Weird machines, exploitability, and provable unexploitability

Thomas Dullien
thomas.dullien+wm@googlemail.com

ABSTRACT

The concept of *exploit* is central to computer security, particularly in the context of *memory corruptions*. Yet, in spite of the centrality of the concept and voluminous descriptions of various exploitation techniques or countermeasures, a good theoretical framework for describing and reasoning about exploitation has not yet been put forward.

A body of concepts and folk theorems exists in the community of exploitation practitioners; unfortunately, these concepts are rarely written down or made sufficiently precise for people outside of this community to benefit from them.

This paper clarifies a number of these concepts, provides a clear definition of exploit, a clear definition of the concept of a *weird machine*, and how programming of a weird machine leads to exploitation. The papers also shows, somewhat counterintuitively, that it is feasible to design some software in a way that even powerful attackers - with the ability to corrupt memory once - cannot gain an advantage.

KEYWORDS

exploitation, memory corruptions

PROBLEM DESCRIPTION

The concept of *exploit* is central in computer security. While intuitively clear, it has not been formalized - even in the very restricted setting of *memory-corruption* attacks. Two largely disjoint communities have worked on exploring the process of exploitation: 'Exploit practitioners' (EPs) with focus on building working exploits have been investigating the topic at least since the infamous Morris worm, and academic researchers, who really began focusing on the problem with the re-invention and popularization of return-oriented programming (ROP) by Shacham et al [20].

The aversion of the EP-community to formal publishing[1] has lead to an accumulation of folklore knowledge within that community which is not properly communicated to a wider audience; this, unfortunately, often leads to duplicated effort, re-invention, and sometimes even acrimony between members of the two communities [11].

The concept of a *weird machine* is informally familiar in the EP community, but widely misunderstood outside of that community. It has numerous implications, most importantly:

- The complexity of the attacked program works in favor of the attacker.
- Given enough time for the preparation of an exploit, non-exploitability is the exception, not the rule. Even extremely restricted programs consisting of little more than a linked list with standard operations allow an attacker sufficient degrees of freedom.
- Questions of 'exploitability' are often decoupled from issues of control-flow or compromising the instruction pointer of a target: Control flow integrity is just one security property that can be violated, and perfect CFI does not imply security. Attackers aim to violate CFI because it provides the most convenient and powerful avenue to violate security properties, not because it provides the only such avenue.
- If exploitability is a result of target's complexity, the boundary where complexity causes exploitability is much lower than commonly appreciated.

The misunderstandings surrounding *weird machines* are particularly unfortunate as the framework of *weird machines* subsumes many individual techniques; the framework predicts that many exploitation countermeasures are overly specific and bound to be bypassable. Among other things, the bypassing of most early ROP countermeasures could have been easily predicted without the ensuing series of tit-for-tat papers, and the somewhat limited effectiveness of control-flow-integrity (CFI) [1, 9, 22] against many attacks such as counterfeit-object-oriented programming (COOP) [16] as well as the existence of data-oriented-programming (DOP) [12] would have come as less of a surprise.

Contributions. This paper provides the following contributions:

(1) Proper definitions and formalizations of the 'folk theorems' of the EP community.
(2) A clear definition of 'exploit' which better matches real-world requirements than the popular approach of showing Turing-completeness of emergent computation.
(3) A first step toward understanding what distinguishes unexploitable from exploitable programs. The paper presents two implementations of the same program. An attacker with the ability to flip a bit of his choosing can exploit one variant, while the other can be shown to be immune even to this powerful attacker. The differences between the programs hint at differences in computational power obtained by the attacker - which depend on the choice of data structures for the program implementation.

The intuitions behind and implications of 1 and 2 are common knowledge, all formalisms, definitions and proofs, as well as 3, are contributions of the author(s).

We hope that this paper bridges the gap between the two communities and provides a common vocabulary.

RELATED WORK

The concept of a *weird machine* that will be discussed in this paper has found numerous mentions over the years; not all these mentions refer to the same concept. [8] discussed *weird machines* but only provided an informal description, not a definition. The Langsec

[1] Or worse, the incentive structure that keeps the EP community from publishing at all.

Conference'17, July 2017, Washington, DC, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

community uses the term in their literature, often informally and vaguely defined, and with slightly varying meanings. [2–4]

[24] discusses weird machines in proof-carrying code (PCC) that arise when the PCC system fails to capture all necessary and sufficient conditions for safe program execution; Contrary to the present paper, he focuses on computations involving unexpected control flow and the proof-carrying-code scenario. This paper is inspired heavily by the use of 'dueling' finite-state transducers in that paper, though.

Computation (and correctness) in the presence of faults has been studied in [25] , which introduces a lambda-calculus to calculate correctly given hardware faults. [14] studies automatic detection of two classes of heap corruptions in running code by keeping multiple copies of a randomized heap.

By and large, while many academic and non-academic papers have studied concrete exploitation instances, few have considered foundational questions.

We will see later that *weird machines* arise when an abstract, intended machine and a concrete implementation which tries to simulate the abstract machine fails to do so. Studying equivalence between automata which simulate each other at different levels of abstraction has been studied by the model-checking and verification community using stuttering bisimulation extensively. [5, 10, 23] This paper eschews the somewhat specialized language of stuttering bisimulation to allow broader accessibility.

OVERVIEW OF THE PAPER

In order to get to the important results of the paper, a fair bit of set-up and definitions are needed. The paper first defines 'the software the developer intended to write' and a simple computing environment for which this software is written. This is followed by further definitions that permit describing erroneous states and distinguishing between erroneous states with and without security implications. Finally, a precise definition of exploit and *weird machine* is provided.

A running example is used throughout these sections. Two implementations of the same software are introduced, along with a theoretical attacker. We prove that one implementation cannot be exploited while the other implementation can, and discuss the underlying reasons.

Finally, we discuss the implications for exploit mitigations, control-flow integrity, and software security.

1 THE INTENDED FINITE-STATE MACHINE (IFSM)

The design of any real software can be described as a potentially very large and only implicitly specified finite state machine (or transducer, if output is possible)[2]. This FSM transitions between individual states according to inputs, and outputs data when necessary. Since any real software needs to run on a finite-memory computing device, the nonequivalence of a FSM to a Turing machine does not matter - *any* real, finite-input software can be modelled as a FSM (or FST) given a sufficiently large state set.

[2]The bisimulation community uses the concept of *process*, which is similar - but more readers will be familiar with FSMs, and they serve our purpose well enough

For simplicity, we will use the notation IFSM in the rest of the paper even when the machine under discussion is a transducer.

For situations when an IFSM needs to be specified formally, recall that a finite-state transducer can be described by the 7-tuple $\theta = (Q, i, F, \Sigma, \Delta, \delta, \sigma)$ that consists of the set of states Q, the initial state i, the final states F, input- and output alphabets Σ and Δ, a state transition function $\delta : Q \times \Sigma \rightarrow Q$ and the output function σ which maps $Q \times \Sigma \rightarrow \Delta$.

1.1 Software as emulators for the IFSM

Since any real-world software can be modelled as an IFSM, but has to execute on a real-world general-purpose machine, an emulator for the IFSM needs to be constructed. This process is normally done by humans and called *programming* or *development*, but can be done automatically in the rare case that the IFSM is formally specified.

Why consider software as emulator for the IFSM instead of examining software as the primary object of study? The answer lies in the very definition of *bug* or *security vulnerability*: When the security issue arises from a software flaw (in contrast to a hardware problem such as [13]) , it is impossible to even define 'flaw' without taking into account what a bug-free version of the software would have been. Viewing the software as a (potentially faulty) emulator for the IFSM allows the exploration of how software faults lead to significantly larger (in the state-space sense) emulated machines.

1.2 Example IFSM: A tiny secure message-passing server

We introduce an example IFSM with the properties of being small, having a clearly-defined security boundary, and allowing for enough complexity to be interesting. We describe the IFSM informally first and subsequently give a formal example.

Informally, our example IFSM is a machine that remembers a password-secret pair for later retrieval through re-submission of the right password; retrieval removes the password-secret pair. We set an arbitrary limit that the system need not remember more than 5000 password-secret pairs.

A diagram sketching the IFSM is shown on page 3 in Figure 1.

To transform this sketch into a formally defined FSM, we replace the memory of the described machine with explicit states. We denote the set of possible configurations of *Memory* with $\mathcal{M}$:

$$\mathcal{M} := \left\{ \begin{array}{l} \emptyset, \\ \{(p_1, s_1)\}, \\ \ldots, \\ \{(p_1, s_1), \ldots, (p_{5000}, s_{5000})\} \end{array} \middle| \begin{array}{l} \\ p_i, s_i \in bits_{32} \backslash \{0\} \\ p_i \neq p_j \end{array} \right\}$$

The central looping state A in the informal diagram can be replaced by a family of states A_M indexed by a memory configuration $M \in \mathcal{M}$. The starting configuration transitions into $A_\emptyset$, and after reading (p, s), the machine transitions into $A_{\{(p,s)\}}$ and so forth. With the properly adjusted transitions, it is now clear that we have a proper FST (albeit with a large number of individual states).

The formal specification of the example IFSM in the 7-tuple form $\theta = (Q, i, F, \Sigma, \Delta, \delta, \sigma)$ is as follows:

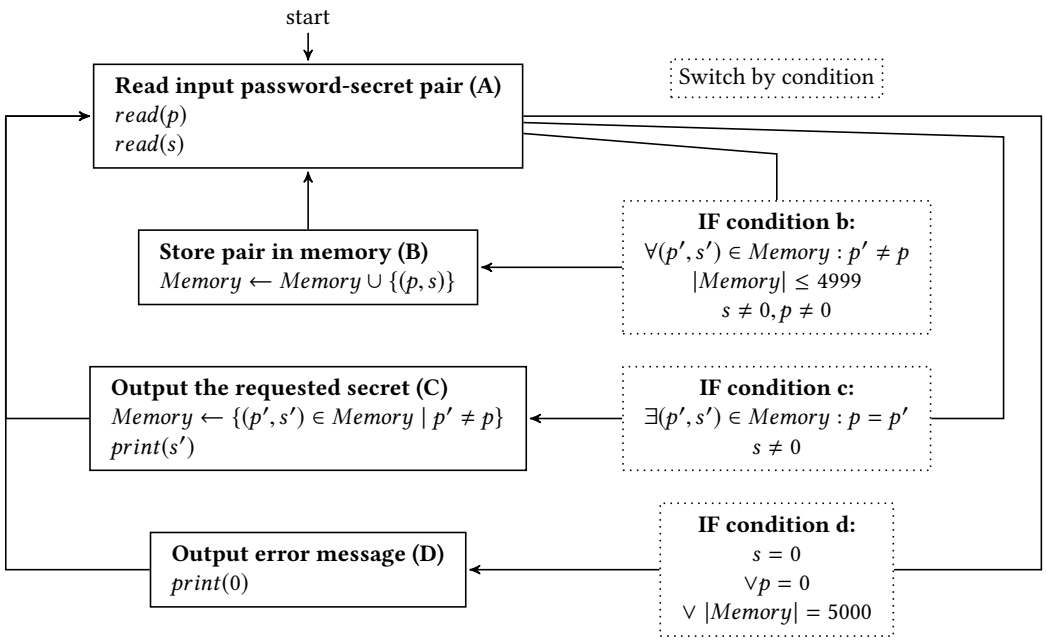


Figure 1: A diagrammatic sketch of the example IFSM

$$Q := \{A_M, M \in \mathcal{M}\}, \qquad i := A_\emptyset, F := \emptyset$$
$$\Sigma := \{(p,s) | p, s \in bits_{32}\}, \qquad \Delta := \{s \in bits_{32}\}$$
$$\delta := A_M \times (p,s) \rightarrow \begin{cases} A_{M \cup (p,s)} \text{ if } & (p,s) \notin M \\ & \wedge |M| \leq 4999 \\ & \wedge s \neq 0 \\ A_{M \setminus (p,s)} \text{ if } (p,s) \in M \\ A_M \text{ otherwise} \end{cases}$$
$$\sigma := A_M \times (p,s) \rightarrow \begin{cases} s' \text{ if } (p,s') \in M \\ 0 \text{ if } s = 0 \vee |M| = 5000 \end{cases}$$

1.3 Security properties of the IFSM

Not every malfunction of a program has security implications. To distinguish between plain erroneous states and erroneous states that have security implications, security properties of the IFSM need to be defined.

Security properties are statements (possibly about probabilities) over sequences of states, inputs, and outputs of the IFSM. They are part of the specification of the IFSM. Not every true statement is a security property, but every security property is a true statement.

The attackers goal is always to violate a security property of the IFSM when interacting with the emulator for the IFSM.

1.3.1 Security properties of the example IFSM. The example IFSM should satisfy the informal notion that "you need to know (or guess) the right password in order to obtain a stored secret".

Intuitively, the attacker should not be able to 'cheat' - there should be no way for the attacker to somehow get better-than-guessing odds to obtain the stored secret from the IFSM.

In order to make this precise, we borrow ideas from the cryptographic community, and define a multi-step game where an attacker and a defender get to take turns interacting with the machine, and we specify that there is no way that the attacker can gain an advantage.

The game mechanics are as follows:

(1) The attacker chooses a probability distribution $\mathcal{A}$ over finite-state transducers Θ_{exploit} that have an input alphabet $\Sigma_{\Theta_{\text{exploit}}} = \Delta$ and output alphabet $\Delta_{\Theta_{\text{exploit}}} = \Sigma$. This means that the attacker specifies one or more finite-state transducers that take as input the outputs of the IFSM, and output words that are the input for the IFSM.
(2) Once this is done, the defender draws two elements p, s from $bits_{32}$ according to the uniform distribution.
(3) The attacker draws a finite-state transducer from his distribution and is allowed to have it interact with the IFSM for an attacker-chosen number of steps n_{setup}.
(4) The defender sends his (p, s) to the IFSM.
(5) The attacker gets to have his Θ_{exploit} interact with the IFSM for a further attacker-chosen number of steps n_{exploit}.

The probability for Θ_{exploit} to obtain the defenders secret should be no better than guessing. Let o_{exploit} be the sequence of outputs that the $\Theta_{exploit}$ produced, and o_{IFSM} the sequence of outputs the IFSM produced during the game. Then our desired security property 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 here is given a random draw from the attacker-specified distribution over transducers. This encodes our desired property: An attacker cannot do better than randomly guessing the password, and the attacker cannot provide a program that does any better.

2 A TOY COMPUTING ENVIRONMENT

The IFSM itself is a theoretical construct. In order to 'run' the IFSM, a programmer needs to build an emulator for the IFSM, and this emulator needs to be built for a different, general-purpose computing environment, which will be introduced next.

For our investigation, the Cook-and-Reckhow [6] RAM machine model is well-suited. Their machine model covers both random-access-machine variants (Harvard architectures) and random-access-stored-program variants (for von-Neumann-Architectures); our discussion applies equally to both, but our concrete example assumes a Harvard architecture.

The machine model consists of a number of registers as well as the following operations:

LOAD(C, r_d)	$: r_d \leftarrow C$	Load a constant
ADD(r_{s_1}, r_{s_2}, r_d)	$: r_d \leftarrow r_{s_1} + r_{s_2}$	Add two registers or a register and constant
SUB(r_{s_1}, r_{s_2}, r_d)	$: r_d \leftarrow r_{s_1} - r_{s_2}$	Subtract two registers or a register and constant
ICOPY(r_p, r_d)	$: r_d \leftarrow r_{r_p}$	Indirect memory read
DCOPY(r_d, r_s)	$: r_{r_d} \leftarrow r_s$	Indirect memory write
JNZ/JZ(r, I_z)		Transfer control to I_z if r is nonzero, zero
READ(r_d)	$: r_d \leftarrow input$	Read a value from input
PRINT(r_s)	$: r_d \rightarrow output$	Write a value to output

While the original model assumes an infinite quantity of infinite-size registers, we fix the size of our registers and the number of these registers arbitrarily. We do this for both theoretical (it simplifies some counting arguments later) and for practical reasons (real machines have finite RAM).

For the purposes of this paper, we fix the size of registers/memory cells to be 32-bit numbers (the set of which we denote $bits_{32}$), and the number of registers/memory cells to 2^{16}. We also denote the memory cells $r_0, \ldots, r_6$ as *registers*. This has no effect at the moment, but will be used later when we introduce attacker models.

The set of possible memory configurations of the machine is denoted by Q_{cpu}; a program for this *cpu* is denoted with ρ, and individual lines in this program is denoted by ρ_i where i is the line number.

Note that the state of the machine is fully determined by the tuple $((q_1, \ldots, q_{2^{16}}) =: \vec{q}, \rho, \rho_i)$: The state of all memory cells, the program that is running, and the line in the program the machine will execute next.

2.1 Example IFSM: What to emulate?

There are many different ways of emulating the IFSM in the toy computing environment. Examining our informal diagram again, emulation needs to be constructed for the three conditional edges in the diagram (labeled b, c, and d) as well as the 3 different state modifications (labeled B, C, D).

2.1.1 Example IFSM emulation: Variant 1. The first emulator of the example IFSM uses registers/cells 0 through 5 as scratch for reading input, and cells 6 to 10006 as a simple flat array for storing pairs of values. It uses no sophisticated data structures and simply searches memory for empty pairs of memory cells, zeroing them in order to release them.

Full source code for the emulator can be found on page 13 in figure 5.

2.1.2 Example IFSM emulation: Variant 2. The first example does not use any sophisticated data structures. The *Memory* of the IFSM is emulated by a simple flat array, at the cost of always having to traverse all 5000 elements of the array when checking for a value.

The second variant implements the same IFSM, but in order to be more efficient, implements *Memory* as two singly linked lists, one for keeping track of free space for password-secret tuples, and one for keeping track of currently active password-secret tuples.

Full source code for the emulator for variant 2 can be found on page 14 in figure 6.

3 ERRORS - REACHING A WEIRD STATE

A common problem when investigating foundations of computer security is the difficulty of even defining exactly what a *bug* is - defining precisely when a program has encountered a flaw and is no longer in a well-defined state. Using the abstraction of the IFSM and viewing the software as an emulator for the IFSM, this becomes tractable.

Intuitively, a program has gone 'off the rails' or a *bug* has occurred when the concrete *cpu* has entered a state that has no clean equivalent in the IFSM - when the state of the *cpu* neither maps to a valid state of the IFSM, nor to an intermediate state along the edges of the IFSM.

To make this notion formal, we define two mappings (remember that Q_{cpu} is the set of possible states of the concrete *cpu* on which the IFSM is emulated, and Q_θ is the set of possible states of the IFSM):

Definition 3.1 (Instantiation). Given an IFSM θ and a target machine *cpu* on which the IFSM is emulated by means of a program ρ, the *instantiation* mapping

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

is a mapping that maps states of the IFSM to the set of states of the concrete *cpu* that can be used to represent these states. Note that it is common that one state in the IFSM can be represented by a large number of states of the target machine.

Definition 3.2 (Abstraction). Given an IFSM θ and a target machine *cpu* on which the IFSM is emulated by means of program ρ, the partial *abstraction* mapping

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

maps a concrete state of the target machine to the IFSM state that it represents. Note that this is a partial mapping: There are many states of *cpu* which do not map to an IFSM state. We denote the set of states on which α is defined as Q_{cpu}^{sane}.

During the process of emulating the IFSM, the target machine is necessarily in states on which $\alpha_{\theta,cpu,\rho}$ is not defined - since following an edge in the IFSM diagram often involves multi-step state modifications to reach a desired target state of the IFSM. To differentiate these states from erroneous states, we define *transitory states*.

Intuitively, a transitory state is a state occuring during the emulation of an edge in the state machine diagram of the IFSM that is always part of a benign and intended transition.

Definition 3.3 (Transitory State). Given an IFSM θ and a target machine *cpu* on which the IFSM is emulated by means of the program ρ, a *transitory* state q^{trans} of the *cpu* is a state that satisfies all of the following:

(1) there exists $S, S' \in Q_\theta$ and $\sigma \in \Sigma$ so that $\delta(S, \sigma) = S'$ - the transition from S to S' given input σ is an existing transition in the IFSM, hence an intended transition.
(2) there exists $q_S \in \gamma_{\theta,cpu,\rho}(S), q_{S'} \in \gamma_{\theta,cpu,\rho}(S')$ and a sequence of state transitions

$$q_S \rightarrow^n q^{trans} \rightarrow^{n'} q_{S'}$$

so that $\alpha_{\theta,cpu,\rho}$ is not defined on all intermediate states and so that all sequences of transitions from q^{trans} lead to $q_{S'}$, irrespective of any addition input and before the machine performs any output.

The set of transitory states will be denoted Q_{cpu}^{trans} from here on.

Clearly, if irrespective of any attacker actions (input) the machine always transitions into a well-defined and intended state without any observable effects, the transitory state is not relevant for the security properties of the IFSM.

Example 3.4 (Example mappings for Emulator Variant 1). For our very simple first example, we can provide the relevant mappings explicitly. An element of Q_{cpu} can be described by the state of all memory cells ($\vec{q}$) and the program line ρ_i. Let $\tau(i) := 2i + \text{base}$. Then

$$\gamma_{\theta,cpu,\rho}(A_M) = \left\{ \begin{array}{l} \vec{q} \in Q_{cpu} \text{ so that} \\ \forall (p,s) \in M \; \exists i \in \mathbb{N}^{<5000} \text{ with} \\ \quad (q_{\tau(i)}, q_{\tau(i)+1}) = (p,s) \\ \wedge (\forall i \neq j \text{ with } (q_{\tau(i)}, q_{\tau(i)+1}) = (q_{\tau(j)}, q_{\tau(j)+1}) \\ \Rightarrow q_{\tau(i)} = q_{\tau(i)+1} = 0) \end{array} \right\}$$

Once we have γ, we can define α in terms of it: Let $q' \in Q_{cpu}$. Then

$$\alpha_{\theta,cpu,\rho} : \bigcup_{M \in \mathcal{M}} \gamma_{\theta,cpu,\rho}(A_M) \rightarrow Q$$

$$\alpha_{\theta,cpu,\rho}(q') = A_M \text{ with } q' \in \gamma_{\theta,cpu,\rho}(A_M)$$

Now we have all the pieces in place to define erroneous and non-erroneous states.

3.1 Defining *weird states*

With the above definitions we can partition the set of possible states Q_{cpu} into three parts: States that directly correspond to states of the IFSM, transitory states that are just symptoms of the emulator transitioning between valid IFSM states, and all the other states.

These other states are the object of study of this paper, and the principal object of study of the exploit practitioner community. They will be called *weird* states in the remainder of this paper - to reflect the fact that they arise unintentionally and do not have any meaningful interpretation on the more abstract level of the ISFM.

Definition 3.5 (Weird state). Given an IFSM θ, the computing environment *cpu* and the program ρ that is supposed to emulate θ, the set Q_{cpu} can be partitioned into disjoint sets as follows:

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

An element of Q_{cpu} that is neither in Q_{cpu}^{sane} nor in Q_{cpu}^{trans} is called a *weird state*, and the set of all such states is denoted as Q_{cpu}^{weird}.

3.1.1 *Possible sources of* weird states.

There are many possible sources for *weird states*. Some of these sources are:

Human Error in the construction of the program ρ. This is probably the single most common source of *weird states* in the real world: Since the process of constructing ρ is based on humans that often have to work on a non-existent or highly incomplete specification of the IFSM, mistakes are made and program paths through ρ exist that allow entering a *weird state*. Real-world examples of this include pretty much all memory corruption bugs, buffer overflows etc.

Hardware Faults during the execution of ρ. While deterministic computing is a convenient abstraction, the hardware of any real-world computing system is often only *probabilistically deterministic*, e.g. deterministic in the average case with some low-probability situations in which it non-deterministically flips some bits. A prime example for this is the widely-publicized Rowhammer hardware issue [13] (and the resulting exploitation [17]) .

Transcription Errors that are introduced into ρ if ρ is transmitted over a channel that can introduce errors. Examples of this include ρ being stored on a storage medium / harddisk which due to environmental factors or hardware failure corrupts ρ partially.

4 WEIRD MACHINES: EMULATED IFSM TRANSITIONS APPLIED TO WEIRD STATES

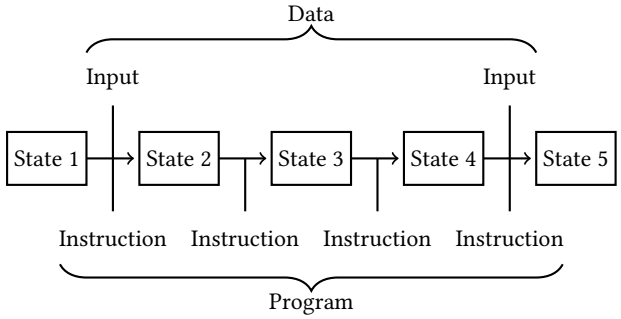
Given the definition of weird states, we now need to examine what happens to the emulated IFSM when ρ can be made to compute on a weird state.

4.1 Interaction as a form of programming

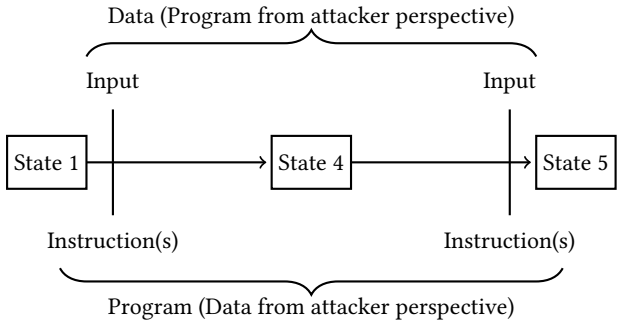
Before examining computation on weird states, though, we need to clarify to ourselves that sending input to a finite state transducer is a form of programming. The set of symbols that can be sent for a restricted instruction set, and the state transitions inside the finite state transducer are the semantics of these instructions. Sending

input is the same thing as programming. This change of perspective is crucial.

The classical perspective views a program as being a sequence of instructions that, combined with some input, drive the machine through a series of states:



We can summarize the sequence of instruction that drives the machine from state 1 to state 5 into one instruction, and summarize the intermediate states, too: From the outside, they are unobservable.



The symmetry of the resulting diagram makes it clear that every finite-state transducer (and as a result, every piece of real-world software) can be viewed from two angles: As an intended machine where the contents of memory, combined with the code, operate on input data - but, from the attacker perspective, as an unintended machine where the input data, combined with the code, operates on the contents of memory. Each side views what it can control as the program, and what it does not control as the data. Mathematically, there is no distinction between the two perspectives.

Under normal conditions, this dual perspective does not matter: By sending symbols to the IFSM, the attacker can of course cause the IFSM to change state - this is obvious and unremarkable. The dual perspective becomes important as soon as a weird state is entered and the attacker obtains much more liberty to modify states than anticipated.

4.2 The weird machine

To recapitulate: There is the machine that the programmer intends to have, the IFSM. Since he only has the *cpu* available, he generates the program ρ to simulate the IFSM on the general *cpu*. This program emulates all the state transitions of the IFSM so that a state from Q_{cpu}^{sane} gets transformed into another state from Q_{cpu}^{sane}, whilst traversing a number of states from Q_{cpu}^{trans}.

Now we consider an attacker that has the ability to somehow move the *cpu* into a weird state - a state that has no meaningful equivalent in the IFSM, and that will also not necessarily reconverge to a state that does. This initial weird state will be called $q_{init} \in Q_{cpu}^{\text{weird}}$.

Once the attacker has achieved this, a new computing device emerges: A machine that transforms the states in Q_{cpu}, particularly those in Q_{cpu}^{weird}, by means of transitions that were meant to transform valid IFSM states into each other, and that takes an instruction stream from the attacker (in form of further inputs).

Definition 4.1 (Weird Machine). The *weird machine* is the computing device that arises from the operation of the emulated transitions of the IFSM on weird states. It consists of the 7-tuple

$$(Q_{cpu}^{\text{weird}}, q_{init}, Q_{cpu}^{\text{sane}} \cup Q_{cpu}^{trans}, \Sigma', \Delta', \delta', \sigma')$$

Note that $Q_{cpu}^{\text{sane}} \cup Q_{cpu}^{trans}$ are terminating states for the weird machine; if one of these states is entered, ρ begins emulating the original IFSM again. Further note that the alphabets for input and output may be different from those for the IFSM.

The *weird machine* has a number of interesting properties:

Input as instruction stream The most interesting property of the weird machine is that, contrary to individual lines of ρ transforming states in Q_{cpu}, the weird machine takes the instruction stream from user input: Every input is an opcode that leads to the execution of the emulated transition. While this is true for the IFSM as well, the IFSM can only reach a well-defined and safe set of states. The weird machine on the other hand has a state space of unknown size that can be explored by 'programming' it - sending careful crafted inputs.

Unknown state space The state space is a priori not known: It depends heavily on ρ and q_{init}, and determining the size and shape of Q_{cpu}^{weird} is very difficult. This also means that determining whether the security properties of the IFSM can be violated is a nontrivial endeavour.

Unknown computational power It is a-priori unclear how much computational power a given weird machine will have. Intuitively, since the transitions of the IFSM end up being the 'instructions' of the weird machine, greater complexity of the IFSM appears to imply greater computational power; but the actual way the transitions are implemented is just as important - some constructs will lead to easier exploitation than others.

Emergent instruction set The attacker gets to choose the sequence of instructions, but the instruction set itself emerges from a combination of the IFSM and the emulator ρ. This means that while the machine is programmable, and the semantics of the instructions are well-defined, the instructions themselves are often extremely unwieldly to use. Furthermore, the attacker needs to discover the semantics of his instructions during the construction of the attack and infer them from ρ and q_{init}.

5 DEFINITION OF EXPLOITATION

Definition 5.1 (Exploitation). Given a method to enter a weird state $q_{init} \in Q_{cpu}^{init} \subset Q_{cpu}^{\text{weird}}$ from a set of particular sane states $\{q_i\}_{i \in I} \subset Q_{cpu}^{\text{sane}}$, *exploitation* is the process of *setup* (choosing the right q_i), *instantiation* (entering q_{init}) and *programming* of the weird machine so that security properties of the IFSM are violated.

An exploit is "just" a program for the weird machine that leads to a violation of the security properties. For a given vulnerability (a method to move the machine into a weird state) it is likely that an infinite number of different programs exist that achieve the same goals by different means.

5.1 Exploitability of Variant 1 and Variant 2

A natural question arises when discussing "exploitability": Do the different implementations of our IFSM have different properties with regards to exploitability ? Does the attacker gain more power by corrupting memory in one case than in the other? Is it possible to implement software in a way that is more resilient to exploitation under certain memory corruptions?

In order to answer these questions, we need a model for an attacker.

5.1.1 The attacker model. How does one model the capabilities of an attacker? The cryptographic community has a hierarchy of detailed attacker models (known-plaintext, chosen-plaintext etc.) under which they examine their constructs; in order to reason about the exploitability of the different implementations we define a few attacker models for memory corruptions. Some of these will seem unrealistically powerful - this is by design, as resilience against an unrealistically powerful attacker will imply resilience against less-powerful attackers.

Arbitrary program-point, chosen-bitflip In this model, the attacker gets to stop ρ while executing, flip an attacker-chosen bit in memory, and continue executing.

Arbitrary program-point, chosen-bitflip, registers This model is identical to the above with the exception that the memory cells 0 through 6 are protected from the attacker. This reflects the notion that *cpu* registers exist that are normally not corrupted. The attacker still gets to stop ρ while executing, and gets to choose which bit to flip.

Fixed-program point, chosen bitflip, registers In reality, attackers can usually not stop the program at an arbitrary point to flip bits. It is more likely that a transcription error has happened (e.g. a bug has been introduced into ρ) at a particular program point.

Various other models are imaginable.[3]

For the purpose of examining our two different variants, we choose the second model: Arbitrary program-point, chosen-bitflip anywhere in memory with the exception of the first 7 cells (registers). This choice is made out of convenience; obviously, more powerful attackers exist.

We show next how the variant 2 (which uses the singly-linked lists) is exploitable in this model, while variant 1 that uses flat arrays turns out - possibly counterintuitively - to be not exploitable by this powerful attacker.

5.1.2 Extending the security game. We defined security properties involving an attacker that can specify a probability distribution over finite-state transducers from which an "attacking" transducer is drawn. In order to include our attacker models into this framework, we simply allow the attacker to corrupt memory while the two machines duel. Concretely, step 5 in the game described in 1.3.1 is extended so that the attacker can stop the attacked program at any point, flip a single bit of memory, and then resume execution.

5.1.3 Proof of exploitability of Variant 2. In order to show exploitability, it is sufficient to provide a sequence of steps (including a single chosen bitflip) that helps an attacker violate the assumptions of the security model. This is done using sequential diagrams showing the internal state of the emulator over pages 8, 9 and 10. In this example, the attacker gets to interact with the machine for a few steps; the defender gets to store his secret in the machine, and the attacker then gets to attempt to extract the defenders secret by flipping just a single bit.

The machine begins in it's initial state, e.g. all memory cells are empty and the head of the free list is at zero.

The diagrams on pages 8, 9 and 10 show the first 15 non-register memory cells, along with the points-to-relations between them. Furthermore, the heads of the free and used linked lists are marked. Between two such diagrams, the actions that the defender or attacker takes are listed, and the resulting state is shown in the subsequent diagram.

Following the diagrams, it is clear that an attacker can exploit the linked-list variant of the IFSM emulator using just a single bit-flip. It is also clear that the specific sequence of inputs the attacker sends to the emulator after the bit-flip constitutes a form of program.

5.1.4 Proof of non-exploitability of Variant 1. The proof idea is to show that any attacker that is capable of flipping a single bit can be emulated by an attacker without this capability with a maximum of 10000 more interactions between attacker and emulated IFSM, thus demonstrating that the attacker can not obtain a significant advantage by using his bit-flipping ability. The number 10000 arises from the fact that our IFSM has a maximum of 5000 (p, s)-tuples in memory. In order to replace a particular memory cell in the emulated IFSM, the emulation process needs to fill up to 4999 tuples with temporary dummy values and remove them again thereafter, leading to 9998 extra interactions for targeting a particular cell.

Assumption 1. *We assume that the security property in 1.3.1 holds for Variant 1 provided the attacker can not corrupt memory.*

The proof proceeds by contradiction: Assume that an attacker can specify a distribution over finite state transducers, a particular bit of memory, and a particular point in time when to flip this bit of memory, to gain an advantage of at least knowing one bit of the secret:

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

Let $\Theta_{exploit}$ be a transducer from the specified distribution that succeeds with maximal advantage: No other transducer shall have

[3] Other examples that are worth exploring: Fixed-program-point random-bit flip, Fixed-program-point chosen-bit flip, Fixed-program-point chosen-byte-writing, Fixed-program-point arbitrary memory rewriting etc.

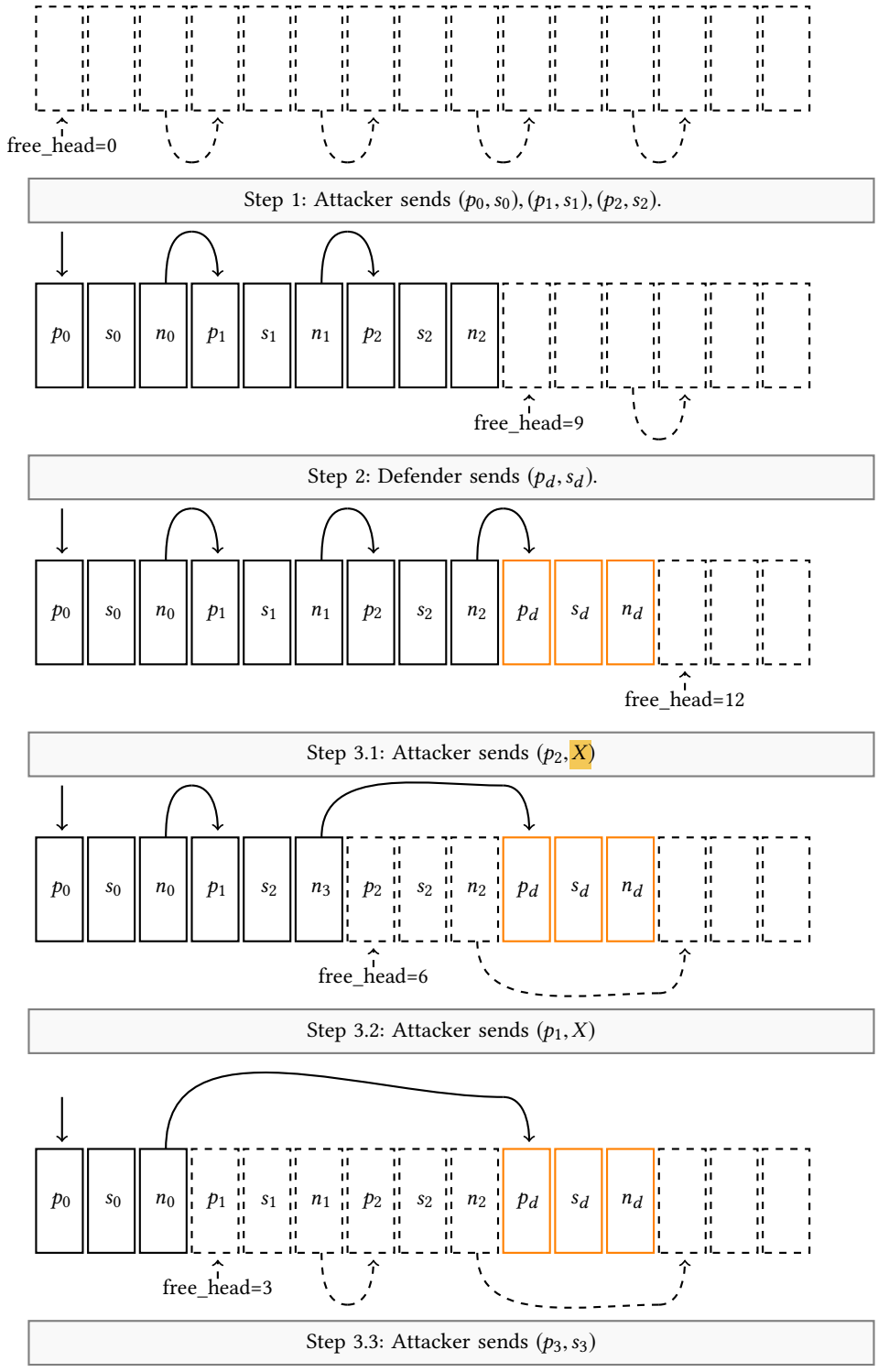
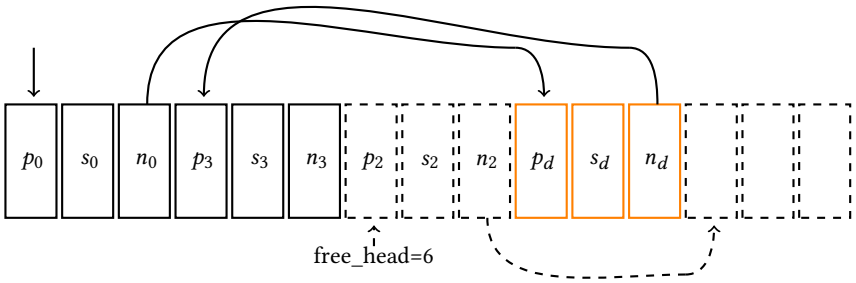
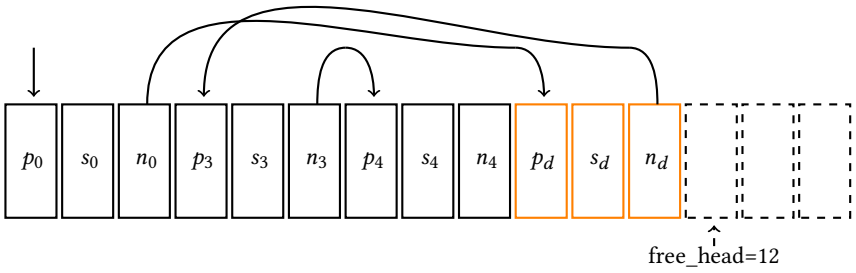


Figure 2: The first part of the attack: Attacker set-up.

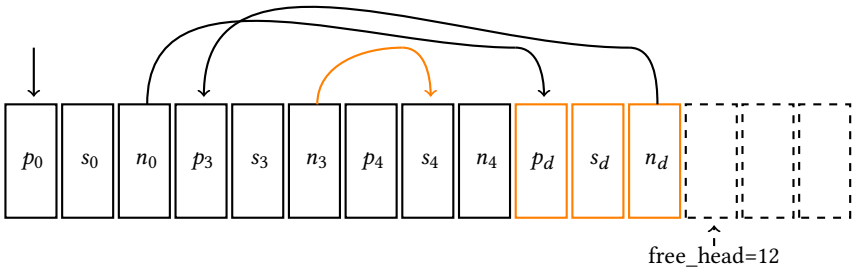
Step 3.3: Attacker sent $(p_2, X), (p_1, X), (p_3, s_3)$



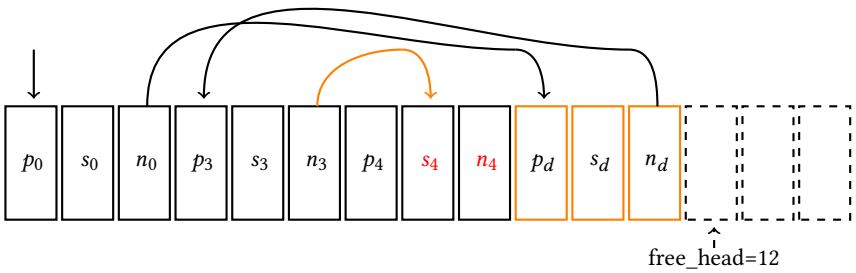
Step 3.4: Attacker sends (p_4, s_4)



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



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



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

Figure 3: The attacker uses his memory-corrupting powers.

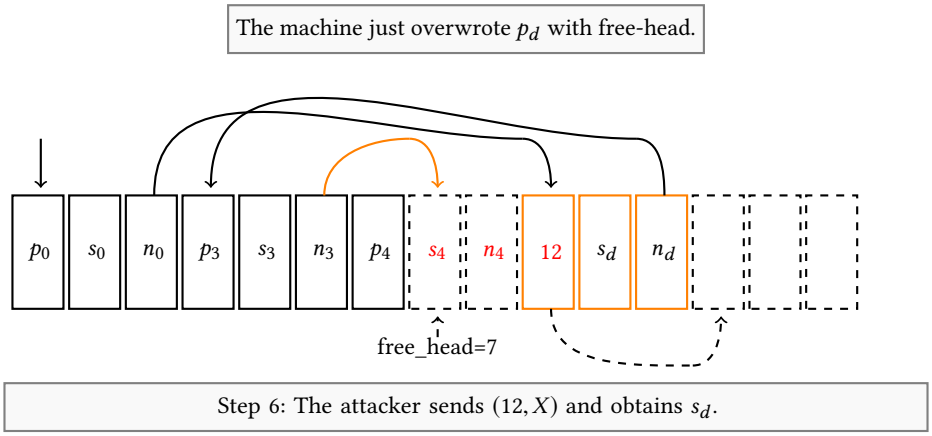


Figure 4: The final successful steps of the attack.

a greater gap between it's probability of success and the security boundary:

$$\Theta_{exploit} = \arg\max_{exploit} P[s \in o_{\text{IFSM}}] - \frac{|o_{\text{exploit}}|}{2^{31}}$$

.

We now state two lemmas describing the set of states reachable by an attacker. No proof is given, but they are easily verified by inspecting the code.

Lemma 1. *All states in Q_{cpu}^{trans} are of the following form: $q \in Q_{cpu}^{sane}$ with exactly one partially-stored tuple (corresponding to program lines 36 and 37) - a short time period where one of the memory cells contains a $p \neq 0$ with a stale s.*

Lemma 2. *An attacker that can flip a bit can only perform the following 5 transitions:*

1. *Replace a (p, s) tuple in memory with $(p \oplus 2^i, s)$.*
2. *Transition a state with memory containing two tuples $(p, s_1), (p \oplus 2^i, s_2)$ into a state where memory contains $(p, s_1), (p, s_2)$.*
3. *Replace a (p, s) tuple in memory with $(p, s \oplus 2^i)$*
4. *Replace a $(p, 2^i)$ tuple with $(p, 0)$*
5. *Replace a $(2^i, s)$ tuple with $(0, s)$*

Note that 1, 3 and 5 are all transitions from Q_{cpu}^{sane} to Q_{cpu}^{sane}. Only 2 and 4 lead to Q_{cpu}^{weird}.

Now consider $S \in Q_{cpu}^{n}$ the sequence of state transitions of Q_{cpu} for a successful attack by $\Theta_{exploit}$.

Theorem 1. *Any sequence of state transitions during a successful attack that use transitions 1, 3, or 5 above can be emulated by an attacker that can not flip memory bits in at most 10000 steps.*

Proof. For all cases, the attacker without the ability to flip bits sends (p_i, x_i) tuples to fill all empty cells preceding the cell in which $\Theta_{exploit}$ flips a bit, performs the action described, and then sends (p_i, x_i) to free up these cells again. We denote an arbitrary value with x.

For case 1: If p was previously known to the attacker, an attacker without the ability to flip bits can simply send (p, x), receive s, and send $(p \oplus 2^i, s)$. If p was not previously known to the attacker, $p \oplus 2^i$ is not either, and the game proceeds normally without attacker advantage.

For case 3: If p was previously known, the attacker sends $(p, 0)$, receives s, and then sends $(p, s \oplus 2^i)$. If p was not known to the attacker, the game proceeds normally without attacker advantage.

For case 5: The value $p = 2^i$ must have been known, and the transition can be emulated by simply sending $(2^i, x)$.

□

This means that the transitions that the attacker gains that help him transit from one sane state to another, but along an unintended path, do not provide him with any significant advantage over an attacker that can not corrupt memory. What about the transitions that lead to weird states?

Lemma 3. *For any sequence of state transitions that successfully violates the security property, there exists a p' which is never sent by either party.*

Proof. Any sequence for which such a p' does not exist is of length $2^{32} - 1$ and can hence not break the security property. □

Theorem 2. *Any sequence of state transitions during a successful attack that uses transition 2 can only produce output that is a proper subsequence of the output produced by an attacker that cannot flip memory bits, with a maximum of 10000 extra steps.*

Proof. For case 2:

Given that the attacker only gets to flip a bit once, the sequence S will of the form

$$(q_{sane})^{n_1} \rightarrow_{t_2} (q_{weird})^{n_2} \rightarrow_{t_2'} (q_{sane})^{n_3}$$

with n_3 possibly zero. The weird state the attacker enters with t_2 is identical to a sane state except for a duplicate entry with the same p. From this state on, there are two classes of interactions that can occur:

1. A tuple (p, x) is sent, which transitions *cpu* via t_2' back into a sane state.
2. A tuple $(p' \neq p, x)$ is sent, which transitions into another state in the same class (sane except duplicate p).

An attacker without bit flips can produce an output sequence that contains the output sequence of the attacker with bit flips as follows:

(1) Perform identical actions until the bit flip.
(2) From then on, if $p \oplus 2^i$ is sent, replace it with p'.
(3) If p is sent and the address of the cell where p is stored is less than the address where p' is stored, proceed normally to receive s_1. Next
    (a) Send (p', x), receive s_2.
    (b) Fill any relevant empty cells.
    (c) Send (p, s_2).
    (d) Free the temporary cells again.
(4) If p is sent and the address of the cell where p is stored is larger than the address where p' is stored, replace the sending of p with p'.
(5) Other operations proceed as normal.

□

Theorem 3. *Any sequence of state transitions during a successful attack that uses transition 4 can only produce output that is a proper subsequence of the output produced by an attacker that cannot flip memory bits.*

Proof. The same properties about the weird state only transitioning into another weird state of the same form or back into a sane state that held in the proof for transition 2 holds for transition 4. To produce the desired output sequence, the attacker without bit flips simply replaces the first query for p after the bit flip with the query $(0, 0)$. □

We have shown that we can emulate any bit-flipping attacker in a maximum of 10000 steps using a non-bit-flipping attacker.

Since we assumed that our bit-flipping attacker can obtain an attack probability

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

it follows that the emulation for the bit-flipping attacker by a non-bit-flipping attacker achieves

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

This contradicts our assumption that the non-bit-flipping attacker cannot beat our security boundary, and hence proves that a bit-flipping attacker cannot get an advantage of even a single bit over a non-bit-flipping attacker.

6 CONSEQUENCES

There are a number of consequences of the previous discussion; they mostly relate to questions about mitigations, demonstrating non-exploitability, and the decoupling of exploitation from control flow.

6.1 Making statements about non-exploitability is difficult

Even experts in computer security routinely make mistakes when assessing the *exploitability* of a particular security issue. Examples range from Sendmail bugs [19] via the famous exploitation of a memcpy with 'negative' length in Apache [18] to the successful exploitation of hardware-failure-induced random bit flips [17]. In all of these cases, large percentages of the security and computer science community were convinced that the underlying memory corruption could not be leveraged meaningfully by attackers, only to be proven wrong later.

It is difficult to reason about the computational power of a given weird machine: After all, a vulnerability provides an assembly language for a computer that has never been programmed before, and that was not designed with programmability in mind. The inherent difficulty of making statements about the non-existence of programs in a given machine language with only empirically accessible semantics may be one of the reasons why statements about non-exploitability are difficult.

Furthermore, many security vulnerabilities have the property that many different initial states can be used to initialize the weird machine, further complicating matters: One needs to argue over all possible transitions into weird states and their possible trajectories thereafter.

6.2 Making statements about non-exploitability is possible

While making statements about non-exploitability is supremely difficult for complex systems, somewhat surprisingly we can construct computational environments and implementations that are provably resistant to classes of memory-corrupting attackers.

This may open a somewhat new research direction: What data structures can be implemented with what level of resiliency against memory corruptions, and at what performance cost?

6.3 Mitigations and their utility

Computer security has a long history of exploit mitigations - and bypasses for these mitigations: From stack cookies [7, 15] via ASLR [21] to various forms of control-flow-integrity (CFI) [1, 9, 22] . The historical pattern has been the publication of a given mitigation, followed by methods to bypass the mitigations for particular bug instances or entire classes of bugs.

In recent years, exploit mitigations that introduce randomness into the states of *cpu* have been very popular, ranging from ASLR [21] via various heap layout randomizations to efforts that shuffle existing code blocks around to prevent ROP-style attacks. It has often been argued (with some plausibility) that these prevent exploitation - or at least "raise the bar" for an attacker. While introducing unpredictability into a programming language makes programming more difficult and less convenient, it is somewhat unclear to what extent layering such mitigations provides long-term obstacles for an attacker that repeatedly attacks the same target.

An attacker that deals with the same target program repeatedly finds himself in a situation where he repeatedly programs highly related computational devices, and it is doubtful that no weird machine program fragments exist which allow an attacker to achieve security violations in spite of not knowing the exact state of *cpu* from the outset. It is imaginable that the added benefit from increasing randomization beyond ASLR vanishes rapidly if the attacker cannot be generically prevented from reading crucial parts of memory.

Mitigations should be preferred that detect corruptions and large classes of weird states in order to terminate the program quickly. [4]

Ideally, mitigations should work independently of the particular way the attacker chooses to program the weird machine. Mitigations that only break a particular way of attacking a vulnerability are akin to blacklisting a particular programming language idiom - unless the idiom is particularly important and unavoidable, odds are that an attacker can work around the missing idiom. While this certainly creates a cost for the attacker, the risk is that this is a one-off cost: The attacker only has to construct a new idiom once, and can re-use it for multiple attacks on the same target.

6.3.1 Limitations of CFI to prevent exploitation. It should be noted that both examples under consideration in this paper exhibited perfect control-flow-integrity: An attacker never subverted control flow (nor could he, in the computational model we used).

Historically,attackers preferred to obtain control over the instruction pointer of *cpu* - so most effort on the defensive side is spent on preventing this from happening. It is likely, though, that the reason why attackers prefer hijacking the instruction pointer is because it allows them to leave the "difficult" world of weird machine programming and program a machine that is well-understood with clearly specified semantics - the *cpu*. It is quite unclear to what extent perfect CFI would render attacks impossible, and depends heavily on the security properties of the attacked program, as well as the other code it contains.

An excessive focus on control flow may set wrong priorities: Exploitation can occur without control flow ever being diverted, and the only thing that can obviously be prevented by perfect CFI are arbitrary syscalls out-of-sequence with the normal behavior of the program. While this in itself may be a worthwhile goal, the amount of damage an attacker can do without subverting control flow is substantial.

6.4 Acknowledgements

This paper grew out of long discussions with and benefited from suggestions given by (in random order): Felix Lindner, Ralf-Philipp Weinmann, Willem Pinckaers, Vincenzo Iozzo, Julien Vanegue, Sergey Bratus, Ian Beer, William Whistler, Sean Heelan, Sebastian Krahmer, Sarah Zennou, Ulfar Erlingsson, Mark Brand, Ivan Fratric, Jann Horn, Mara Tam and Alexander Peslyak.

APPENDIX

Appendix A: Program listing for the flat-array variant

Appendix B: Program listing for the linked-list variant

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*. ACM, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165

[2] Clive Blackwell and Hong Zhu (Eds.). 2014. *Cyberpatterns, Unifying Design Patterns with Security and Attack Patterns.* Springer. https://doi.org/10.1007/978-3-319-04447-7

[3] Sergey Bratus, Julian Bangert, Alexandar Gabrovsky, Anna Shubina, Michael E. Locasto, and Daniel Bilar. 2014. Ẃeird MachinePatterns. See [2], 157–171. https://doi.org/10.1007/978-3-319-04447-7_13

[4] Sergey Bratus, Michael E. Locasto, Len Sassaman Meredith L. Patterson, and Anna Shubina. 2011. Exploit Programming: From Buffer Overflows to Weird Machines and Theory of Computation. *j-LOGIN* 36, 6 (Dec. 2011), 13–21. https://www.usenix.org/publications/login/december-2011-volume-36-number-6/exploit-programming-buffer-overflows-weird

[5] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. 1988. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theor. Comput. Sci.* 59 (1988), 115–131. https://doi.org/10.1016/0304-3975(88)90098-9

[6] Stephen A. Cook and Robert A. Reckhow. 1972. Time-bounded Random Access Machines. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing (STOC '72)*. ACM, New York, NY, USA, 73–80. https://doi.org/10.1145/800152.804898

[7] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7 (SSYM'98)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1267549.1267554

[8] Thomas Dullien. 2011. Exploitation and state machines. In *Infiltrate Offensive Security Conference*. Miami Beach, Florida. http://www.slideshare.net/scovetta/fundamentals-of-exploitationrevisited

[9] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, Washington, DC, USA, 575–589. https://doi.org/10.1109/SP.2014.43

[10] Jan Friso Groote and Frits W. Vaandrager. 1990. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence.. In *ICALP* (2009-09-19) *(Lecture Notes in Computer Science)*, Mike Paterson (Ed.), Vol. 443. Springer, 626–638. http://dblp.uni-trier.de/db/conf/icalp/icalp90.html#GrooteV90

[11] Sean Heelan. 2010. Misleading the public for fun and profit. *https://sean.heelan.io/2010/12/07/misleading-the-public-for-fun-and-profit/* (Dec. 2010).

[12] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Chua Leong, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *37th IEEE Symposium on Security and Privacy, San Jose, CA, US, May 2016*.

[13] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 361–372. https://doi.org/10.1145/2678373.2665726

[14] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2007. Exterminator: Automatically correcting memory errors with high probability. In *In Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM.* Press.

[15] Gerardo Richarte. 2002. Four different tricks to bypass StackShield and StackGuard protection. *World Wide Web* 1 (2002).

[16] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications.. In *IEEE Symposium on Security and Privacy*. IEEE, IEEE Computer Society, 745–762. http://dblp.uni-trier.de/db/conf/sp/sp2015.html#SchusterTLDSH15

[17] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. http://googleprojectzero.blogspot.fr/2015/03/exploiting-dram-rowhammer-bug-to-gain.html. (March 2015).

[18] GOBBLES Security. 2002. Ending a few arguments with one simple attachment. *BugTraq Mailing List* 1 (June 2002).

[19] LSD Security. Technical analysis of the remote sendmail vulnerability. Email posted to Bugtraq Mailing List, http://seclists.org/bugtraq/2003/Mar/44 month = mar, year =. (????).

[20] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, New York, NY, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[21] PaX Team. 2003. https://pax.grsecurity.net/docs/aslr.txt. Text File. (March 2003).

[22] PaX Team. 2015. RAP: RIP ROP. https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf. (Oct. 2015).

[23] Rob J. van Glabbeek and W. Peter Weijland. 1996. Branching Time and Abstraction in Bisimulation Semantics. *J. ACM* 43, 3 (May 1996), 555–600. https://doi.org/10.1145/233551.233556

[24] Julien Vanegue. 2014. The Weird Machines in Proof-Carrying Code. *2014 IEEE Security and Privacy Workshops* 0 (2014), 209–213. https://doi.org/10.1109/SPW.2014.37

[4] Strong stack cookies are one example of a mitigation that will deterministically detect a particular class of corruptions if a given program point ρ_i is reached.

```
.const firstIndex 6
.const lastIndex 6 + (5000*2)
BasicStateA:
    READ    r0              # Read p
    READ    r1              # Read s
CheckForNullSecret:
    JZ      r1, OutputErrorMessage
    JZ      r0, OutputErrorMessage
CheckForPresenceOfP:        # Run through all possible array entries.
    LOAD    firstIndex, r3
    LOAD    lastIndex, r4
CheckForCorrectP:
    ICOPY   r3, r5          # Load the stored p of the tuple
    SUB     r5, r0, r5      # Subtract the input p
    JZ      r5, PWasFound
    ADD     r3, 2, r3       # Advance the index into the tuple array.
    SUB     r3, r4, r5      # Have we checked all elements of the array?
    JNZ     r5, CheckForCorrectP
PWasNotFound:
    LOAD    firstIndex, r3
    LOAD    lastIndex, r4
SearchForEmptySlot:
    ICOPY   r3, r5
    JZ      r5, EmptyFound
    ADD     r3, 2, r3
    SUB     r3, r4, r5
    JZ      r5, NoEmptyFound
    J       SearchForEmptySlot
NoEmptyFound:
OutputErrorMessage:
    SUB     r0, r0, r0
    PRINT   r0
    J       BasicStateA
EmptyFound:
    DCOPY   r3, r0  # Write the password
    ADD     r3, 1, r3   # Adjust the pointer
    DCOPY   r3, r1  # Write the secret.
    J       BasicStateA
PWasFound:
    LOAD    0, r4
    DCOPY   r3, r4  # Zero out the stored p
    ADD     r3, 1, r3
    ICOPY   r3, r5  # Read the stored s
    PRINT   r5
    J       BasicStateA
```

variant1A.s

Figure 5: Listing for variant1A.s

[25] David Walker, Lester Mackey, Jay Ligatti, George A. Reis, and David I. August. 2006. Static Typing for a Faulty Lambda Calculus. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/1159803.1159809

```
const free_head 5  # Head of the free list.
const used_head 6  # Head of the used list.
    J       InitializeFreeList
BasicStateA:
    READ    r0              # Read p
    READ    r1              # Read s
    SUB     r2, r2, r2      # Initialize a counter for number of elements.
CheckForNullSecret:
    JZ      r1, OutputErrorMessage # Zero secret not allowed.
    JZ      r0, OutputErrorMessage # Zero password not allowed.
    LOAD    used_head, r3 # The list consists of [p, s, nxt] tuples.
CheckForPresenceOfP:
    JZ      r3, EndOfUsedListFound
    ICOPY   r3, r4          # Load 'p' of the entry.
    SUB     r4, r0, r4      # Compare against the password
    JZ      r4, PWasFound # Element was found.
    ADD     r3, 2, r3       # Advance to 'next' within [p, s, nxt]
    ICOPY   r3, r3          # Load the 'next' pointer.
    J       r3, CheckForPresenceOfP
EndOfUsedListFound:
    LOAD    free_head, r3
    JZ      r3, OutputErrorMessage # No more free elements available?
    ICOPY   r3, r2       # Get the first element from the free list
    DCOPY   r2, r0       # Write the [p, ?, ?]
    ADD     r2, 1, r4
    DCOPY   r4, r1       # Write the [p, s, ?]
    LOAD    used_head, r0
    ICOPY   r0, r1       # Load used_head to place it in 'next'
    DCOPY   r0, r2       # Rewrite used_head to point to new element
    ADD     r2, 2, r4    # Point to 'next' field
    ICOPY   r4, r2       # Load the ptr to the next free element into r2
    DCOPY   r4, r1       # Write the [p, s, next]
    DCOPY   r3, r2       # Write the free_head -> next free element
    J       BasicStateA
PWasFound:
    ADD     r3, 1, r2
    ICOPY   r2, r1     # Load the stored secret.
    PRINT   r1     # Output the secret.
    ADD     r3, 2, r2  # Point r2 to the next field.
    LOAD    free_head, r1
    ICOPY   r1, r0  # Read the current pointer to the free list.
    DCOPY   r2, r1  # Point next ptr of current triple to free list.
    DCOPY   r1, r3  # Point free-head to current triple.
    J       BasicStateA
InitializeFreeList:
    LOAD    free_head, r0
LoopToInitialize:
    ADD     r0, 3, r1        # Advance to the next element.
    ADD     r0, 2, r0        # Advance to the next pointer inside.
    DCOPY   r0, r1           # Write the next pointer.
    ADD     r1, 0, r0        # Set current elt = next element.
    SUB     r0, 5000*3+7, r2  # Have we initialized enough?
    JNZ     r2, LoopToInitialize
TerminateFreeList:
    SUB     r0, 1, r0
    DCOPY   r0, r2      # Set the last next-pointer 0 to terminate
                        # the free list.
WriteInitialFreeHead:
    LOAD    used_head+1, r0
    LOAD    free_head, r1
    DCOPY   r1, r0     # Set the free-head to point to the first triple.
    J       BasicStateA
OutputErrorMessage:
    SUB     r0, r0, r0
    PRINT   r0
    J       BasiceStateA
```

variant2A.s

Figure 6: Listing for variant2A.s