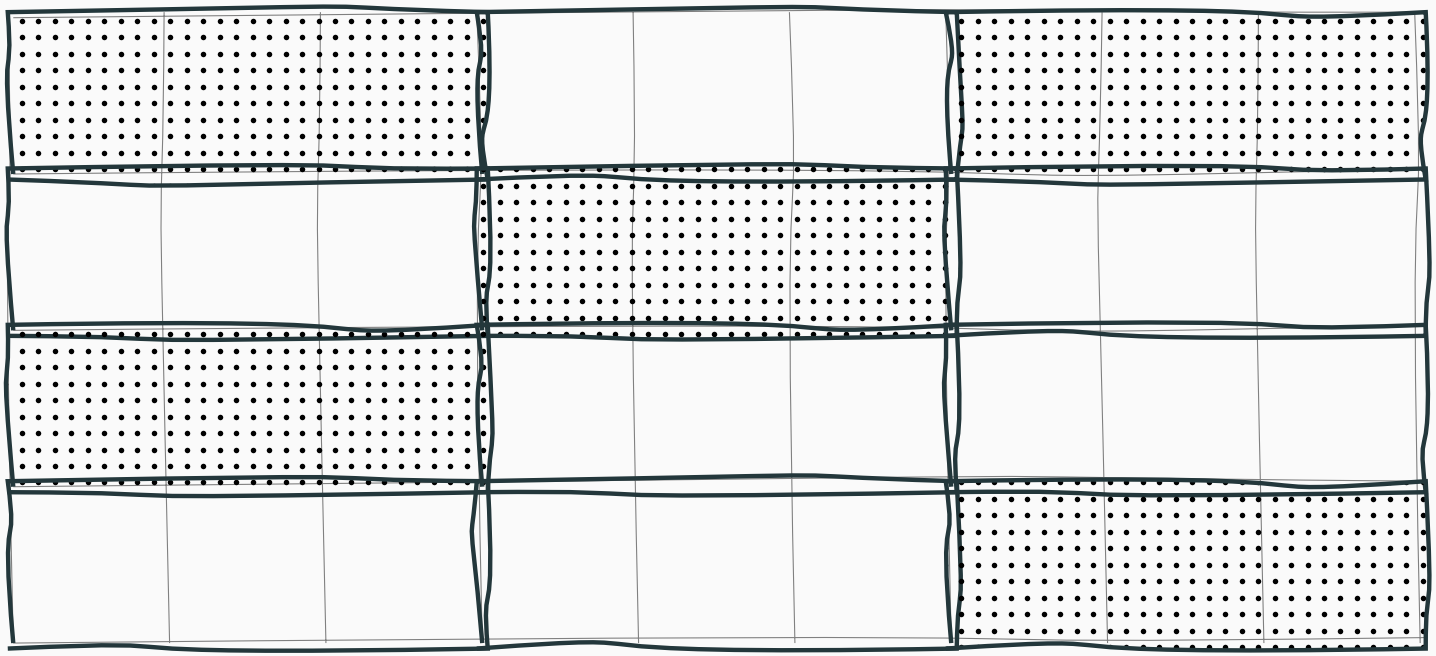
The linked-list implementation

Linearly sweeping memory is computationally expensive and not very CSy.

Let us replace the flat array with a linked list for free and used tuples.

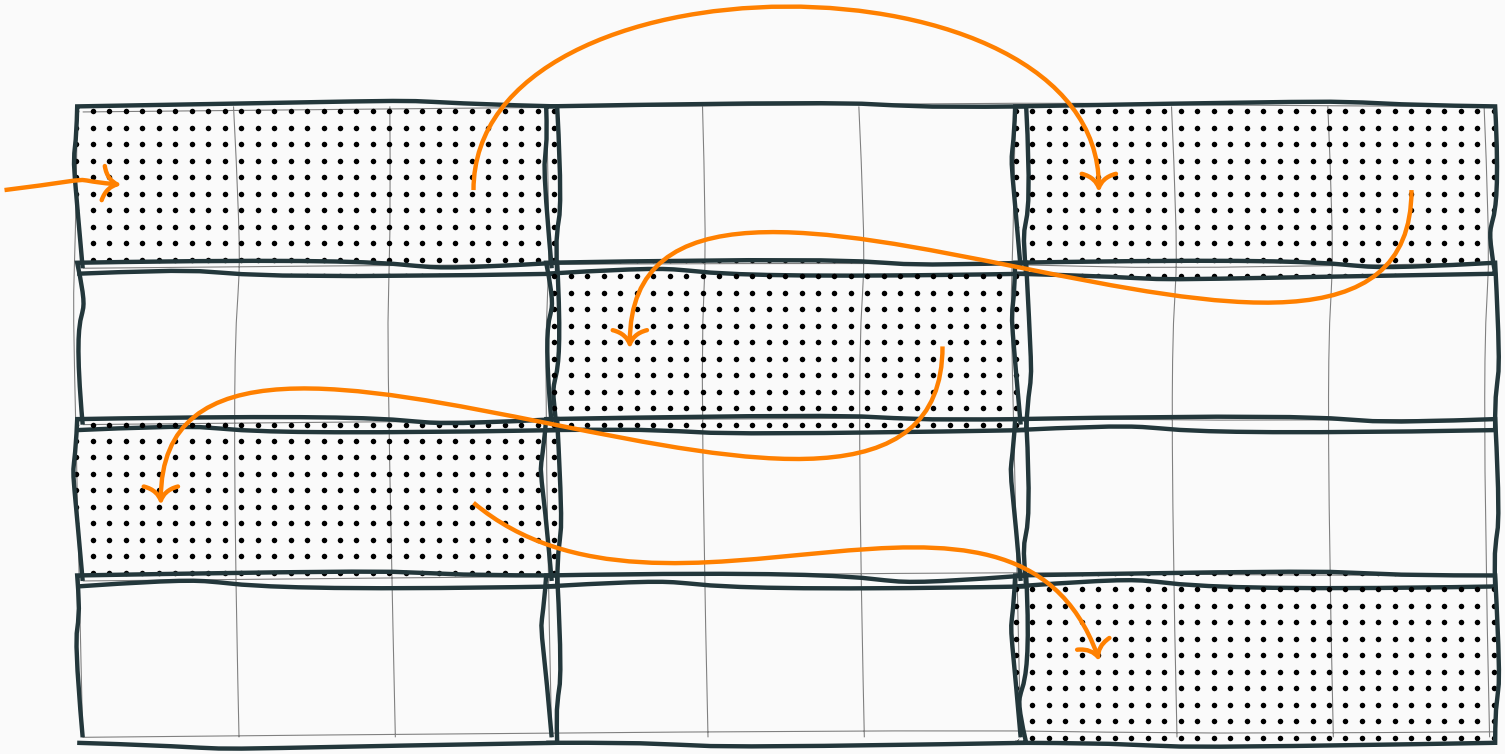
Divide memory into 3-tuples: $(p, s, next)$. Keep track of two list heads for free and used tuples.

Linked-List implementation



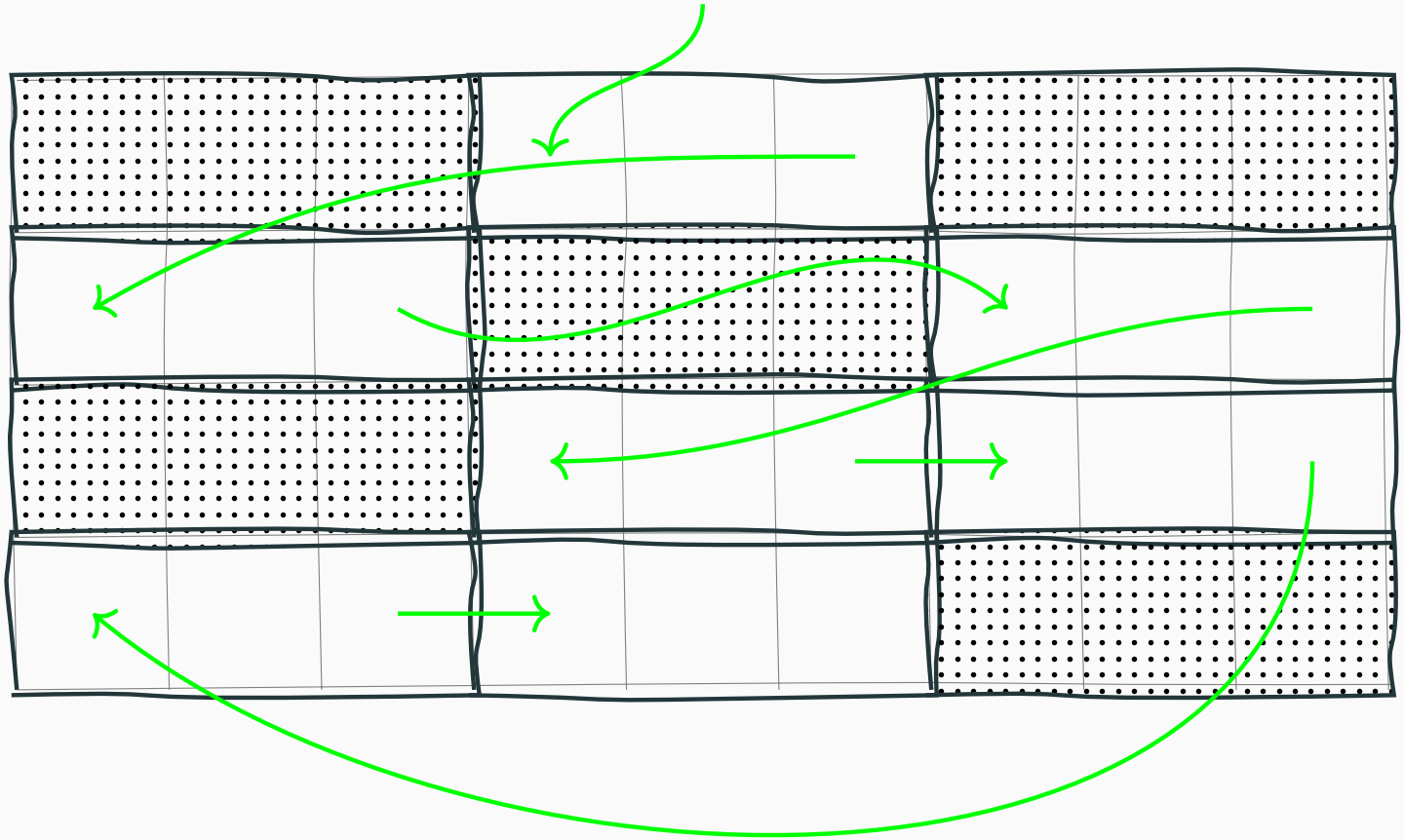
Linked-List implementation

Keep all used blocks in a linked list.



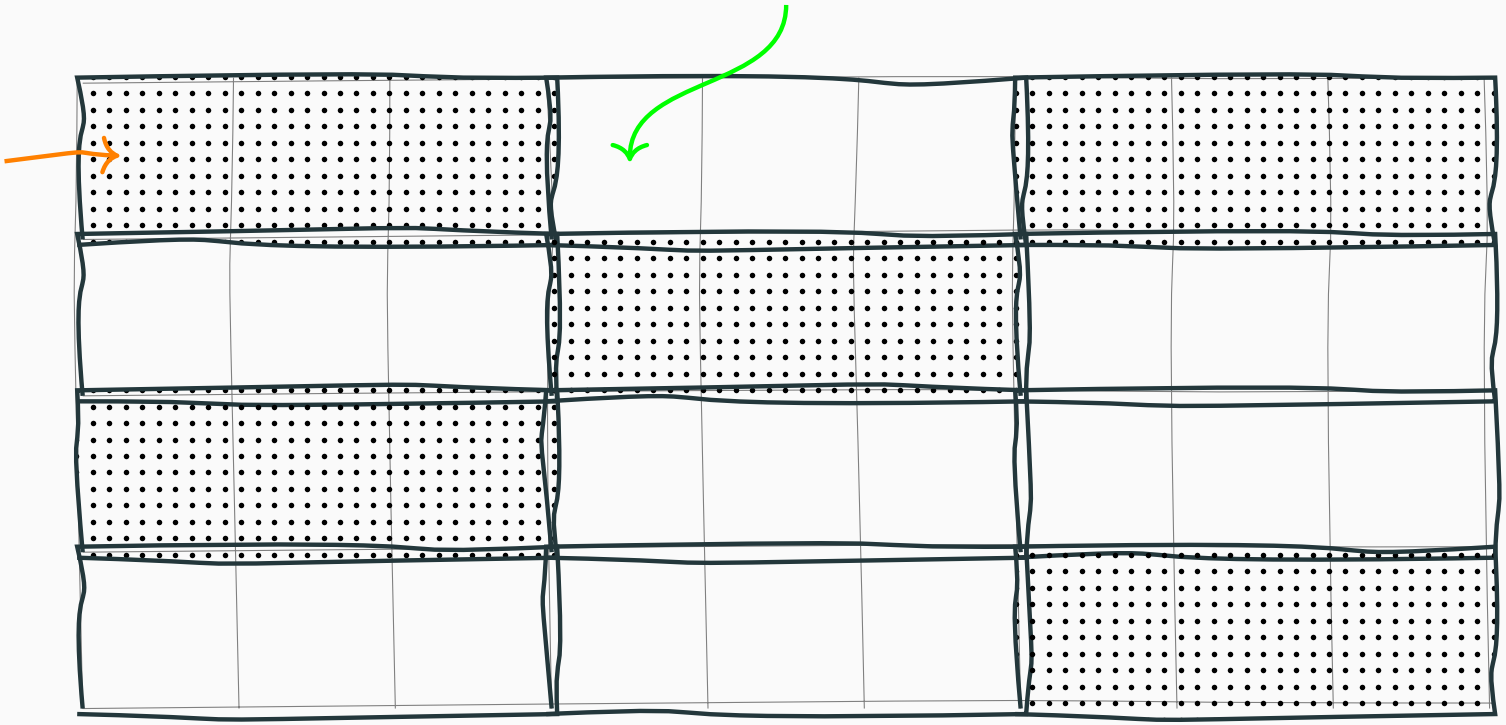
Linked-List implementation

Keep all free blocks in a linked list.



Linked-List implementation

Keep two list heads in registers.



Linked-list implementation details

Very detailed step-by-step slides for what happens are in the appendix, but most people will be familiar with linked-list unlinks.

We will look at the machine implementation in more detail later.

Right now, we need to define what *security properties* are.

Security properties of the IFSM

The security properties of the IFSM are "what we want to be true" for the IFSM.

This is needed to define "winning" for an attacker: He wins when he defeats the security properties of the IFSM.

The security property is just a statement about input and output sequences of the IFSM.

Security properties of the IFSM

A secret-keeping machine should not allow the secrets to be retrieved by someone who does not know the password.

"If the system stores a pair (p, s) it should be practically infeasible for the attacker to obtain s if he does not know p".

Intended states, CPU states, and weird states

Intended states

There is one IFSM state for each possible configuration of *Memory*.

At the same time, we saw that there can be multiple states of the CPU that represent the same IFSM state (sets do not care about ordering).

You can map between intended states and sets of CPU states: Every state of the IFSM can be represented by one or more CPU states.

But many CPU states do not represent any state of the IFSM.

CPU states

CPU states can fall into one of three categories:

Sane states: A CPU state which corresponds to a state of the IFSM.

Transitory states: A CPU state which inevitably irrespective of further input or output leads to a sane state. These occur when the emulator emulates a transition arrow of the IFSM using multiple instructions.

Weird states: A CPU state that does not correspond to a state of the IFSM, and hence cannot be interpreted as a representation of the IFSM.

Weird states

How can the CPU enter a weird state?

Programming mistakes

Bitflips in DRAM

Broken data on your harddisk that gets paged in

... and a myriad of other ways ...

Emulated transitions on weird states

The program that executes on the CPU usually cannot know it has entered a weird state.

It will happily continue executing until an exception occurrs.

This means that it will emulate transitions intended for a **sane** state on a **weird** state.

This transforms one weird state into (usually) another weird state.

Emulated transitions on weird states

Once a weird state is entered, many other weird states can be reached by applying the transitions intended for sane states on them.

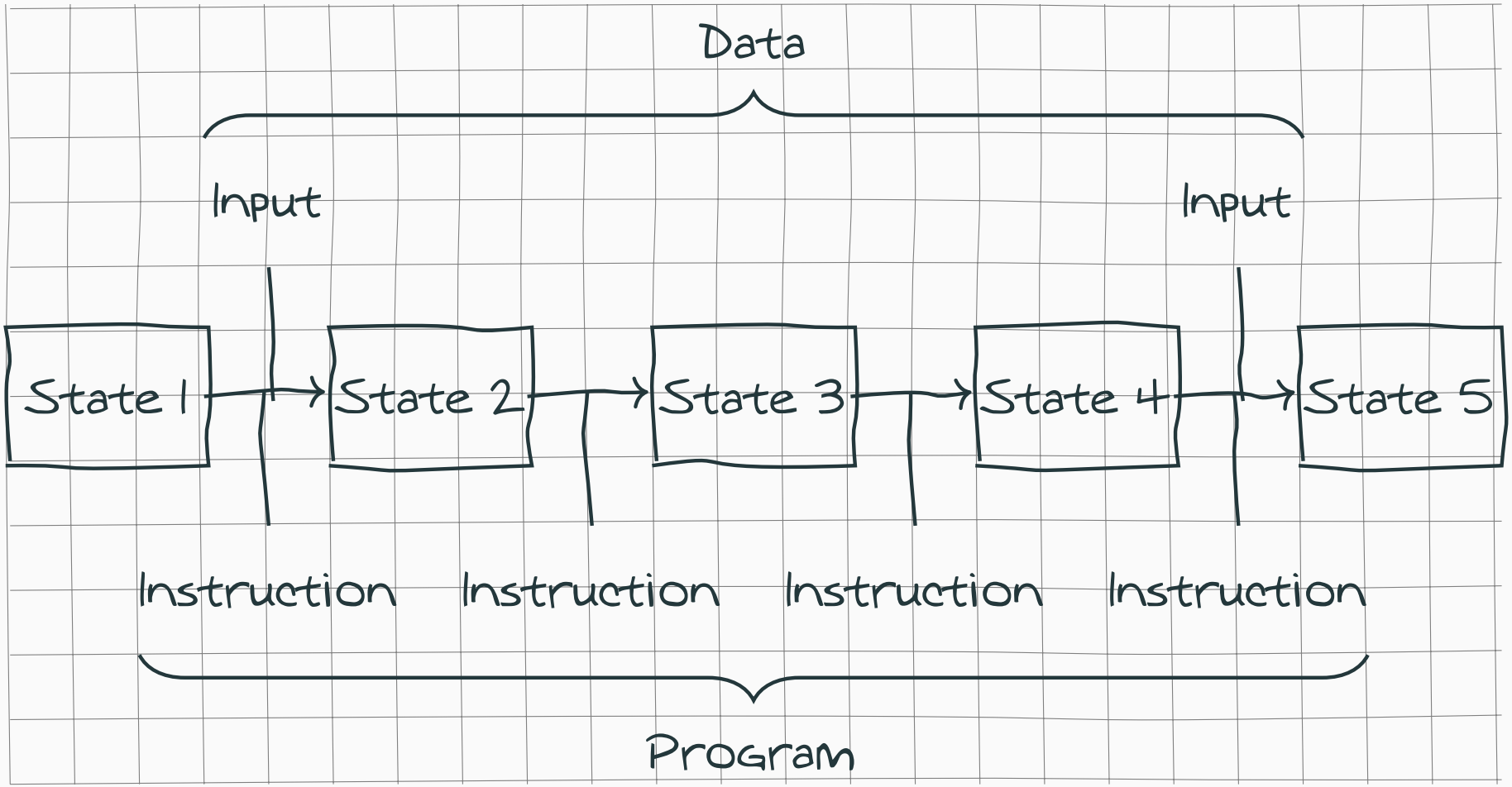
A new computational device emerges, the *weird machine*.

The *weird machine* is the computing device that arises from the operation of the emulated transitions of the IFSM on weird states.

...time for a radical change of perspective...

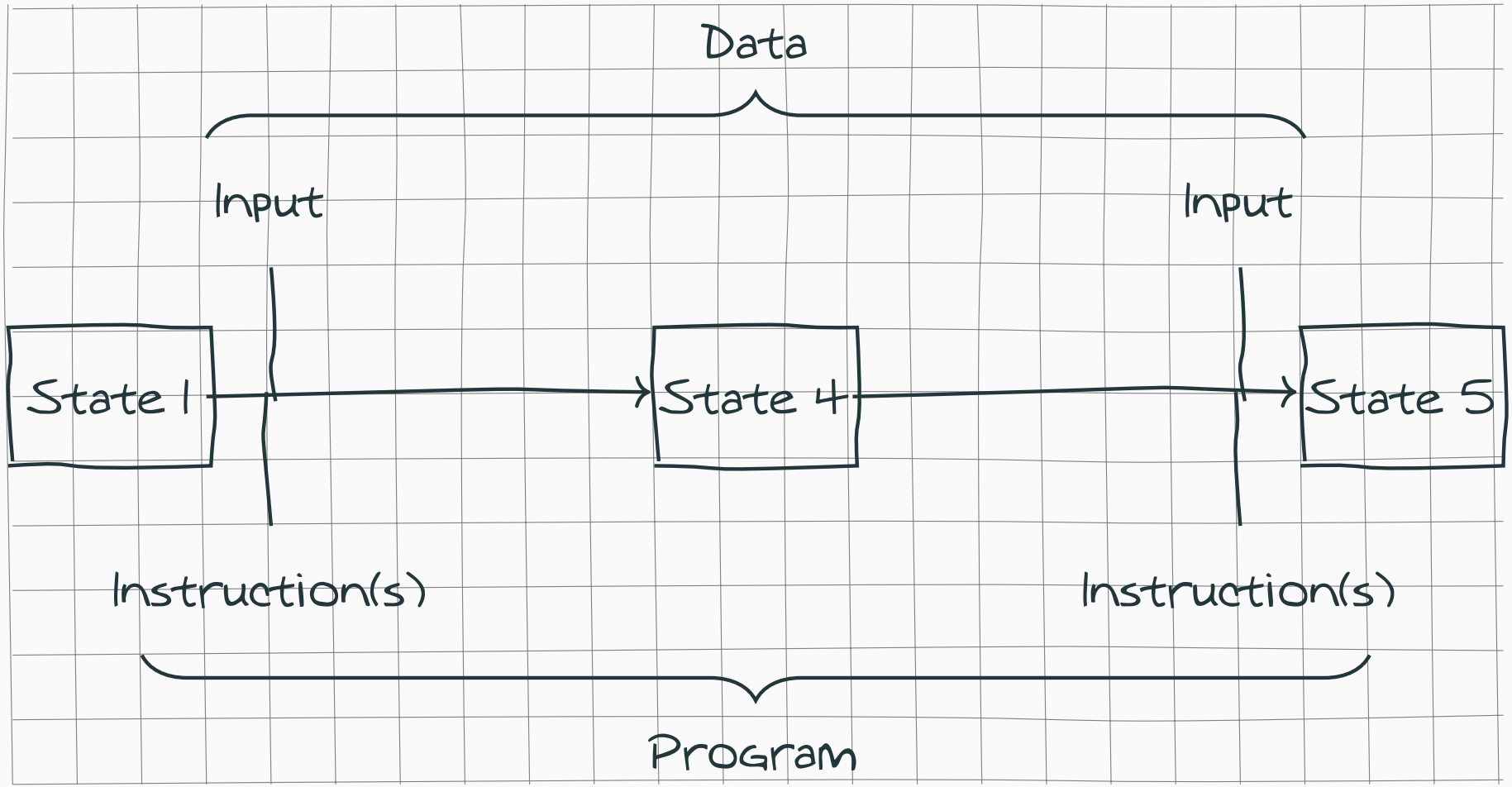
Input streams as instruction streams

Normal model of computation

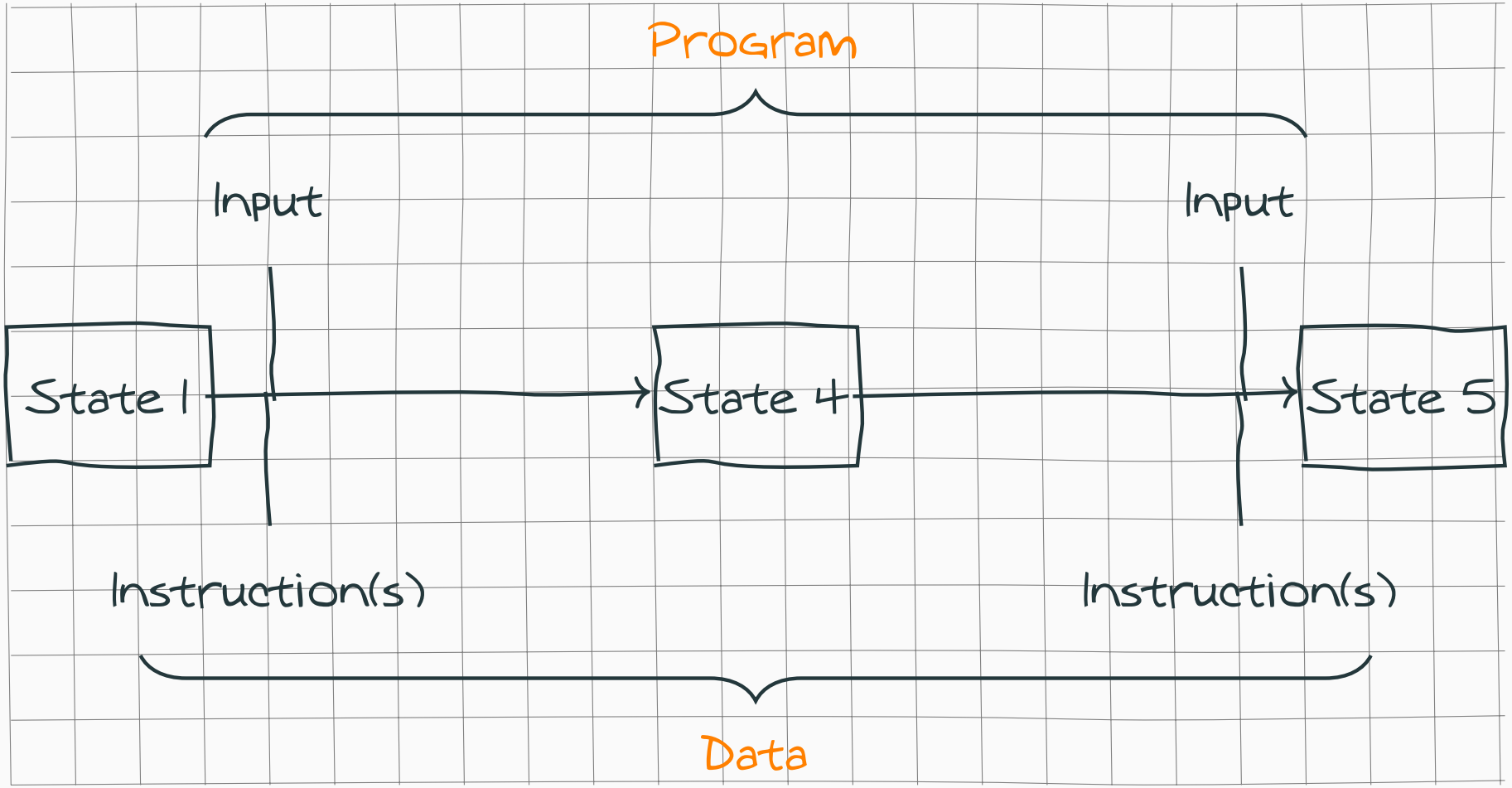


Normal model of computation

Summarize the middle transitions.



Attacker model of computation



Switching perspective

Any computing device that accepts input and reacts to this input by executing a different program path can be viewed as a computing device where the *inputs* are the program.

If no weird state can ever be entered, this is no problem, and intended.

If there is a way to enter a weird state, all hell breaks loose: The attacker is now programming your computer.

The essence of exploitation

Given a method to enter a weird state from a set of particular sane states, *exploitation* is the process of

setup (choosing the right sane state),

instantiation (entering the weird state) and

programming of the weird machine

so that security properties of the IFSM are violated.

Provably exploitable and provably unexploitable code

Attacker model

Cryptography has a variety of attacker models of different strengths.

Plaintext-only. Chosen-ciphertext. Chosen-plaintext. Adaptive chosen-plaintext.

These attacker models define what an attacker is allowed to do and generalize many real-world attacks.

They are usually quite powerful, and allow reasoning about large classes of attacks and defenses.

Attacker model

Let us introduce a simple (yet powerful) attacker model for memory corruptions:

A **chosen-bitflip-once** attacker is allowed to corrupt exactly one bit of memory exactly once, when the machine is waiting for input.

There are many other attacker models that are sensible and imaginable - but we won't discuss them here.

Attacker model

Our attacker model works as follows:

1. Attacker gets to send inputs to the machine
2. Defender gets to send an input (p, s) to the machine
3. Attacker gets to send inputs to the machine
4. Attacker gets to flip exactly one bit of his choosing
5. Attacker gets to send more inputs to the machine to obtain s

The chosen-bitflip-once attacker is a fairly strong attacker model, but not unreasonably so.

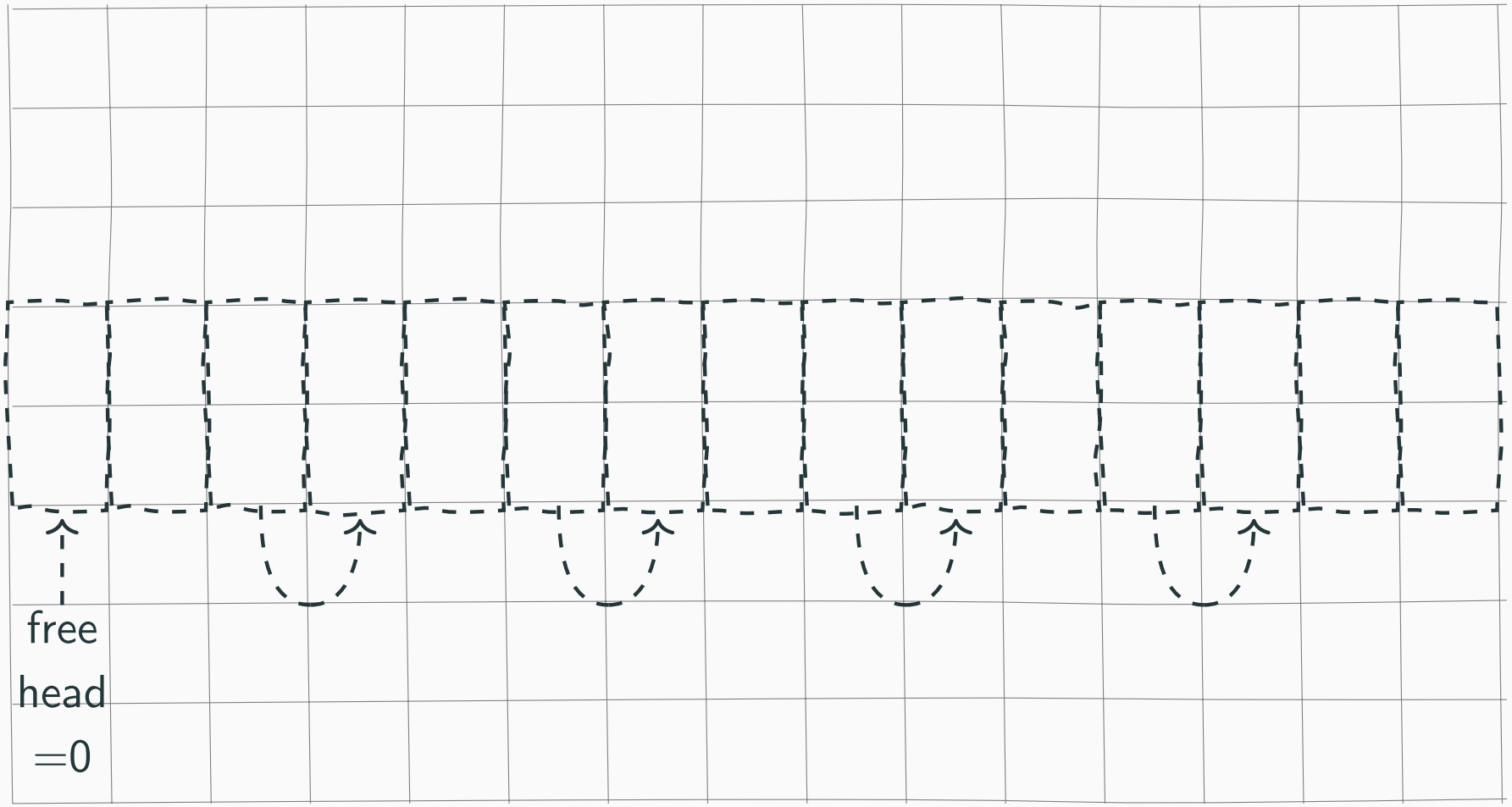
Proving exploitability of the linked-list variant

The time-honored way of showing exploitability is providing an exploit.

We follow the same route: We describe a sequence of operations that allows the attacker to obtain the secret.

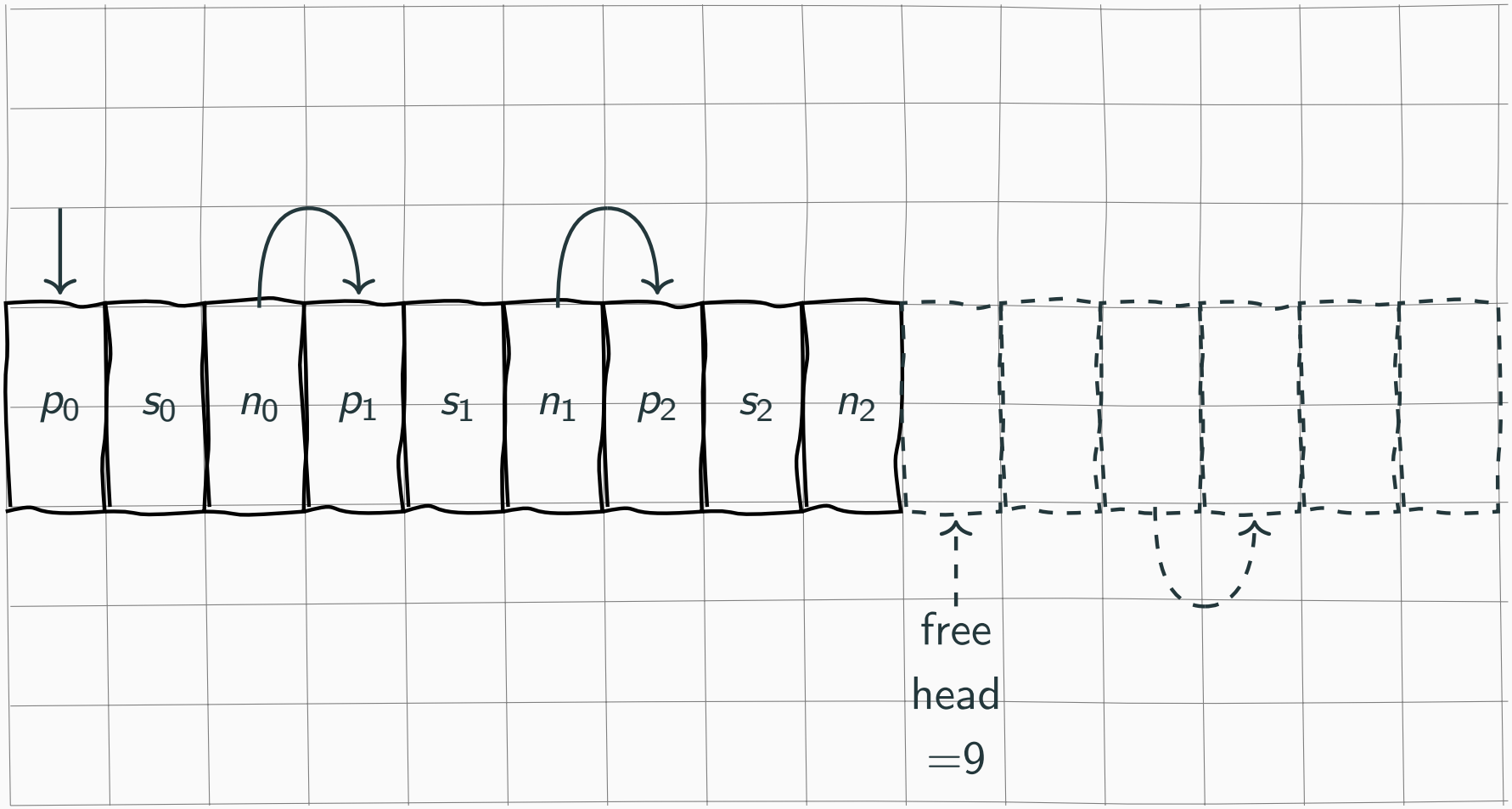
Exploiting the linked-list implementation

Initial state is a list of empty 3-tuples.



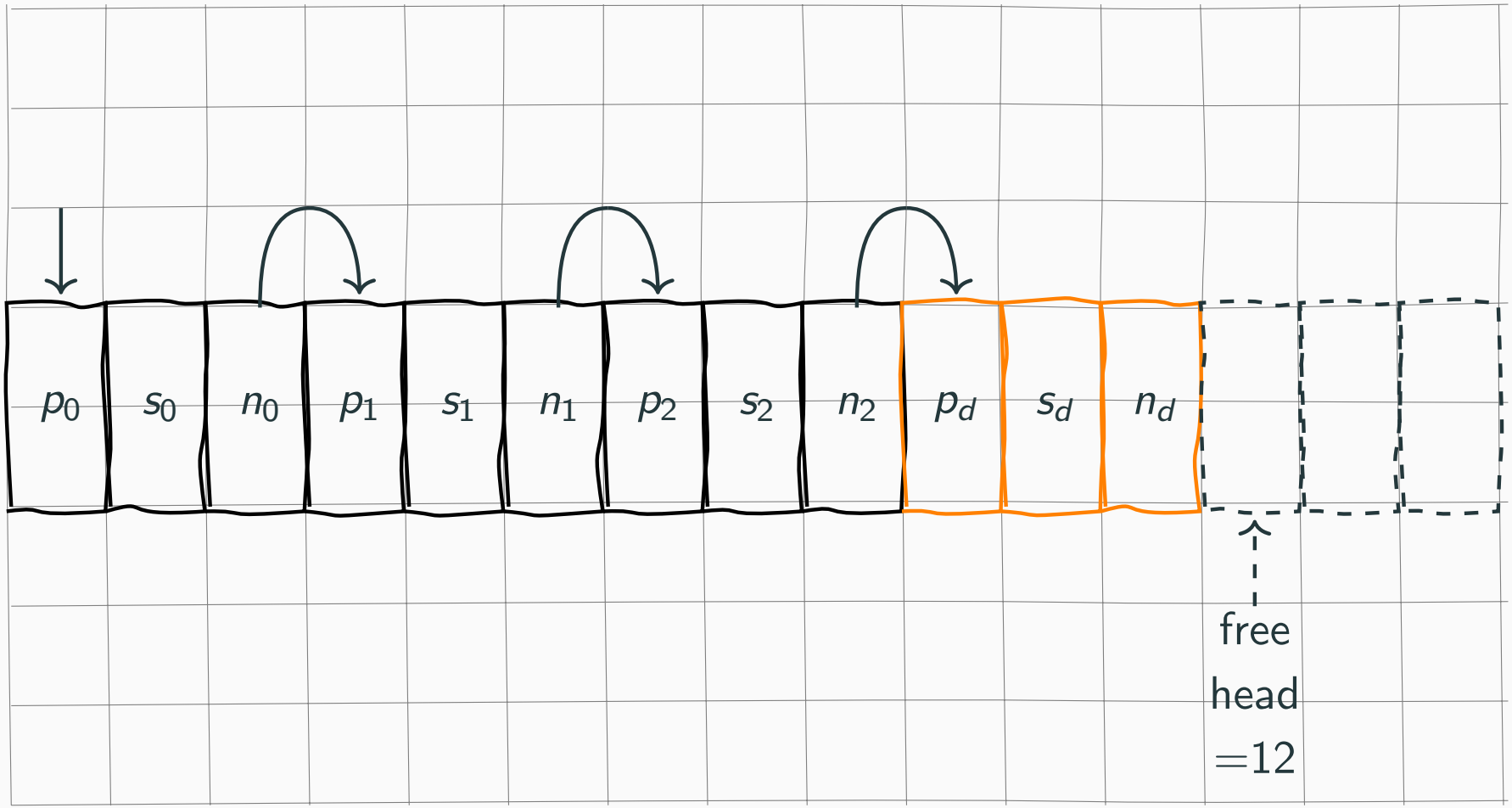
Exploiting the linked-list implementation

Attacker sends $(p_0, s_0), (p_1, s_1), (p_2, s_2)$ sequence.



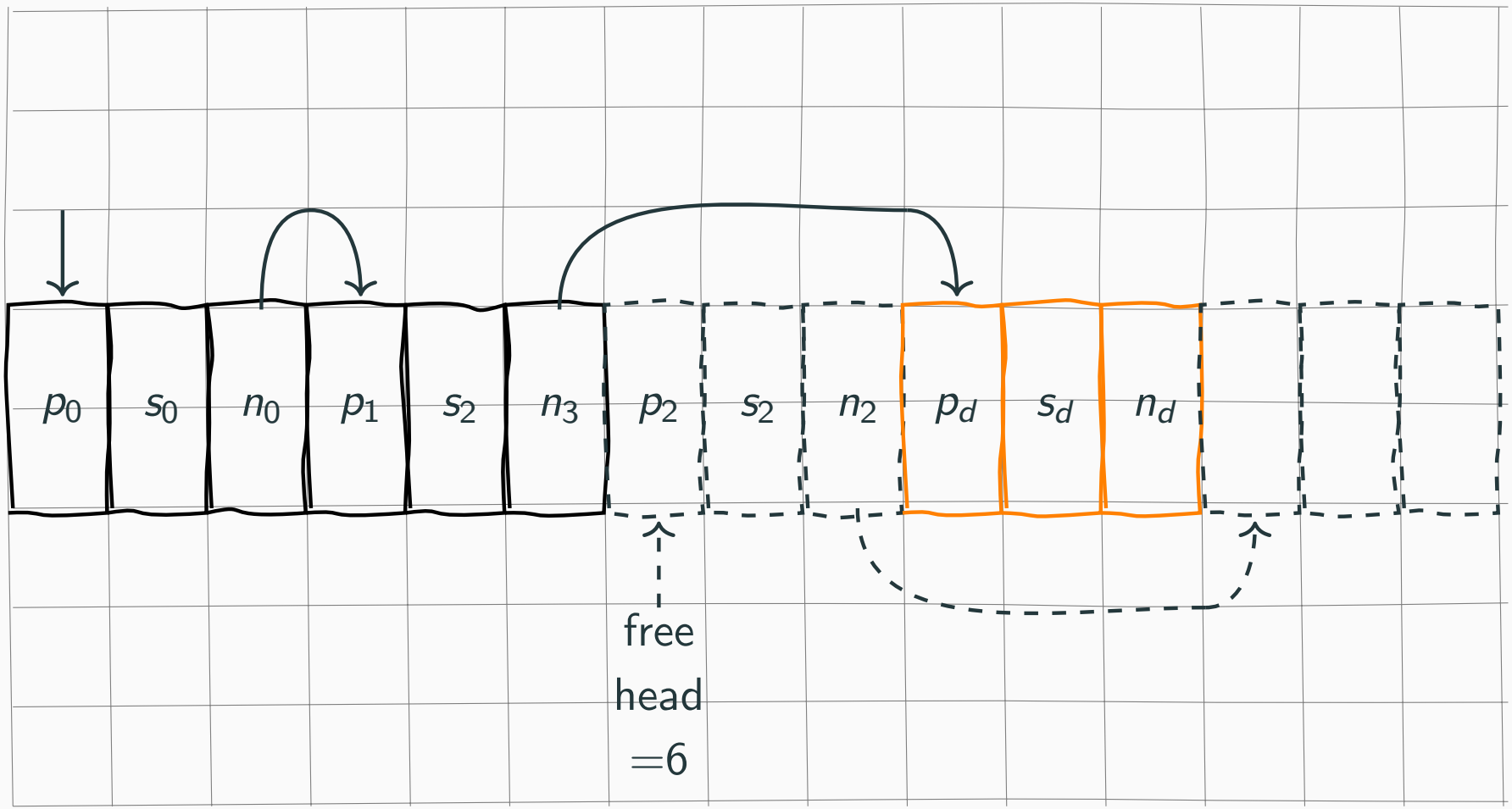
Exploiting the linked-list implementation

Attacker sends $(p_0, s_0), (p_1, s_1), (p_2, s_2)$ sequence. Defender sends (p_d, s_d).



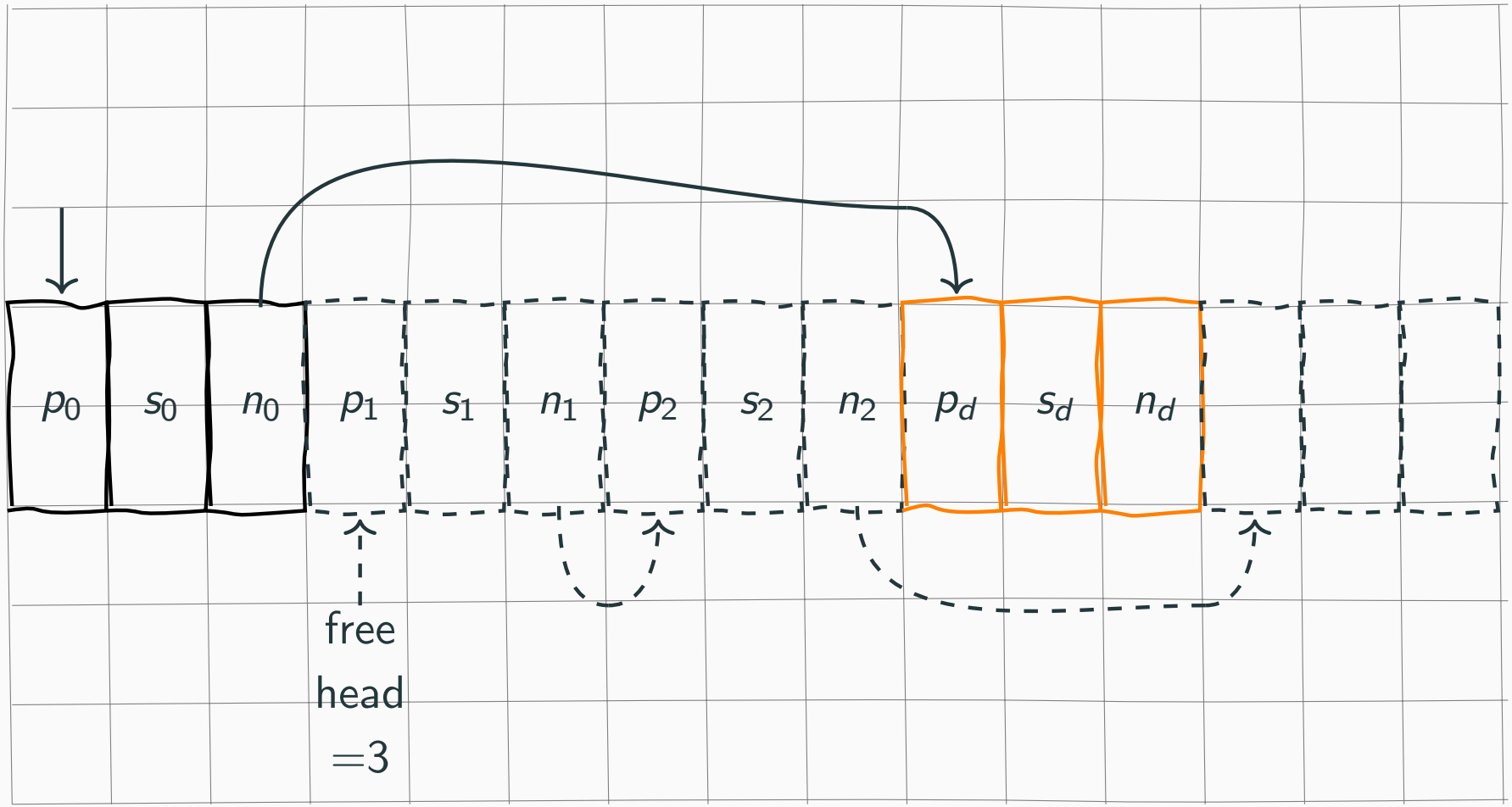
Exploiting the linked-list implementation

Attacker sends $(p_0, s_0), (p_1, s_1), (p_2, s_2)$ sequence. Defender sends (p_d, s_d). Attacker sends (p_2, X).



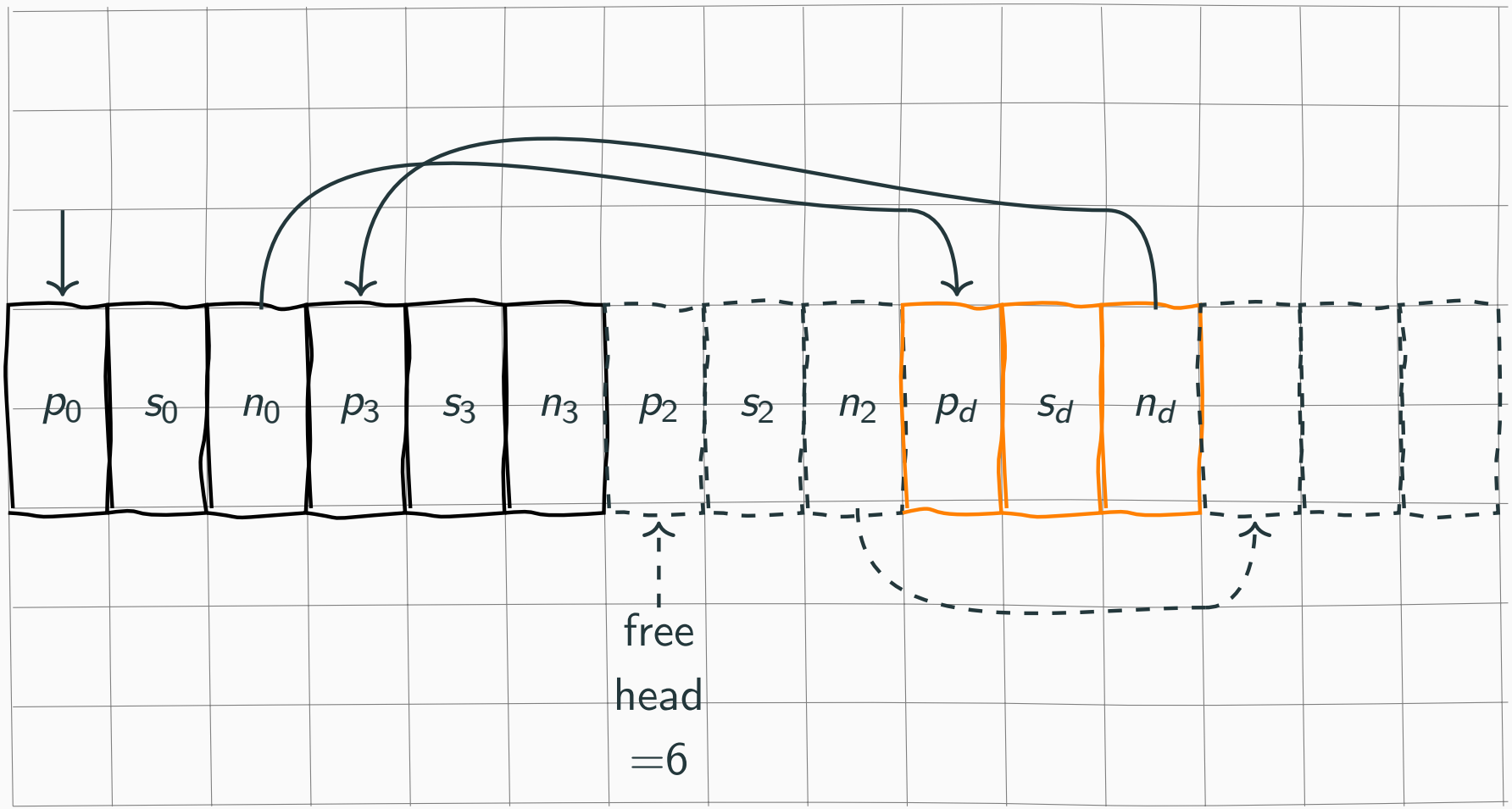
Exploiting the linked-list implementation

Attacker sends $(p_0, s_0), (p_1, s_1), (p_2, s_2)$ sequence. Defender sends (p_d, s_d). Attacker sends $(p_2, X), (p_1, X)$.



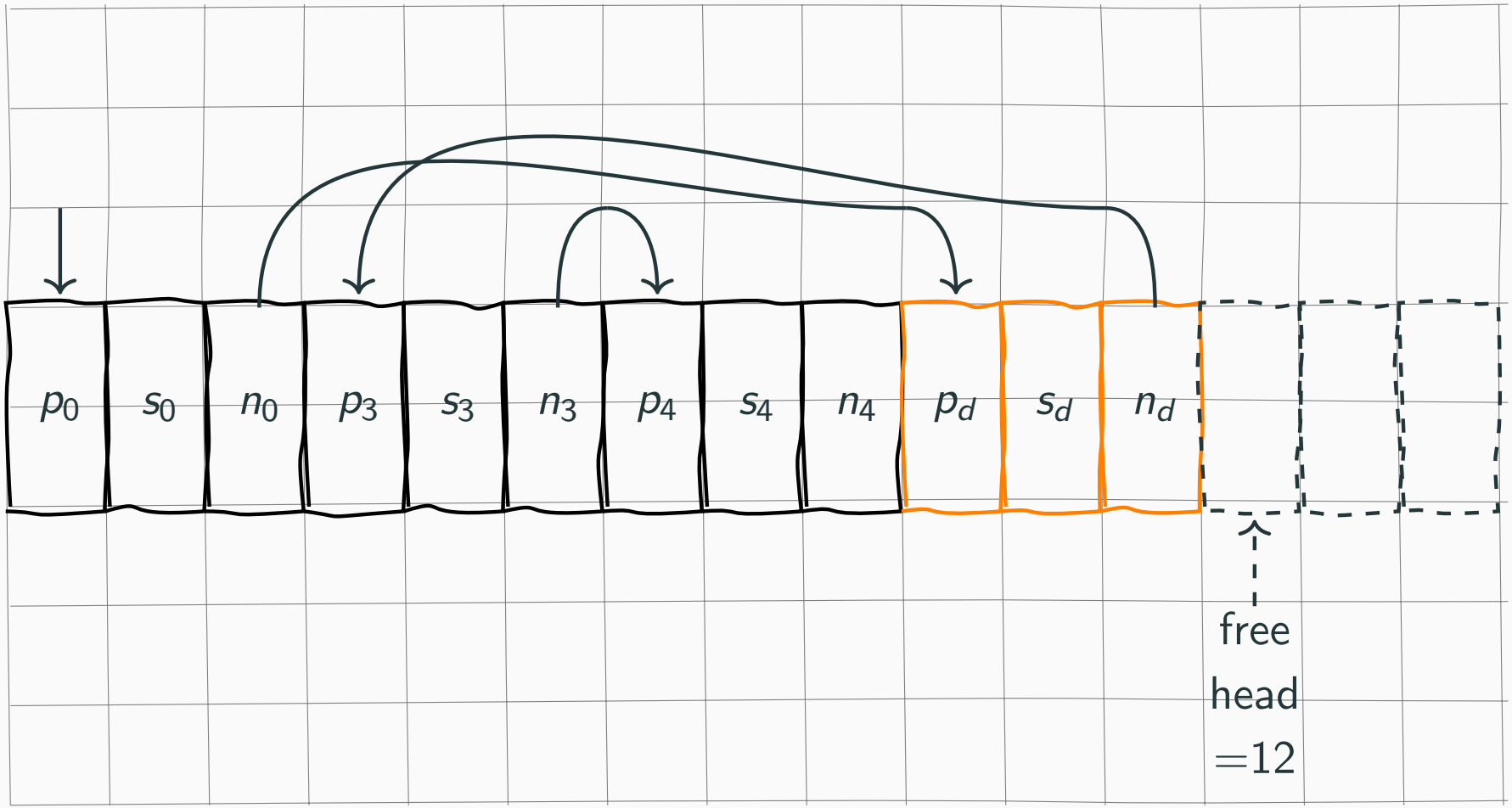
Exploiting the linked-list implementation

Attacker sends $(p_0, s_0), (p_1, s_1), (p_2, s_2)$ sequence. Defender sends (p_d, s_d). Attacker sends $(p_2, X), (p_1, X), (p_3, s_3)$.



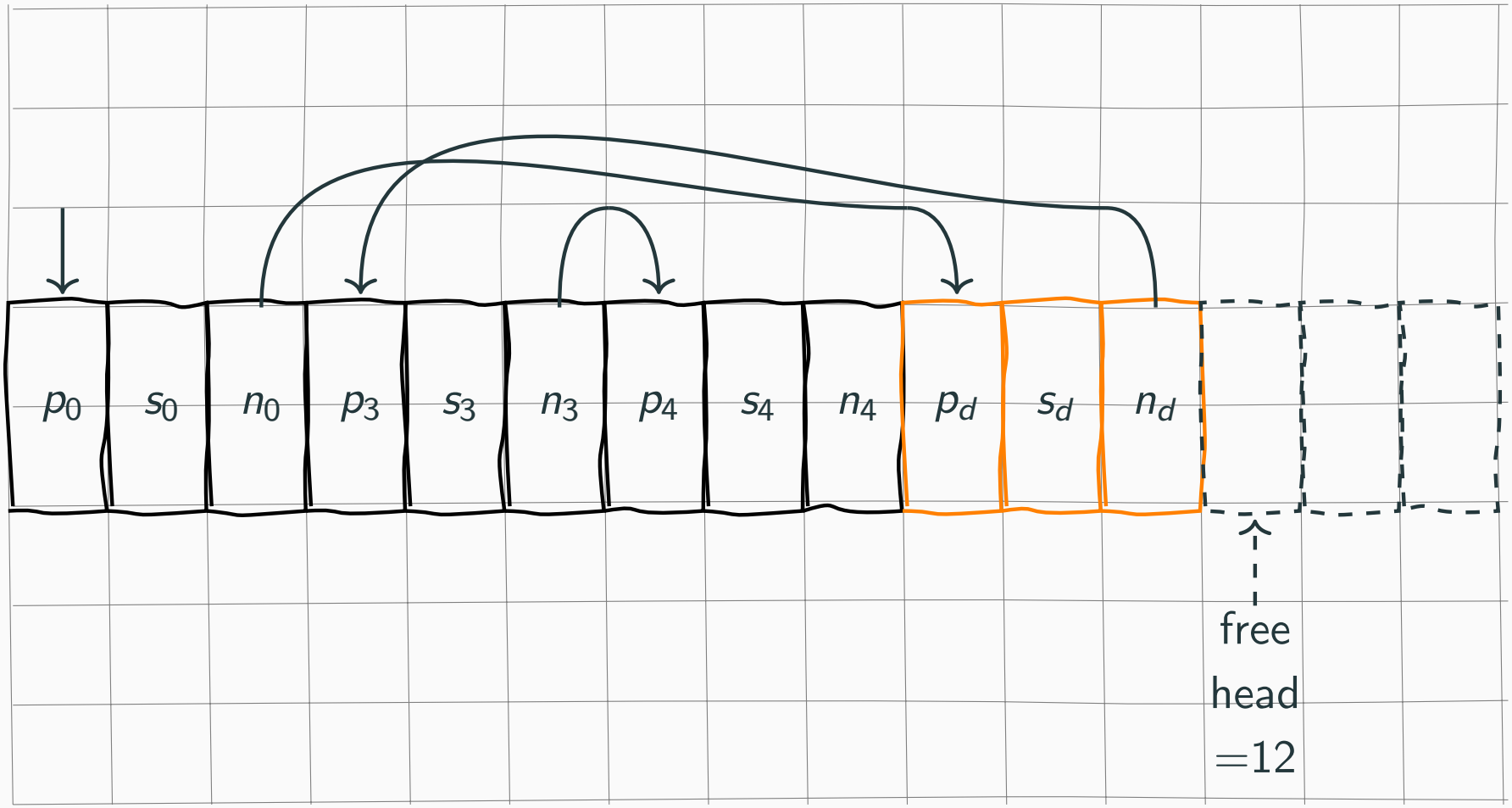
Exploiting the linked-list implementation

Attacker sends $(p_0, s_0), (p_1, s_1), (p_2, s_2)$ sequence. Defender sends (p_d, s_d). Attacker sends $(p_2, X), (p_1, X), (p_3, s_3), (p_4, s_4)$.



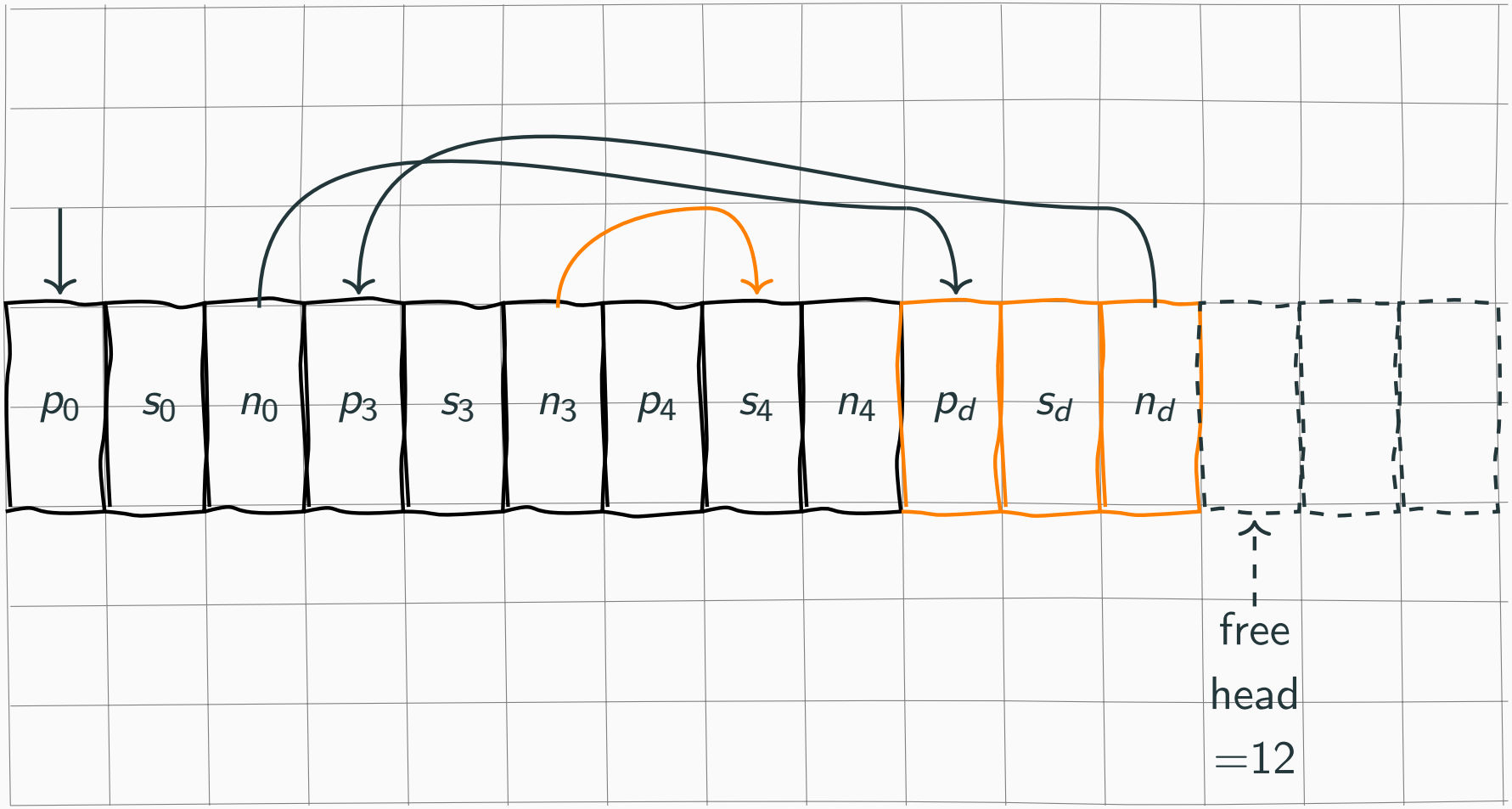
Exploiting the linked-list implementation

The machine is still in an entirely sane state.



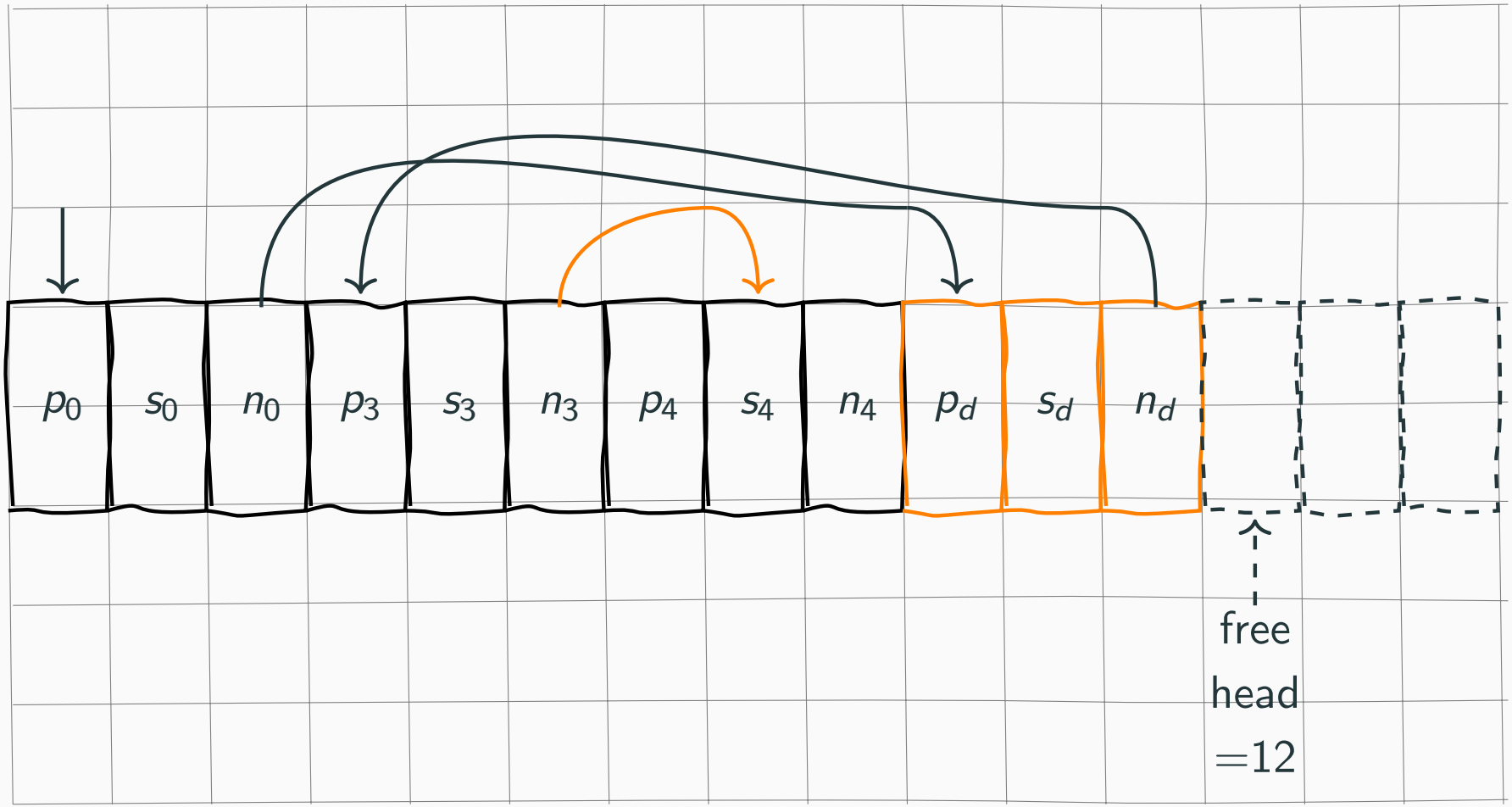
Exploiting the linked-list implementation

The attacker gets to corrupt one bit, incrementing n_3.



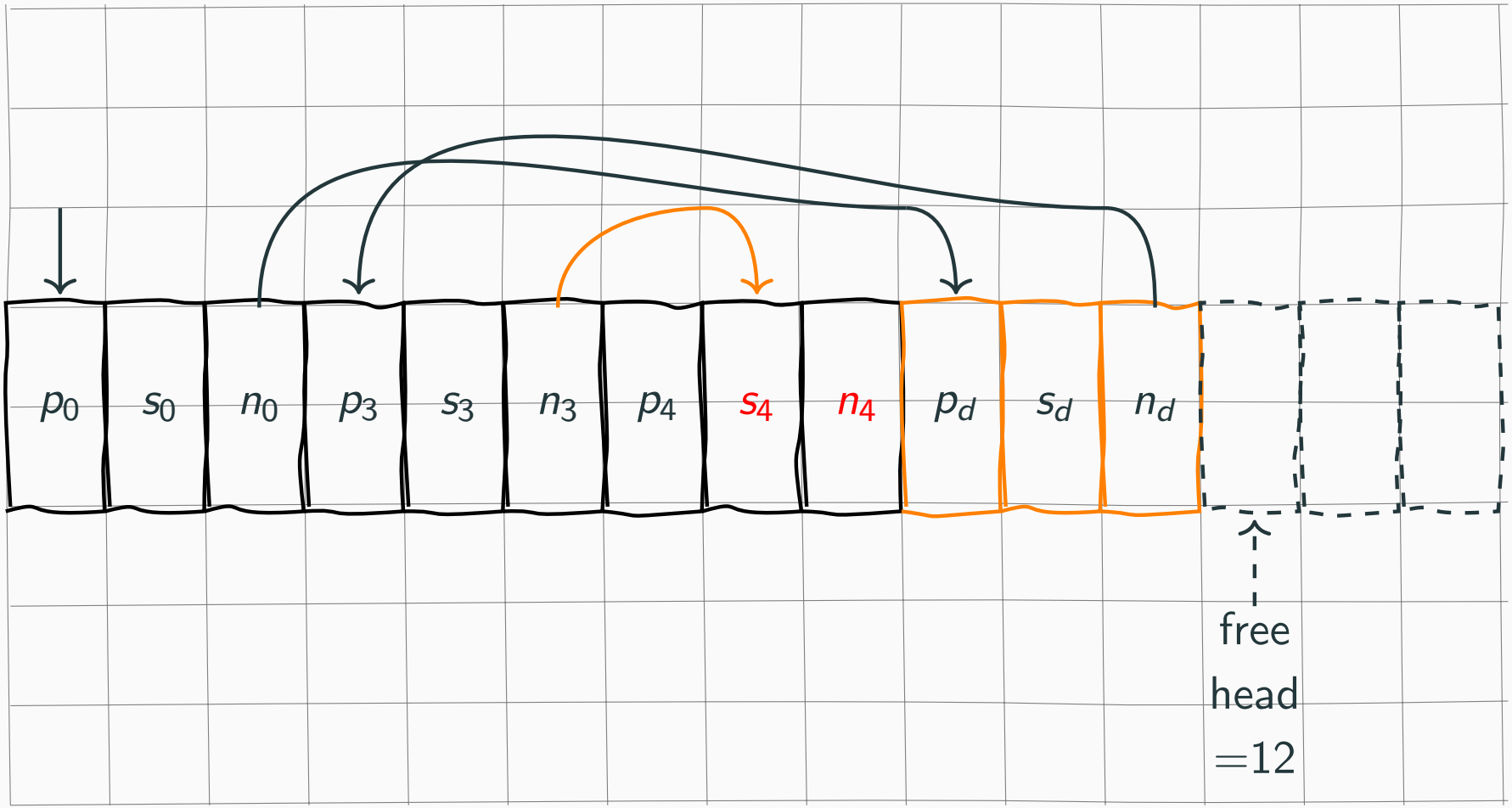
Exploiting the linked-list implementation

The attacker gets to corrupt one bit, incrementing n_3. He then sends in (s_4, X).



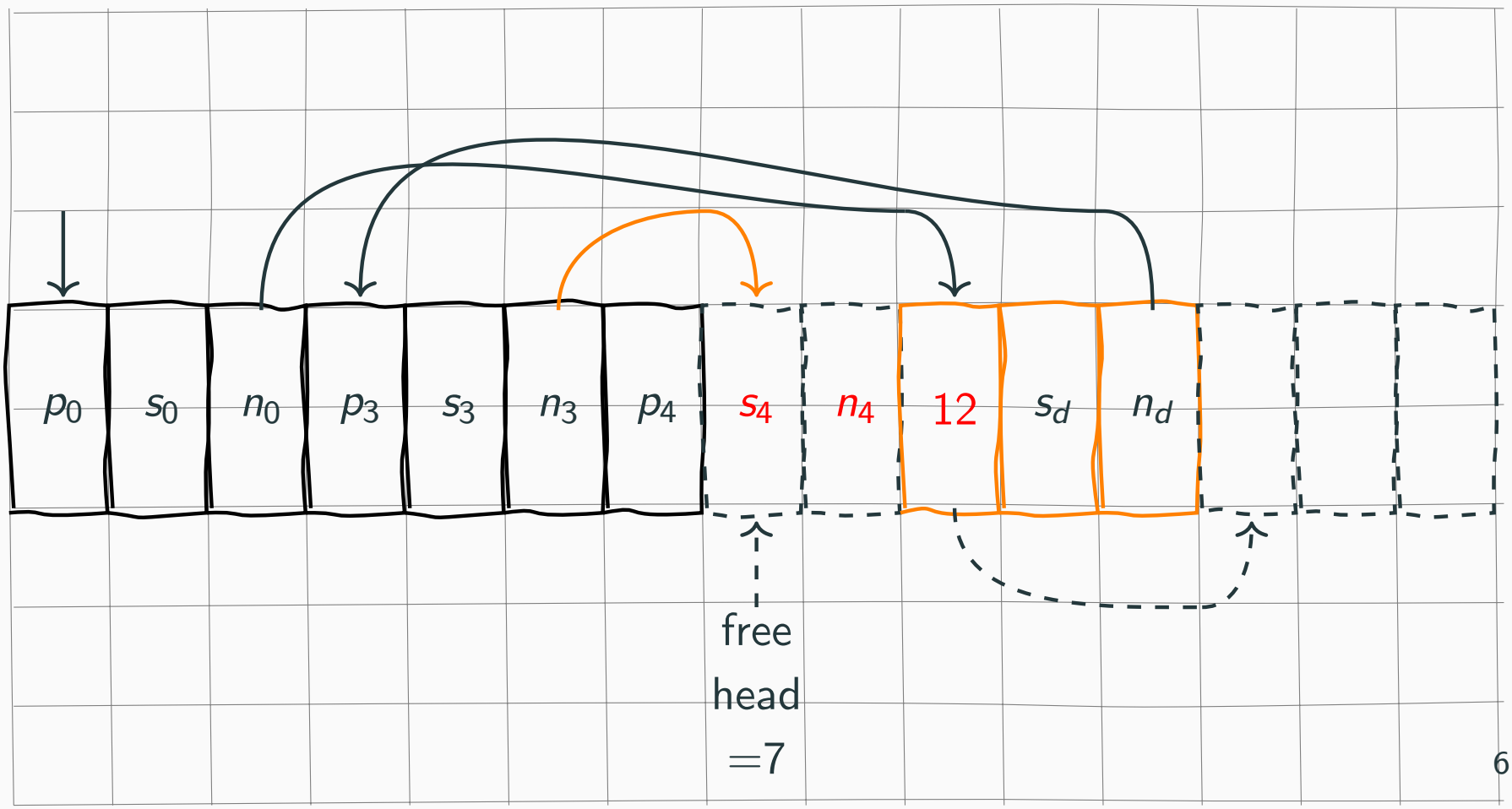
Exploiting the linked-list implementation

The machine follows the linked list, interprets the stored s_4 as password, and outputs n_4.



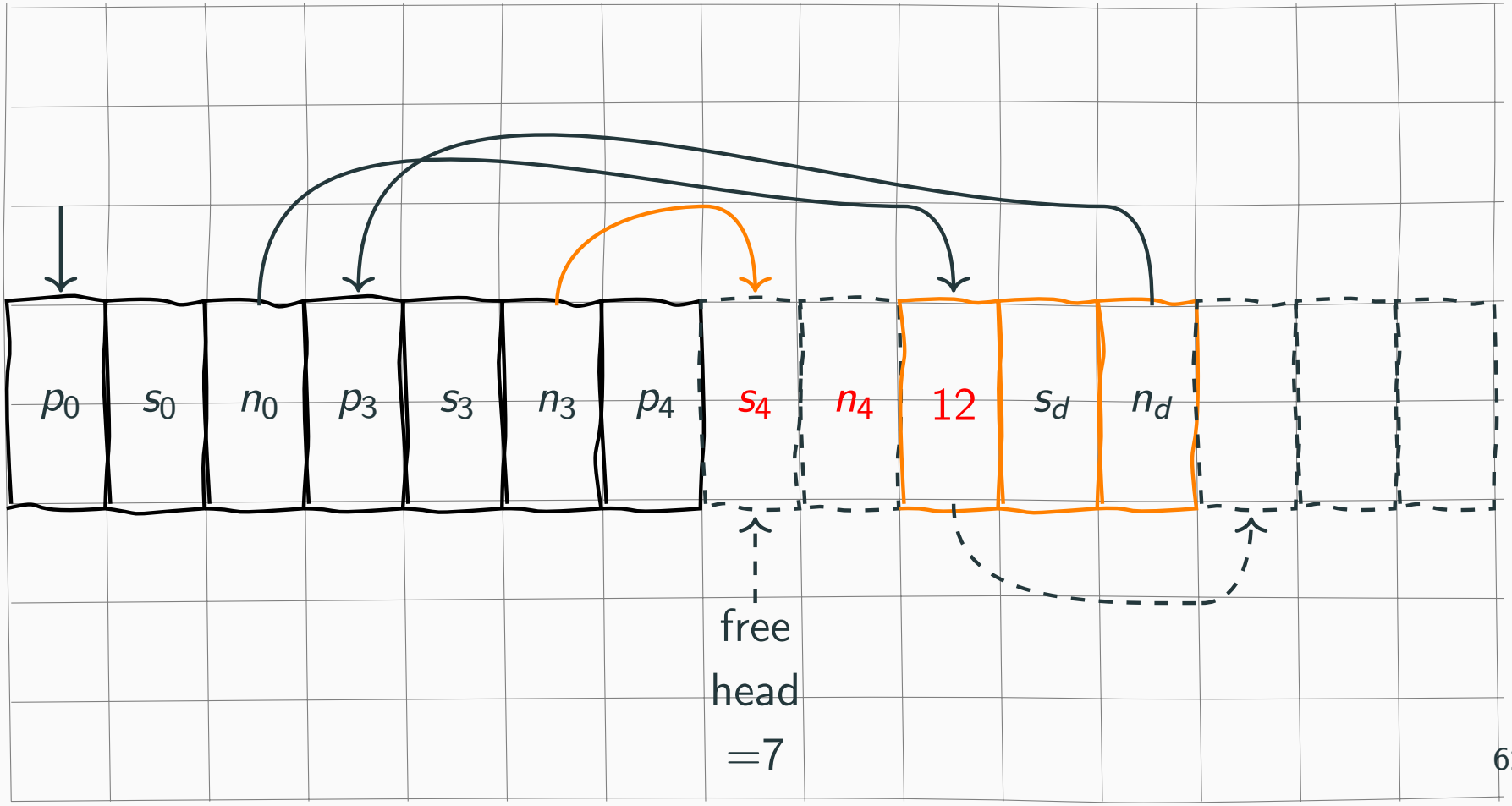
Exploiting the linked-list implementation

The machine follows the linked list, interprets the stored s_4 as password, and outputs n_4. It then sets the three cells to be free and overwrites the stored p_d with the free-head.



Exploiting the linked-list implementation

The machine follows the linked list, interprets the stored s_4 as password, and outputs n_4. It then sets the three cells to be free and overwrites the stored p_d with the free-head. Attacker sends $(12, X)$ and obtains s_d.



Proving non-exploitability of the flat-array variant

The proof-of-exploitability was constructive.

Proving non-exploitability has a different flavor.

It is simply enumerating all initially reachable weird states and showing they lead nowhere.

Proving non-exploitability of the flat-array variant

By flipping a bit, an attacker can enter one of the following states:

1. A sane state where he modifies a previously-saved p by one bit.
2. A sane state where he modifies a previously-saved s by one bit.
3. A weird state where a password-field of a $(0, 0)$-pair is set to $\neq 0$ but the secret field of the pair is 0.
4. A weird state where a secret-field of a $(0, 0)$-pair is set to $\neq 0$.

The sane states do not give him any advantage in obtaining s_d. What about the weird states?

Proving non-exploitability of the flat-array variant

Examine the initial weird states:

3. A weird state where a password-field of a $(0, 0)$-pair is set to $\neq 0$ but the secret field of the pair is 0.
   - 3.1 The machine can be made to spit out 0 given the right password, then reverts to a sane state.
   - 3.2 This is weird behavior, but does not violate security properties
4. A weird state where a secret-field of a $(0, 0)$-pair is set to $\neq 0$.
   - 4.1 The machine will treat the pair as empty and overwrite it, reverting to a sane state.
   - 4.2 This is weird behavior, but does not violate security properties

The attacker is out of options.

What does all of this mean?

Interpreting all of this (1 of N)

Exploitation has very little to do with hijacking RIP/EIP.

All programs we considered had perfect control-flow integrity, and no instruction pointer was hijacked.

Exploitation is programming a weird machine.

Properties of weird machines

Weird machines get more powerful as the emulator for the IFSM gets more complex.

It seems that the presence of a linked list in the same address space where a bit gets flipped may already imply "anything can happen" if the attacker can cause unlink/link operations.

Weird machines are created out of the IFSM emulator. They are not designed to be programmed, and the attacker writes "weird machine assembler".

Properties of weird machines

Since the weird machine arises from its "host", different versions of the same "host" may create similar weird machines.

This leads to attacker specialization: "X is the expert for IE exploitation."

Attackers also end up writing libraries for recurrent exploitation of the same targets. Nobody wants to write assembler from scratch, especially not weird machine assembler.

Interpreting all of this (2 of N)

Many mitigations break one particular weird machine program.

Many mitigations can be "programmed around" by the attacker.

Many mitigations can be "bypassed once" (for a given target), then the attacker has a "library" to repeat the feat when he finds the next bug.

Interpreting all of this (3 of N)

It is actually possible to write non-exploitable programs.

Exploring the space of non-exploitable programs seems worthwhile.

What datastructures can be implemented in non-exploitable manner?

At what computational cost?

Interpreting all of this (4 of N)

"Provably correct" is not the same as non-exploitable. These are orthogonal concepts.

"Provably correct" means there is no software-only path to induce a weird state.

Non-exploitable means all weird states a given attacker can reach resolve to harmless sane states.

Interpreting all of this (5 of N)

"Turing-complete" is a fashionable way of showing the power of an offensive technique in some academic papers.

It is a misuse of the term. It is trivial to produce a IFSM that simulates a two-counter machine that is "Turing-complete", but no security properties are violated.

The right way of showing power is demonstrating violation of security properties.

If you want to show more, demonstrate violation of **all possible** security properties. So you want to show you can reach any pathological state. (Turing-complete computation on arbitrary memory can do this).

Interpreting all of this (6 of N)

When trying to write non-exploitable programs, it is very hard to keep "indirection" under control.

If one memory cell refers to another (directly or indirectly), it is often not easy to assure it remains sane.

Theoretical computer science distinguishes between RAM machines with indirect addressing and without. Is something similar at play here?

Summary

Summary

Exploitation is programming emergent weird machines.

It does not require EIP/RIP, and is not a bag of tricks.

Theory of exploitation is still in embryonic stage.

A lot of interesting questions - where will we stand in 10 years?

Questions?

Acknowledgements

A lot of people have discussed the topic with me and thus influenced my views. In random order: Felix Lindner, Ralf-Philipp Weinmann, Willem Pinckaers, Vincenzo Iozzo, Julien Vanegue, Sergey Bratus, William Whistler, Sean Heelan, Sebastian Krahmer.

A lot of thanks for their patience and insight.

I have spoken to so many people about exploitation in recent years. If you feel you should've been listed here, just assume you were :-)

I also have to thank Mara Tam for useful feedback and outside perspective.

References

Creating a proper full bibliography is difficult and out of scope for this conference talk. Some starting points:

[3] [2] [1]

References I

S. Bratus, J. Bangert, A. Gabrovsky, A. Shubina, M. E. Locasto, and D. Bilar.

'weird machine' patterns.

In C. Blackwell and H. Zhu, editors, *Cyberpatterns, Unifying Design Patterns with Security and Attack Patterns*, pages 157–171. Springer, 2014.

L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto.

Security applications of formal language theory.

IEEE Systems Journal, 7(3):489–500, 2013.

J. Vanegue.

The weird machines in proof-carrying code.

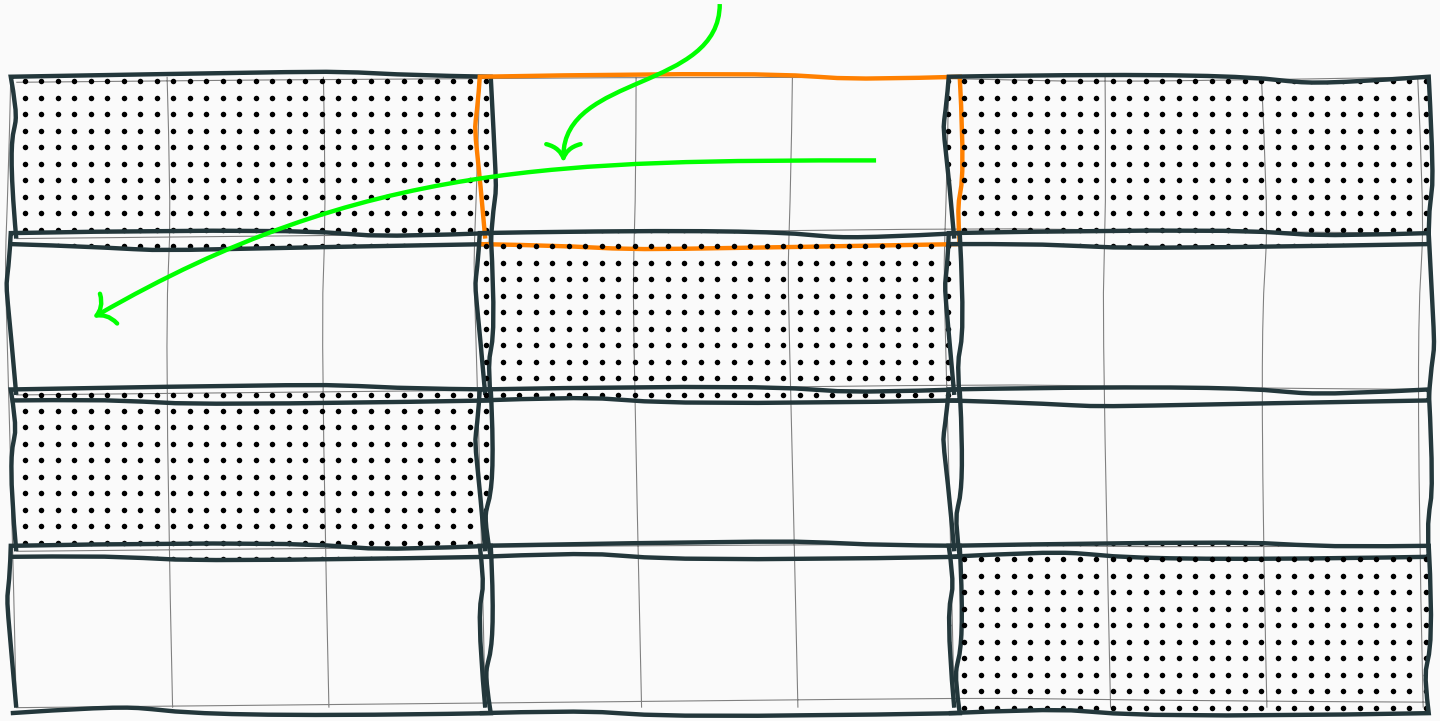
In *IEEE Symposium on Security and Privacy Workshops*, pages 209–213, 2014.

Appendix slides

Linked-list implementation

Transition (B): $\mathit{Memory} \leftarrow \mathit{Memory} \cup \{(p, s)\}$.

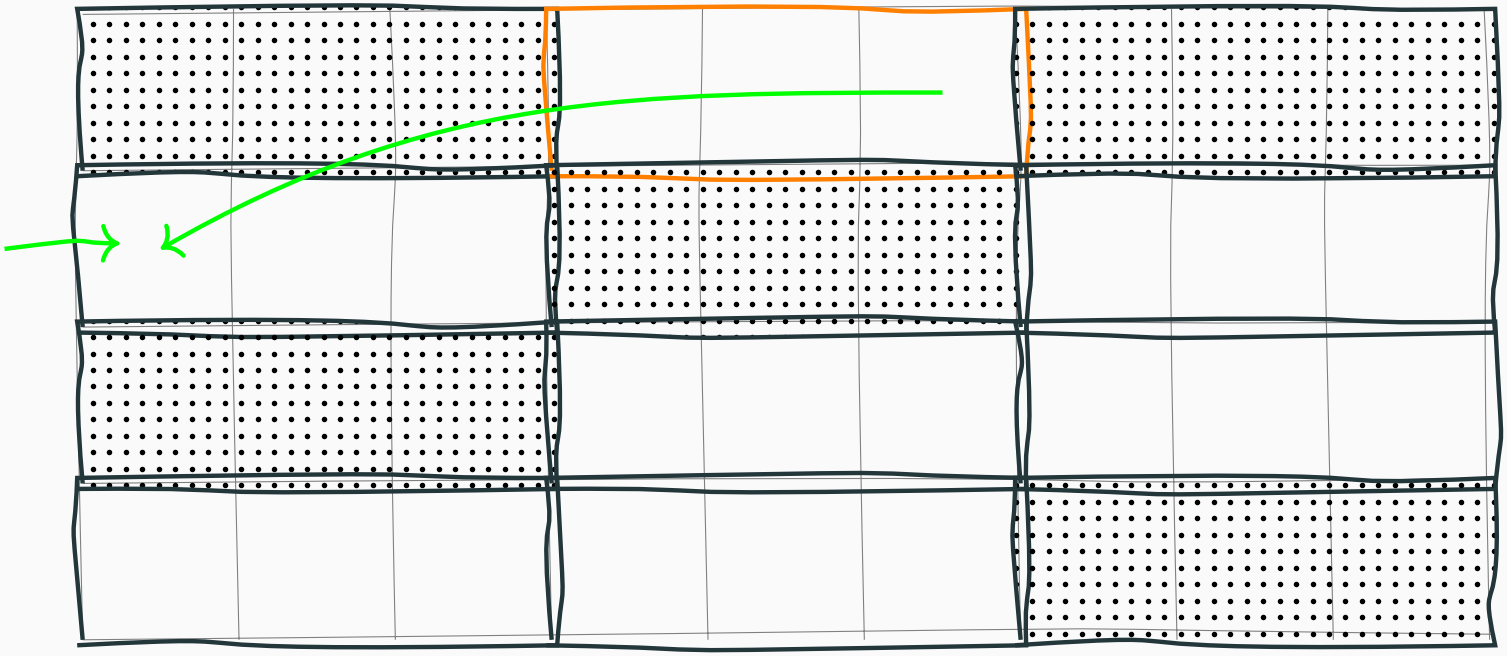
Take first element from free-list head.



Linked-list implementation: Writing

How do we store something in memory?

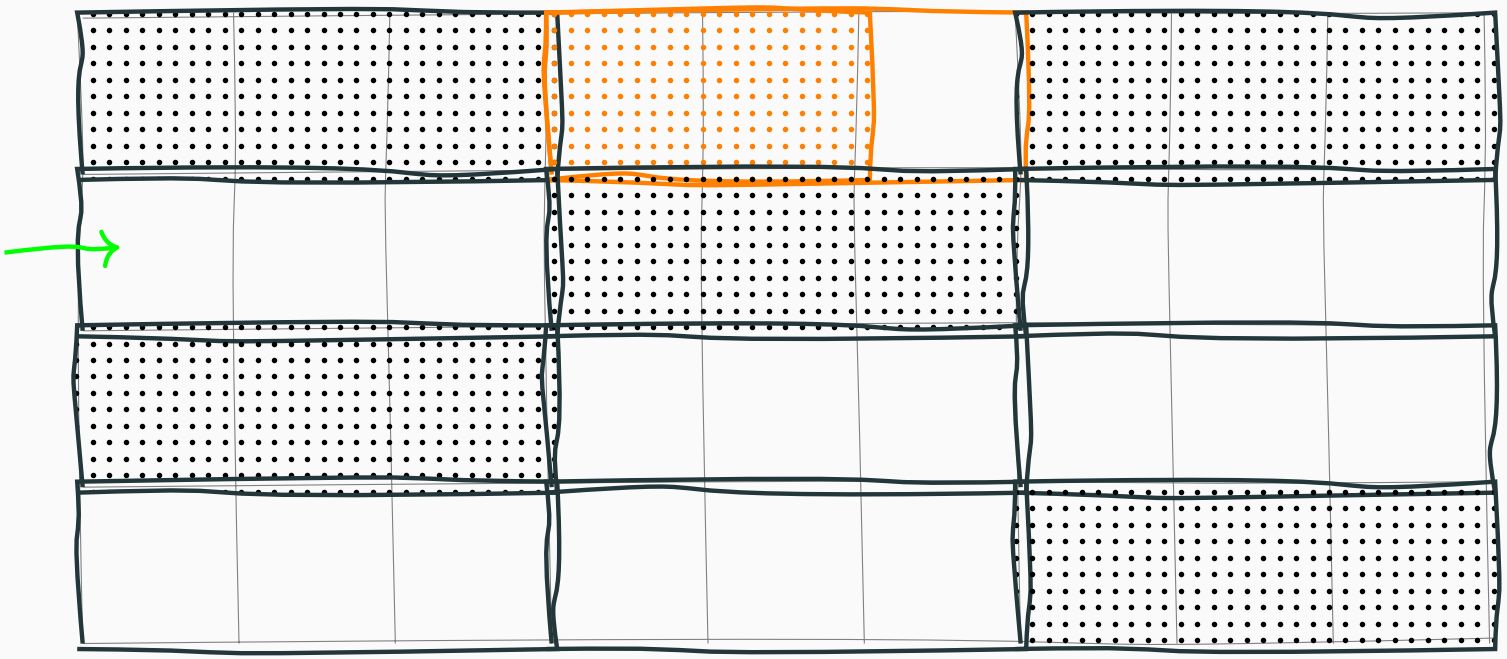
Take first element from free head. Adjust free-list head.



Linked-list implementation

How do we store something in memory?

Take first element from free head. Adjust free-list head. Fill in p, s.

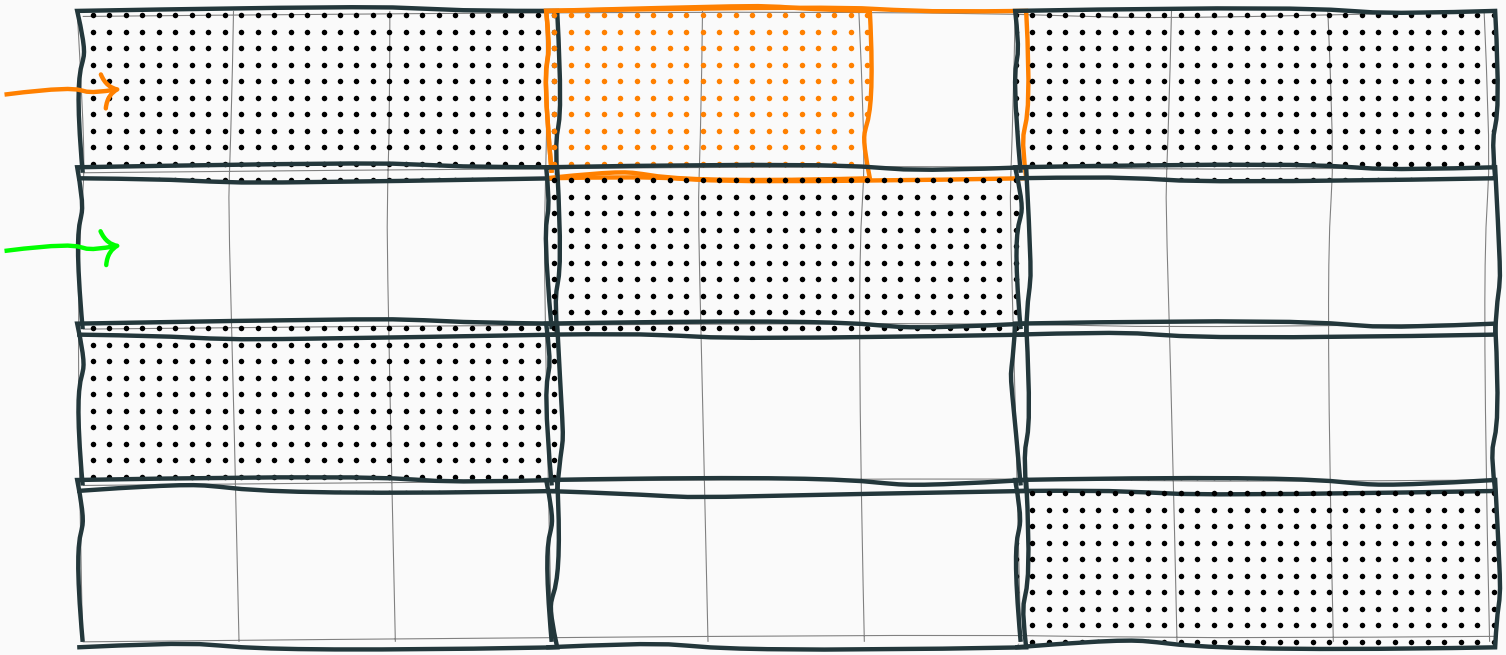


Linked-list implementation

How do we store something in memory?

Take first element from free-list head. Adjust free-list head. Fill in p, s.

Fill in *next* from the used-list head.

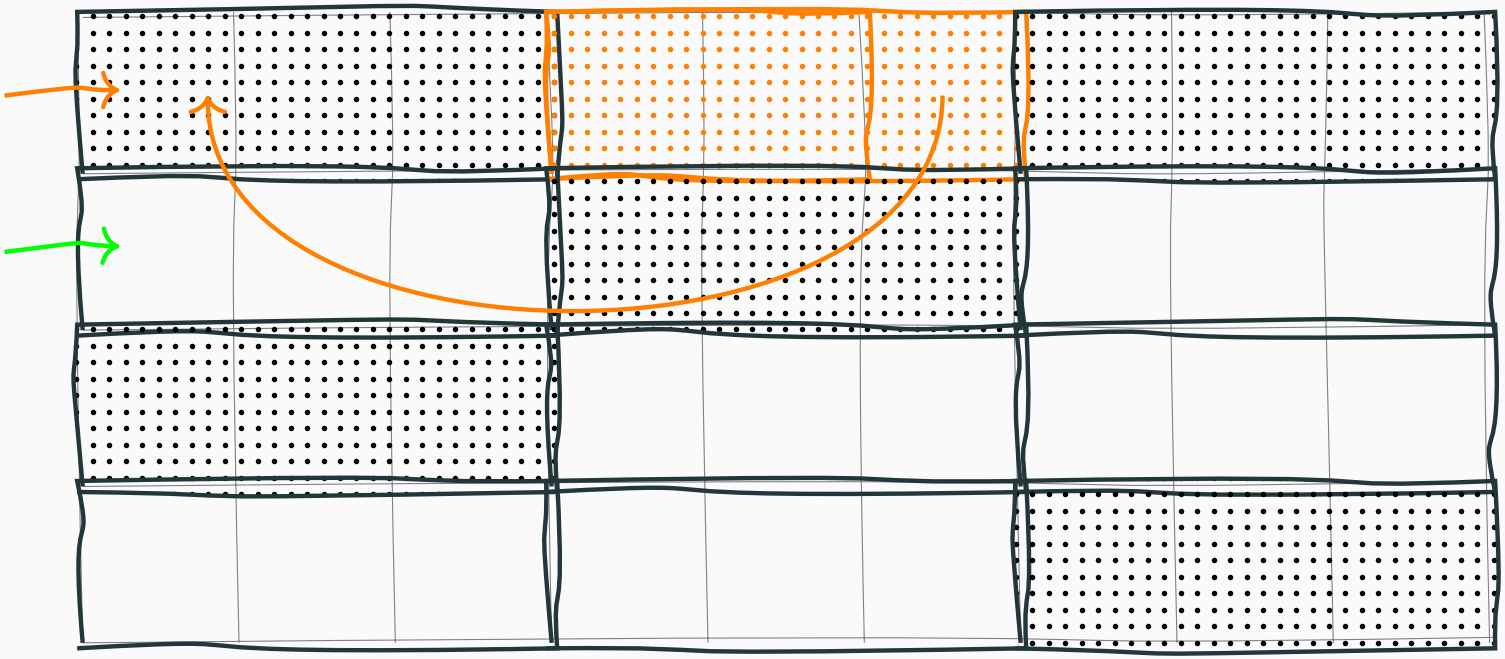


Linked-list implementation

How do we store something in memory?

Take first element from free-list head. Adjust free-list head. Fill in p, s.

Fill in *next* from the used-list head.

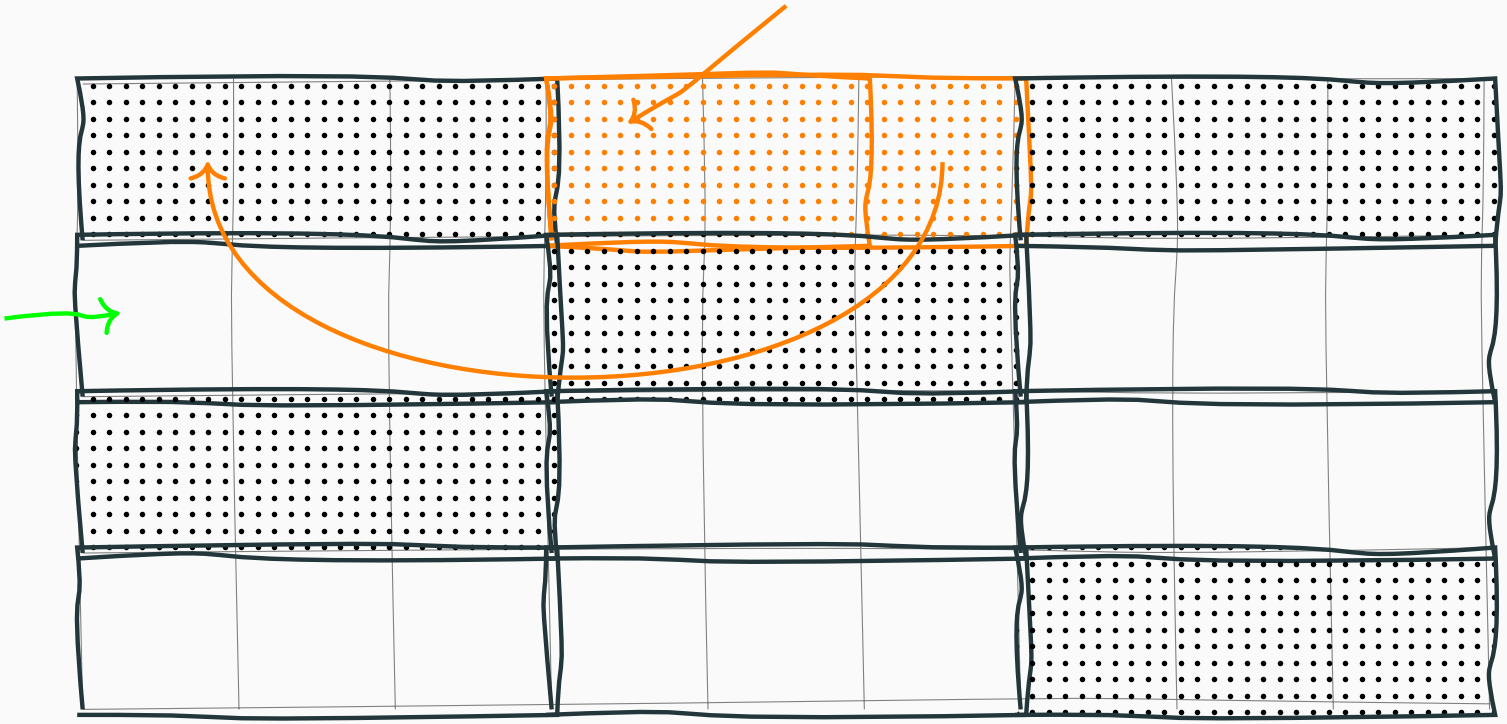


Linked-list implementation

How do we store something in memory?

Take first element from free-list head. Adjust free-list head. Fill in p, s.

Fill in *next* from the used-list head. Adjust used-list head.



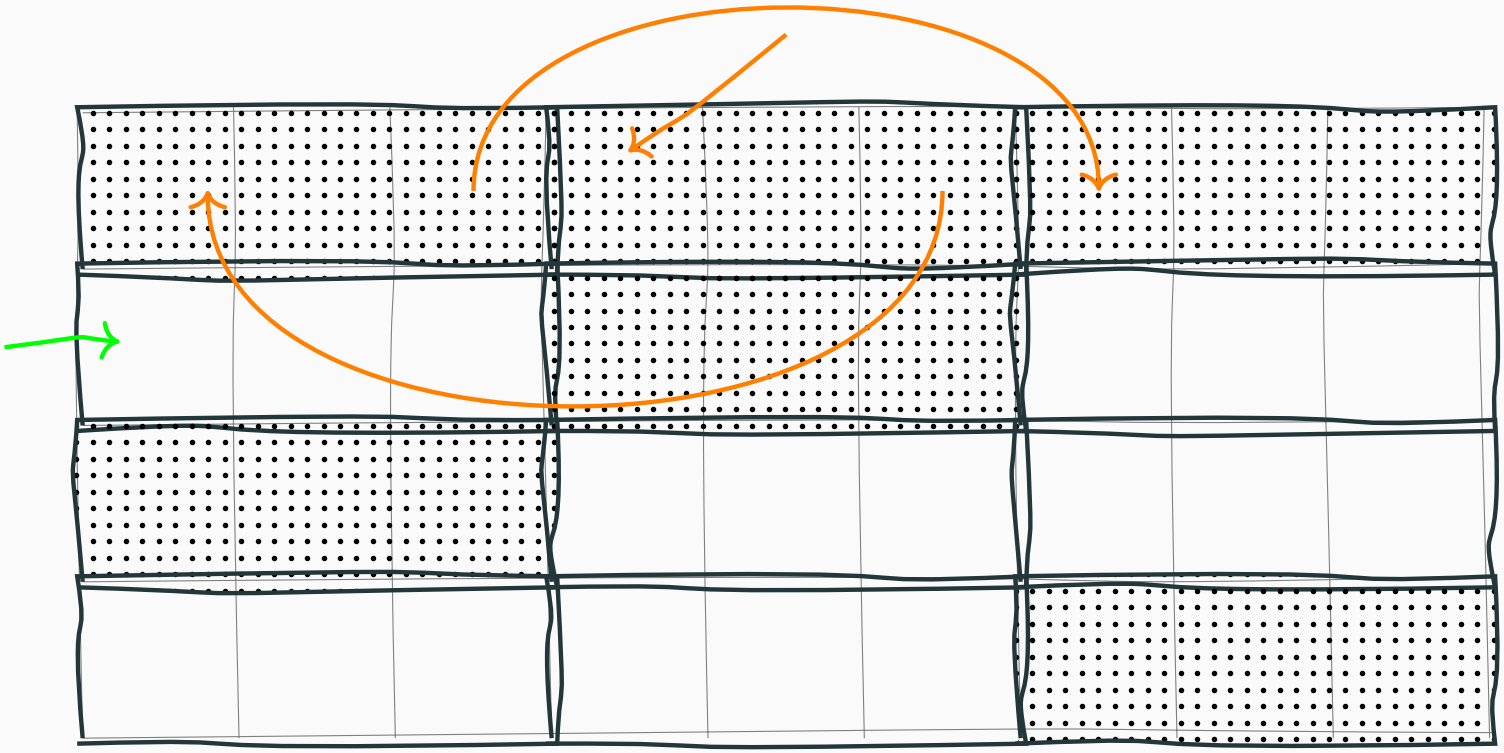
Fill in *next* from the used-list head. Adjust used-list head.

Linked-list implementation

Transition (B): $\mathit{Memory} \leftarrow \mathit{Memory} \cup \{(p, s)\}$.

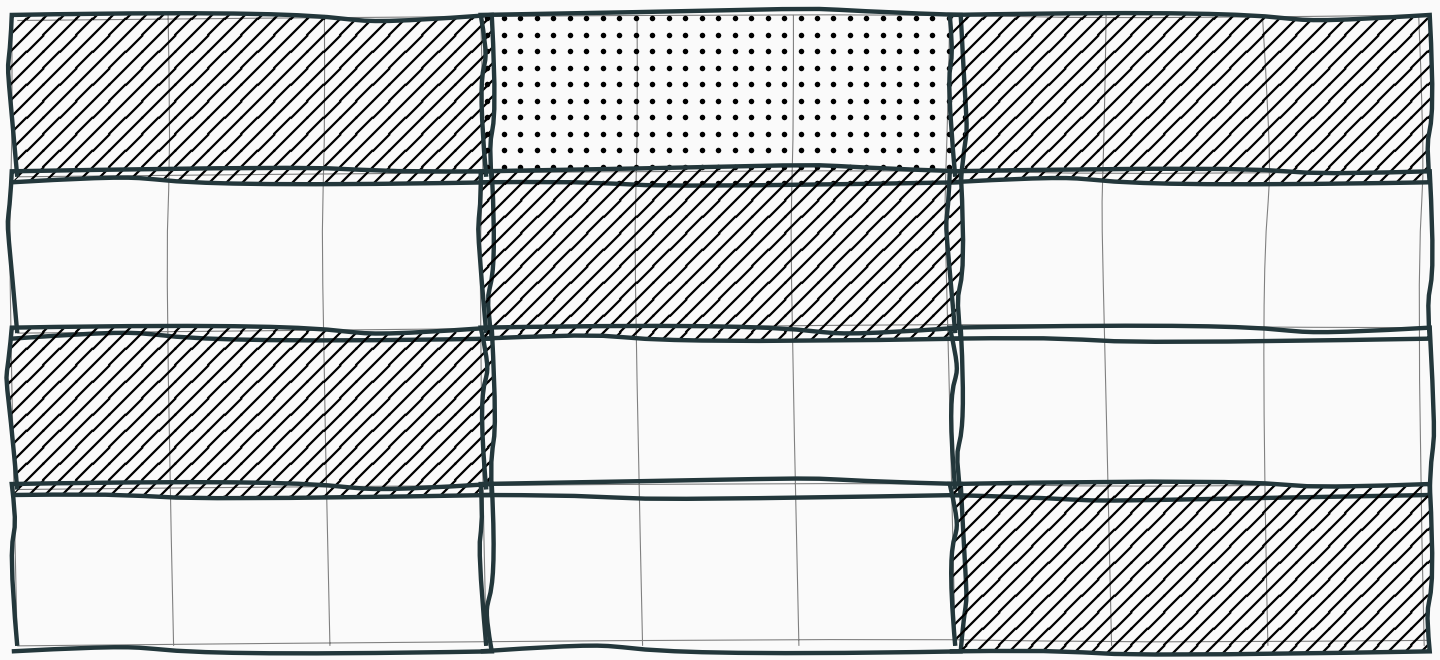
Take first element from free-list head. Adjust free-list head. Fill in p, s.

Fill in *next* from the used-list head. Adjust used-list head. Done.



Linked-list implementation

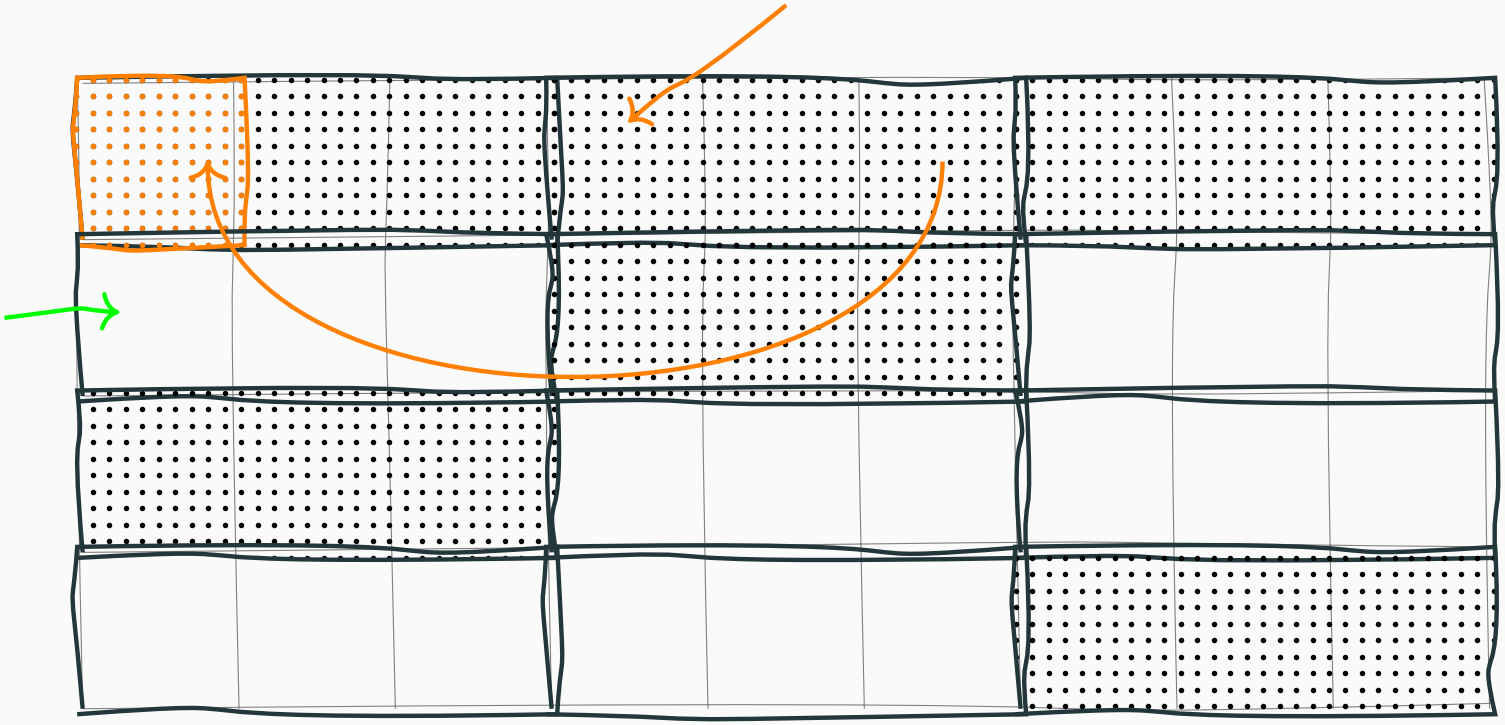
Meditative slide for everybody to rest for a minute.



Linked-list implementation

Transition C: $Memory \leftarrow \{(p', s') \in Memory | p' \neq p\}$.

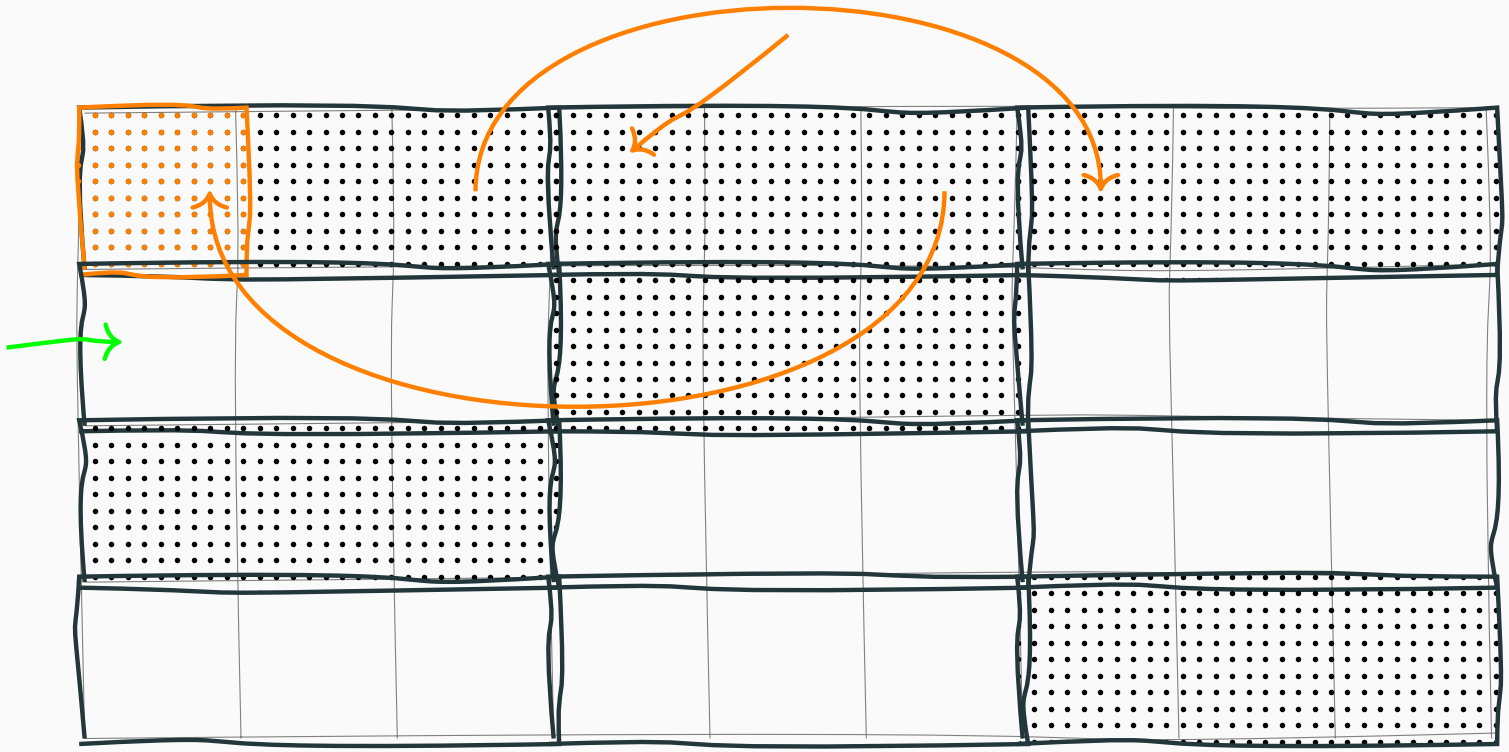
Follow used-list until p is found.



Linked-list implementation

Transition C: $Memory \leftarrow \{(p', s') \in Memory | p' \neq p\}$.

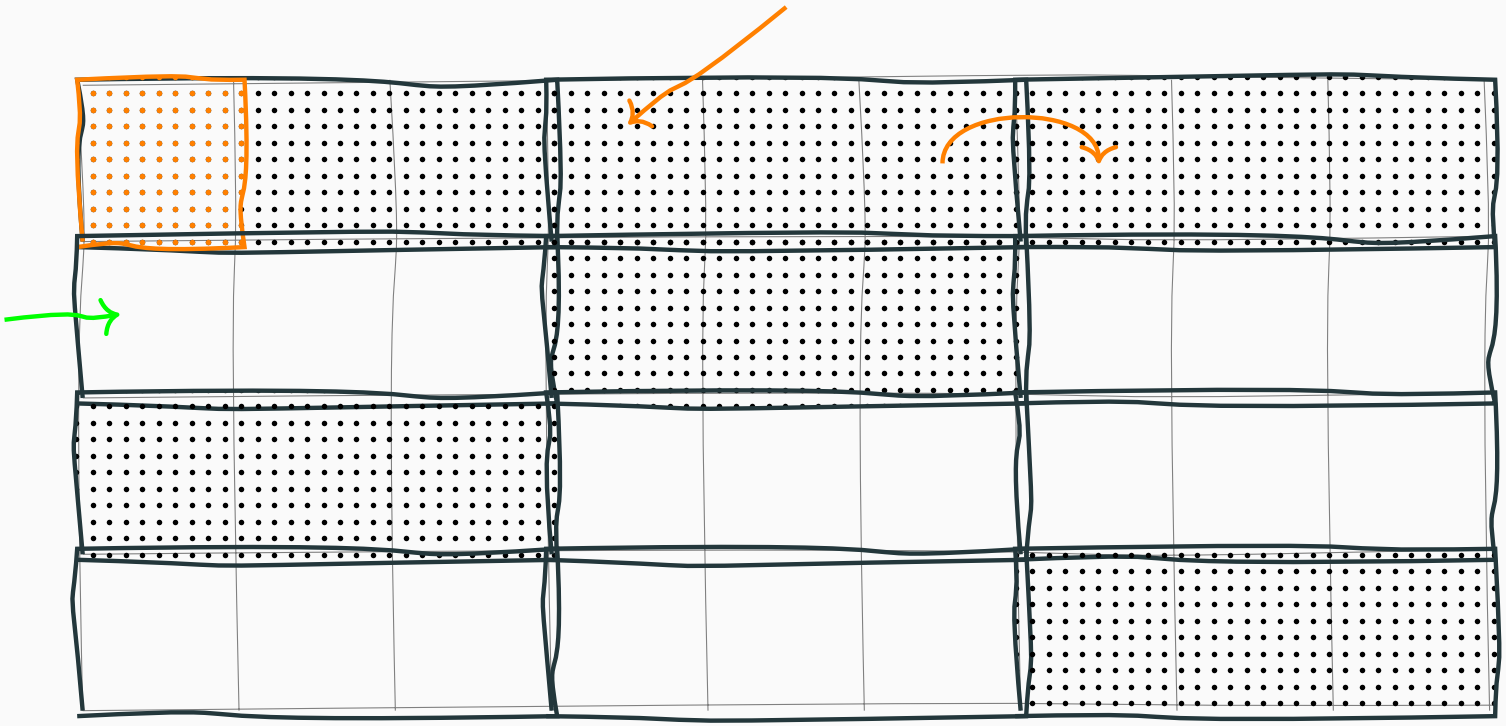
Follow used-list until p is found. Look at *next*.



Linked-list implementation

Transition C: $Memory \leftarrow \{(p', s') \in Memory | p' \neq p\}$.

Follow used-list until p is found. Adjust *next* that led us here so that the block with p is removed from the used-list.

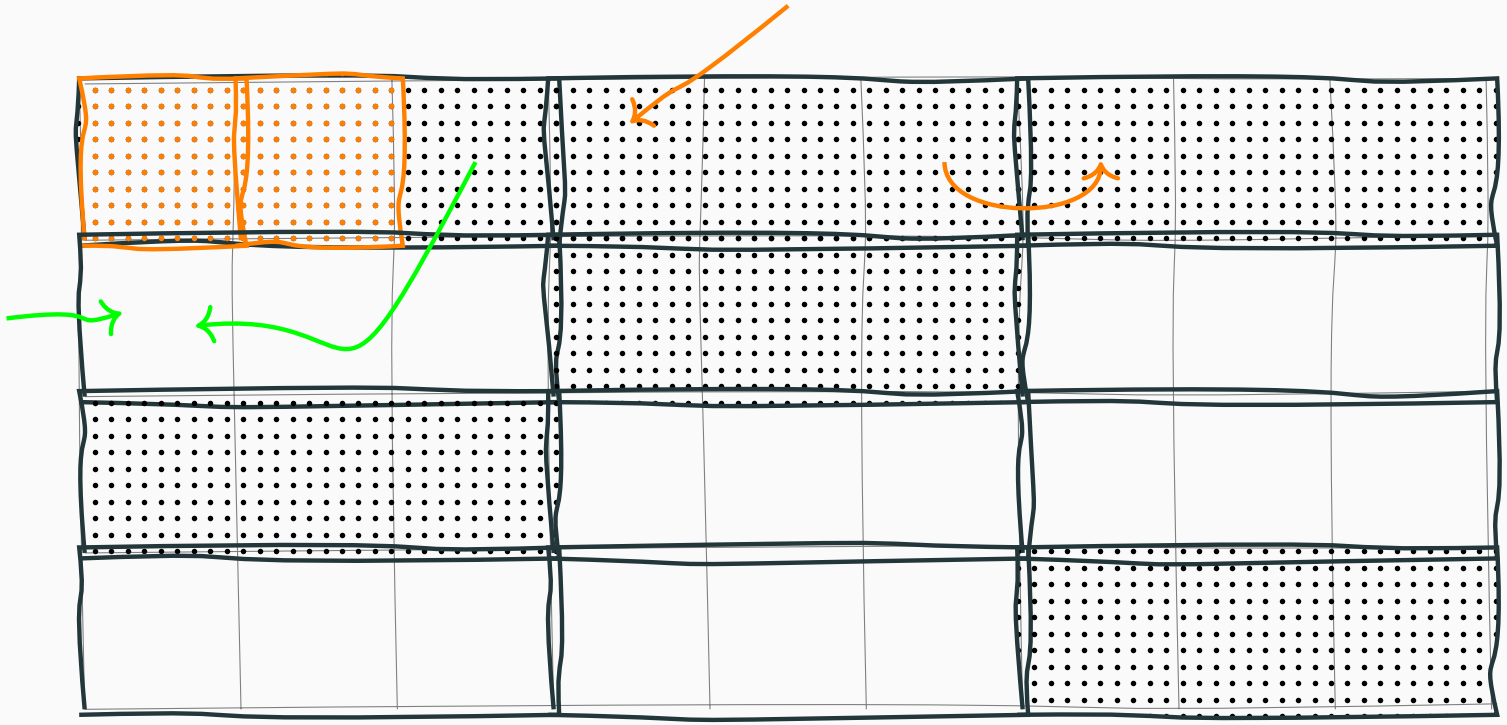


Linked-list implementation

Transition C: $Memory \leftarrow \{(p', s') \in Memory | p' \neq p\}$.

Follow used-list until p is found. Adjust *next* that led us here so that the block with p is removed from the used-list.

Output s, and replace *next* with free-list head.



Linked-list implementation

Transition C: $Memory \leftarrow \{(p', s') \in Memory | p' \neq p\}$.

Follow used-list until p is found. Adjust *next* that led us here so that the block with p is removed from the used-list.

Output s, and replace *next* with free-list head. Adjust free-list head.