Fundamentals of “security exploits”

… and implications for the design of future secure computing architectures & languages

(extended edition with proof sketches)

Thomas Dullien - optimyze - January 2019

thomasdullien@optimyze.cloud - https://optimyze.cloud

optimyze

Memory Corruptions: A problem since 1976

- First mention of buffer overflows and their exploitation is from 1976 or even 1972

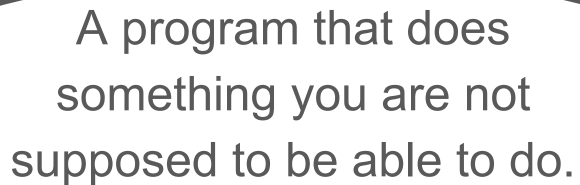
- 46-42 year later memory corruptions are still the primary source of security vulnerabilities

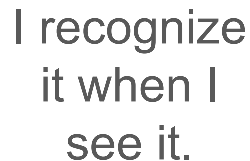
- Why is this not solved yet?

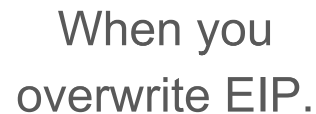
optimyze

What is this thing we call an 'exploit'?

Central term in computer security - but ill-defined.







optimyze

“Security exploits” are poorly understood

- Computer Science studies “what happens when things go right”.
- Electrical engineering studies “how to make things go right almost all of the time” (reliability).
- Formal methods in Computer Science studies “how to prove things do not go wrong”
- Security is about “what are the consequences when something goes wrong”, which is … none of the above.

Why define it?

Real-world decisions are being made:

Exploit Mitigations

Trade-offs involving:

- Performance
- Programmability
- Complexity

vs.
“difficulty of exploitation”.

Exploitability

Do random bitflips in memory (Rowhammer) constitute a security problem?

Can they be “exploited”?

Do we need to tell customers?

optimyze

Raising Bars

Mitigations are often accepted on the assurance that they “raise the bar”.
Unanswered questions:

- How much will the bar be raised?
- Will it cost the attacker effort once, or every time he exploits something?
- Is the cost (performance, programmability, complexity) worth the gain?

The discussion needs clear terms & understanding, lest we end up like the MitiGator:

MitiGator

MitiGator: Raising the bar until no more exploits can be seen.

My background

- First exploit written in 1998 - 21 years ago, presented on Win32 Heap Exploits in 2002
- Built tools/paper about patch diffing (“BinDiff”) in 2003/2004. Started a company focused on reverse engineering tools.
- Company acquired by Google in 2011 (response to Aurora).
- 2011-2015: Working on malware analysis infrastructure.
- 2016-2019: Vulnerability research at Google Project Zero

- Since 2009 or so: Thinking about “foundations” for exploitation.
- December 2017: Finally got a paper about it published.

This talk

- **Informal introduction to the concept “weird machine”**
- **Proof sketch**
- **Relationships to other fields**
    - Automata and model checking (“stuttering bisimulation”)
    - Dynamical systems (Lyapunov Exponents, Cellular Automata)
    - Minimal turing machines and emergent computation

Weird machines - what are they?
Part 1: The intuition.

Three perspective shifts are necessary

Software as emulator for an intended finite-state machine (or transducer).

Software as something that restricts the reachable state space of a device.

Symmetry between providing input to a finite-state machine and “programming”.

1. What is software?

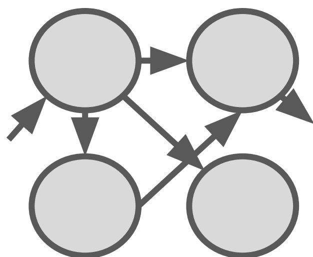
Software is an emulator ...

... for a finite state machine that we would like to have.

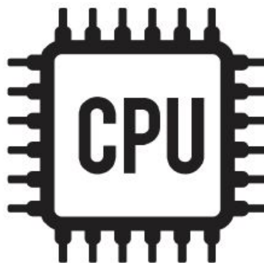
optimyze

Why do people write software?

- They want to solve a concrete problem by means of a (finite state) machine (or more precisely transducer - a finite-state machine with output)
- They do not have the machine to solve it
- They write software that emulates the finite state machine they want on a general-purpose CPU



What I need



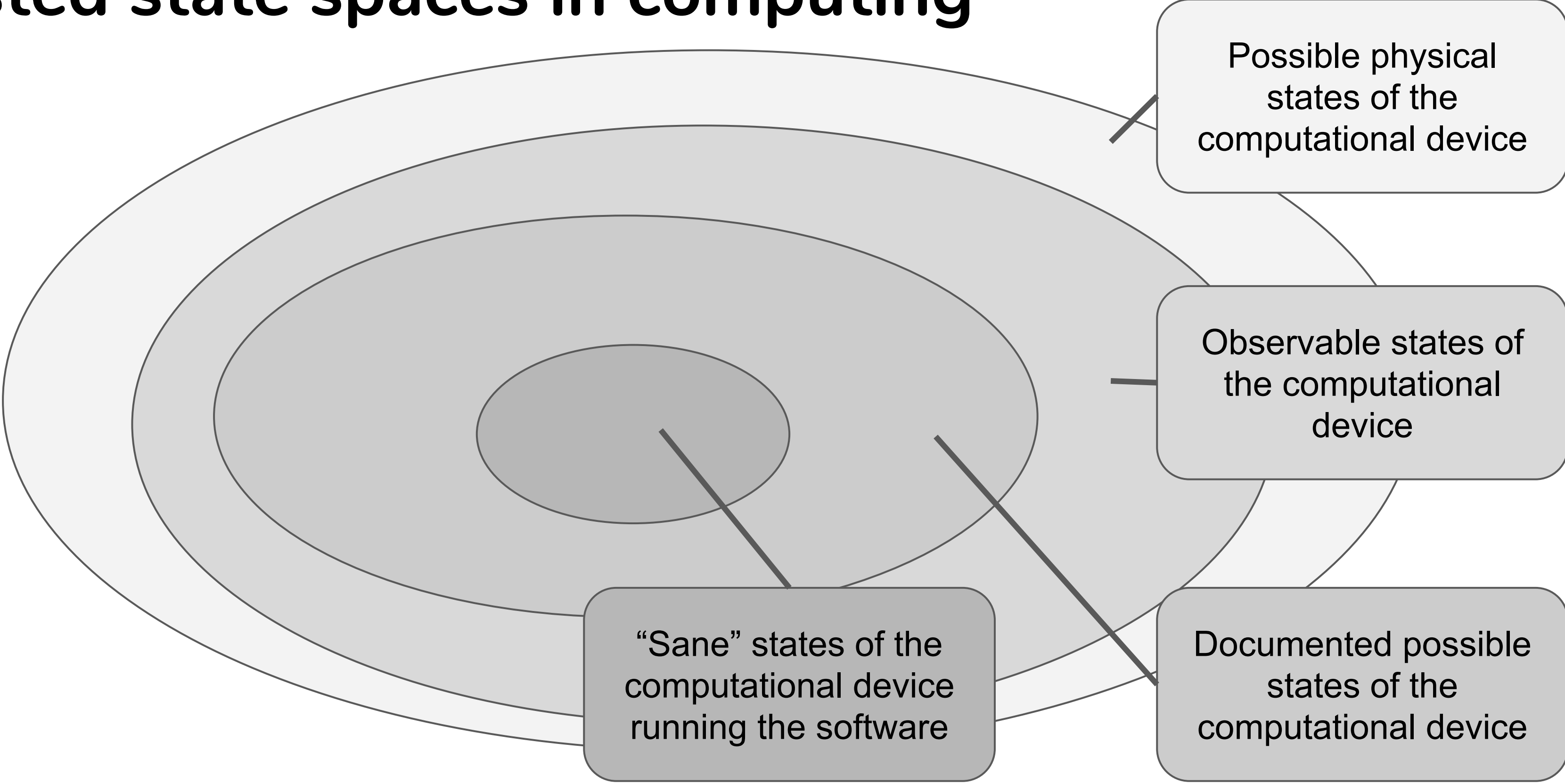
What I have

2. What is software?

Software is a restriction …

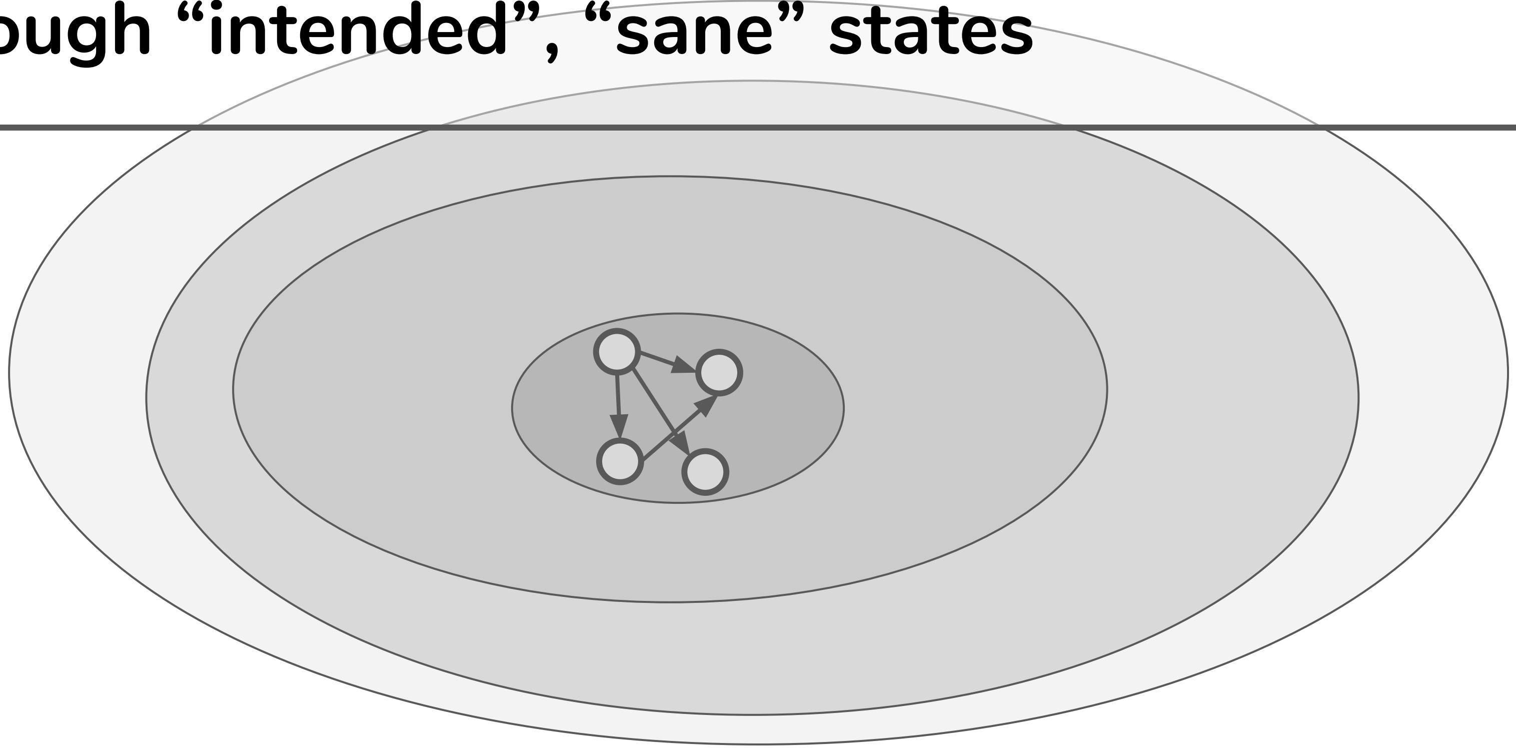
… that restricts the number of reachable states in a general-purpose CPU.

Nested state spaces in computing



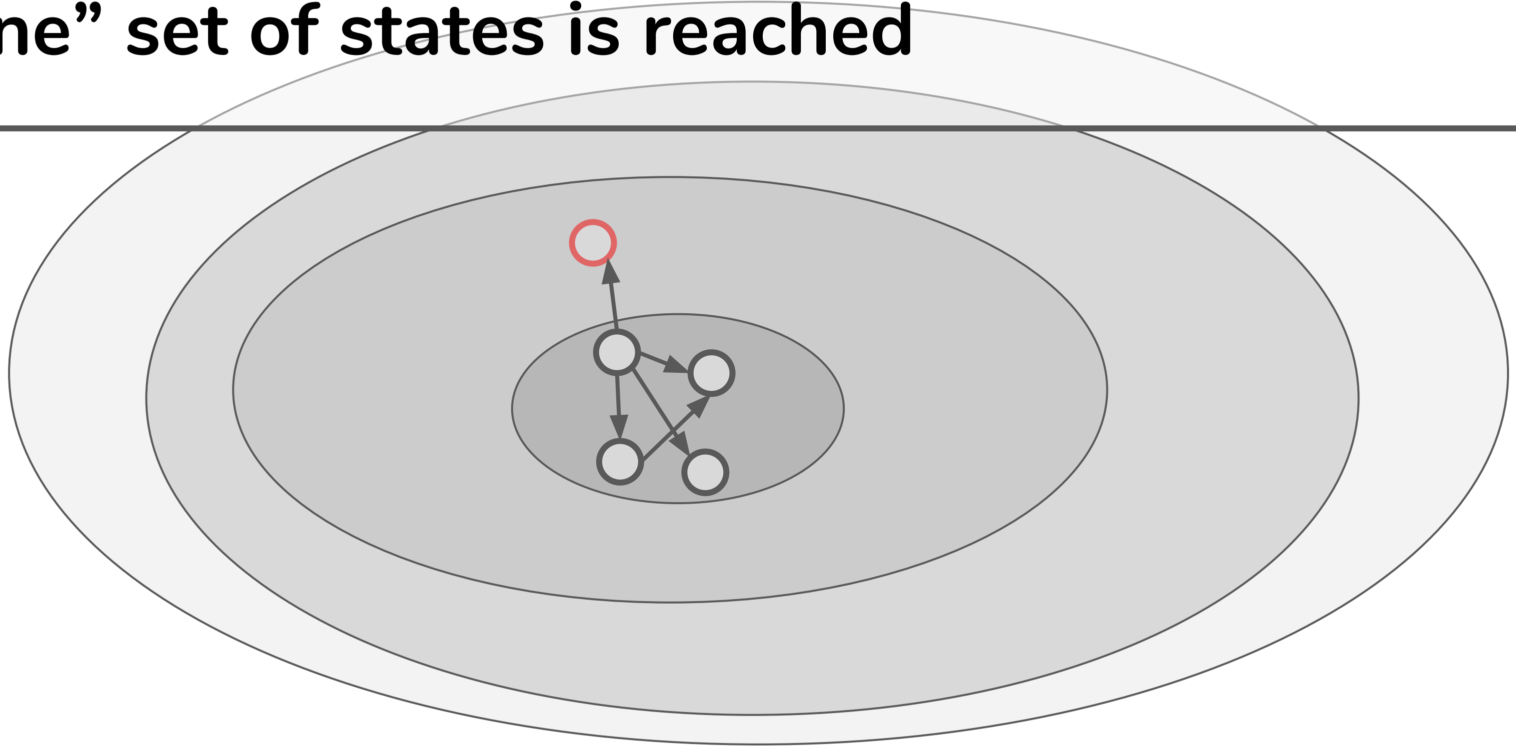
optimyze

Program execution should follow trajectories through “intended”, “sane” states



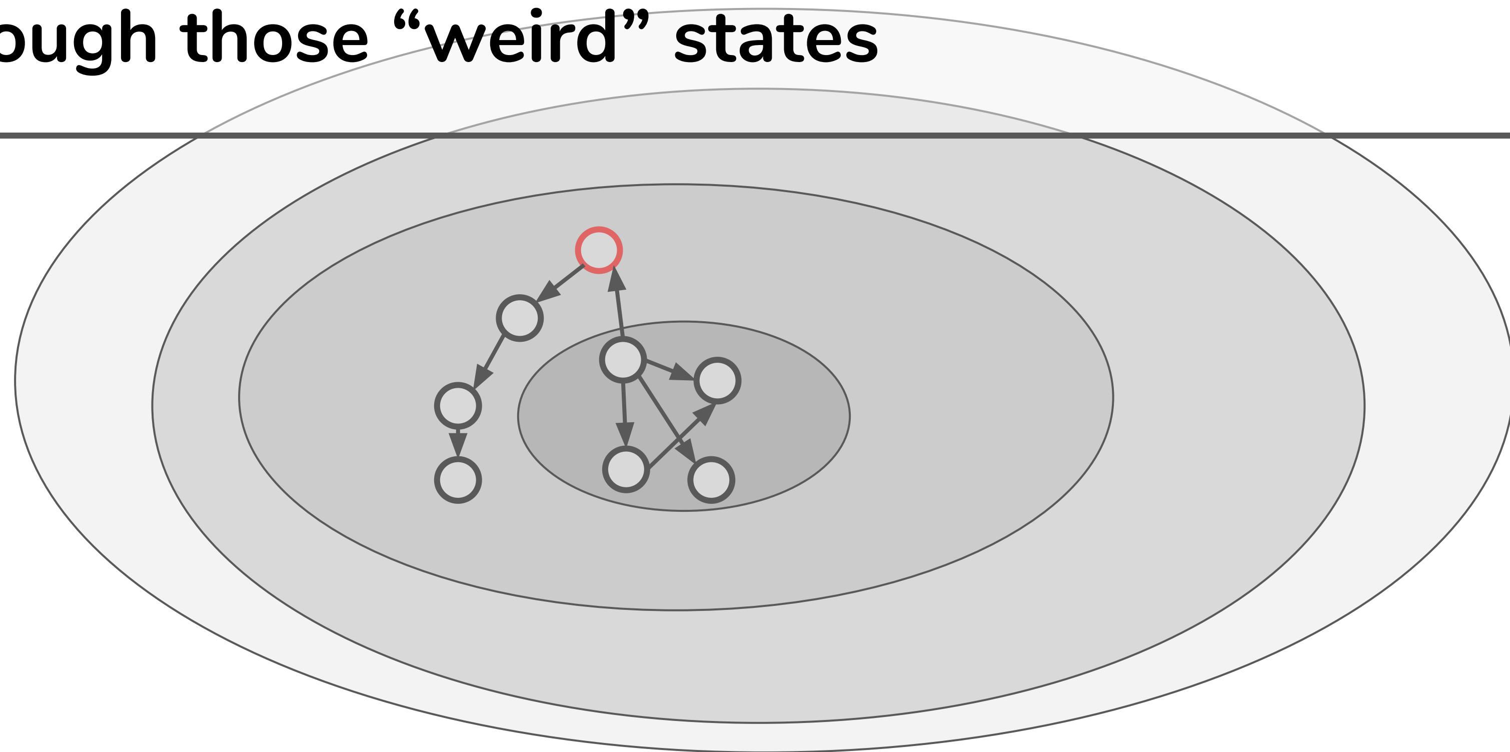
optimyze

During exploitation, a state outside the “intended”, “sane” set of states is reached



optimyze

The attacker carefully controls the trajectory through those “weird” states



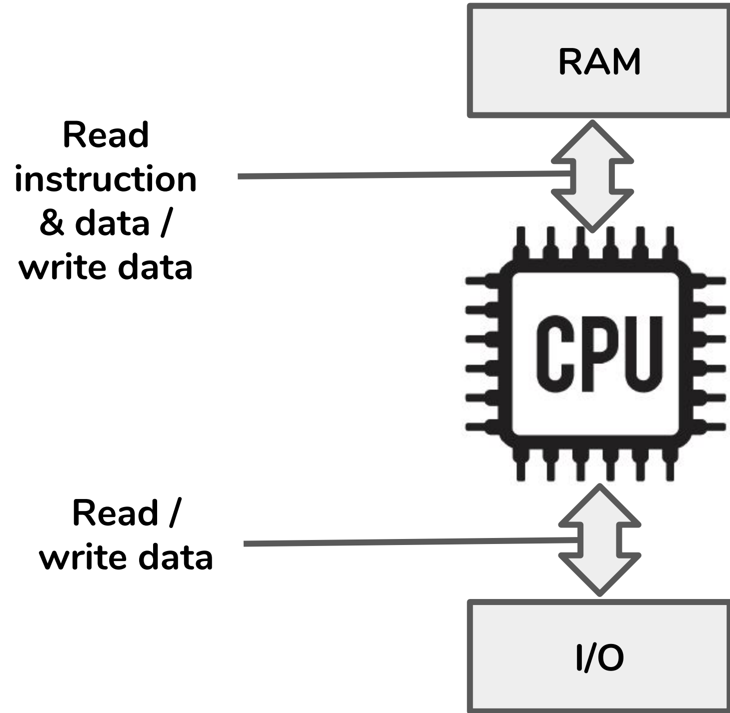
optimyze

3. What is programming?

Providing input to a computer…

… is almost indistinguishable from programming.

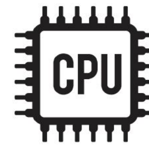
Symmetry between input and programming



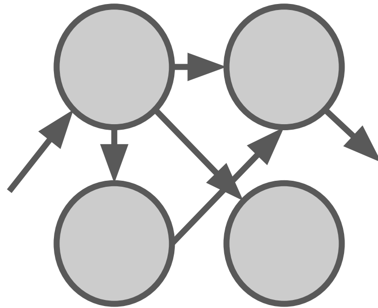
optimyze

Back to weird machines

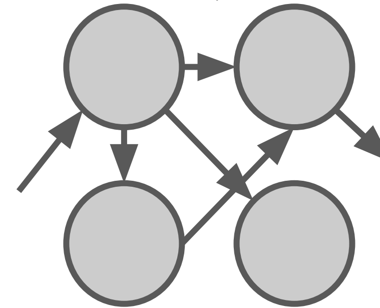
The intended and emulated FSM



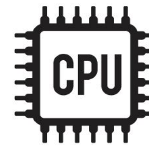
Intended finite state machine (IFSM)



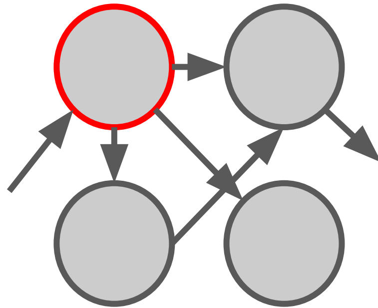
Emulated version of the IFSM



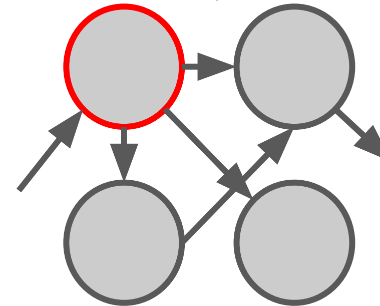
The intended and emulated FSM



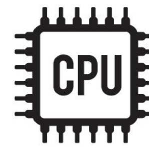
Intended finite state machine (IFSM)



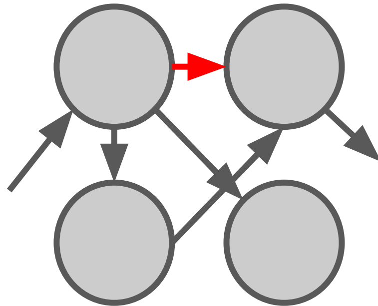
Emulated version of the IFSM



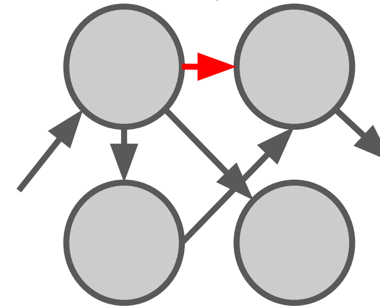
The intended and emulated FSM



Intended finite state machine (IFSM)

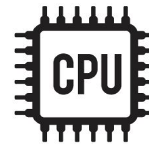


Emulated version of the IFSM

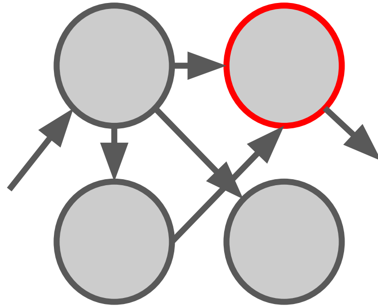


optimyze

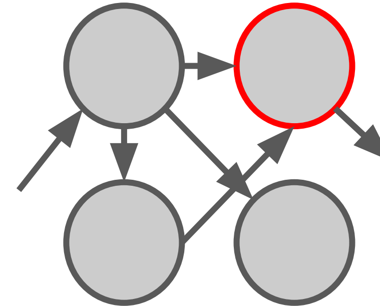
The intended and emulated FSM



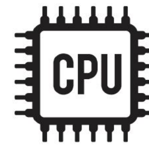
Intended finite state machine (IFSM)



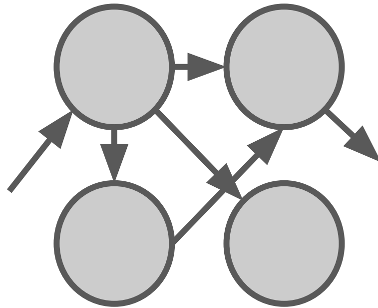
Emulated version of the IFSM



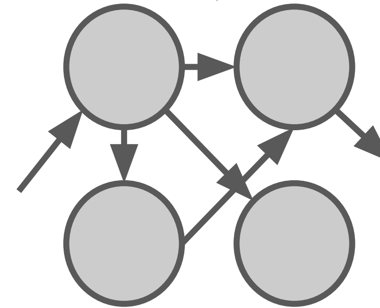
The intended and emulated FSM



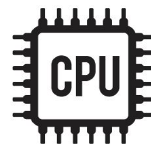
Intended finite state machine (IFSM)



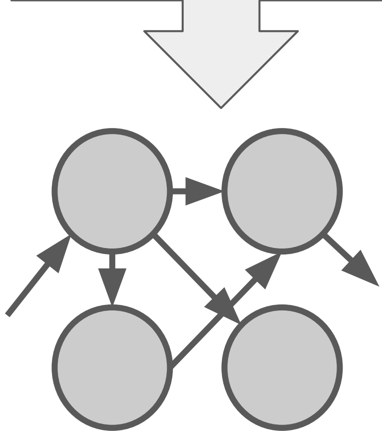
Emulated version of the IFSM



The intended and emulated FSM

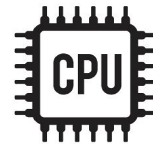


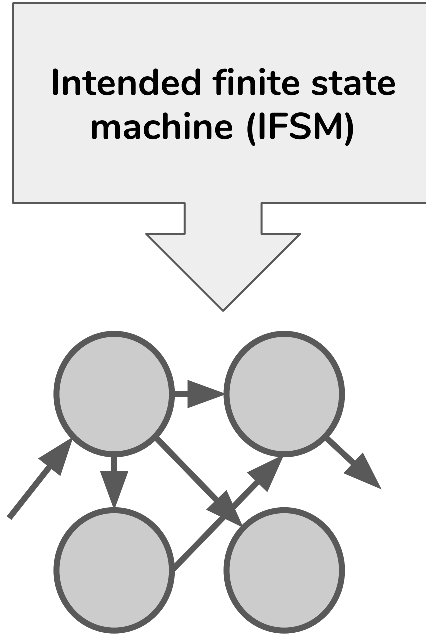
Intended finite state machine (IFSM)



optimyze

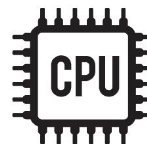
The intended and emulated FSM



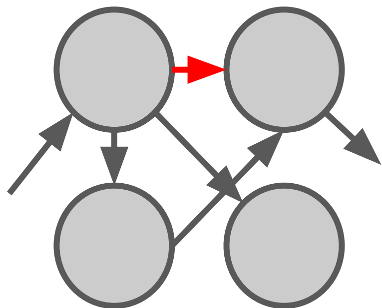


optimyze

The intended and emulated FSM

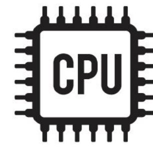


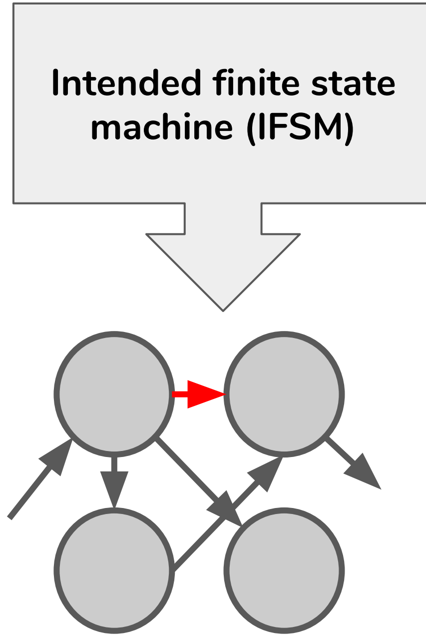
Intended finite state machine (IFSM)



optimyze

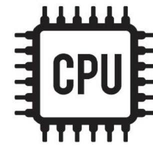
The intended and emulated FSM

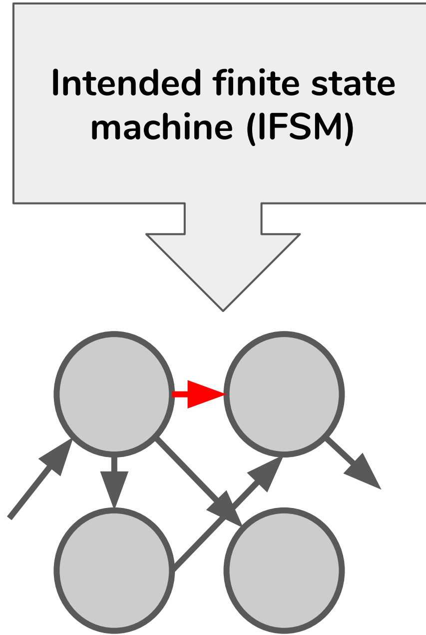




optimyze

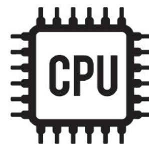
The intended and emulated FSM

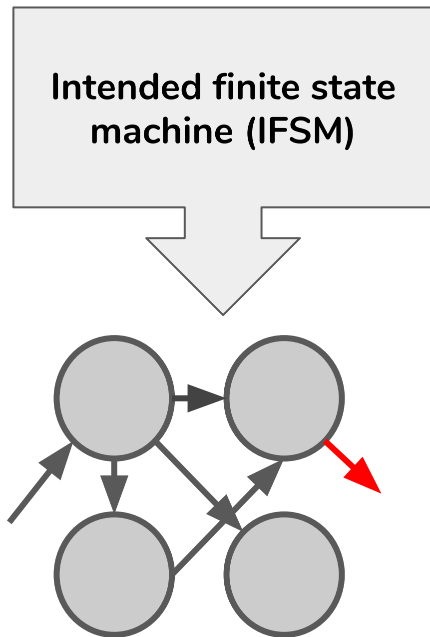


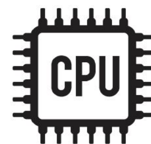


optimyze

The intended and emulated FSM

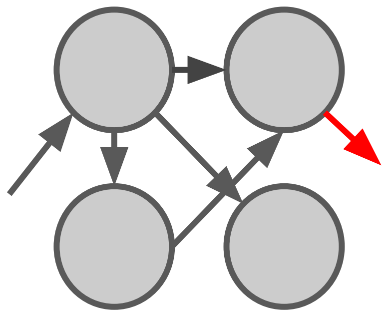






The intended and emulated FSM

Intended finite state machine (IFSM)



optimyze

The weird machine

- Starting from a “sane” state, we somehow enter a state that does not correspond to any state in the IFSM. A **weird** state.
- By repeatedly applying transformations designed to transform sane state to stane state, we reach a new weird state.
- We obtain a machine that can be driven through various states by our input. It contains many more states than the IFSM, and is emergent, not designed.

Attacker goal: Reach a state that violates a security requirement.
Write a program for the weird machine.

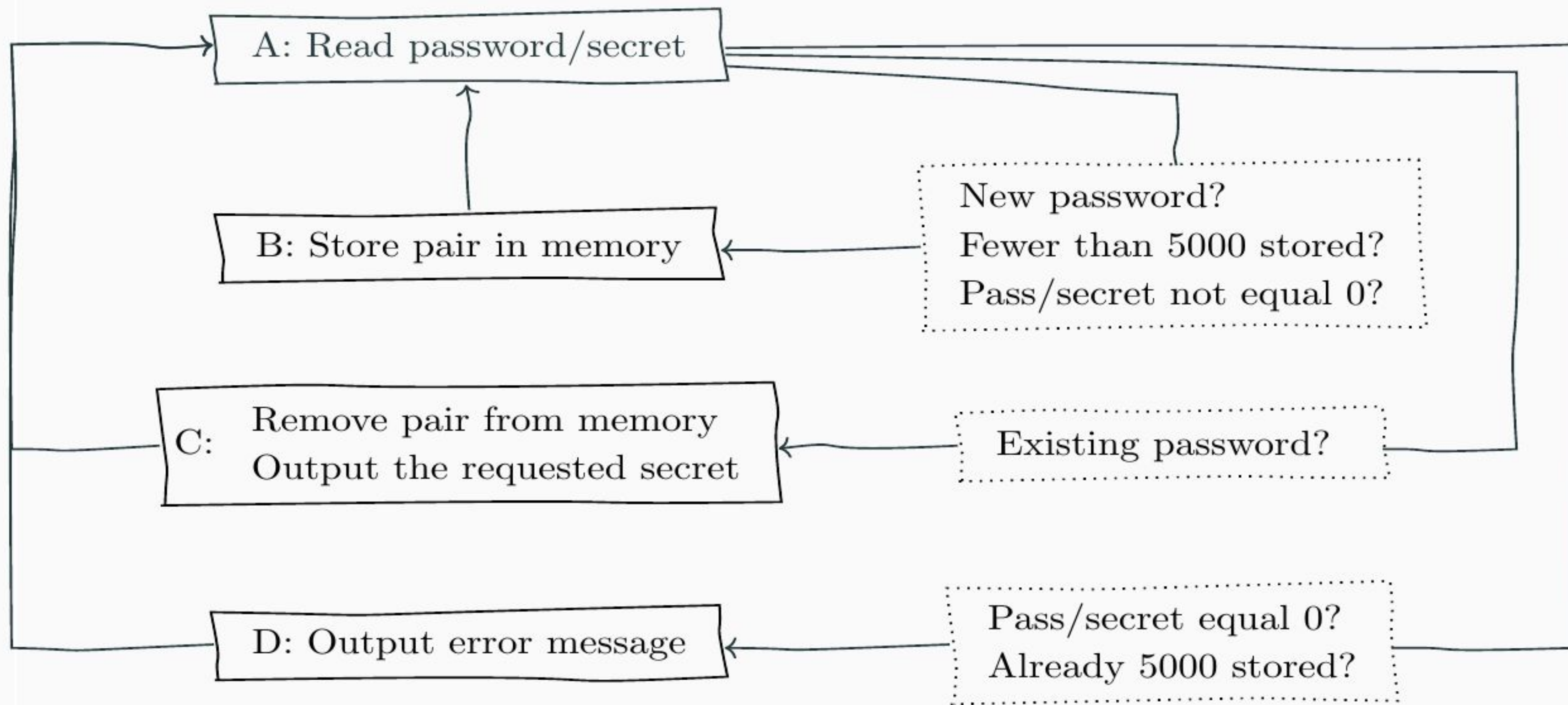
Weird machines - what are they?
Part 2: Definitions etc.

The Intended Finite State Machine

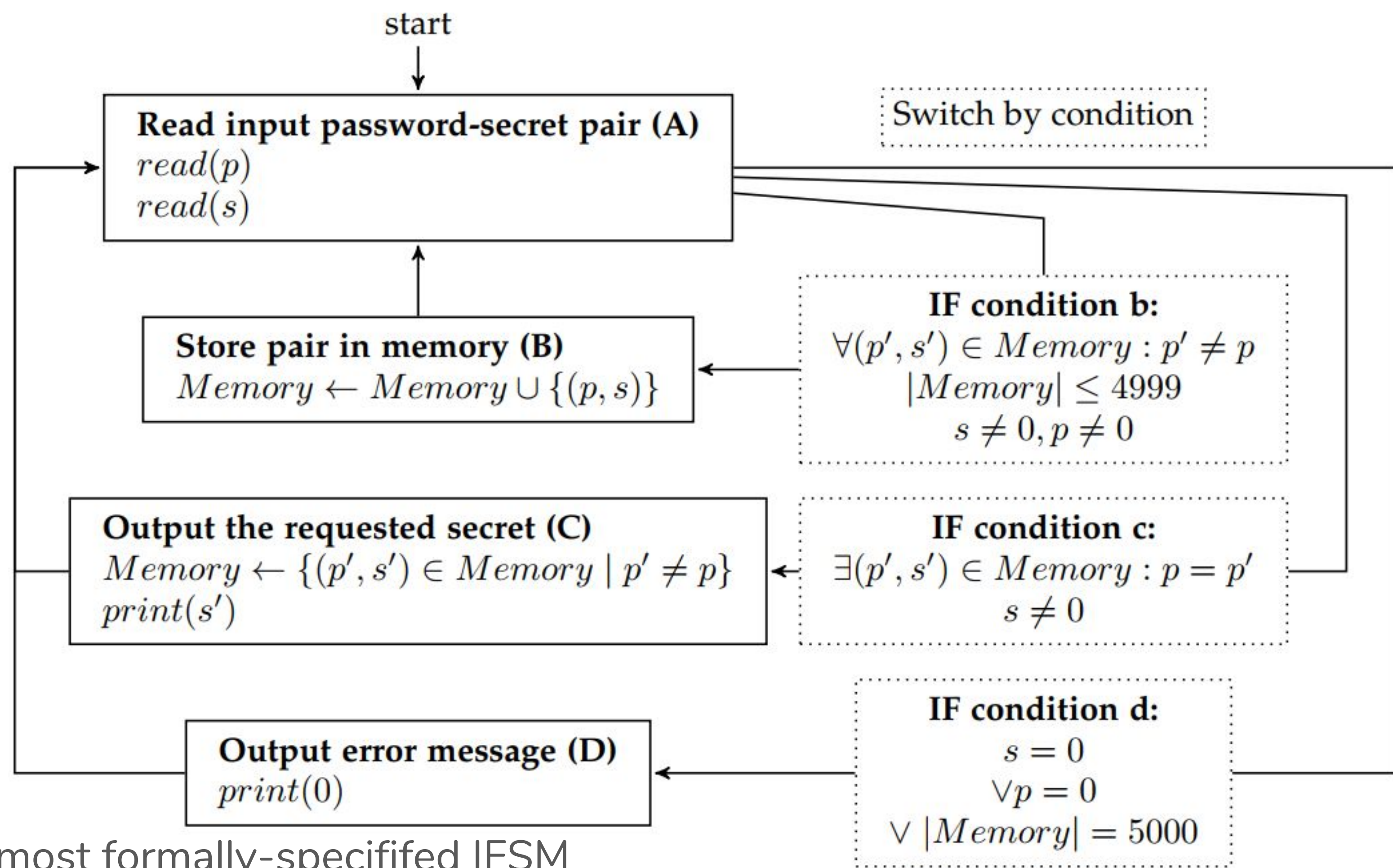
- The state machine we want to have is called the “intended finite state machine” (***IFSM***)
- It is usually not explicitly specified
- It is “perfect by design” - i.e. it is the state machine that fully implements our intents.
- It cannot, by definition, have security problems (one can specify formal “security properties”)

Let us examine the smallest interesting example we can come up with.

optimyze



An informally-specified IFSM



An almost formally-specifed IFSM

optimyze

Small enough to specify (don't read this now!)

$$Q := \{A_M, M \in \mathcal{M}\}, \qquad i := A_\emptyset, F := \emptyset$$

$$\Sigma := \{(p,s) | p, s \in bits_{32}\}, \qquad \Delta := \{s \in bits_{32}\}$$

$$\delta := A_M \times (p,s) \quad \rightarrow \begin{cases} A_{M \cup (p,s)} \text{ if } & \begin{array}{l} (p,s) \notin M \\ \wedge |M| \leq 4999 \\ \wedge s \neq 0 \end{array} \\ A_{M \setminus (p,s)} \text{ if } (p,s) \in M \\ A_M \text{ otherwise} \end{cases}$$

$$\sigma := A_M \times (p,s) \quad \rightarrow \begin{cases} s' \text{ if } (p,s') \in M \\ 0 \text{ if } s = 0 \vee |M| = 5000 \end{cases}$$

optimy

What is security?

Security are properties of the IFSM …

… that we want to hold in the presence of an adversary.

Security Properties

- Security properties are “what do we want to be true for the IFSM”.
- Informally, in our example:

You need to know (or guess) the right password in order to obtain the stored secret.

- There should be no way to automatically extract the secret without the password with any probability better than guessing

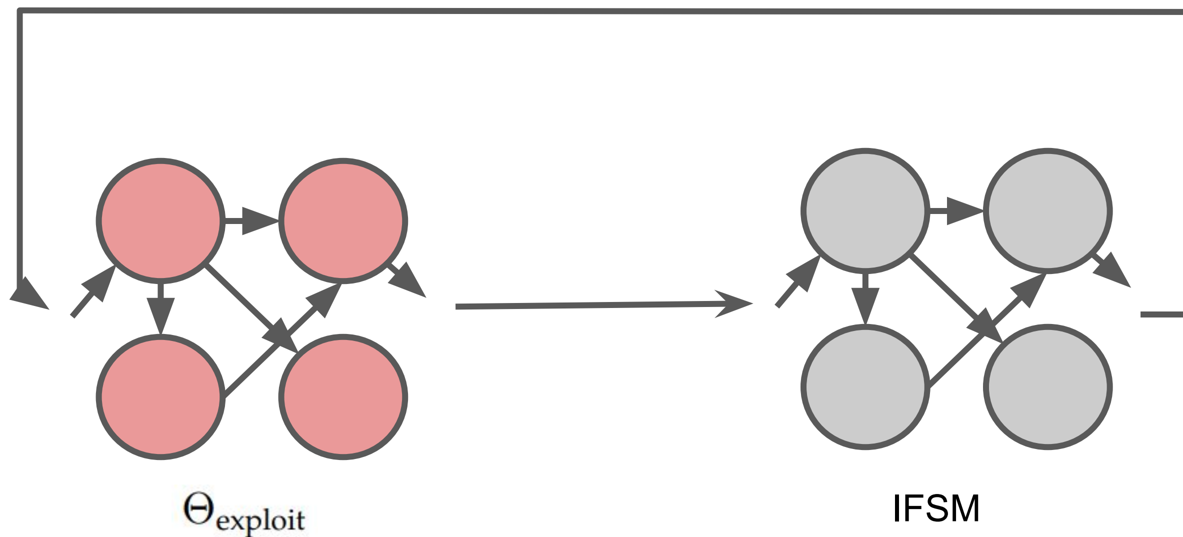
Formal security properties of the IFSM

- Multi-step game with attacker & defender that interact with the IFSM
- Game flow:
    a. Attacker chooses a distribution over finite-state transducers that have as input alphabet the output alphabet of the IFSM, and that have as output alphabet the input alphabet of the IFSM
    b. Defender draws p, s uniformly at random from $bits_{32}$
    c. Attacker draws a finite-state transducer Θ_{exploit} from his distribution and connects it to the IFSM. The transducer is allowed to interact with the IFSM for n_{setup} steps
    d. The defender sends p,s to the IFSM
    e. The attacker is allowed to have his transducer interact with the IFSM for n_{exploit} steps.

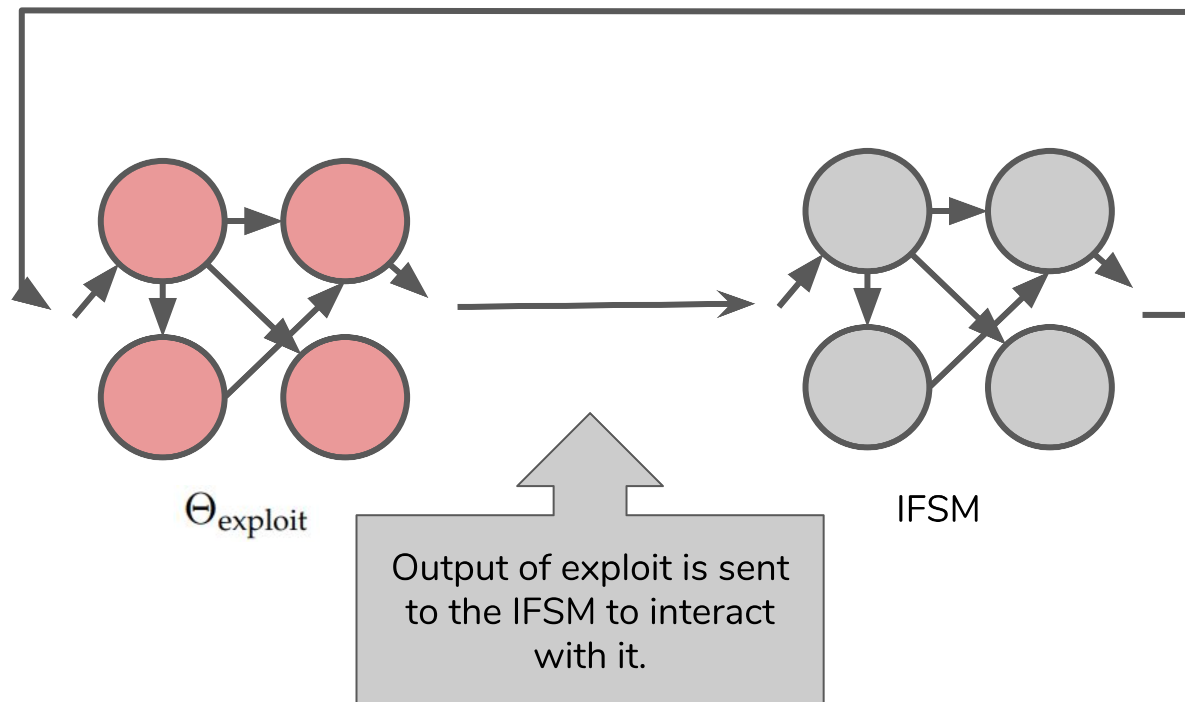
Let o_{IFSM} be the sequence of outputs that the IFSM sends to the exploit during this game.

optimyze

Dueling transducers: Diagram



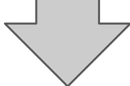
Dueling transducers: Diagram

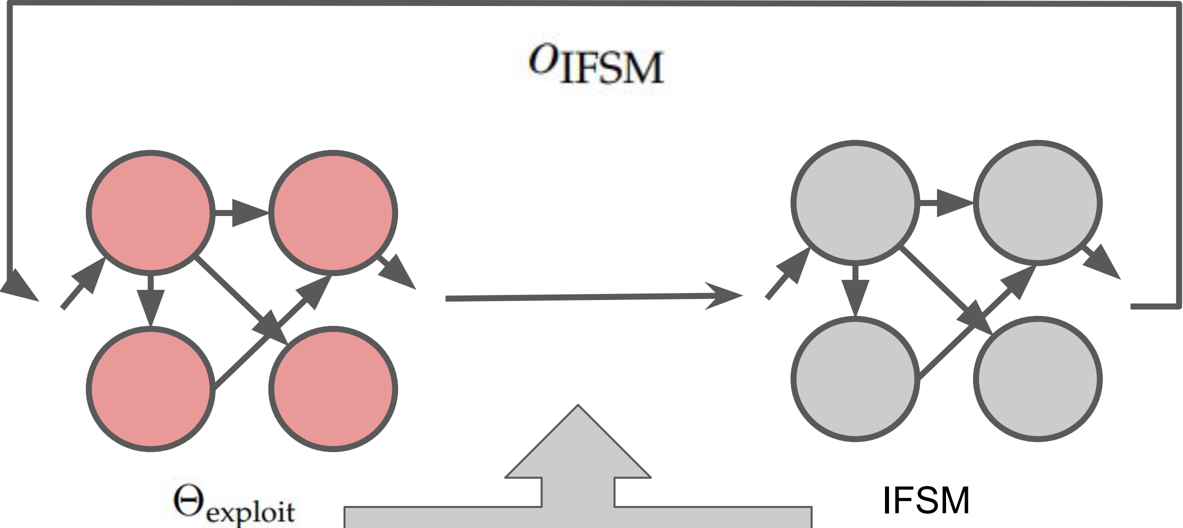


optimyze

Dueling transdu

Output of the IFSM is sent to the exploit.





Output of exploit is sent to the IFSM to interact with it.

Formal security properties of the IFSM

The desired security property of our game is:

$$P[s \in o_{\text{IFSM}}] \leq \frac{n_{\text{setup}} + n_{\text{exploit}}}{|bits_{32}|} = \frac{|o_{\text{exploit}}|}{2^{32}}$$

“The probability that the attacker gets the defender’s secret s sent to him needs to no better than what he could obtain through random guessing, no matter what program/exploit he chooses.”

Implementation

We will write software to **emulate** the IFSM on a simple RAM machine.

optimyze

A simple computing environment

- Standard RAM machine with 2^16 memory cells that can hold 32 bit values each.
- Memory cells r0 to r6 are *registers* - memory locations that cannot be accessed indirectly.
- Harvard architecture: Program is not stored in main memory.
- Program denoted with ρ , individual lines ρ_i
- Instructions execute in zero time (timing attacks out of scope for this paper / talk)
- State fully determined by $((q_1, \ldots, q_{2^{16}}) =: \vec{q}, \rho, \rho_i)$
- Set of all states is Q_{cpu}

$\text{LOAD}(C, r_d) \quad : r_d \leftarrow C$

$\text{ADD}(r_{s_1}, r_{s_2}, r_d) \quad : r_d \leftarrow r_{s_1} + r_{s_2}$

$\text{SUB}(r_{s_1}, r_{s_2}, r_d) \quad : r_d \leftarrow r_{s_1} - r_{s_2}$

$\text{ICOPY}(r_p, r_d) \quad : r_d \leftarrow r_{r_p}$

$\text{DCOPY}(r_d, r_s) \quad : r_{r_d} \leftarrow r_s$

$\text{JNZ/JZ}(r, I_z)$

$\text{READ}(r_d) \quad : r_d \leftarrow input$

$\text{PRINT}(r_s) \quad : r_s \rightarrow output$

Two implementations of the IFSM

1. One implementation that simulates the set “Memory” by using a flat linear array & sequential scanning.

2. One that simulates the set “Memory” by using two singly-linked lists and two list-heads.

Foreshadowing: The first one will be provably unexploitable, the second will be provably exploitable.

What is “sane” behavior of a program?

When exactly does a program go “off the rails” ?

Using the IFSM and our abstractions, we can define both “sane” states of the CPU, and erroneous states with unpredictable consequences.

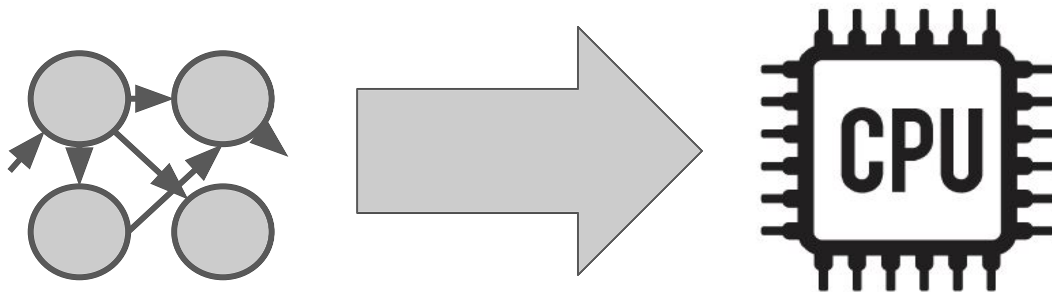
Correspondence between IFSM and CPU

We can map between IFSM states and CPU states, but not perfectly.

$$\gamma\theta, cpu, \rho : Q_\theta \rightarrow \mathfrak{P}(Q_{cpu})$$

Instantiation: Map from states of the IFSM to sets of memory states of the CPU. **Note that one IFSM state can be represented by many CPU states!**

“Give me the set of all possible states of the CPU that can represent a given state of the IFSM”.



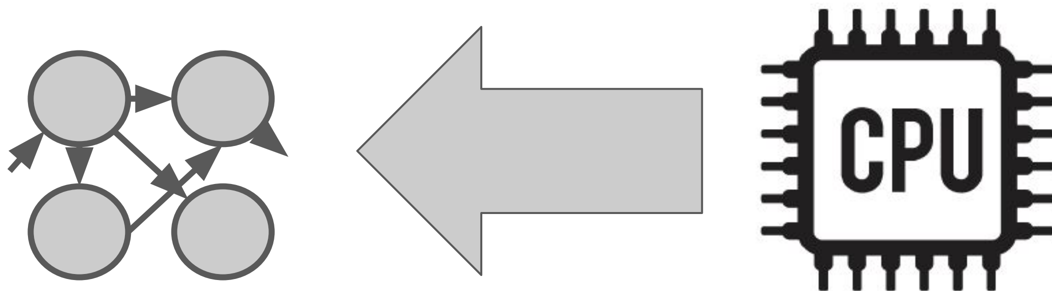
Correspondence between IFSM and CPU

We can map between IFSM states and CPU states, but not perfectly.

$$\alpha_{\theta,cpu,\rho} : Q_{cpu} \to Q_{\theta}$$

Abstraction: Partial map from states of the CPU to states of the IFSM. **Note that there are many CPU states that do not correspond to an IFSM state!**

“What IFSM state does this state of my CPU correspond to?”



Sane states

- CPU States on which the abstraction partial map is defined (e.g. that have a clear correspondence to states of the IFSM) are called ***sane states***.

Transitory states

- A transition between two IFSM states can usually not be performed atomically in the CPU

- Many CPU instructions are usually required to emulate an IFSM transition

- As a consequence, the CPU travels through "***transitory states***":
    - A CPU state occurring during the emulation of an edge that is always part of a benign and intended transition of the IFSM

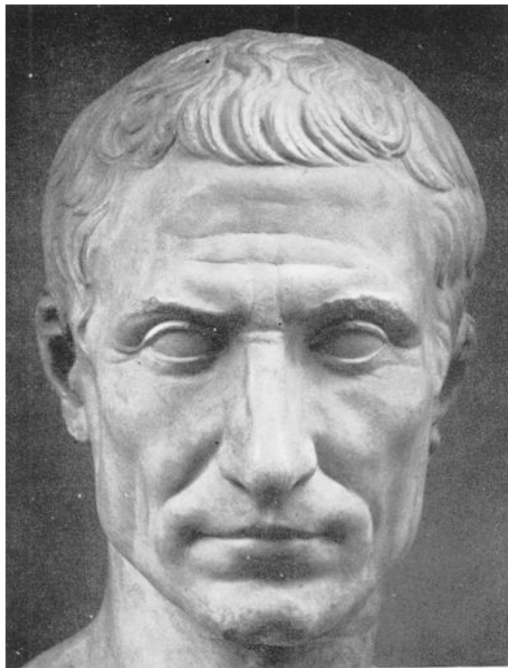
Weird states

- Sane states correspond cleanly to IFSM states. Transitory states correspond to intermediate states while emulating a transition. But there are many possible CPU states that are in neither set.

- These states, which should never be observed, are called ***weird states***.

$$Q_{cpu} = Q_{cpu}^{\text{sane}} \dot{\cup} Q_{cpu}^{\text{trans}} \dot{\cup} Q_{cpu}^{\text{weird}}$$

Weird states



CPU state space est omnis divisa in partes tres:

$$Q_{cpu} = Q_{cpu}^{\text{sane}} \dot{\cup} Q_{cpu}^{\text{trans}} \dot{\cup} Q_{cpu}^{\text{weird}}$$

optimyze

Weird states

Weird states can be entered in many different ways in reality:

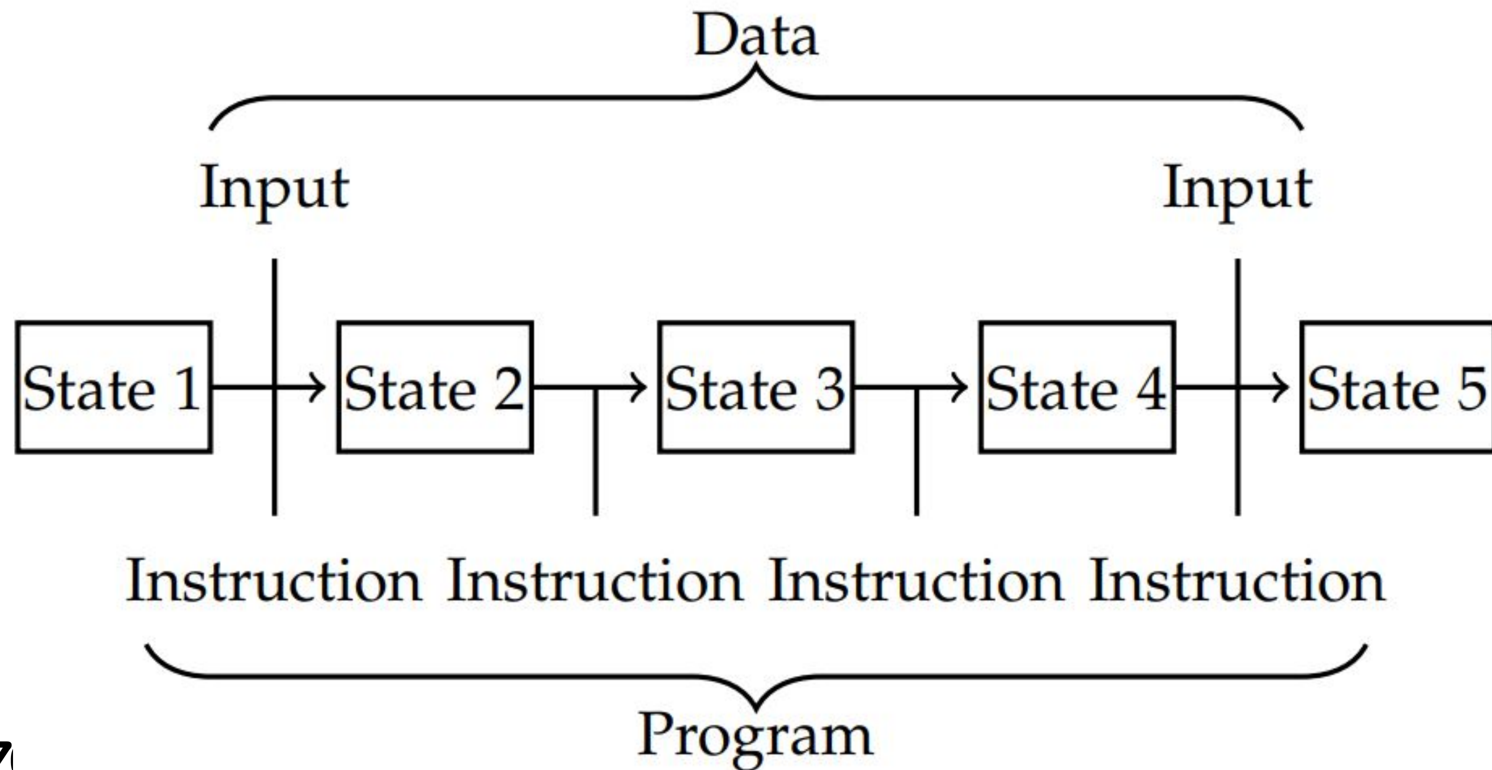
- Human Error (software bugs)
- Hardware Faults (Rowhammer)
- Transcription Errors
- Mis-specification of the semantics of CPU
- ...

What is programming?

Providing input to a computer...

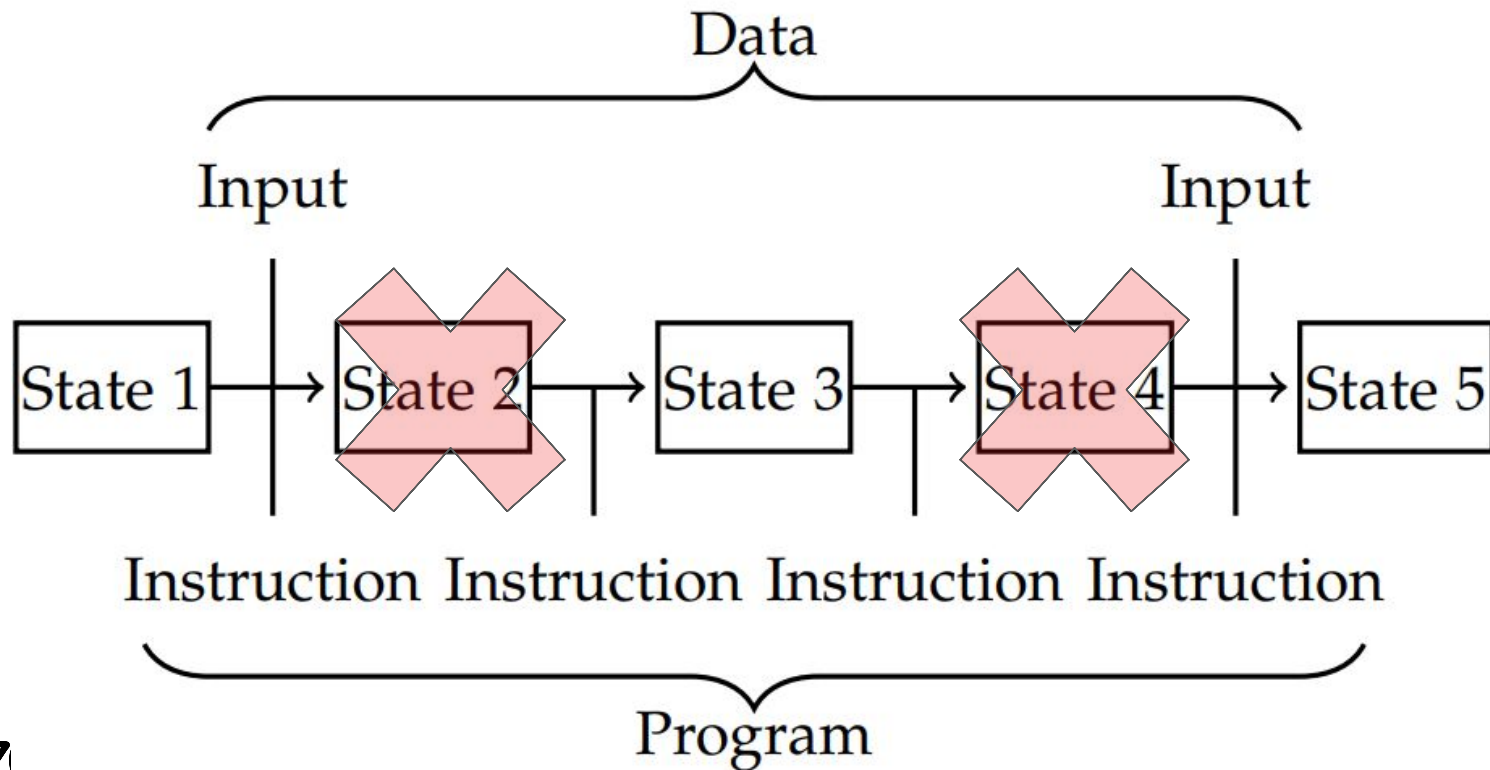
... is almost indistinguishable from programming.

Classical view of programming



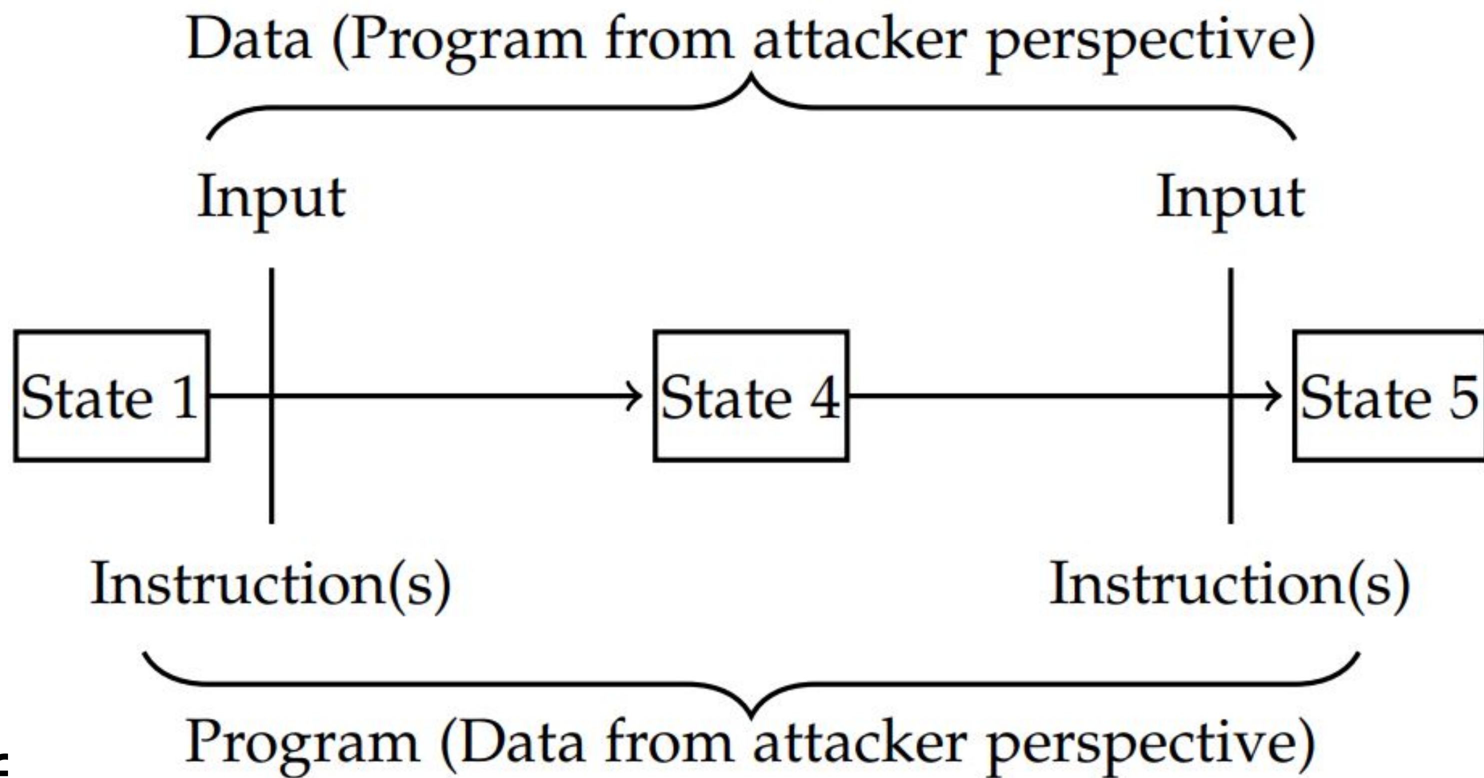
optimyz

Summarize non-observable states



optimyz

Attacker view of programming



optimyze

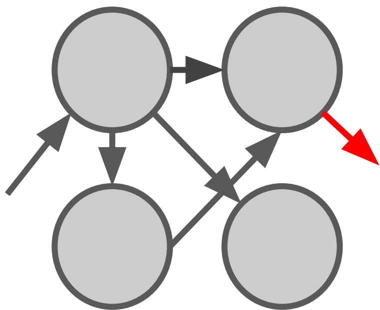
What is “exploitation” ?

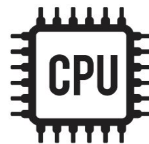
Programming an emergent computational device, the “weird machine”, to violate security properties.

optimyze

The weird machine

After entering a weird state, the emulated transitions will operate happily on it.





optimyze

The weird machine

There is now a new computational device: ***The weird machine.***

- Transforms states in Q_{cpu}^{weird} via emulated transitions designed to transform IFSM states.
- Takes the input stream of the IFSM as instruction stream
- $Q_{cpu}^{\text{sane}} \cup Q_{cpu}^{trans}$ are terminating states for the weird machine (because the IFSM resumes execution)

The weird machine: Properties

The weird machine has a number of interesting properties:

- IFSM Inputs as instruction stream
- Unknown state space (gets explored by exploit author)
- Unknown computational power (gets explored by exploit author)
- Emergent instruction set:
    - Results from interaction of the CPU, the program rho, and the IFSM
    - Often extremely unwieldly to program
    - Semantics need to be “discovered”

Exploitation

Exploitation is …

- the *setup* (choosing the right sane state),
- *initialization* (moving the emulated IFSM into an initial weird state),
- and *programming* of a weird machine

An “exploit” is such a triple that causes a violation of the security properties of the IFSM.

Can we prove anything?

Exploitability? Unexploitability?

A 1-slide summary of results followed by a long proof sketch.

Two implementations of the IFSM

1. One implementation that simulates the set “Memory” by using a flat linear array & sequential scanning.

2. One that simulates the set “Memory” by using two singly-linked lists and two list-heads.

The first one **can be proven to be unexploitable**, the second is provably **exploitable** in an attacker model where the attacker can flip 1 bit in memory.

How to read the proof?

- Not practical to prove any non-toy systems unexploitable
- The important part of the paper is the intuition we covered so far

The proof is in the paper for three reasons:

- As a ploy to get the paper published. Nobody publishes just a definition + intutition.
- Complexity works in favor of the attacker. What is the minimum complexity needed until things become exploitable?
- To illustrate that control-flow integrity is insufficient for security (and that people should not obsess about it)

optimyze

Proving (non-)exploitability: Attacker model

If we want to reason about exploitability, we need an attacker model. Many plausible models exist (see cryptographic community for a "zoo of attackers").

Simple attacker for this talk:

Single chosen bit-flip.

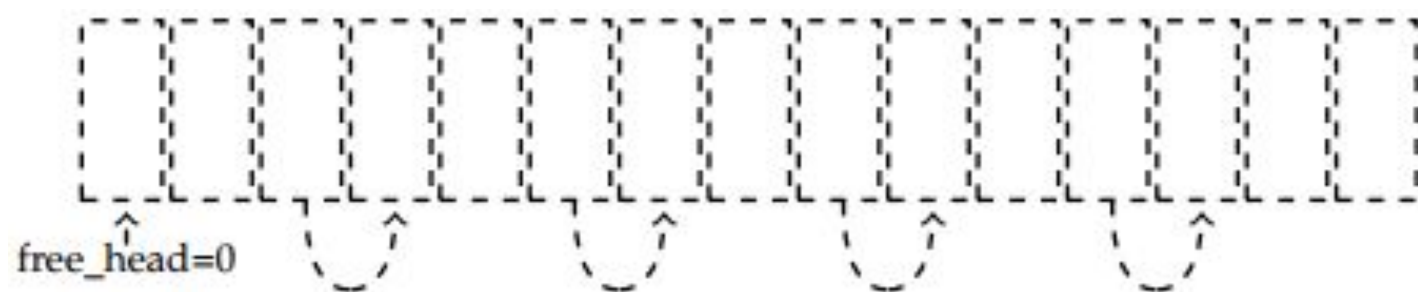
Formal attacker model in the security game

- Game flow:
  a. Attacker chooses a distribution over finite-state transducers that have as input alphabet the output alphabet of the IFSM, and that have as output alphabet the input alphabet of the IFSM
  b. Defender draws p, s uniformly at random from $bits_{32}$
  c. Attacker draws a finite-state transduce Θ_{exploit} from his distribution and connects it to the IFSM. The transducer is allowed to interact with the IFSM for n_{setup} steps
  d. The defender sends p,s to the IFSM
  e. The attacker is allowed to have his transducer interact with the IFSM for n_{exploit} steps. **At any step, but only once, he is allowed to flip an attacker-chosen bit in memory (not in registers).**

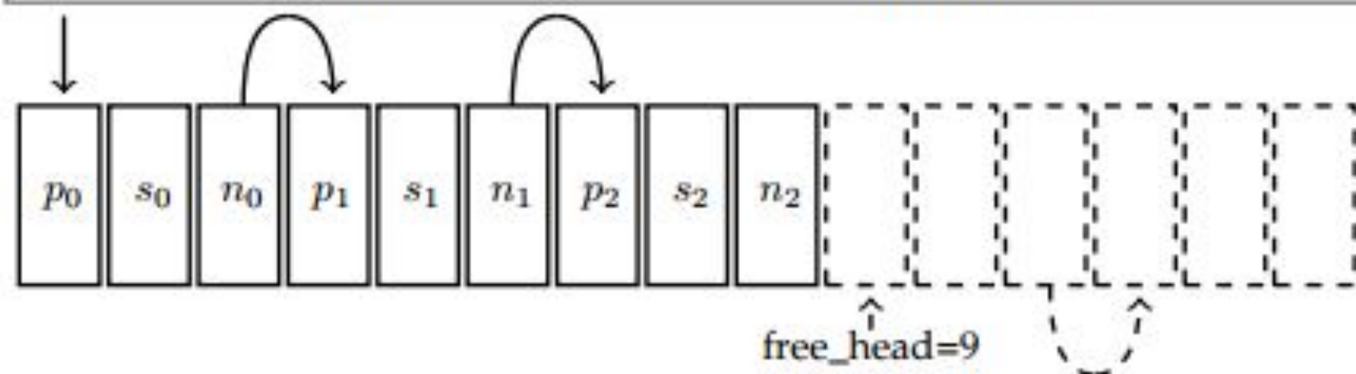
optimyze

Exploitability of the linked-list implemented IFSM

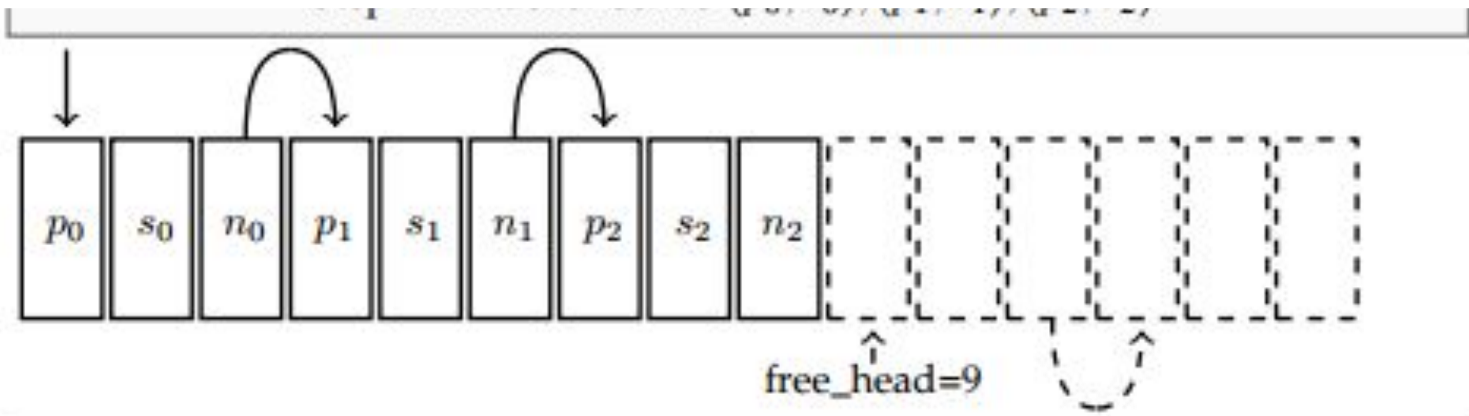
Exploitability of the IFSM implemented using two singly linked lists is shown by simply providing an exploit in the above game.



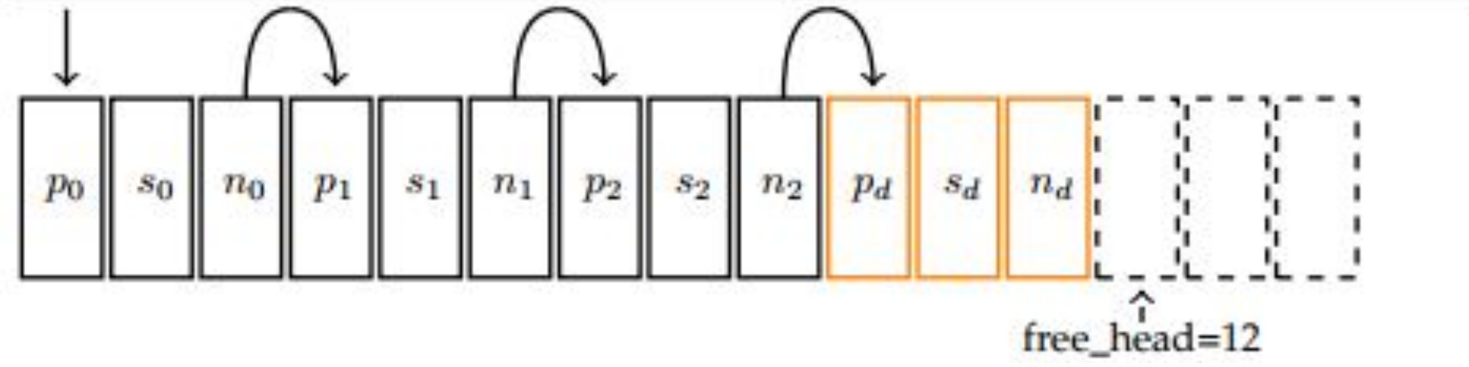
Step 1: Attacker sends $(p_0, s_0), (p_1, s_1), (p_2, s_2)$.



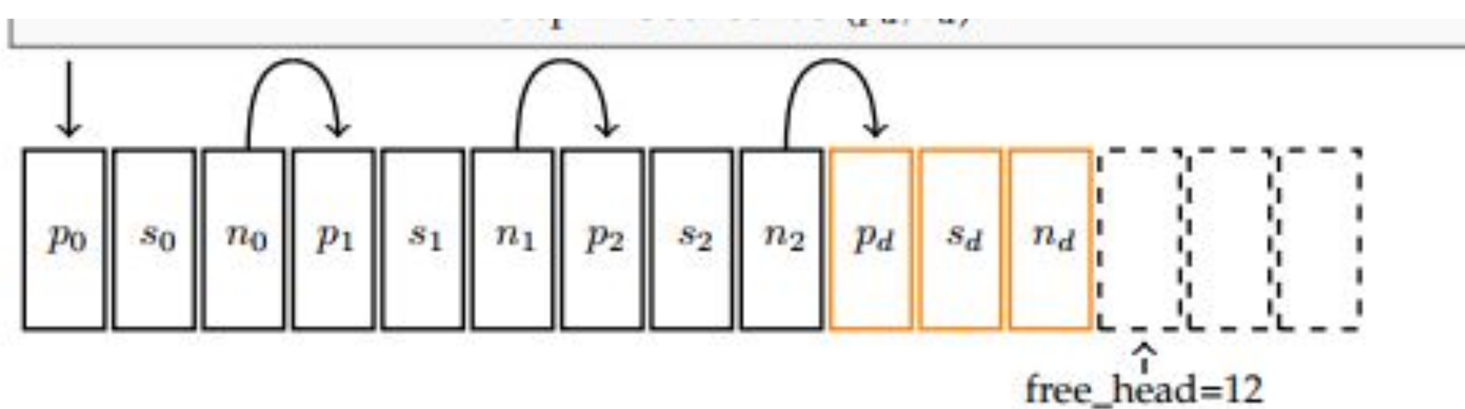
optim

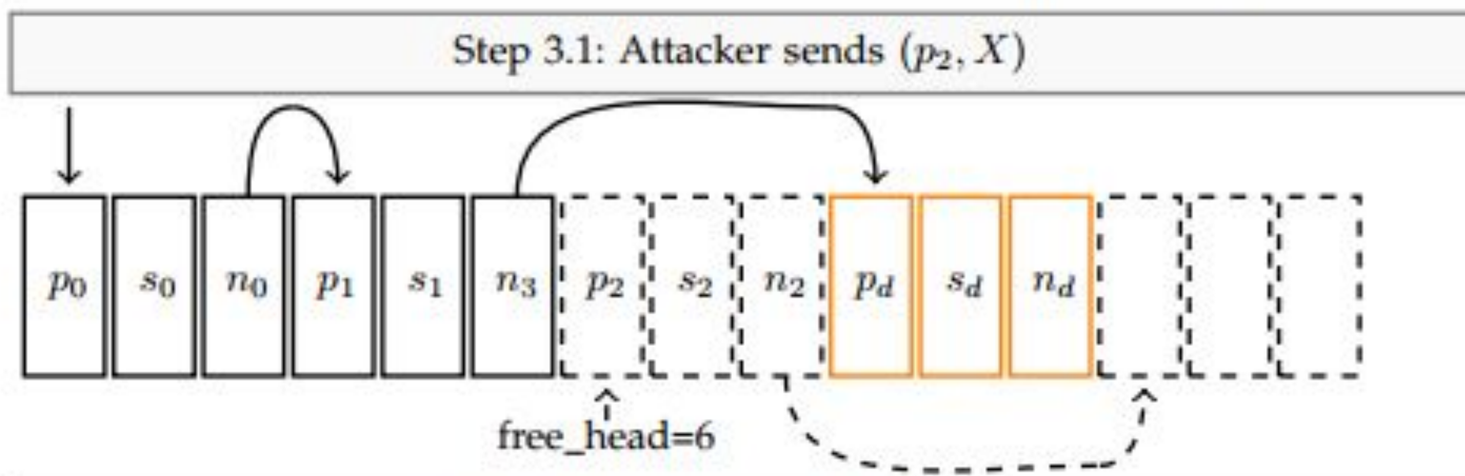


Step 2: User sends (p_d, s_d).

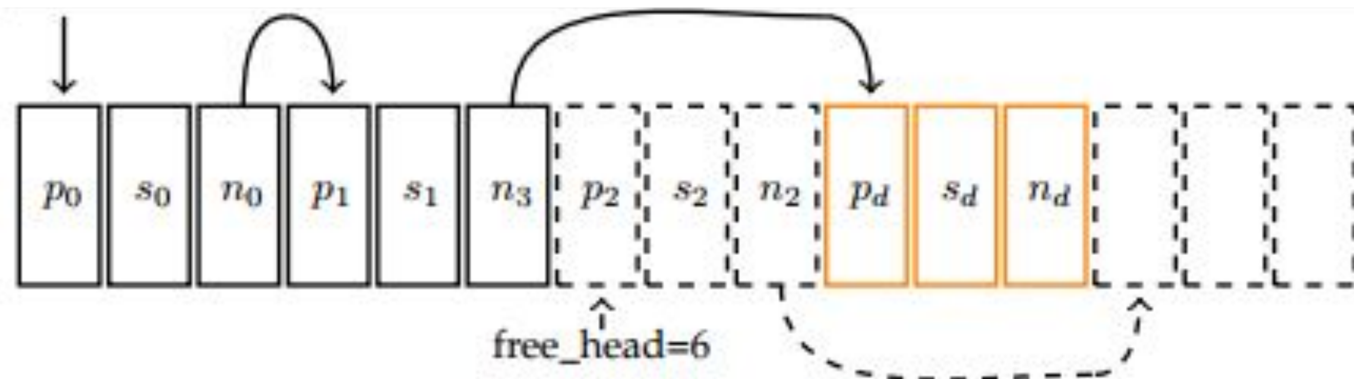


End of the initialization phase.
The attacker begins the main part of his attack.

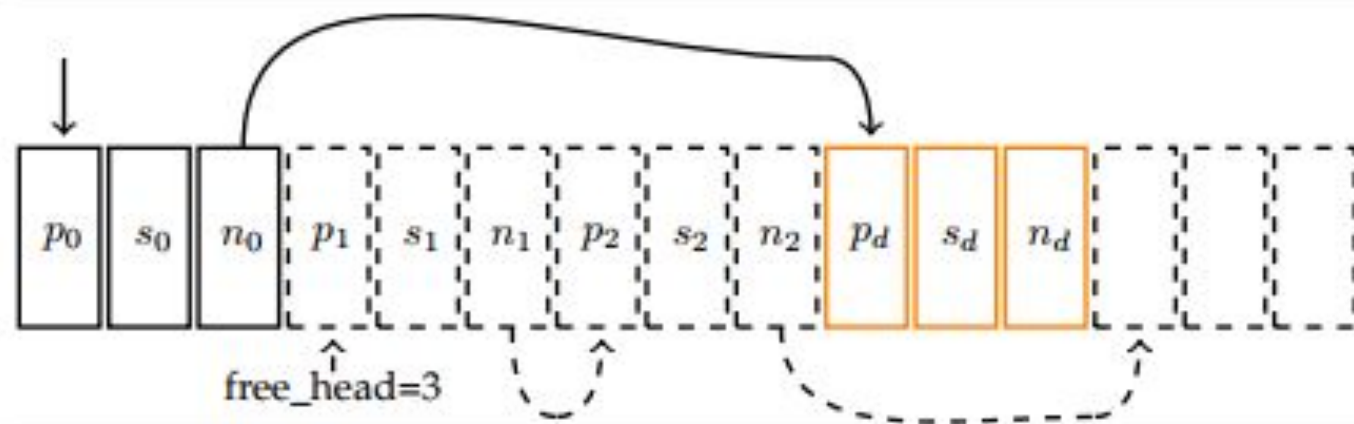




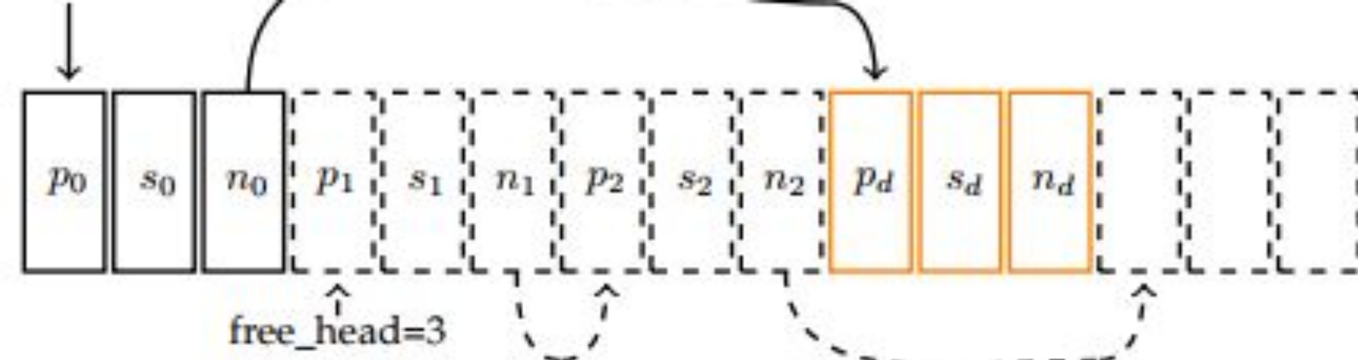
optim



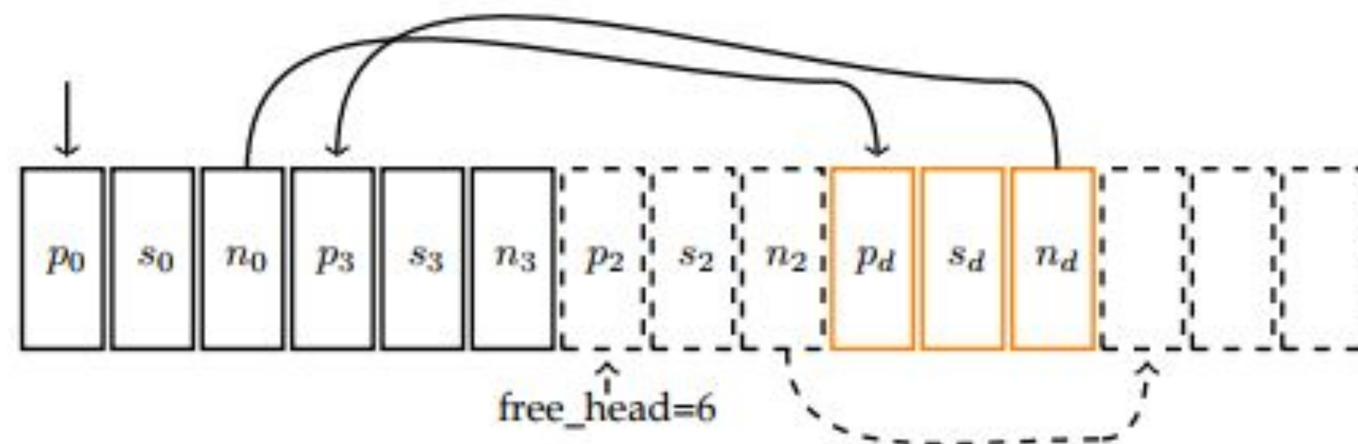
Step 3.2: Attacker sends (p_1, X)



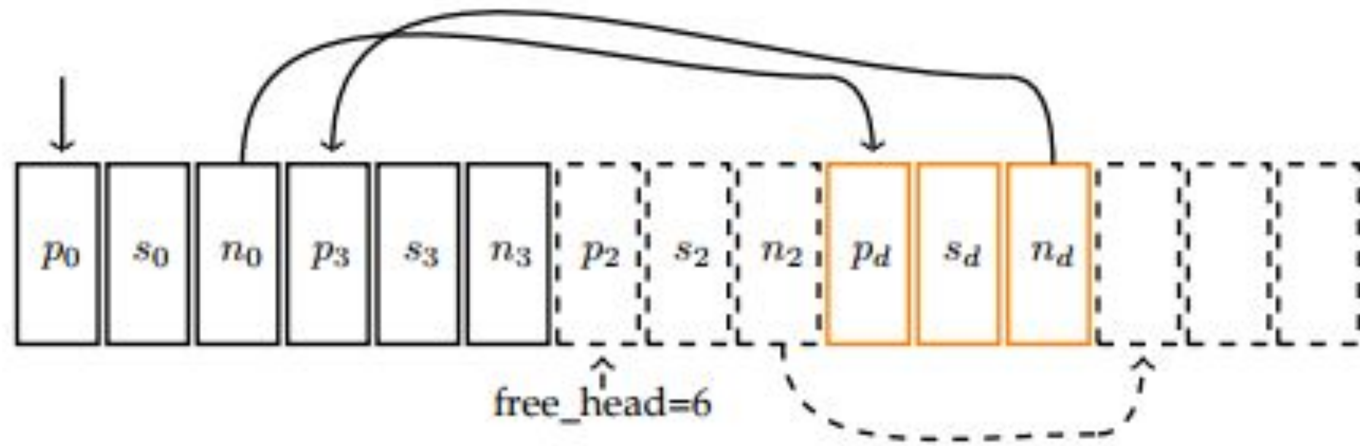
optim



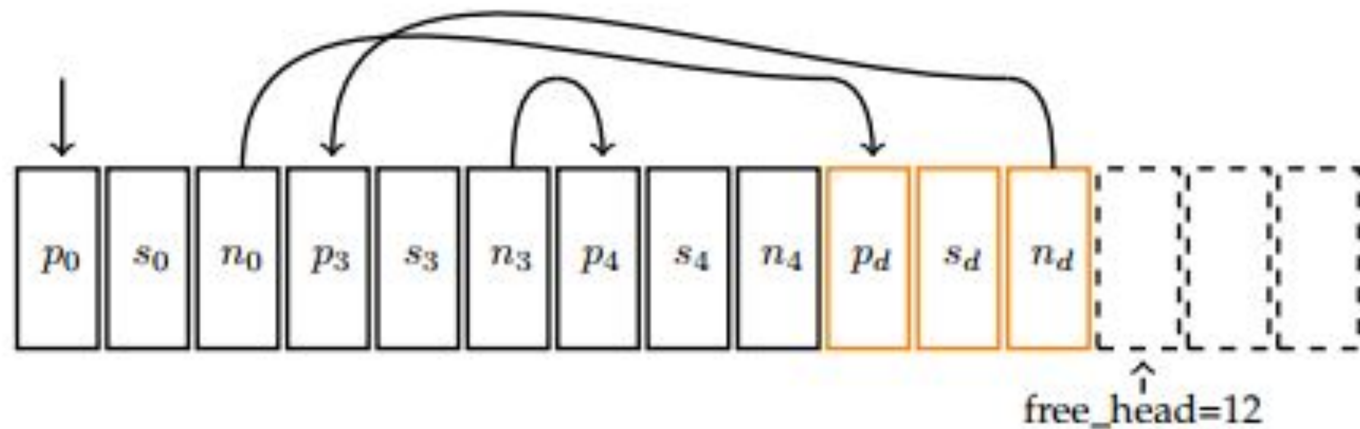
Step 3.3: Attacker sends (p_3, s_3)



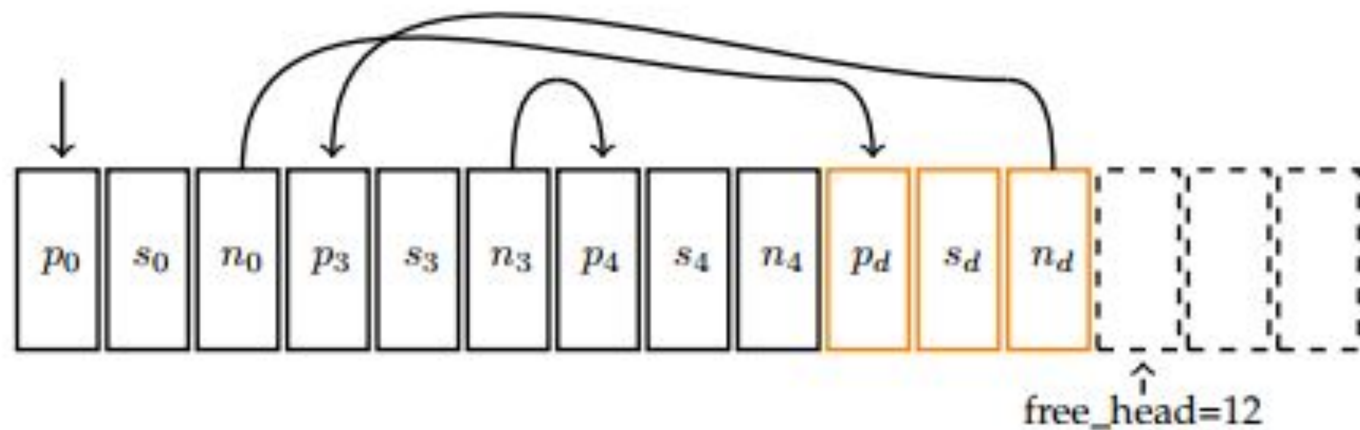
optim



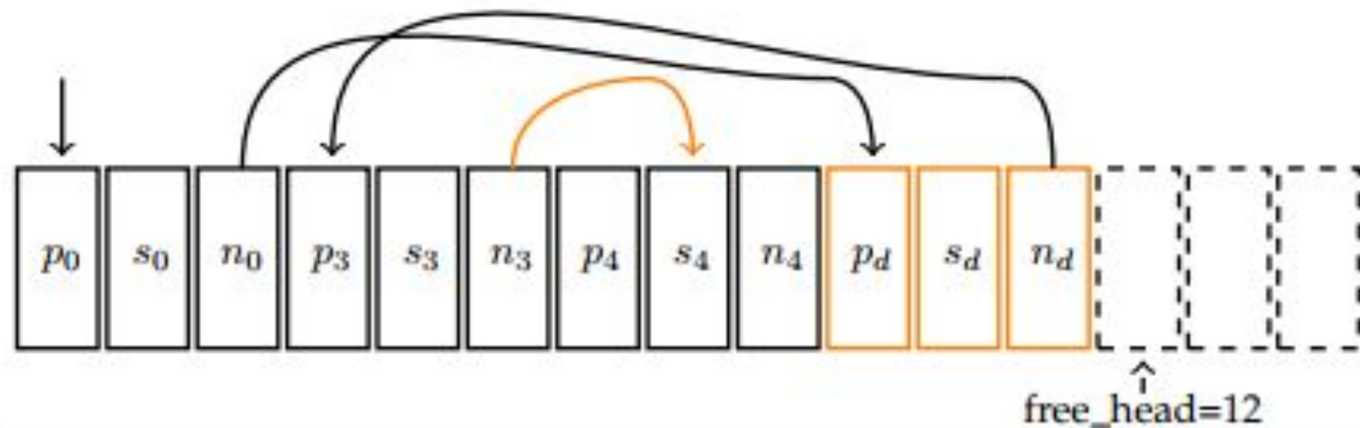
Step 3.4: Attacker sends (p_4, s_4)



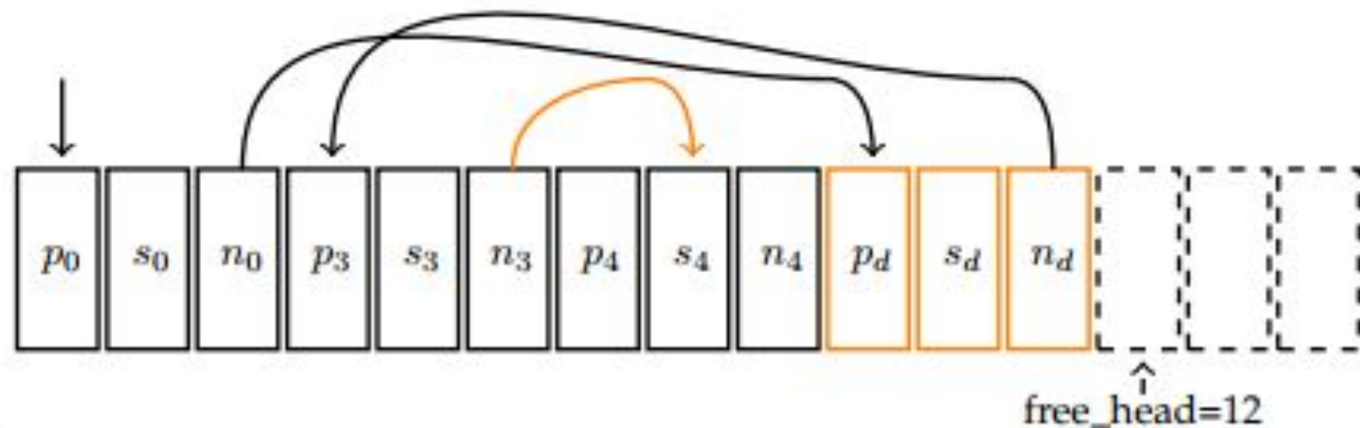
optimyze



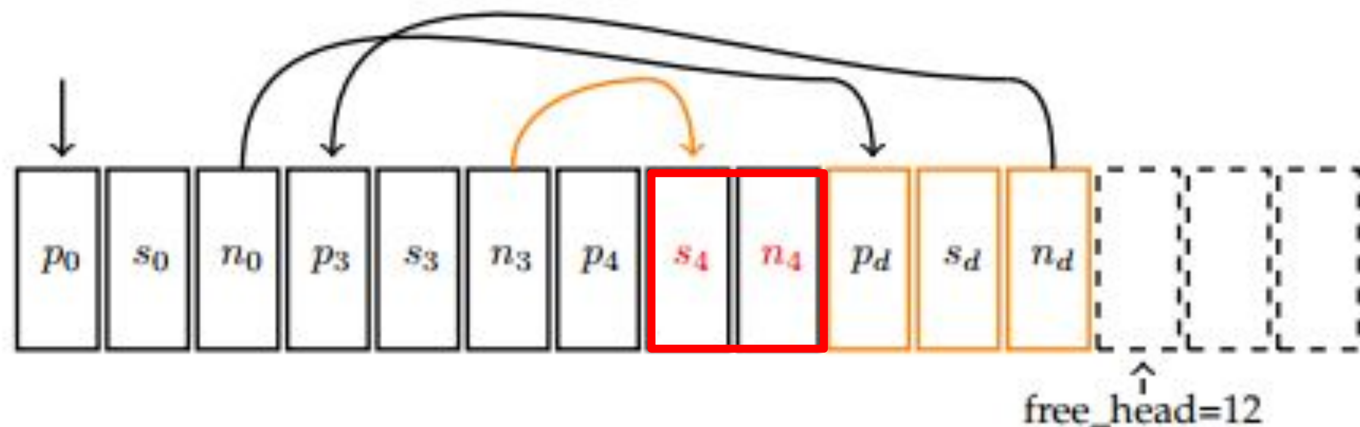
Step 4: The attacker gets to corrupt a single bit, and increments n_3.



optimyze



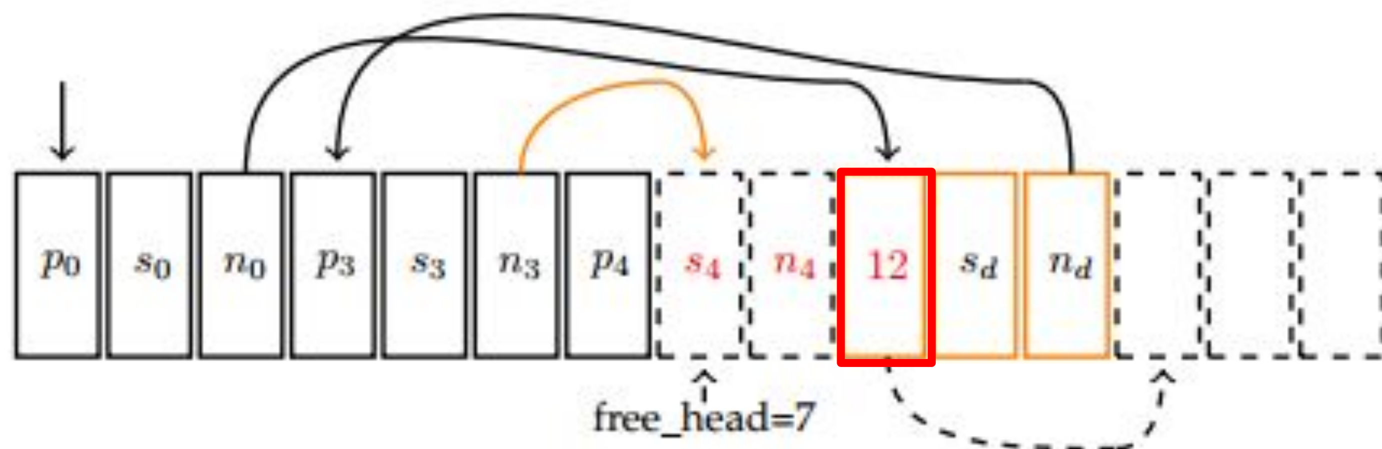
Step 5: The attacker sends (s_4, X). The machine follows the linked list, interprets s_4 as password, and outputs n_4.



optimyze

The machine then sets the three cells to be free,
and overwrites the storted p_d with free-head.

The machine just overwrote p_d with free-head.



Step 6: The attacker sends $(12, X)$ and obtains s_d.

optimyze

The linked-list implementation is exploitable …

… even by an attacker that just gets to corrupt one bit

… in spite of perfect control flow integrity

Proving non-exploitability

Proving non-exploitability has a different flavor.

Idea of proof:

- Any sequence of outputs o_{IFSM} that can be produced by an attacker **with** the ability to flip bits is a proper subsequence of the o_{IFSM} that can be produced by an attacker **without** the ability to flip bits in a maximum of 10000 extra interactions.

“An attacker does not gain any significant advantage through his ability to flip bits”

Assume that we have a bit-flipping exploit

Assume the bit-flipping attacker can provide a transducer so that

$$P[s \in o_{\mathrm{IFSM}}] > \frac{n_{\mathrm{setup}} + n_{\mathrm{exploit}}}{|bits_{31}|} = \frac{|o_{\mathrm{exploit}}|}{2^{31}}$$

and furthermore that for a non-bit-flipping attacker, the security property holds:

$$P[s \in o_{\mathrm{IFSM}}] \leq \frac{n_{\mathrm{setup}} + n_{\mathrm{exploit}}}{|bits_{32}|} = \frac{|o_{\mathrm{exploit}}|}{2^{32}}$$

Proof sketch from here on

1. Enumerate all possible transitions that an attacker can perform by flipping a single bit
   a. Result: Only 5 possible transitions:
      i. 3 are from sane state to sane state:
         1. (p, s) → (p XOR 2^i, s)
         2. (p, s) → (p, s XOR 2^i)
         3. (2^i, s) → (0, s)
      ii. 2 are from sane state to weird state:
         1. (p, s1), (p XOR 2^i, s2) → (p, s1), (p, s2)
         2. (p, 2^i) → (p, 0)
2. Show that any output sequence using 3 sane-to-sane transitions can be emulated by a maximum of 10000 extra operations without flipping a bit.
3. ....

optimyze

Proof sketch from here on

1. Enumerate transitions, 3 sane → sane, 2 sane → weird
2. Show the first 3 can be emulated without bitflips in less than 10000 steps
3. Show that any output sequence for the other 2 can be emulated in less than 10000 extra steps.
   a. Examine the sequence of inputs to the target and outputs of the target that can be produced by an attacker that uses one of the sane → weird transitions
   b. Construct a sequence of inputs to the target, which, without bitflips, produces a sequence of inputs/outputs so that the o_{IFSM} of the attacker with bitflips is a proper subsequence of the o_{IFSM} of the attacker without bitflips.

optimyze

Proof sketch from here on

1. Enumerate transitions, 3 sane → sane, 2 sane → weird
2. Show the first 3 can be emulated without bitflips in less than 10000 steps
3. Show that any output sequence for the other 2 can be emulated in less than 10000 extra steps.

So if we have an attacker **with** bitflips for whom $P[s \in o_{\text{IFSM}}] > \frac{|o_{\text{exploit}}|}{2^{31}}$, then we can build an attacker **without bitflips** for whom

$$P[s \in o_{\text{IFSM}}] > \frac{|o_{\text{exploit}}| + 10000}{2^{31}} > \frac{|o_{\text{exploit}}|}{2^{32}}$$

This contradicts our assumption that an attacker without bitflips cannot perform better than guessing.

optimyze

Proof details

Not very pretty.

Mostly rote exhaustion of the possible weird state transitions.

I would love to have an elegant proof, but I do not.

If you are interested, please dig into the paper (and make me aware of any flaws that need fixing)

Consequences …

… and implications.

optimyze

Consequences (I)

- Making statements about (non-)exploitability (or the lack thereof) is very hard
- Making statements about non-exploitability is **possible**, if not practical

Consequences (II)

- Multiple bugs in the same target will produce related weird machines
- While not identical, attackers can likely re-use fragments of a previous attack in a new attack
  a. Similar to “library functions” for the weird machine
  b. Already common in practice with Browser Javascript heap grooming libraries
- Impact on mitigations that randomize the programming environment (ASLR etc.) is unclear:
  a. Will the attacker be able to use one “weird machine code fragment” that he constructed to break a mitigation in a previous attack in a new attack?
  b. Which mitigations will be hard to bypass in repeated-attacks-against-same-software scenarios?

Consequences (III)

- Control-flow-integrity is no panacea. Our examples had perfect CFI.
- The extent to which CFI will make exploitation impossible is extremely unclear, and highly dependent on both the IFSM and the implementation.

- Registers, e.g. memory that was not accessible through indirection and in which the attacker can not flip a bit, play a crucial role.
- The data structure which allowed indirection also seems to cause trouble.
- Complexity seems to give advantage to the attacker (more transitions).

- RAM machines with indirect addressing have a different computational power than those that do not. Is the same difference at play here?

Consequences (IV)

- Exploitation is programming. Automated exploitation is a form of code synthesis.

- Contrary to real-world programs, attacks are successful even if they only work probabilistically.

Closing notes

- The single-bit-flip attacker model is not the best or most realistic attacker model.
- The example IFSM is tiny and not a realistic piece of software.

Both were not chosen because they reflect real software deployed today, but because they allowed a close examination of “where does the line between exploitability and non-exploitability lie”.

Relationships to other fields

What fields study related phenomena?

1. Model checking community: “Stuttering bisimulation”
2. The dynamical systems and cellular automata community: “Discrete Dynamical Systems, Cellular Automata”
3. Subset of the cellular automata community: “Emergent computation” and “Minimal Turing machines”

What intuitions can we take from these fields?

1. A computer is a big discrete dynamical system

2. Software keeps the system on well-defined trajectories - the IFSM corresponds to the states that we **wish** are the only reachable ones

3. Small deviation from an intended trajectories tends to blow up the set of reachable states

What intuitions can we take from these fields?

4. Attackers try to cause deviation from intended trajectory

5. Attackers then try to "drive" the dynamical system to an advantageous state

6. The goal of secure systems engineering: Make this harder.
    a. Option: Systems that revert to intended trajectories (see unexploitability proof)
    b. Option: Systems that detect deviant trajectories
    c. (Worst) option: Systems that "shake up" the trajectories

optimyze

Future hardware and language design

The roots of insecurity

- Almost all security issues have a de-synchronization between IFSM and concrete CPU state at their root
- “Weird machines” emerge, and end up being too powerful (in terms of reaching other states)
- Hardware does not have capability to detect “off-the-rails” execution

Where do we go from here?

- Preventing "going off the rails" is difficult: Hardware is not perfect, especially in hostile environments.
- Co-design of programming language and hardware.
- Ways of communicating IFSM invariants to the hardware.

Better Programming Languages

- Better programming language support for specifying and reasoning about IFSM states would be great.
- Synthesis of software from IFSM specification.
- Automated derivation of invariants for the IFSM.

Real-world example: Memory tagging (1)

- Each memory allocation generates N random bits and stores them in the returned pointer.
- Memory is “tagged” with this pointer (possibly stored inline in DRAM in special bits)
- Memory dereference validates: Is the pointer still pointing to memory with the right tag?

optimyze

Real-world example: Memory tagging (2)

- Enforced IFSM invariant: Pointers do not cross allocations.
- Note: Language constraint and programming convention that is enforced on the hardware layer.
- Much stronger systems can be imagined:
  - Fully typed memory?
  - Other constraints about pointer-access-patterns

Questions?

optimyze

Thank you.

Detailed talk with proof sketch:
https://vimeo.com/252868605

Full paper:
https://ieeexplore.ieee.org/document/8226852