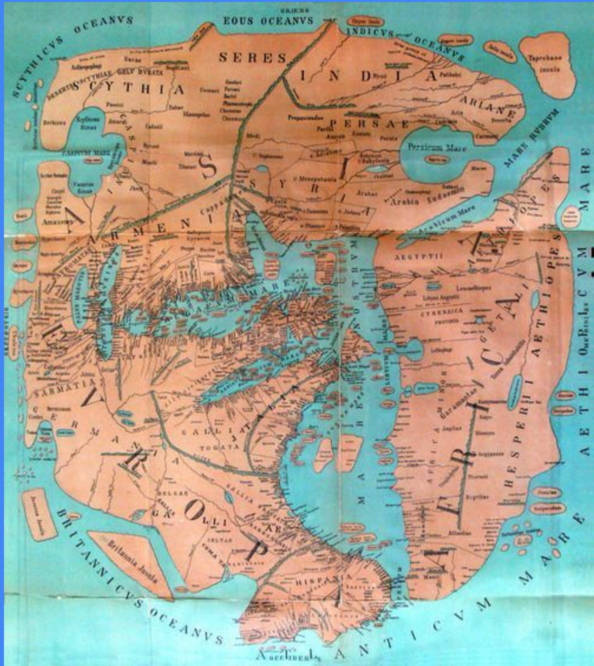
Proving un-exploitability of parsers

An imaginary roadmap for unknown territories

Thomas Dullien / “Halvar Flake”
Google Project Zero



Introduction

- Wrote my first “exploit” in 1998
- Trained as a mathematician (cryptography, computational commutative algebra); some background with abstract interpretation etc.
- Since 2009 or 2010 increasingly interested in fundamental questions - “what is an exploit” ? - necessary to formalize “folklore”
- Work on “exotic” exploits (Rowhammer, JS Bytecode corruption etc.)

Introduction

- During sabbatical 2015/2016 and after my return to P0 I wrote a paper about theoretical foundations of “exploitability” and “weird machines”
- “Weird machines, exploitability, and provable unexploitability” [Paper][Talk]
- Key results of the paper:
    - Formalisation of “what is an exploit”
    - Formalisation of intended machines & weird machines
    - Insight that exploitability is a mostly orthogonal concept to correctness
    - Non-exploitability can be proven in some extremely restricted cases

What comes next?

- Results in the paper are quite “weak”
- 60%+ of the paper is just introducing concepts, clarifying definitions, and “learning to walk” with those definitions
- Now that we have the machinery, and have made the first two wobbly steps, where do we want to go?

This talk

1. **Recap** of the key concepts from the paper
2. What were the **important tricks** that helped us prove non-exploitability in the restricted case?
3. What extra scaffolding would we need if we wanted to prove non-exploitability of something more complex - like a parser?

This talk

1. **Recap** of the key concepts from the paper

2. What w itability in
   the res

Highly speculative and likely incomplete and wrong.

3. What extra scaffolding would we need if we wanted to prove non-exploitability of something more complex - like a parser?

Recap: Key concepts from the paper

Intended finite state machine (or transducer)

IFSM

Software PROG as emulator to simulate IFSM on real CPU

Concretization mapping: IFSM state to set of possible CPU states that represent it.

Abstraction mapping: Partial mapping from CPU states to valid IFSM states.

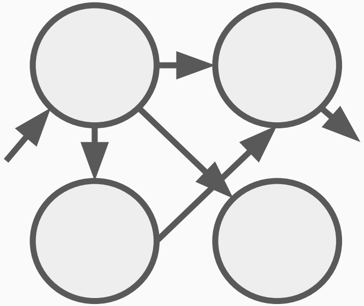
Sane, transitory, and weird states.

Security Properties as assertion over results of a game between ...

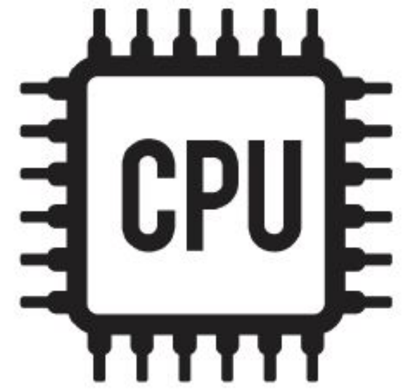
… two dueling transducers.

Weird machine programming

Intended Finite-State Machine (or transducer)



What I want



What I have

Recap: Key concepts from the paper

Intended finite state machine (or transducer)

IFSM

Software PROG as emulator to simulate IFSM on real CPU

Concretization mapping: IFSM state to set of possible CPU states that represent it.

Abstraction mapping: Partial mapping from CPU states to valid IFSM states.

Sane, transitory, and weird states.

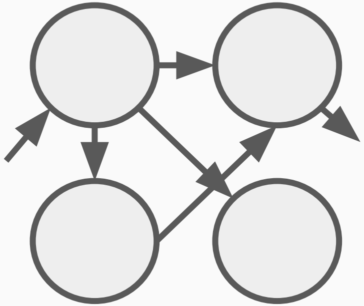
Security Properties as assertion over results of a game between

...

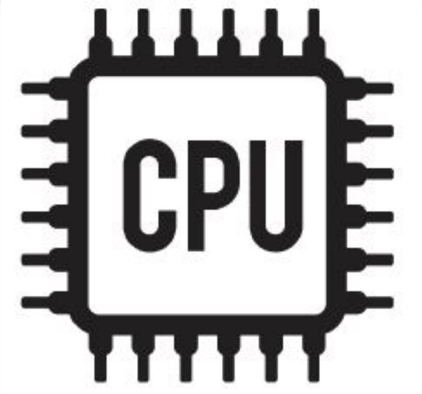
… two dueling transducers.

Weird machine programming

Instantiation mapping



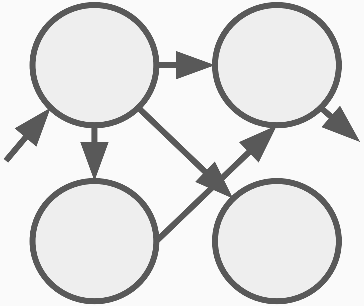
What I need



What I have

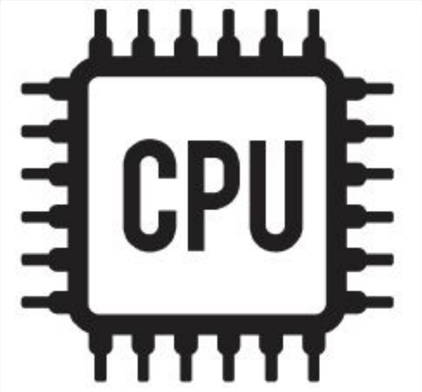
$$\gamma\theta, cpu, \rho : Q_\theta \rightarrow \mathfrak{P}(Q_{cpu})$$

Abstraction



What I need

partial



What I have

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

Recap: Key concepts from the paper

Intended finite state machine (or transducer)

IFSM

Software PROG as emulator to simulate IFSM on real CPU

Concretization mapping: IFSM state to set of possible CPU states that represent it.

Abstraction mapping: Partial mapping from CPU states to valid IFSM states.

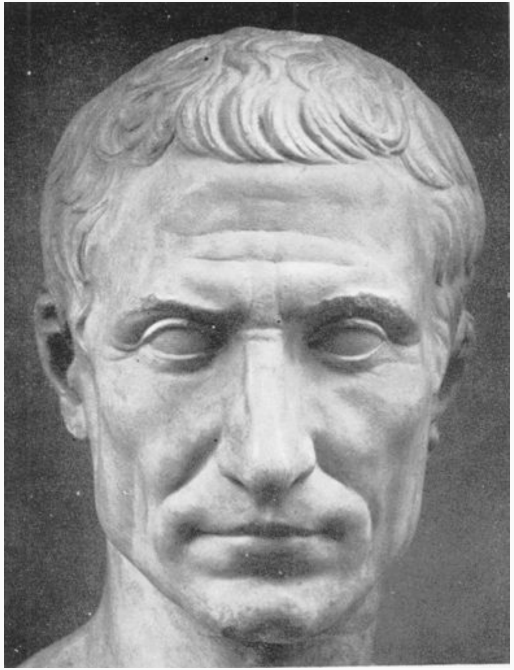
Sane, transitory, and weird states.

Security Properties as assertion over results of a game between ...

… two dueling transducers.

Weird machine programming

Sane, weird, and transitory states



CPU state space est omnis divisa in partes tres:

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

Recap: Key concepts from the paper

Intended finite state machine (or transducer)

IFSM

Software PROG as emulator to simulate IFSM on real CPU

Concretization mapping: IFSM state to set of possible CPU states that represent it.

Abstraction mapping: Partial mapping from CPU states to valid IFSM states.

Sane, transitory, and weird states.

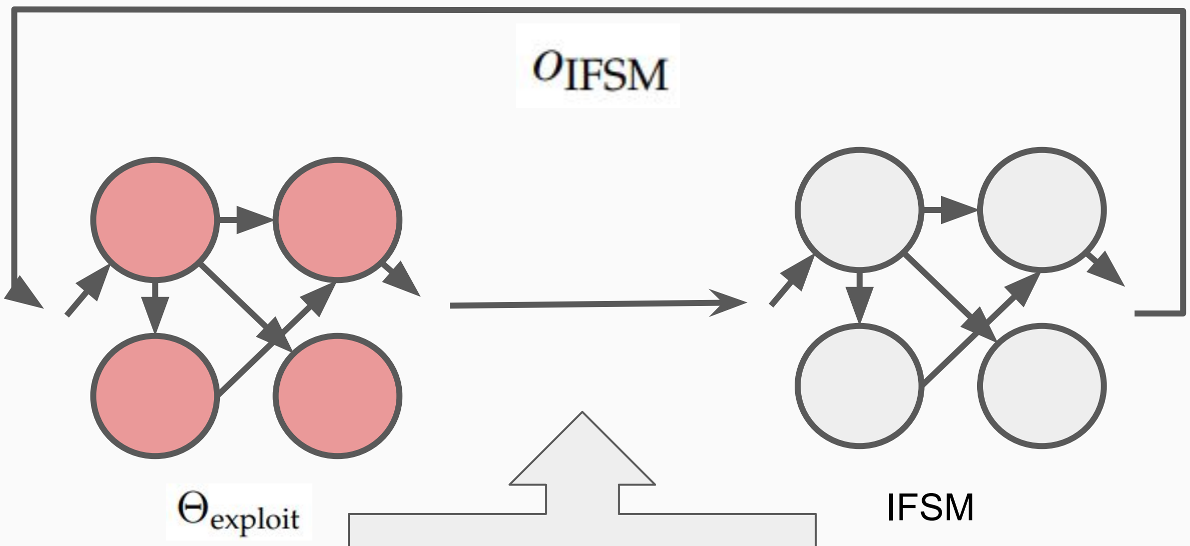
Security Properties as assertion over results of a game between ...

… two dueling transducers.

Weird machine programming

Dueling transducers

Output of the IFSM is sent to the exploit.



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

Security properties

- Define the game between the dueling transducers.
- Decide what you do not want to happen.
- Phrase this as statement about the communication between the transducers and the possible final states of the IFSM
- This “game structure” is adopted from security proofs in Cryptography

Recap: Key concepts from the paper

Intended finite state machine (or transducer)

IFSM

Software PROG as emulator to simulate IFSM on real CPU

Concretization mapping: IFSM state to set of possible CPU states that represent it.

Abstraction mapping: Partial mapping from CPU states to valid IFSM states.

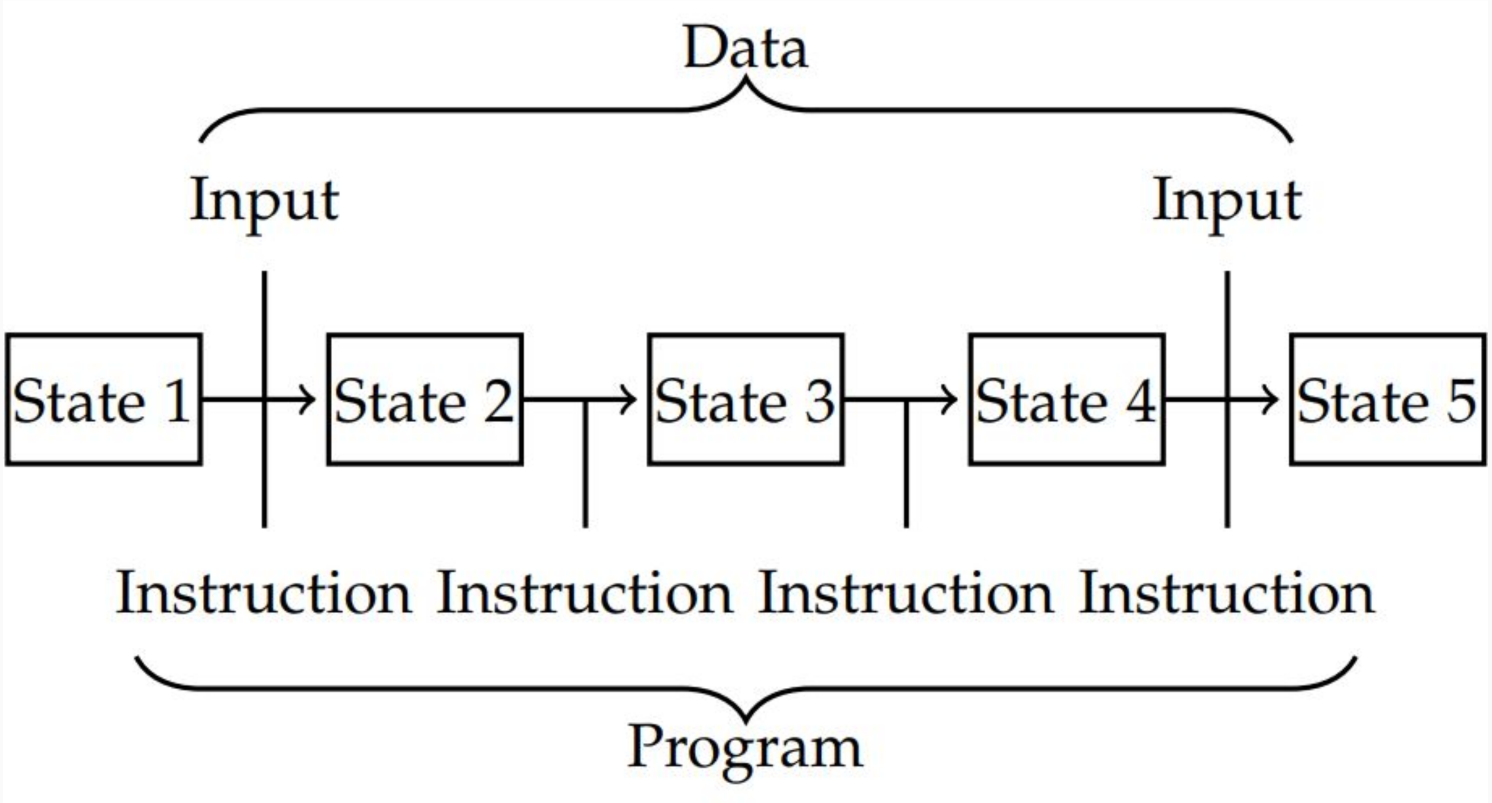
Sane, transitory, and weird states.

Security Properties as assertion over results of a game between …

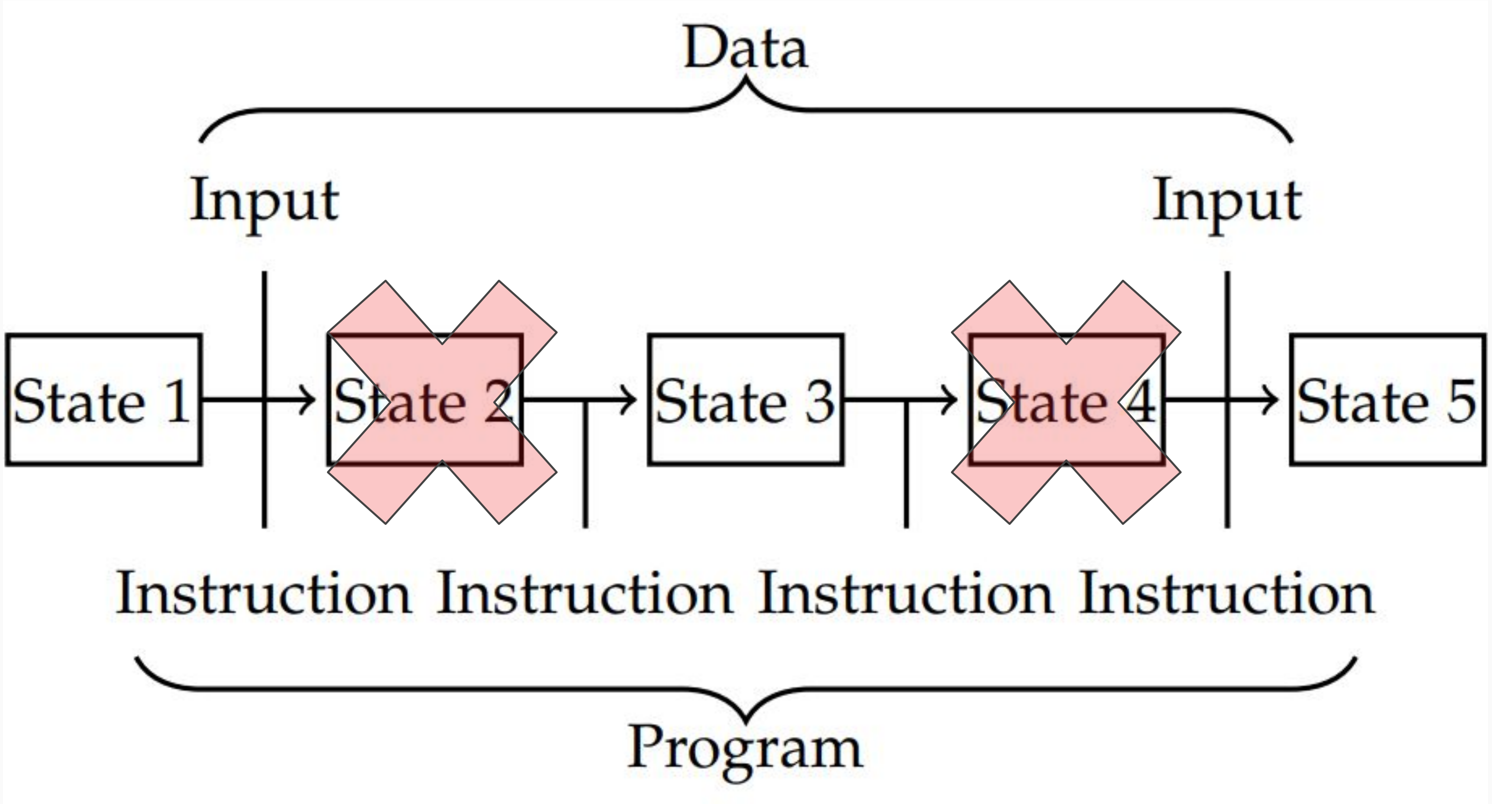
… two dueling transducers.

Weird machine programming

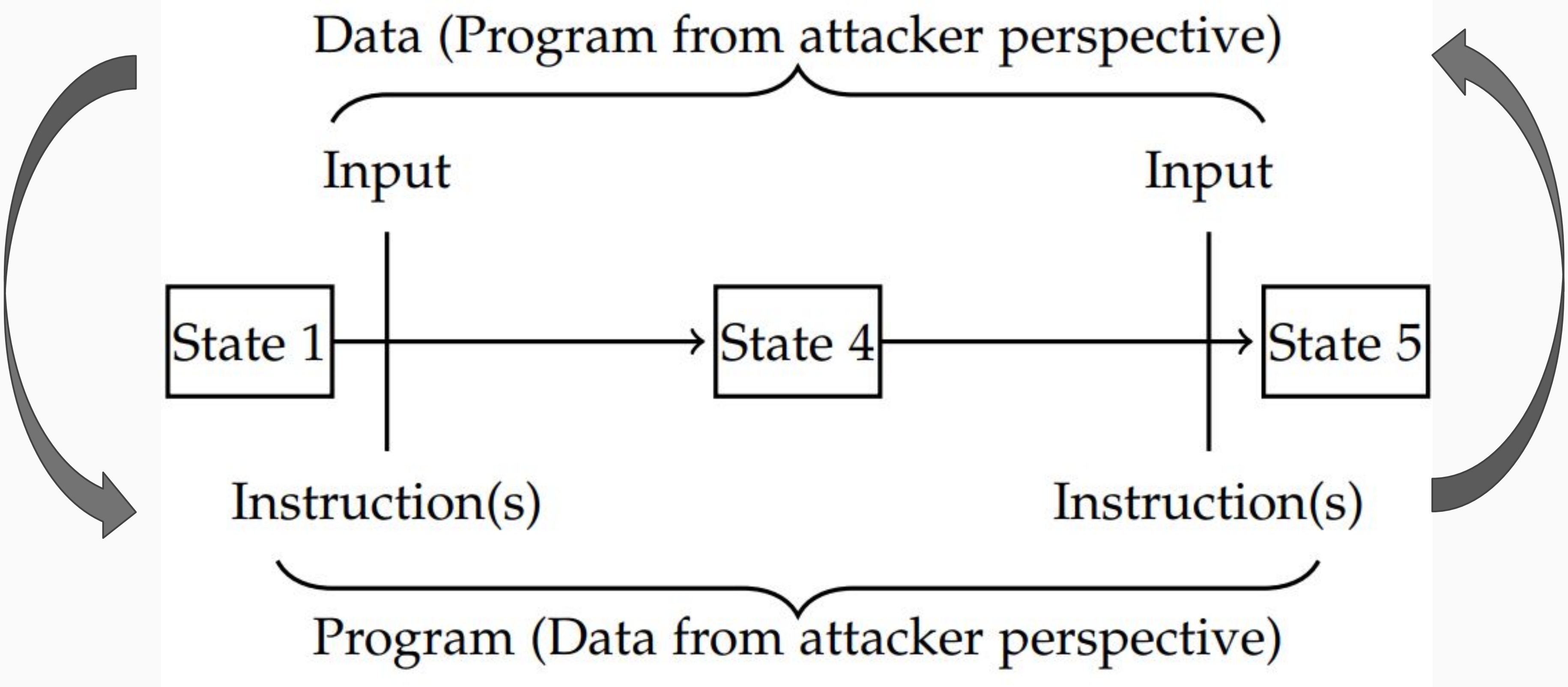
Classical view of programming



Classical view of programming



Attacker view of programming



Data (Program from attacker perspective)

Input

Input

State 1

State 4

State 5

Instruction(s)

Instruction(s)

Program (Data from attacker perspective)

Attacker view of programming

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)

Recap: Key concepts from the paper

Intended finite state machine (or transducer)

IFSM

Software PROG as emulator to simulate IFSM on real CPU

Concretization mapping: IFSM state to set of possible CPU states that represent it.

Abstraction mapping: Partial mapping from CPU states to valid IFSM states.

Sane, transitory, and weird states.

Security Properties as assertion over results of a game between …

… two dueling transducers.

Weird machine programming

Example in the paper: Secret-keeping machine

- Simple IFSM that keeps up to 5000 pairs of (password, secret).
- Attackers should not be able to retrieve a secret for which they do not know the password faster than guessing.
- Attacker model: Attacker is allowed to corrupt one chosen bit, exactly once.
- Two implementations: One linked-list based (exploitable), one based on flat arrays that are linearly traversed that can be proven “unexploitable” by that attacker.

Game in the paper:

- 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. **At any step, but only once, he is allowed to flip an attacker-chosen bit in memory (not in registers).**

Idea underlying the proof of non-exploitability

- Begin by showing (or assuming) that attacker without bit-flip cannot violate security properties (get secret much faster than guessing)
- Assume attacker with bit-flip **can** violate security properties (e.g. get secret much faster than guessing)
- Demonstrate that anything that can be achieved by the attacker with bit-flips **could also** be achieved by an attacker without bitflips with just a small overhead.
- **Contradiction**. This shows that an attacker with bit flips cannot get a significant advantage over an attacker without bit flip.

Proof sketch

- Cleverly summarize possible states of CPU/PROG into a few understandable equivalence classes.
- Show that attacker memory corruption can only lead to a few different equivalence classes of weird or sane states.
- Show that all sane -> sane transitions attacker can cause can be emulated by the weaker attacker.
- Show that all weird -> weird -> weird … transitions reach only a controlled number of equivalent states; show that any output could also be emulated by the weaker attacker.

Paper result ...

- For a very simple and limited IFSM ...
- … and a restricted, but also powerful memory-corrupting attacker ...
- … it is possible to prove unexploitability

What next?

… Sergey asked me ...

… “can you talk about how one could prove non-exploitability of parsers?”

Like asking someone who travelled twenty miles by feet “what is the best way to walk to India from here?”

Here be dragons.

Non-exploitable parsers

- A parser is a transducer that emits a program state at the end
- Any sane input language should lead to a formally-describable IFSM
- Safe compilation from IFSM-description to emulated IFSM is necessary
- This can, if done properly, yield a **correct** parser.

Non-exploitable parsers

- Exploitability is mostly orthogonal to correctness
- A correct program can be exploitable if an attacker has the means to enter a weird state (hardware fault etc.)
- An incorrect program gives the attacker means to enter weird states
- What would we need to build a compiler that can compile a spec of an IFSM to a **non-exploitable** implementation PROG ?

Ingredients needed

- Spec of the IFSM, specification of CPU
- Security Game
- Security Properties
- Attacker model for the weak and strong attacker

Security properties for parsers

- Parsers map input sequences to program states
- A good security property for parsers could be:

No attacker should be able to get the parser to emit an invalid state.

Recipe for proving non-exploitability ...

- Show that PROG (and/or IFSM) preserves the security property against the weak attacker. This should be comparatively “easy”.
- Show that all sane-to-sane transitions the the strong attacker can cause are either intended sane-to-sane transitions, or can be emulated easily by a weak attacker.
- Show that the strong attacker can only cause weird-weird transitions to a small number of equivalence classes of weird states, cannot produce output from the weird states, and when reverting back to a sane state only achieves a transition achievable by a weak attacker.

What would the compiler need to do?

The compiler will need to do the heavy lifting of ensuring that only a few, well-specified equivalence classes of states are reachable.

Controlling **sane** transitions

- **Controlling sane transitions:** Ensure that the attacker can only achieve benign sane-sane transitions. Can probably be done with clever design & layout of data structures in memory.

- Will be very dependent on precise semantics of CPU, and precise capabilities of the attacker

Controlling **weird** transitions

- Ensure that any program state that can be emitted using transitions through weird states can be emitted without those transitions.

- Easiest solution if computational cost is not an issue: Build code that can check whether CPU state is sane, run it before consuming a byte of input.

- Memory tagging is a much weaker, probabilistic variant of this.

- Sanity checks on data structure internals before operating on this are also weak, probabilistic variants of this.

Other possible avenues

- Validating CPU state is sane may be too expensive?
- Commonly done in some high-security embedded circuits (failure on invalid combination of state bits)
- Is doing this cheaply in software possible?
- Is there another way - perhaps “trapping” the attacker in a few harmless equivalence classes of weird states?

Closing words

- We are only slowly coming to grips with what “exploitation” means
- Computers are big recurrence equations that tend to exhibit deterministic chaos
- Security implies making sure that only few points in the state space are reachable, and that those points are well-understood
- Please take my speculation on “the way forward” with a rather huge grain of salt. Sergey Bratus made me do it.

Questions?