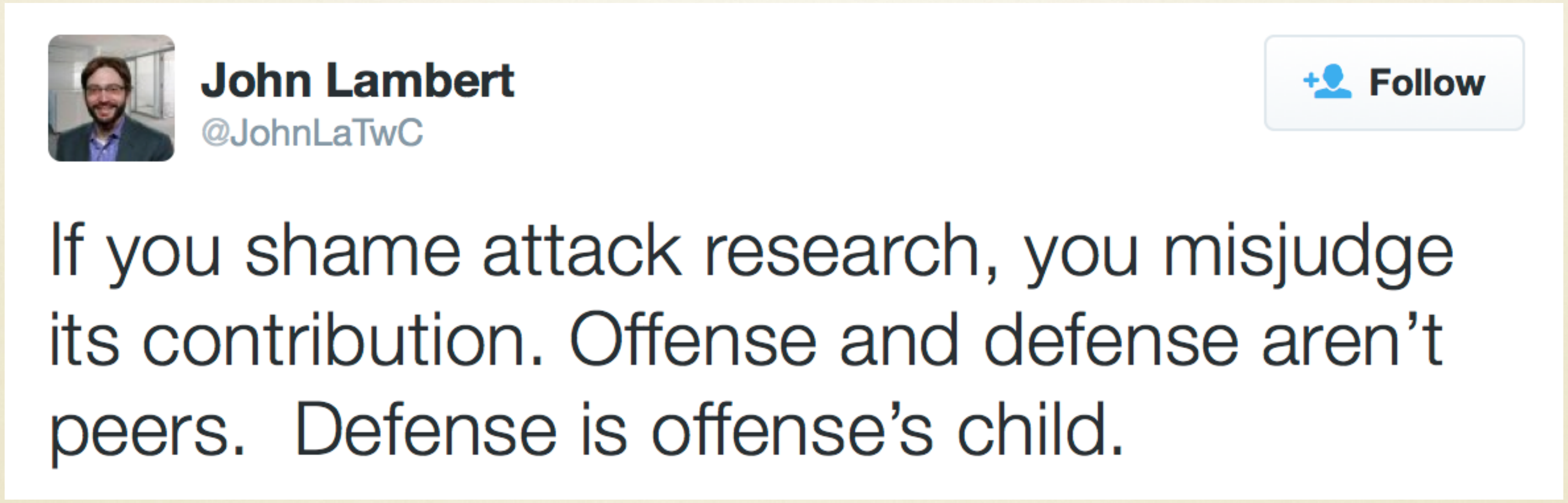
Regulating your nose to spite your face

Sergey Bratus

OFFENCE VS DEFENCE

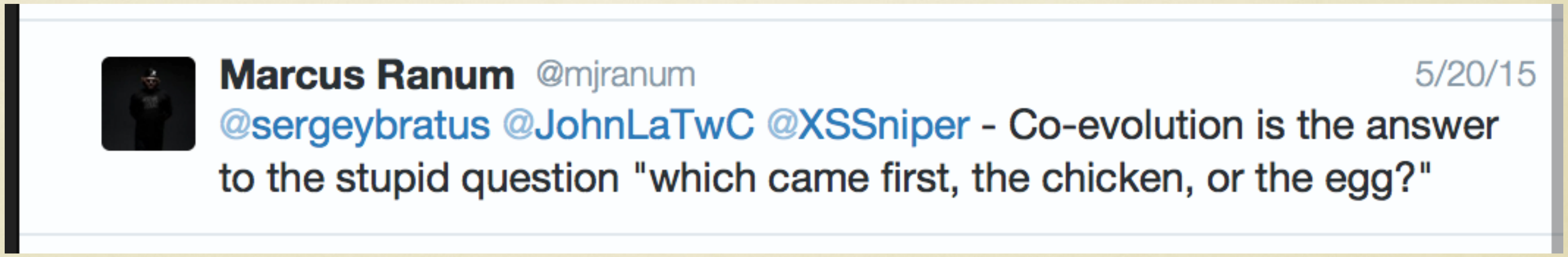
What is their relationship?



John Lambert
@JohnLaTwC

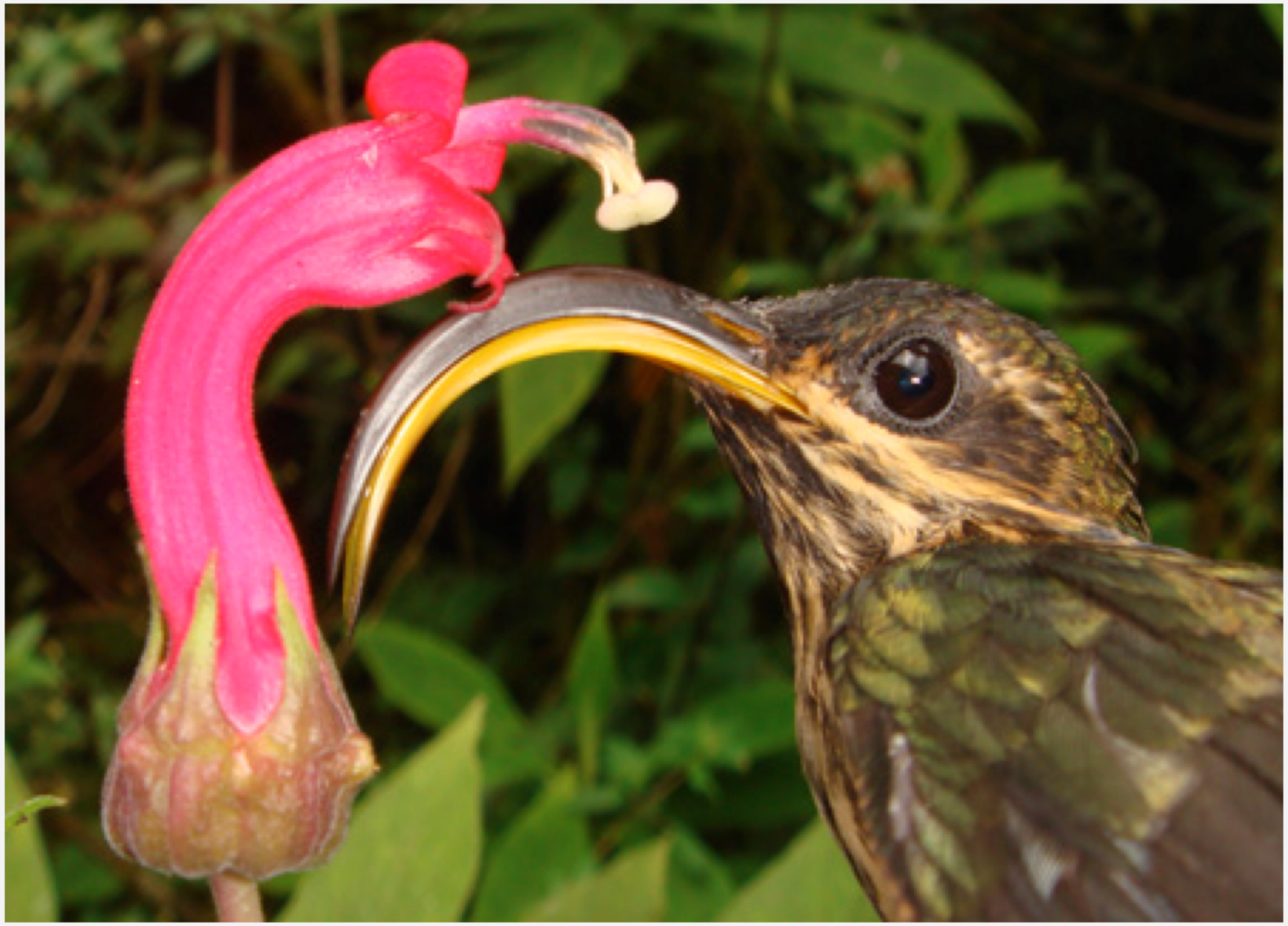
Follow

If you shame attack research, you misjudge its contribution. Offense and defense aren't peers. Defense is offense's child.



Marcus Ranum @mjranum 5/20/15
@sergeybratus @JohnLaTwC @XSSniper - Co-evolution is the answer to the stupid question "which came first, the chicken, or the egg?"

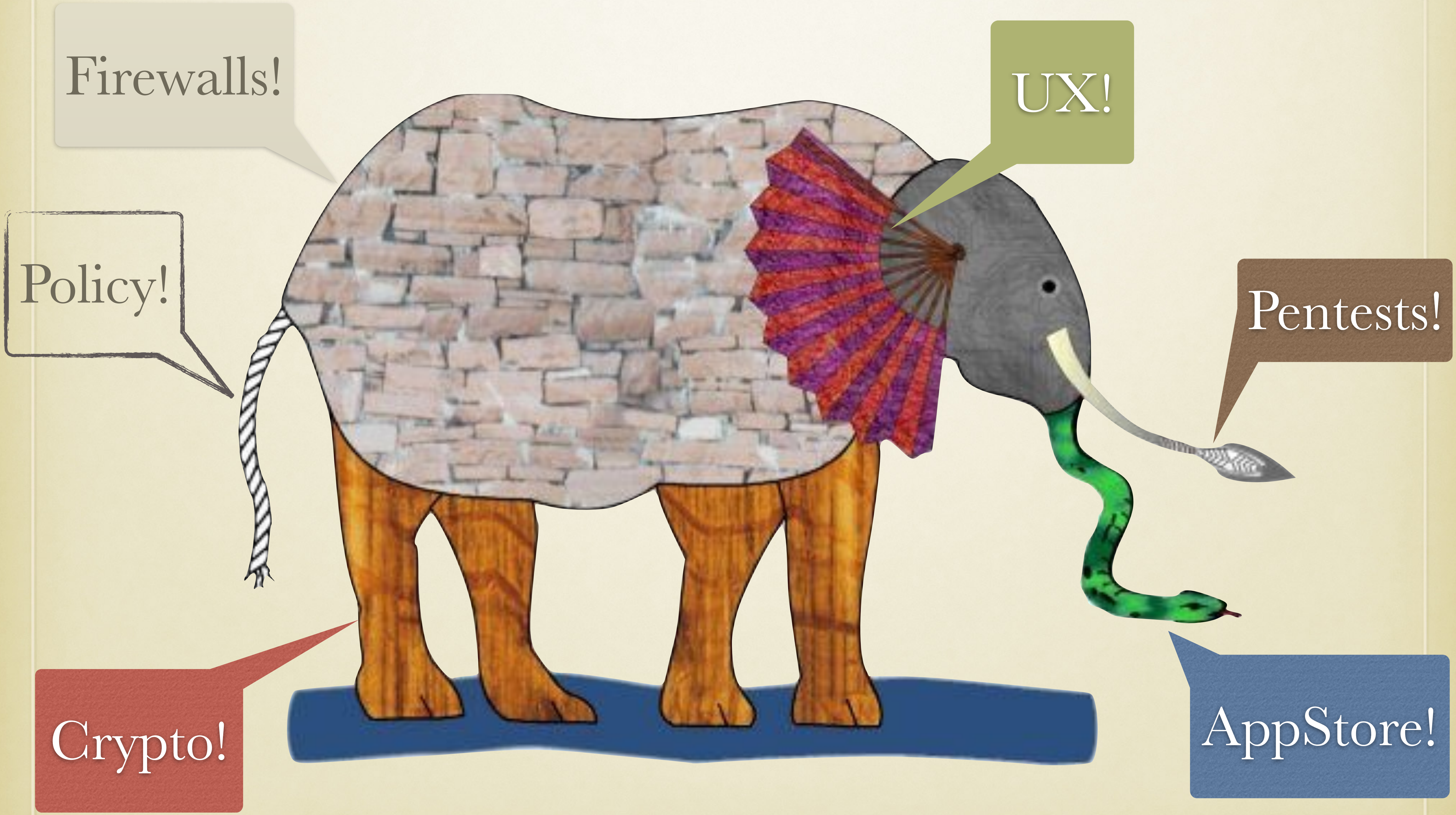
IT'S CO-EVOLUTION, BUT WHAT IS (CO-)EVOLVING?


The Buff-tailed Sicklebill (Eutoxeres condamini), a hermit hummingbird, beside one of the flowers to which they are specialized, showing how the flower and recurved bill have co-evolved. Photo by Christopher Witt, University of New Mexico.

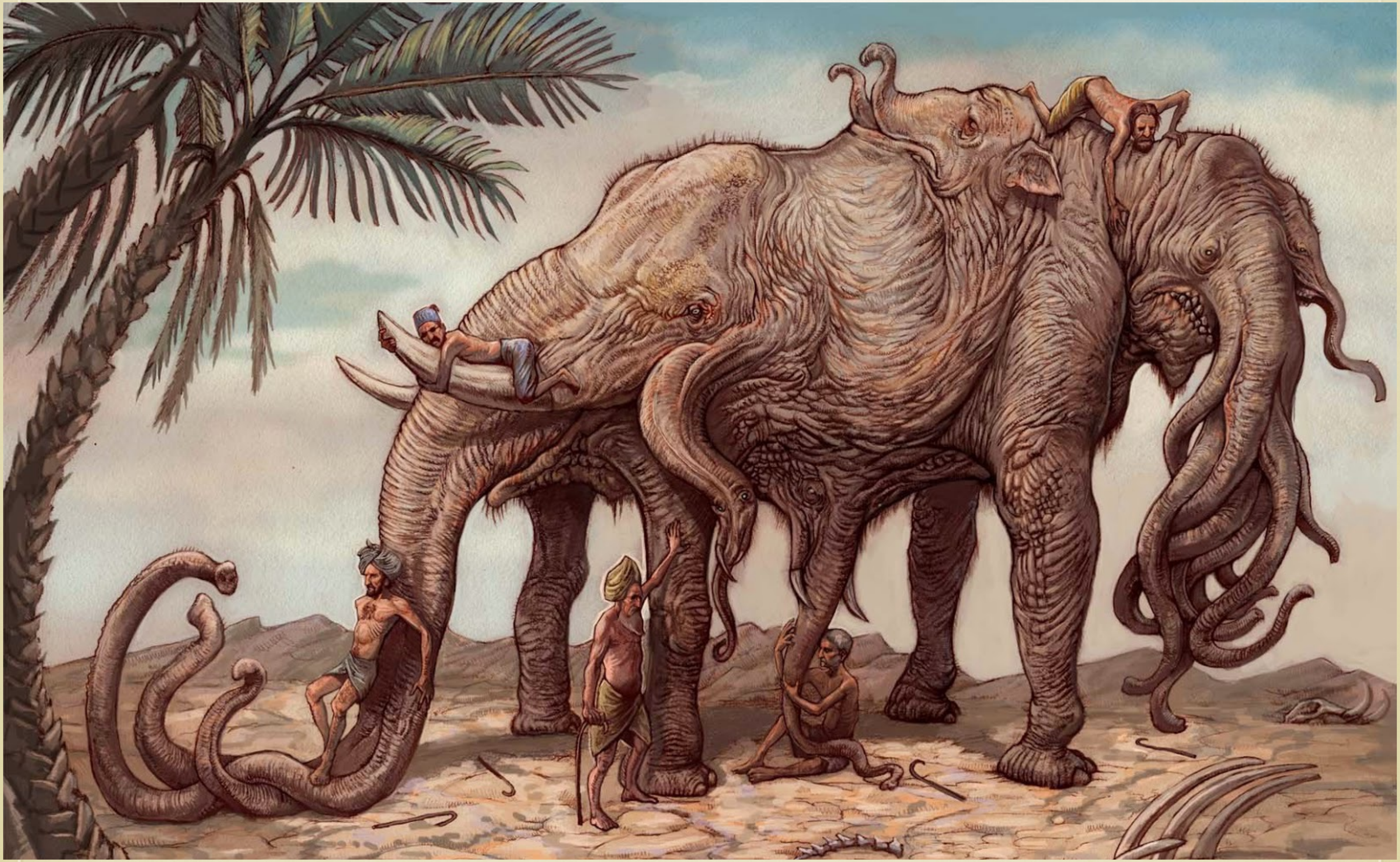
IT'S CO-EVOLUTION, BUT WHAT IS (CO-)EVOLVING?

- Is the co-evolution of offensive and defensive computing "random" & meaningless?
- Is it about **adapting** to some "random" external conditions?
- Does it improve our understanding of how to build computers we could finally **trust**?

"The security elephant"



"The real security elephant"



http://davidmengart.blogspot.de/2014/03/of-elephants-and-blind-men.html

OFFENSIVE COMPUTING

- Problem: we build computing systems with behaviors we don't **understand** & cannot **predict**
  - **Emergent** computation kills security & trust
- Offensive computing has higher goals that merely annoying defenders:
  - revealing **mechanisms** of emergent computation
  - revealing actual **impact** of **features** & **bugs**
  - systematically explores **unexpected**/illegal **states**

"WHAT CAN THIS COMPUTE?"

- This core question of computer science is daily, empirically explored by offense
- Goes back to Hilbert's 10th problem, Church, and Turing
- **Undecidability** lurks, emergent Turing completeness abounds

Exploits are proofs

- Exploits are **proofs by construction** that unexpected computation and its supporting **automata** exist within the target
  - Sometimes these are Universal Turing machines
- Our current exploit/proof **methods** are neither methodical nor precise; only our **results** are
  - "Can't argue with root shell"

Methods

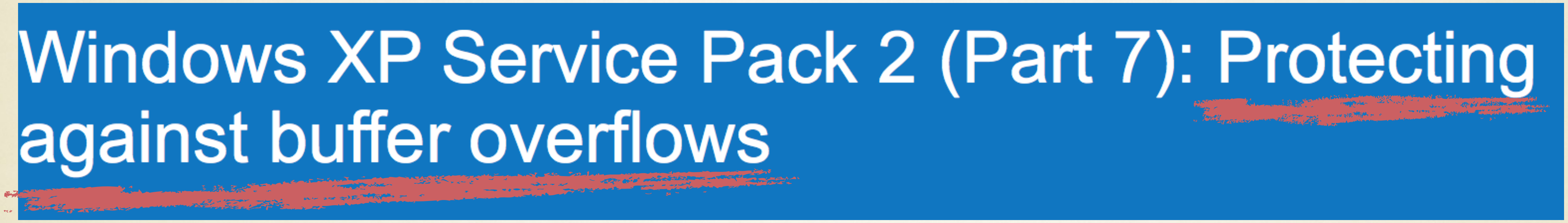
THREE TALES OF CO-EVOLUTION

- When first introduced, security measures will be misunderstood & **mislabeled**. Their actual value will become clear when they are **circumvented**.
- Offensive proof-of-concept release enables **detection** of similar technique **already** in the wild.
- Security-critical features **start outside** of security domain & get understood as such only when successfully attacked.

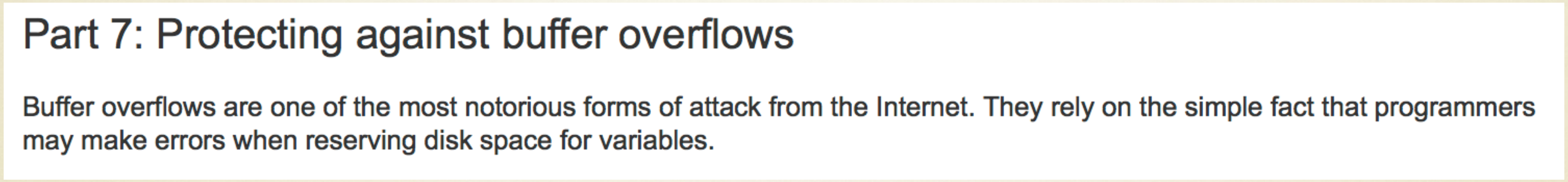
Hacker research becomes an industry standard: DEP+ASLR

- Standard function prologues and epilogues are an **automaton**, distributed through code, stack frames are its programs
  - This automaton realizes the “control flow graph”
- Programmed since Aleph One’s Phrack 49:14 ("reuse of a RET")
  - OpenWall & PaX enforce/emulate **non-executable** stacks
- Solar Designer, Tim Newsham, gera, Nergal, and others discover stack frame-chaining programming techniques, 1997-2001
  - **PaX** mitigates with **ASLR**: http://pax.grsecurity.net/docs/aslr.txt
- Known as **ROP** (named by Hovav Shacham) since 2007
- DEP+ASLR -- pioneered by **hackers**! -- become an industry standard

DEP in WinXP SP2: a "buffer overflow" protection

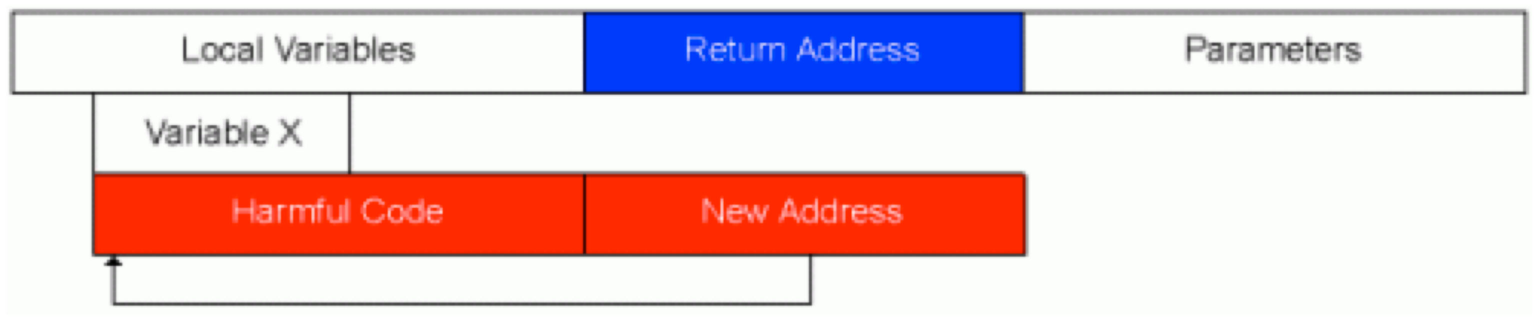


Windows XP Service Pack 2 (Part 7): Protecting against buffer overflows



Part 7: Protecting against buffer overflows

Buffer overflows are one of the most notorious forms of attack from the Internet. They rely on the simple fact that programmers may make errors when reserving disk space for variables.

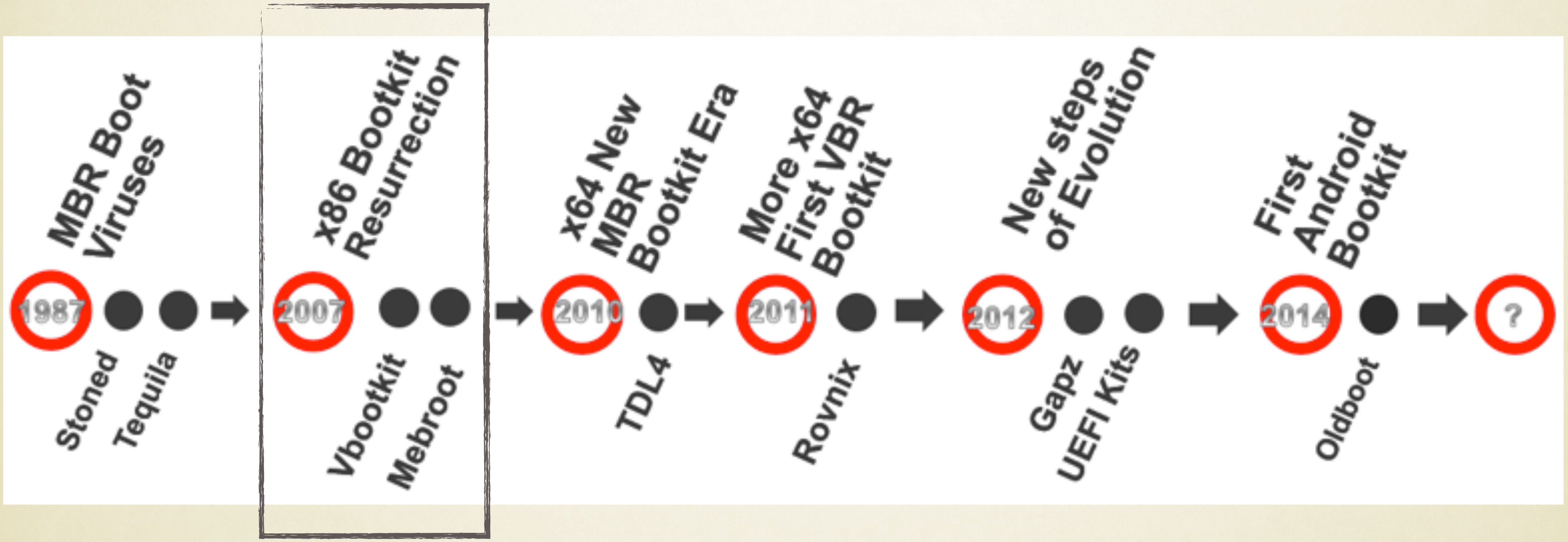


What does Data Execution Prevention do?

Data Execution Prevention (DEP) monitors programs to verify whether they are using system memory securely. To do this, DEP software, either alone or with compatible microprocessors, marks memory locations as "non-executable." If an program tries to run a code (malicious or not) from one of these protected locations, DEP closes the program and notifies you by sending a warning message.

https://support.microsoft.com/en-us/kb/889741

Return of the Bootkit



- **Vbootkit,** PoC bootkit for Vista, BlackHat USA **2007**
  - **Merbroot**, first modern bootkit in the wild, **2007**
- Vbootkit x64, Stoned, BH 2009: PoC bypass digital signature check
  - Olmarik (**TDL4**), 2010: first x64 sign. check bypass in the wild

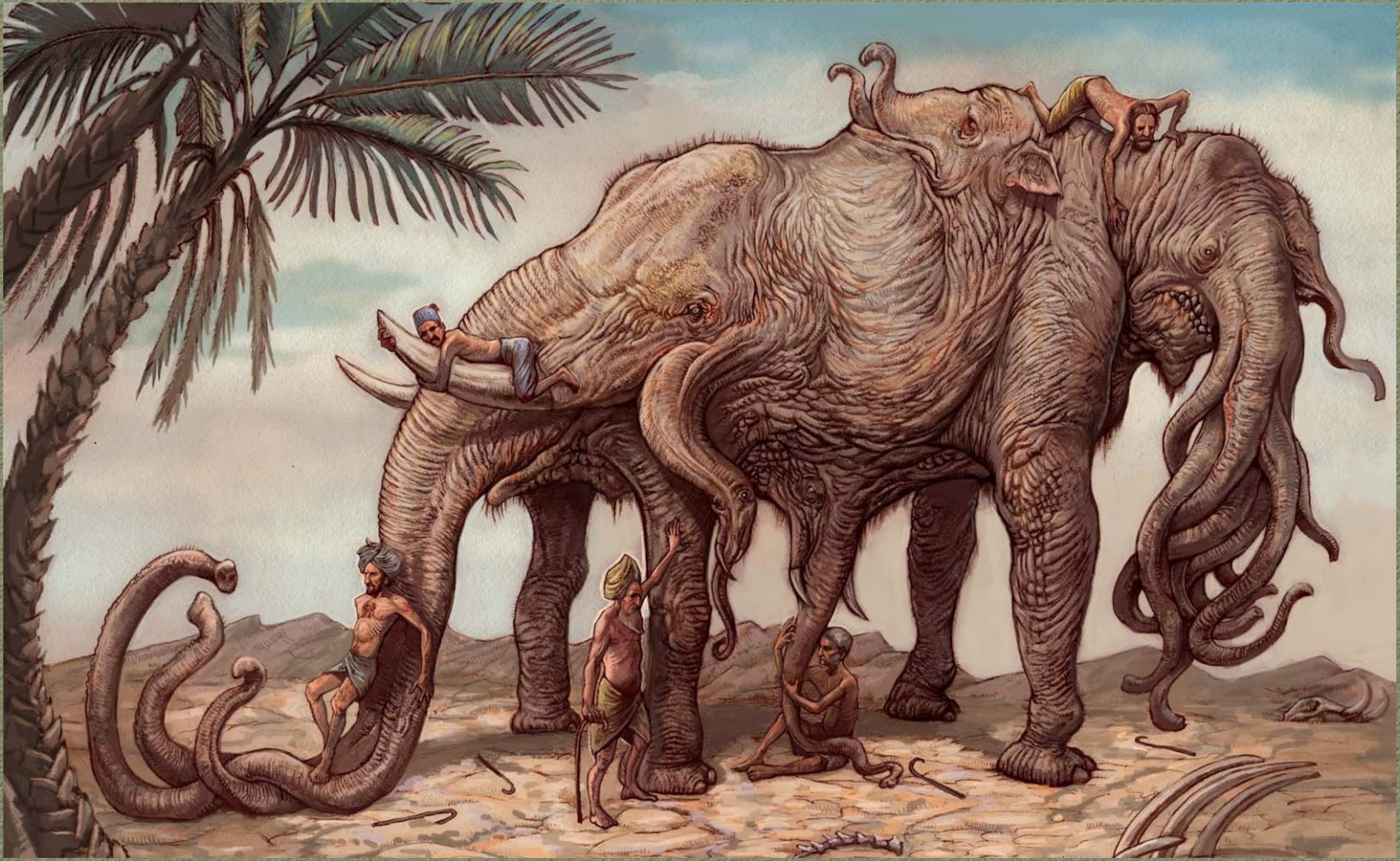
Adventures of SMM

- Security-critical features start as power management, performance, admin/inventorying, etc. - **outside** of the security domain
- Attackers expose their actual **execution models**
- Seeing these models, defenders understand the features' actual worth, and invest in (**re**)**designing** them properly
- New security models eventually result

Adventures of SMM

- Duflot, 2006: payload in SMM to bypass OpenBSD secure levels
- BSDaemon et al., Phrack 65:7, "*System Management Mode Hacks: Using SMM for Other Purposes*", 2008
- Sherry Sparks, 2008: *"SMM Rootkits: A New Breed of OS Independent Malware"*
- Filip Wecherowski, Phrack 66:11, "*A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers*", 2009
- Joanna Rutkowska, 2009, "*Attacking SMM Memory via Intel® CPU Cache Poisoning*"
- Intel's SMM for Authenticated Variables, MITRE's signed BIOS, ... -- SMM now a core element of security policy

Shape of the beast



OFFENSE DEFINES THE UNIVERSE IN WHICH DEFENSE LIVES

Programmer's model	Functions, variables
Binary representation/ runtime	"Gadgets", overwrites, out-of-type references, illegal states, ...
Physical representation	Glitches, noise, cross-talk, ...

Offense defines the universe in which defense lives

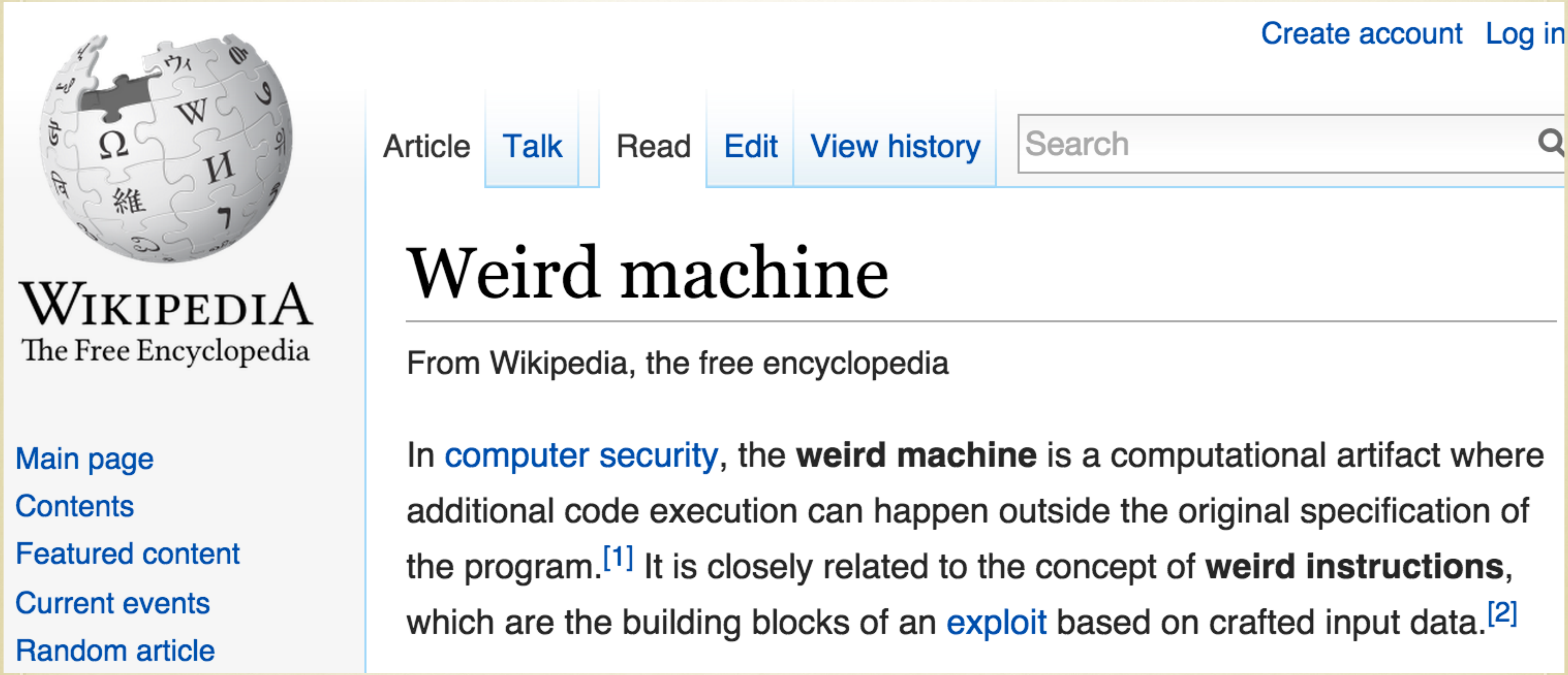
The unexpected computation the defense must counter occurs within offense's universe, on **offense's computation models** and **modes of programming**.

Are all bugs shallow? Hell No.

- "Given enough eyes, all bugs are shallow" - not until we grow eyes that can see solutions to **NP-complete** problems (or **undecidable** ones)
  - "Bugs are just bugs" - and an ISA is just an ISA :)

We don't understand a word until we can use it in a **sentence**
We don't understand a math fact until we can use it in a **proof**
We don't understand a bug or a <u>feature</u> until we can use them in an **exploit**

"WEIRD MACHINES"



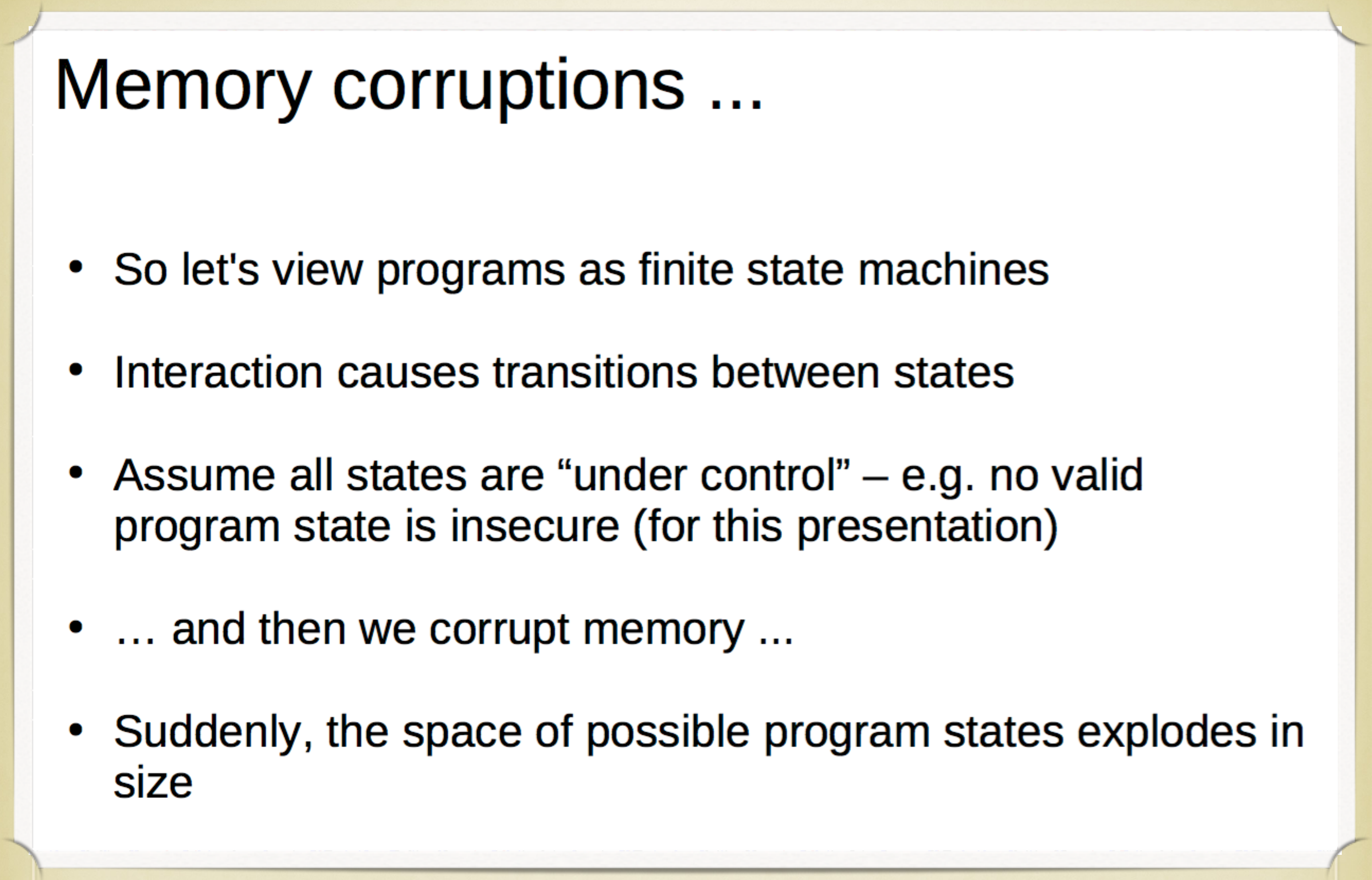
Weird machine

From Wikipedia, the free encyclopedia

In computer security, the **weird machine** is a computational artifact where additional code execution can happen outside the original specification of the program.[1] It is closely related to the concept of **weird instructions**, which are the building blocks of an exploit based on crafted input data.[2]

Exploitation is setting up, instantiating, and programming the weird machine

-Thomas Dullien, Infiltrate 2012

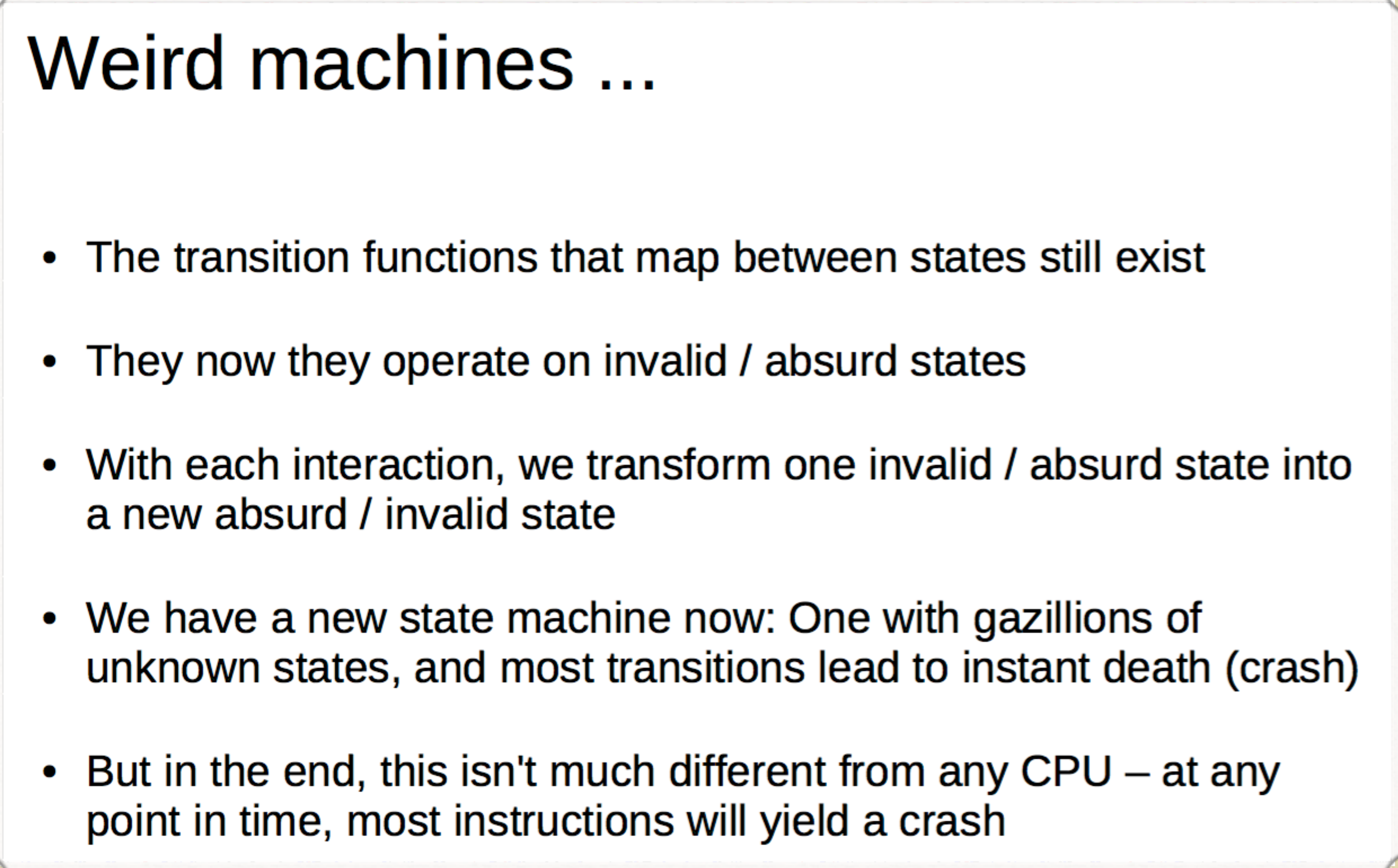


Memory corruptions ...

- So let's view programs as finite state machines
- Interaction causes transitions between states
- Assume all states are "under control" – e.g. no valid program state is insecure (for this presentation)
- … and then we corrupt memory ...
- Suddenly, the space of possible program states explodes in size

-Thomas Dullien, Infiltrate 2012

http://titanium.immunityinc.com/infiltrate/archives/Fundamentals_of_exploitation_revisited.pdf

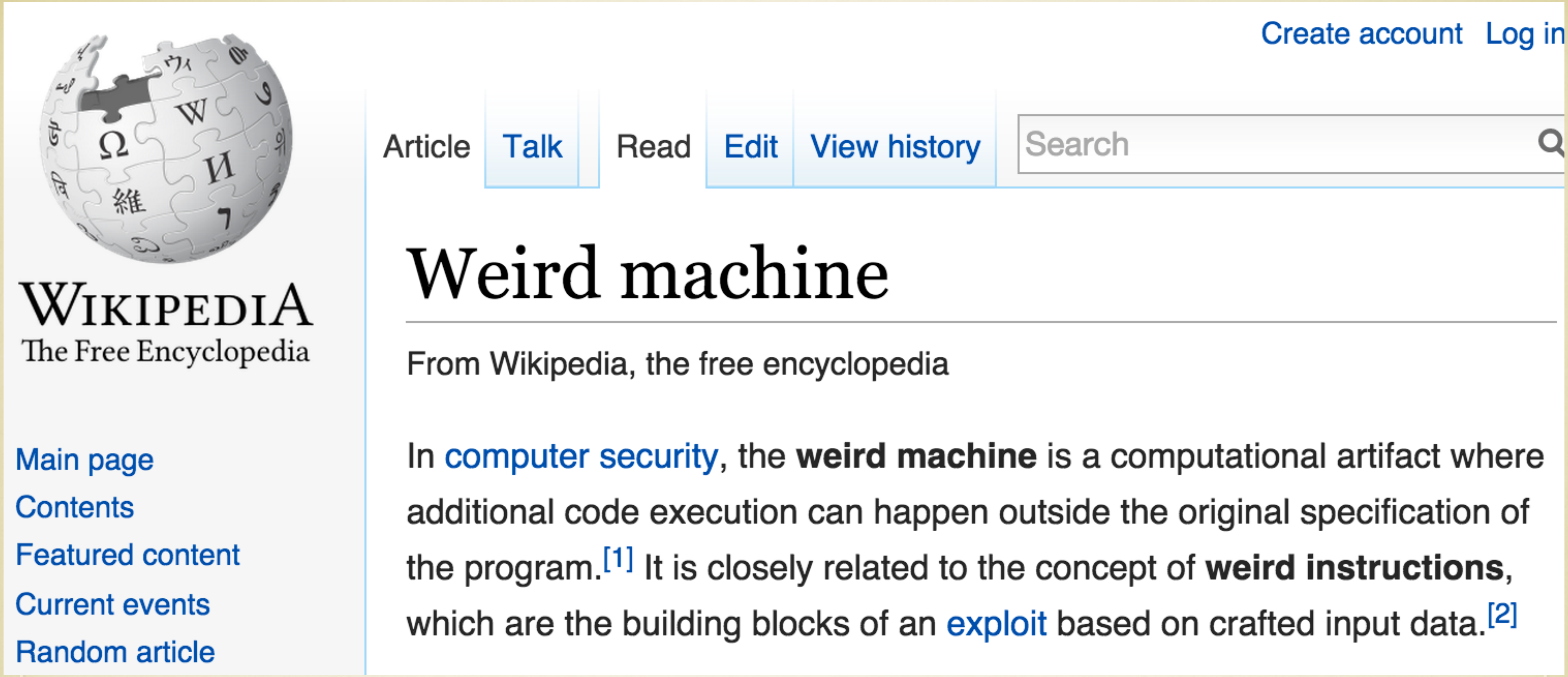


Weird machines ...

- The transition functions that map between states still exist
- They now they operate on invalid / absurd states
- With each interaction, we transform one invalid / absurd state into a new absurd / invalid state
- We have a new state machine now: One with gazillions of unknown states, and most transitions lead to instant death (crash)
- But in the end, this isn't much different from any CPU – at any point in time, most instructions will yield a crash

-Thomas Dullien, Infiltrate 2012

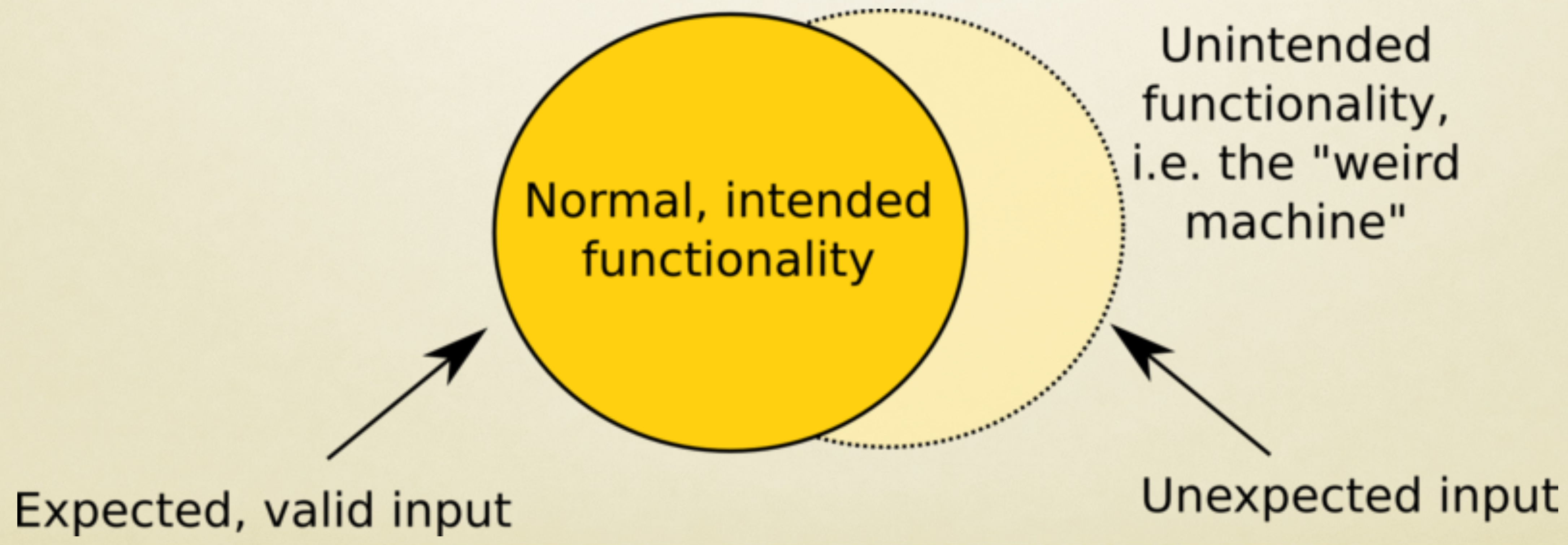
http://titanium.immunityinc.com/infiltrate/archives/ Fundamentals_of_exploitation_revisited.pdf



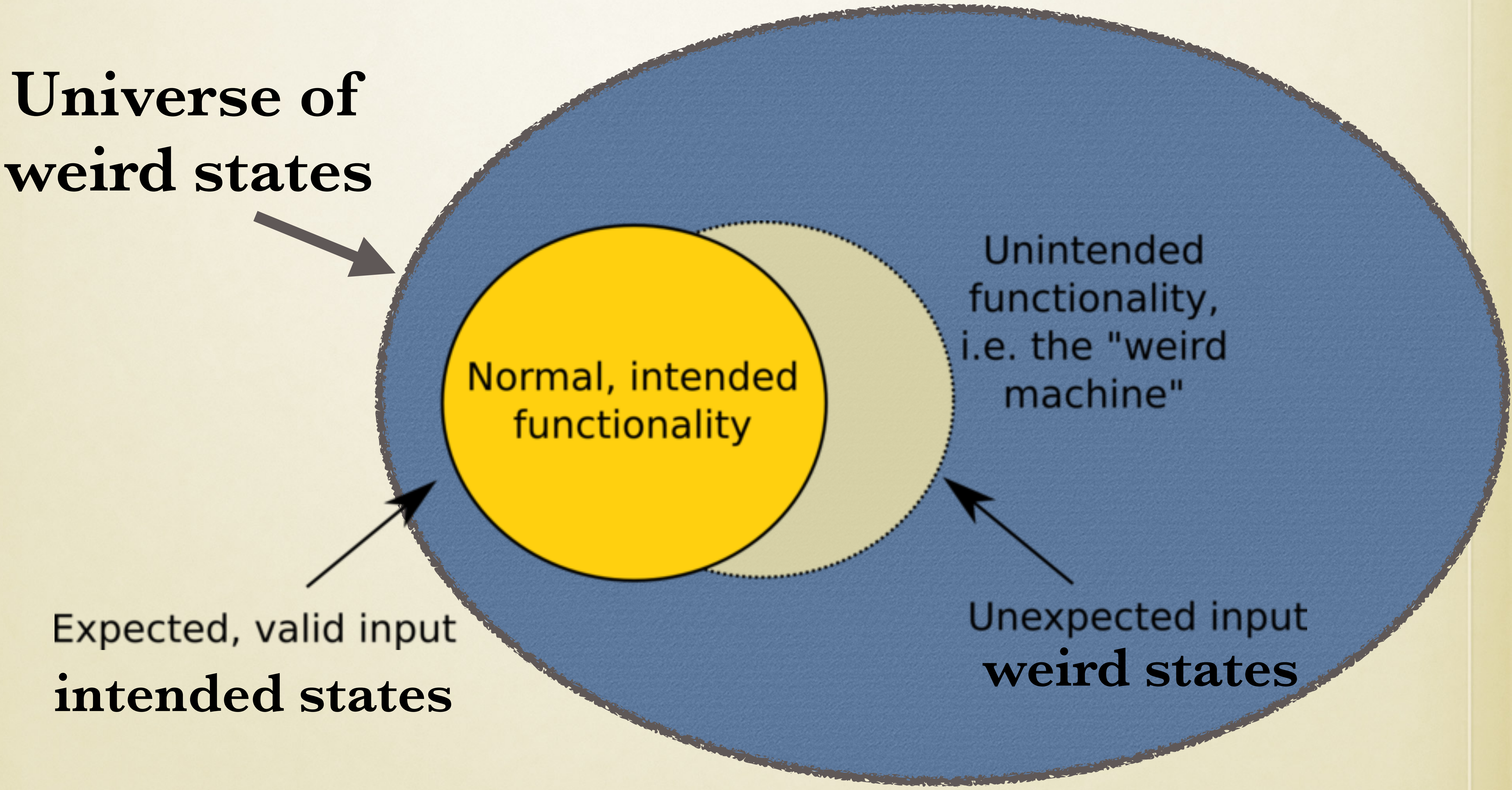
Weird machine

From Wikipedia, the free encyclopedia

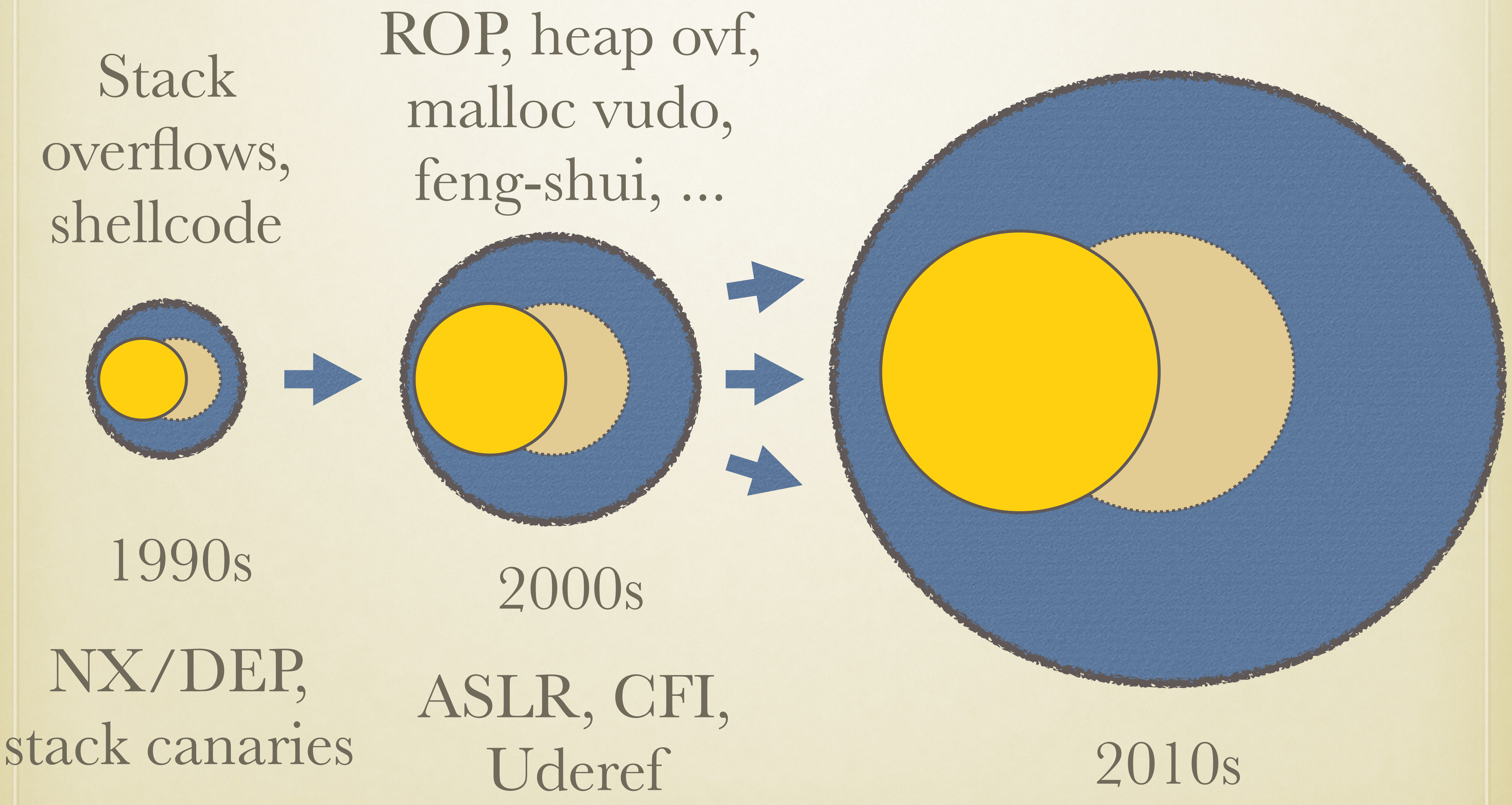
In computer security, the **weird machine** is a computational artifact where additional code execution can happen outside the original specification of the program.[1] It is closely related to the concept of **weird instructions**, which are the building blocks of an exploit based on crafted input data.[2]


Normal, intended functionality
Unintended functionality, i.e. the "weird machine"
Expected, valid input
Unexpected input

Domain of offensive computing



The universe of known "weird states" expands


Stack overflows, shellcode
ROP, heap ovf, malloc vudo, feng-shui, ...
1990s
2000s
2010s
NX/DEP, stack canaries
ASLR, CFI, Uderef

METAPHOR FOR OFFENSE:
ALGEBRA

METAPHOR FOR OFFENSE: ALGEBRA

كتاب الخوارزمي

[illegible] الشيخ الأجل ابو عبد الله

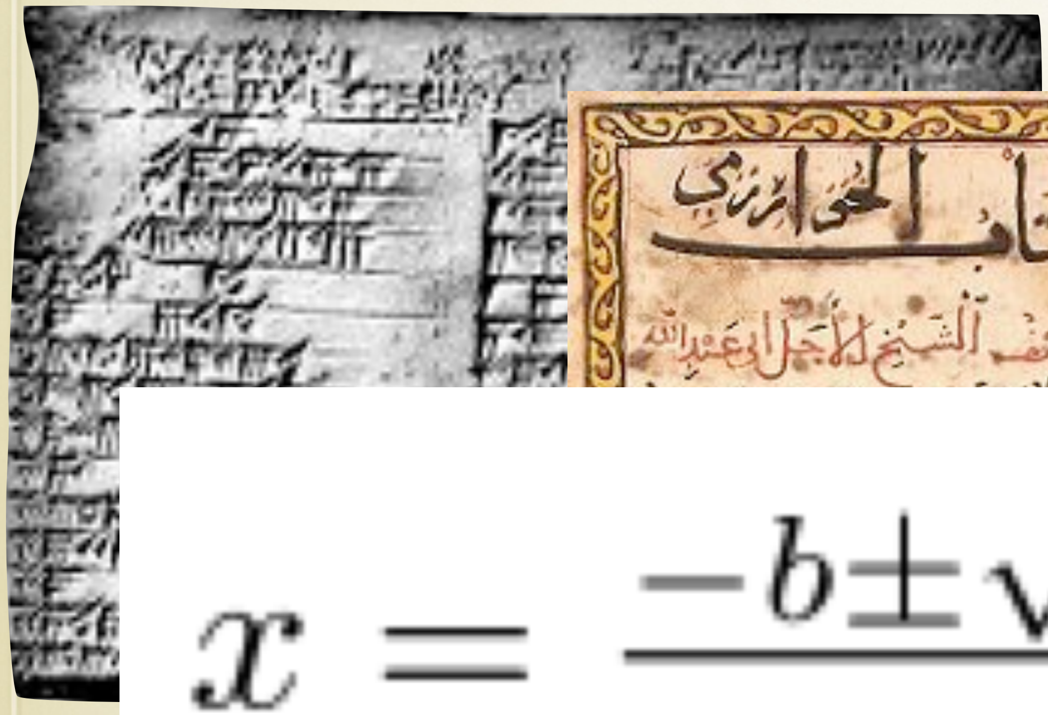
محمد بن موسى الخوارزمي رضي الله عنه [illegible]

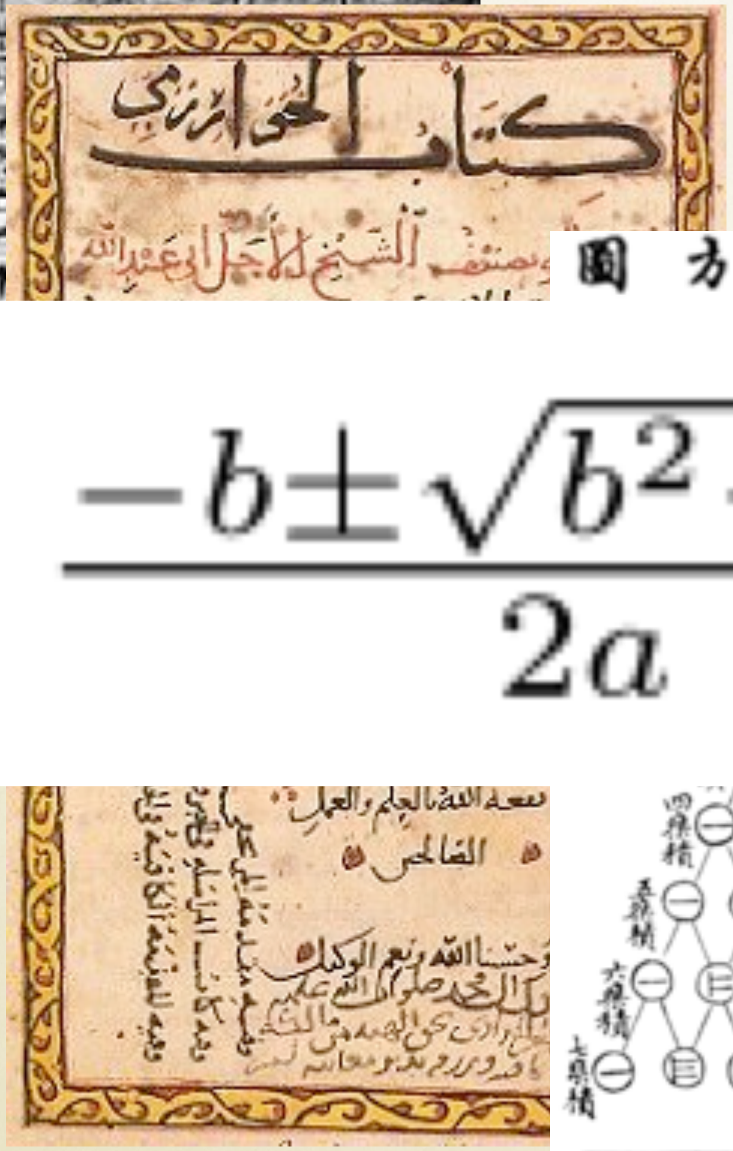
[illegible]

METAPHOR FOR OFFENSE: ALGEBRA

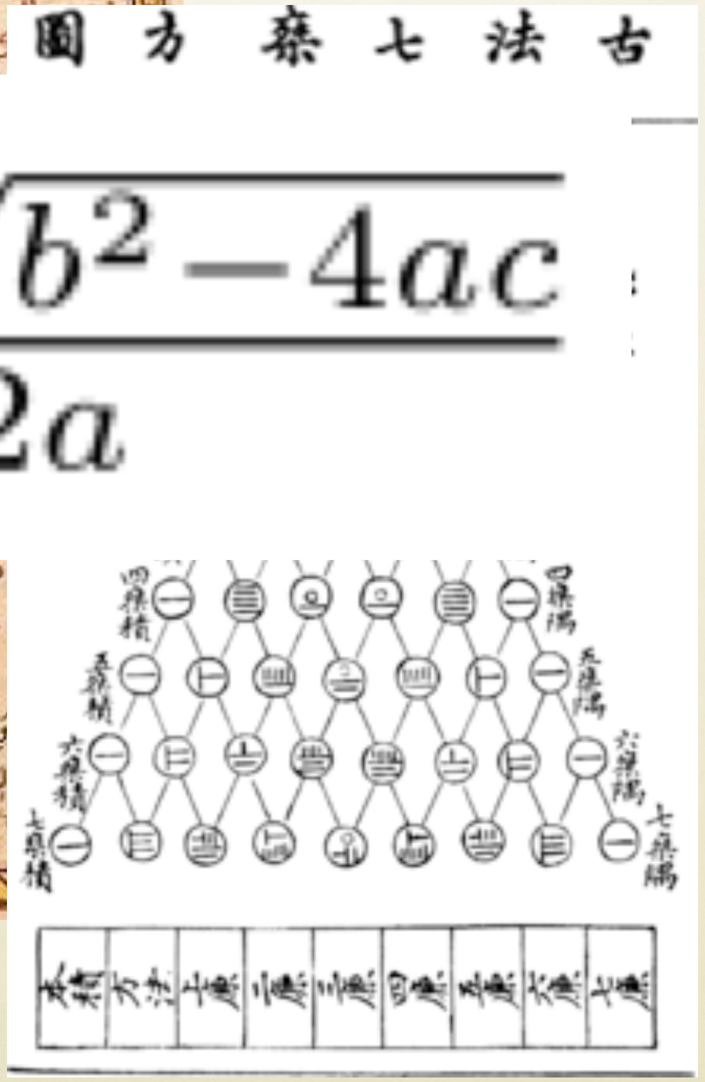
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

METAPHOR FOR OFFENSE: ALGEBRA




كتاب الخوارزمي

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$



François Viète

METAPHOR FOR OFFENSE: ALGEBRA

- Computation existed since forever
- Algebraic notation **abstracted** it, allowed us to **represent** & **combine** "all formulas" in given variables
    - "***Art of calculation on symbols***"
- Some formulas don't make sense, many can be simplified
- Offense enumerates "variables", "formulas" and properties of **weird states**

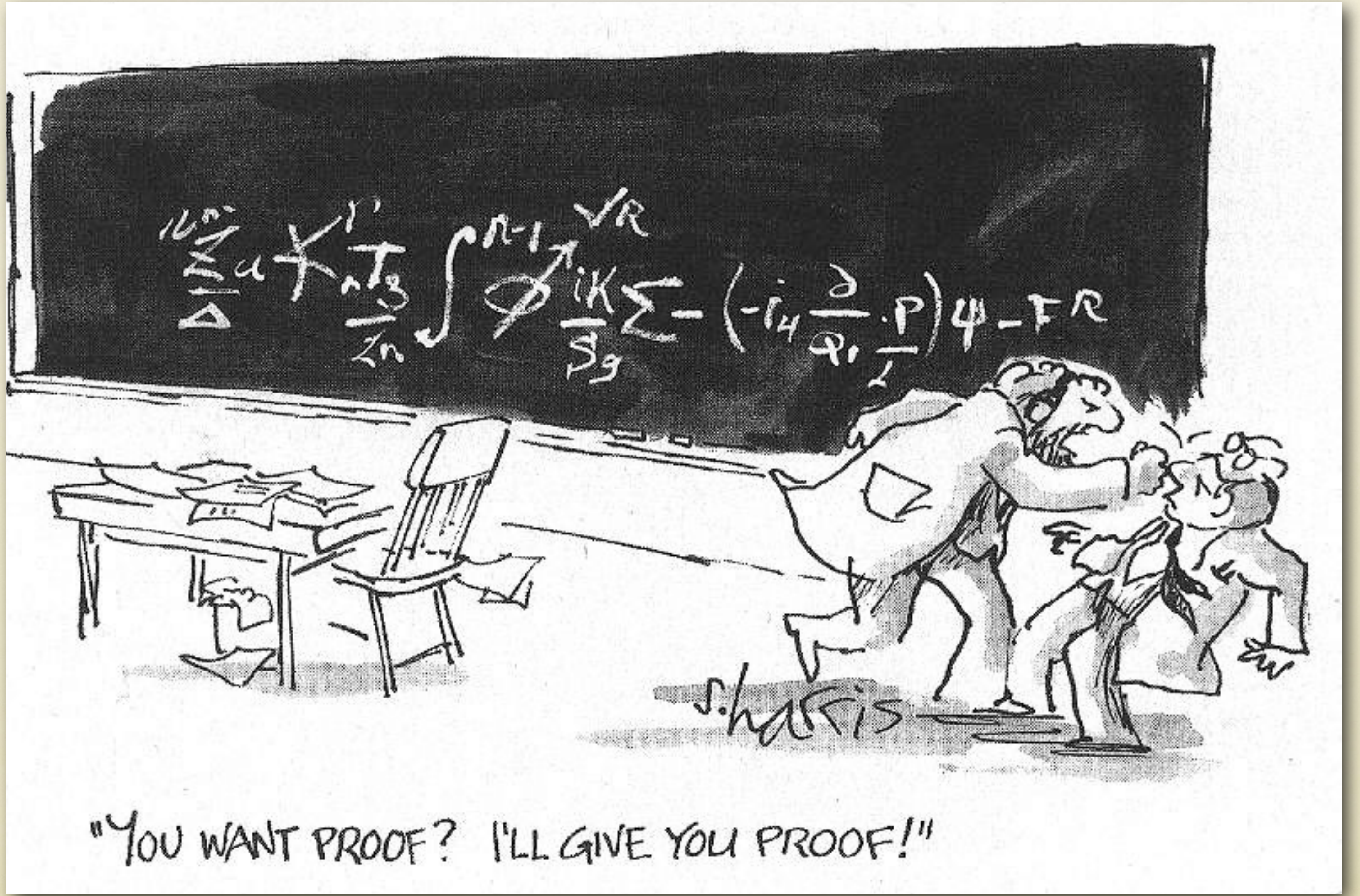
François Viète

METAPHOR FOR OFFENSE: ALGEBRA

- Effective defense needs to know & **represent all illegal** state
  - Compare with functional programmers' mantra: "***Make illegal state unrepresentable***"
- Algebra is necessary (but not sufficient) for **automation** of computations
  - It takes a lot more to **automate proofs**, but it starts with algebra

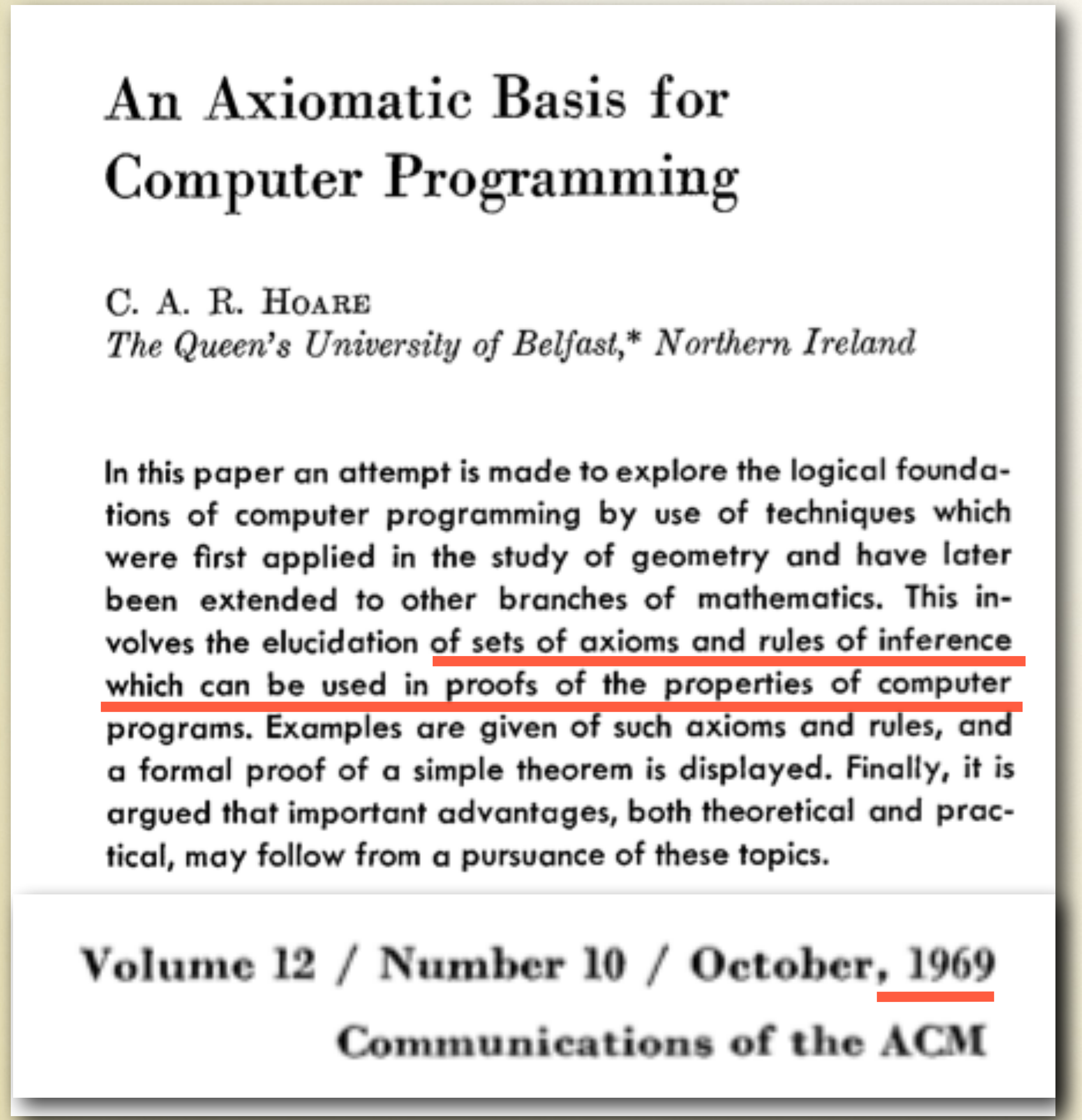
François Viète

ABOUT PROOFS



"YOU WANT PROOF? I'LL GIVE YOU PROOF!"

Program Verification



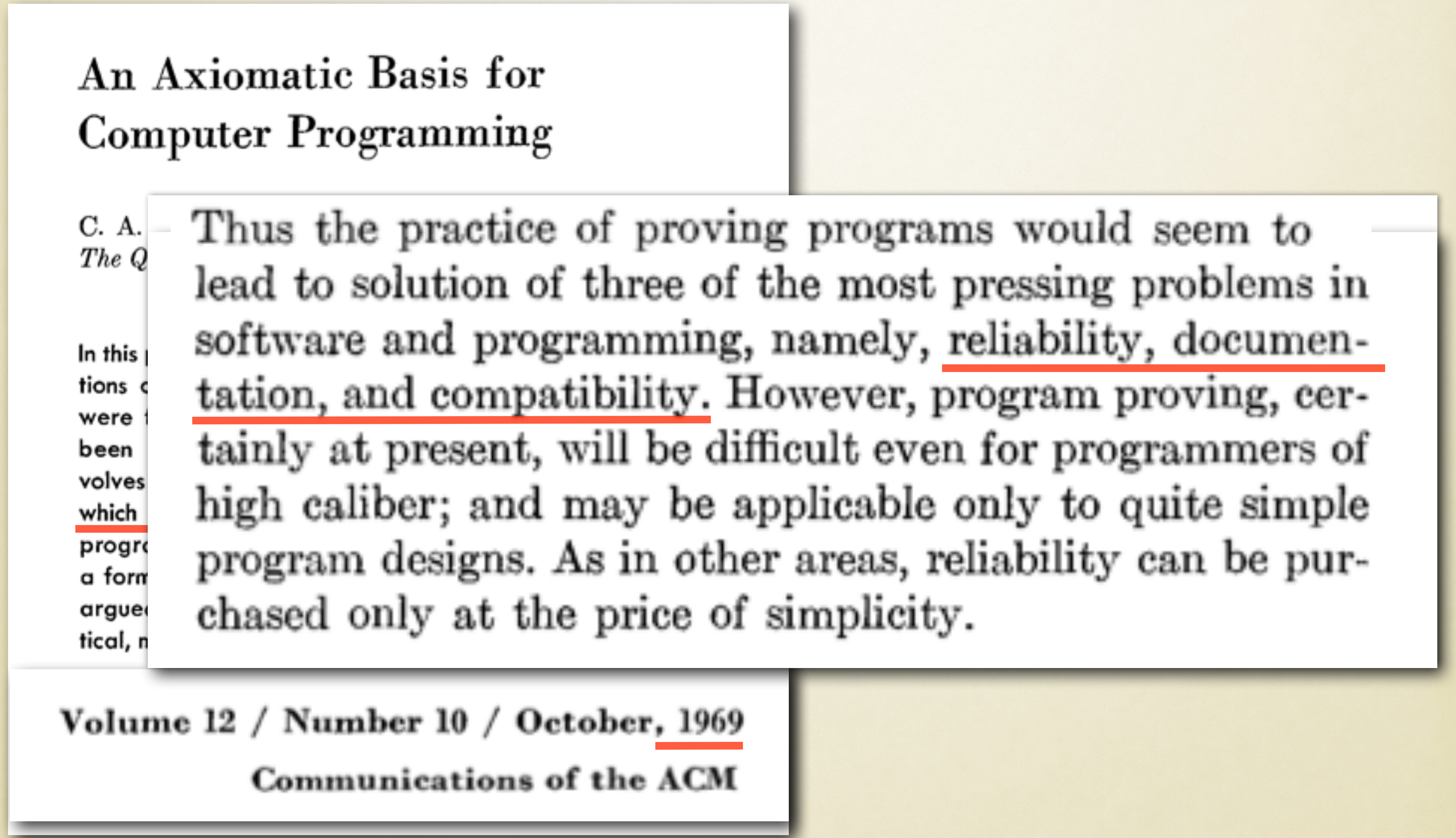
An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

Volume 12 / Number 10 / October, 1969
Communications of the ACM

Program Verification


An Axiomatic Basis for
Computer Programming

C. A.
The Q

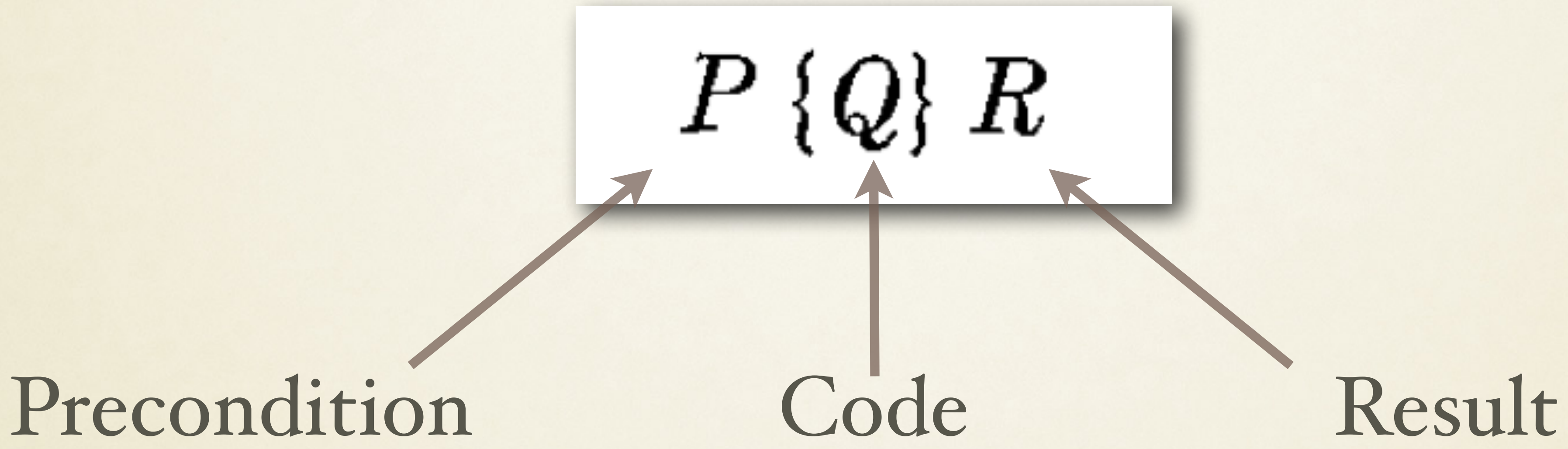
In this
tions
were
been
volves
which
progr
a form
argue
tical, n

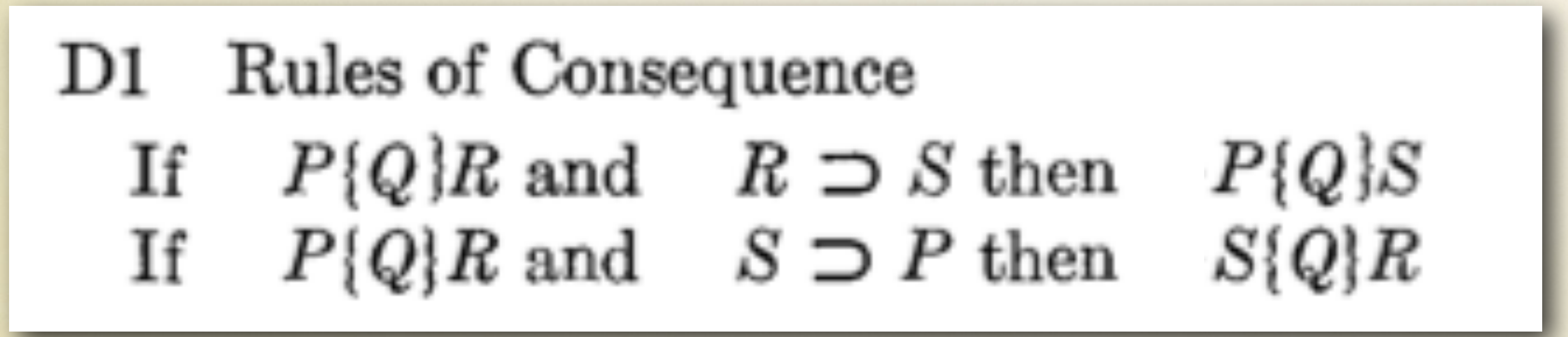
Thus the practice of proving programs would seem to lead to solution of three of the most pressing problems in software and programming, namely, reliability, documentation, and compatibility. However, program proving, certainly at present, will be difficult even for programmers of high caliber; and may be applicable only to quite simple program designs. As in other areas, reliability can be purchased only at the price of simplicity.

Volume 12 / Number 10 / October, 1969

Communications of the ACM

P { Q } R

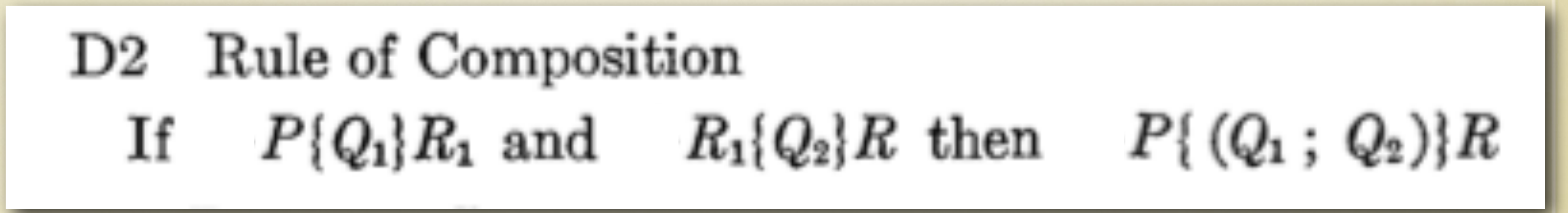




D1 Rules of Consequence

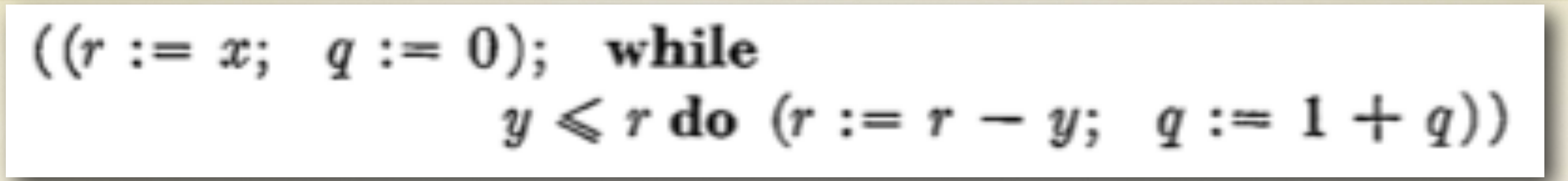
If $P\{Q\}R$ and $R \supset S$ then $P\{Q\}S$

If $P\{Q\}R$ and $S \supset P$ then $S\{Q\}R$

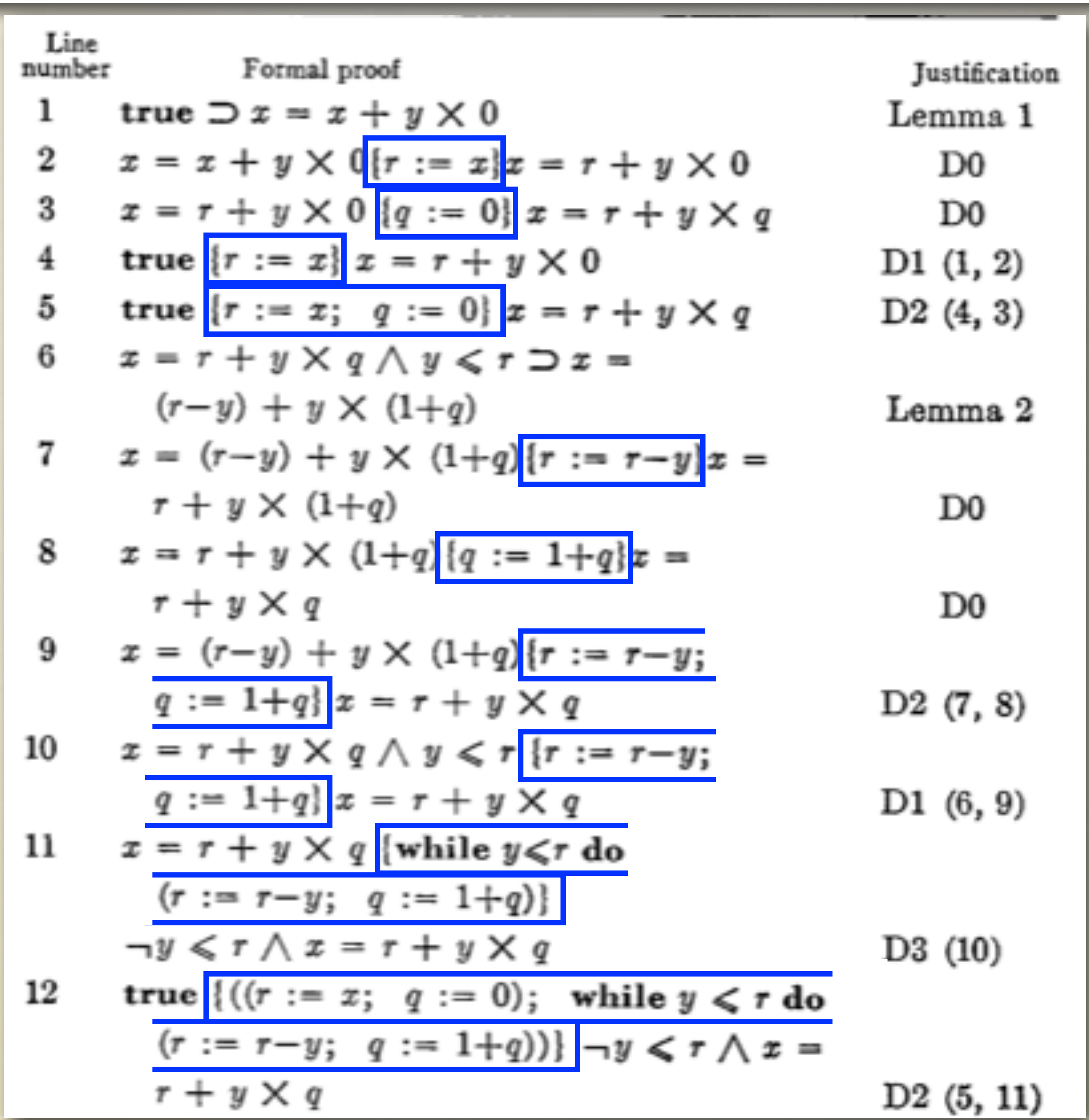


D2 Rule of Composition

If $P\{Q_1\}R_1$ and $R_1\{Q_2\}R$ then $P\{(Q_1\ ;\ Q_2)\}R$

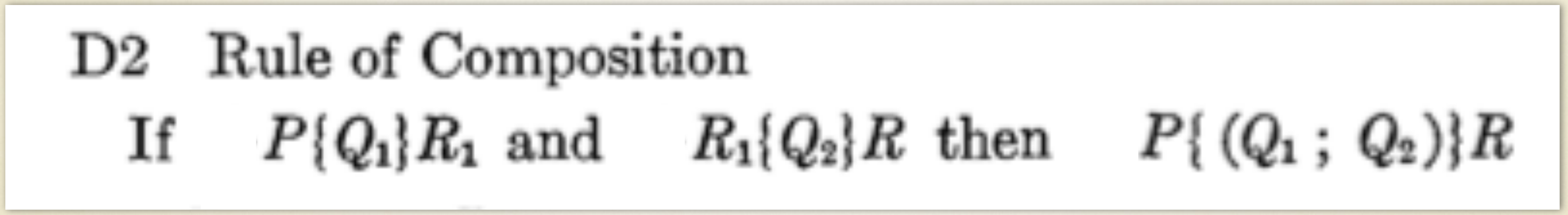


$$((r := x;\ \ q := 0);\ \ \textbf{while}\ y \leqslant r\ \textbf{do}\ (r := r - y;\ \ q := 1 + q))$$



Line number	Formal proof	Justification
1	$\textbf{true} \supset x = x + y \times 0$	Lemma 1
2	$x = x + y \times 0\ \boxed{\{r := x\}}\ x = r + y \times 0$	D0
3	$x = r + y \times 0\ \boxed{\{q := 0\}}\ x = r + y \times q$	D0
4	$\textbf{true}\ \boxed{\{r := x\}}\ x = r + y \times 0$	D1 (1, 2)
5	$\textbf{true}\ \boxed{\{r := x;\ \ q := 0\}}\ x = r + y \times q$	D2 (4, 3)
6	$x = r + y \times q \wedge y \leqslant r \supset x = (r-y) + y \times (1+q)$	Lemma 2
7	$x = (r-y) + y \times (1+q)\ \boxed{\{r := r-y\}}\ x = r + y \times (1+q)$	D0
8	$x = r + y \times (1+q)\ \boxed{\{q := 1+q\}}\ x = r + y \times q$	D0
9	$x = (r-y) + y \times (1+q)\ \boxed{\{r := r-y;}$ $\boxed{q := 1+q\}}\ x = r + y \times q$	D2 (7, 8)
10	$x = r + y \times q \wedge y \leqslant r\ \boxed{\{r := r-y;}$ $\boxed{q := 1+q\}}\ x = r + y \times q$	D1 (6, 9)
11	$x = r + y \times q\ \boxed{\{\textbf{while}\ y \leqslant r\ \textbf{do}}$ $\boxed{(r := r-y;\ \ q := 1+q)\}}$ $\neg y \leqslant r \wedge x = r + y \times q$	D3 (10)
12	$\textbf{true}\ \boxed{\{((r := x;\ \ q := 0);\ \ \textbf{while}\ y \leqslant r\ \textbf{do}}$ $\boxed{(r := r-y;\ \ q := 1+q))\}}\ \neg y \leqslant r \wedge x = r + y \times q$	D2 (5, 11)

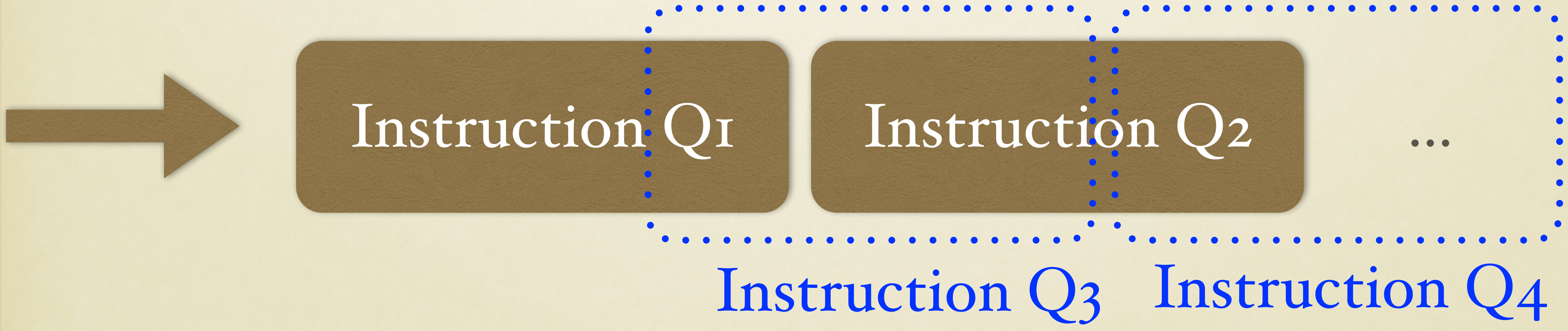
Composition is weird



D2 Rule of Composition

If $P\{Q_1\}R_1$ and $R_1\{Q_2\}R$ then $P\{(Q_1 ; Q_2)\}R$

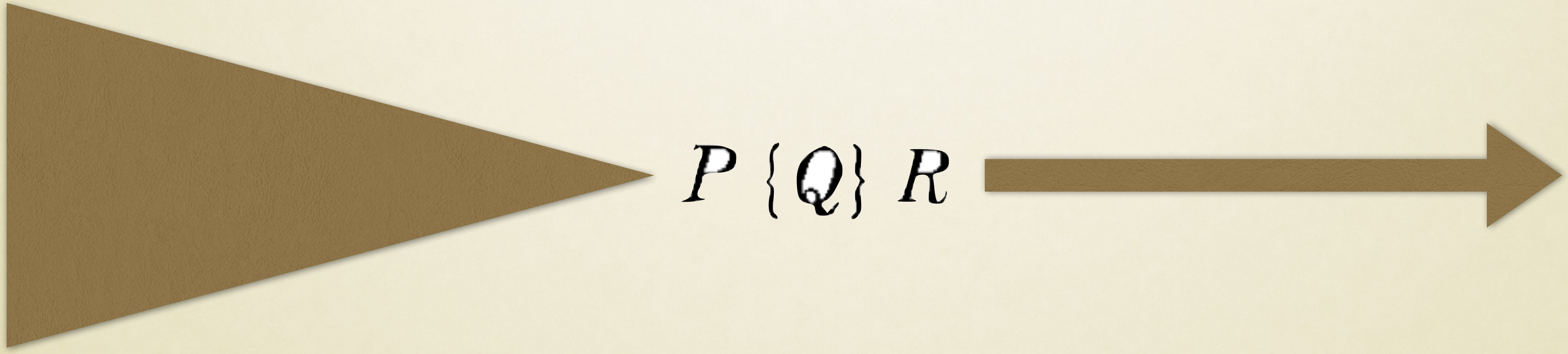
So you put together { Q1 ; Q2 }. How many programs did you actually create?



HOW MUCH CAN PROVEN CODE SURPRISE YOU?

Assume Q is proven correct, P { Q } R

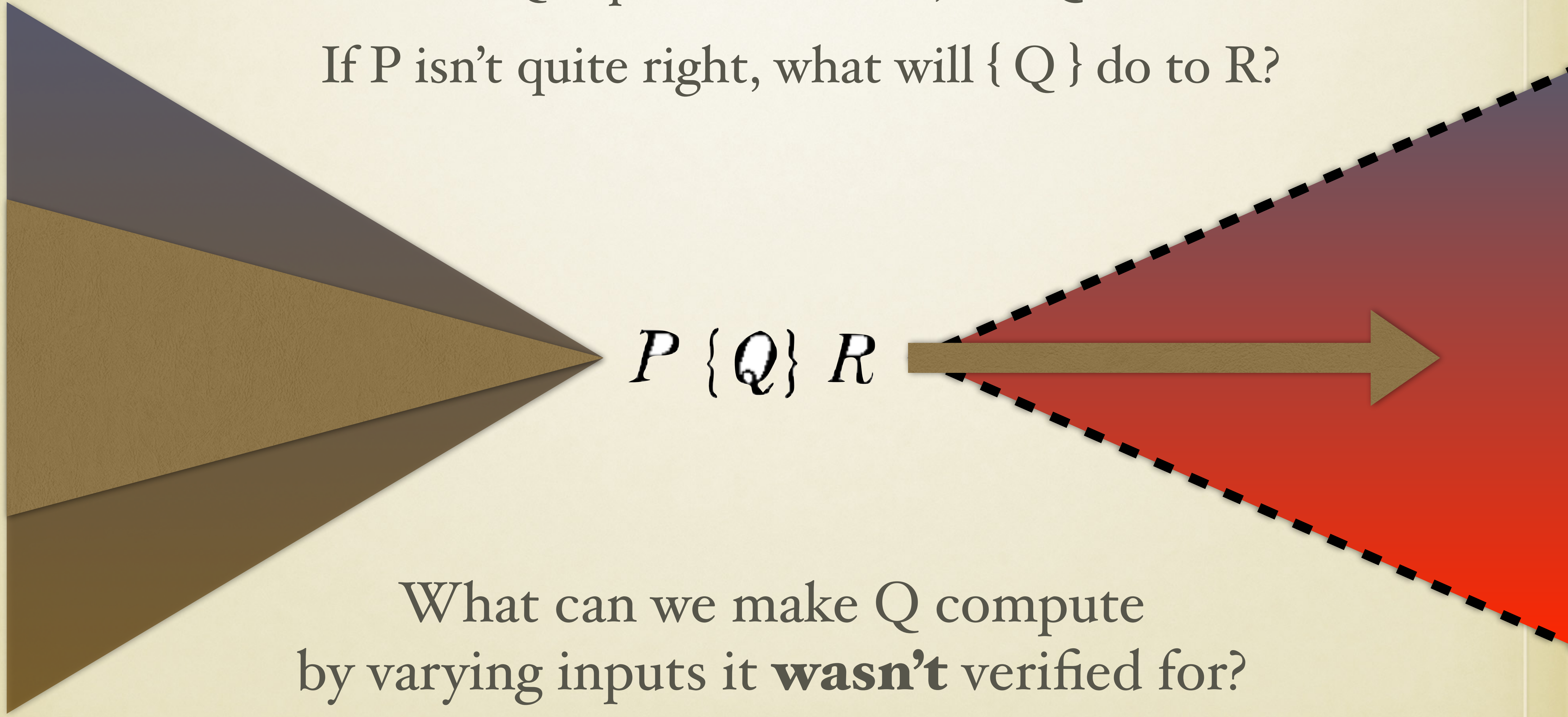
If P isn't quite right, what will { Q } do to R?



HOW MUCH CAN PROVEN CODE SURPRISE YOU?

Assume Q is proven correct, **P { Q } R**

If P isn't quite right, what will { Q } do to R?



What can we make Q compute
by varying inputs it **wasn't** verified for?

Any Input is a Program

"Everything is an interpreter"

-Greg Morrisett

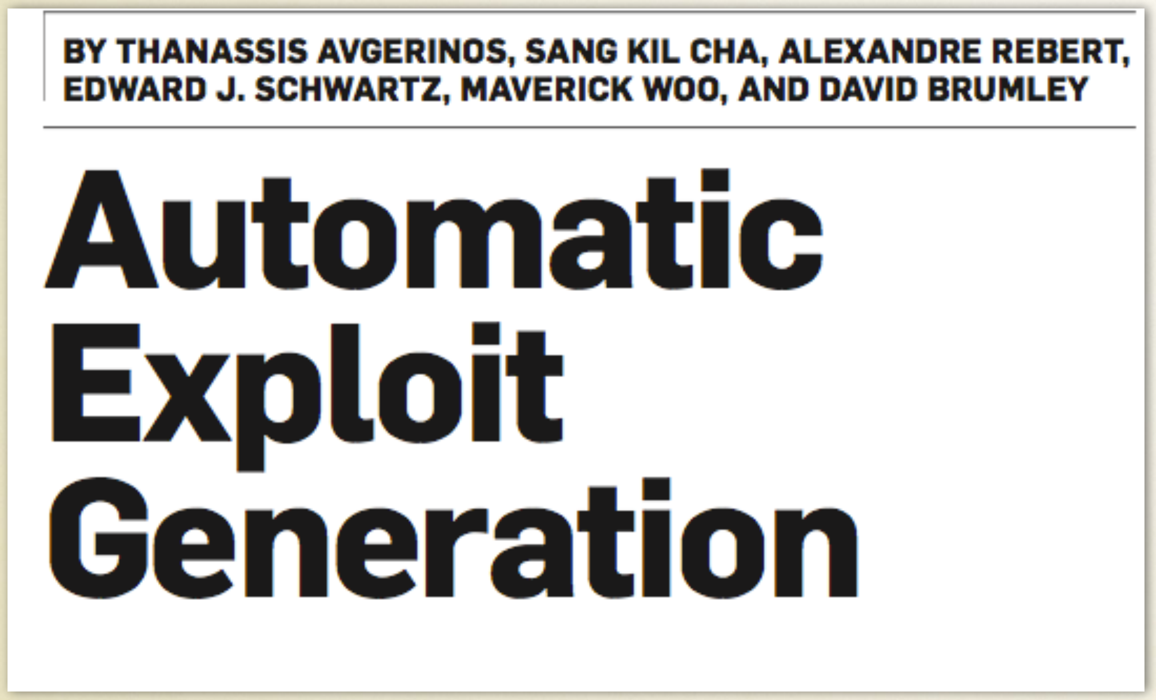
The illusion that your program is manipulating its data is powerful. But it is an illusion: The data is controlling your program.

-Taylor Hornby (@DefuseSec)

With the right inputs, "everything" is Turing-complete

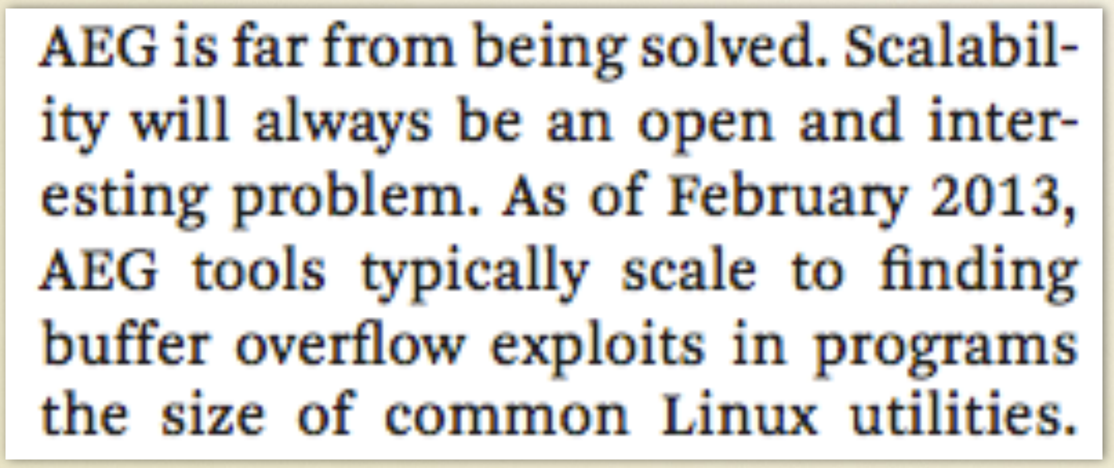
- Your ld.so ELF **loader/relocator** is Turing-complete
    - PE (*LOCREATE,* Uninformed 6:3), Mach-O are fun, too
- So is the DWARF **exception handler** helpfully linked into your C/C++ program
- So is your **x86 MMU** on x86 GDT + IDT + TSS + PTEs
- Good luck predicting effects of those inputs!

Exploitation is a program verification task!

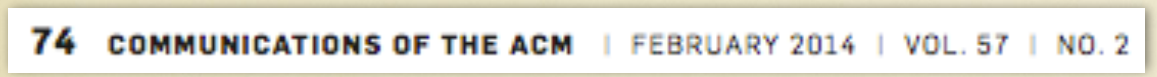


BY THANASSIS AVGERINOS, SANG KIL CHA, ALEXANDRE REBERT, EDWARD J. SCHWARTZ, MAVERICK WOO, AND DAVID BRUMLEY

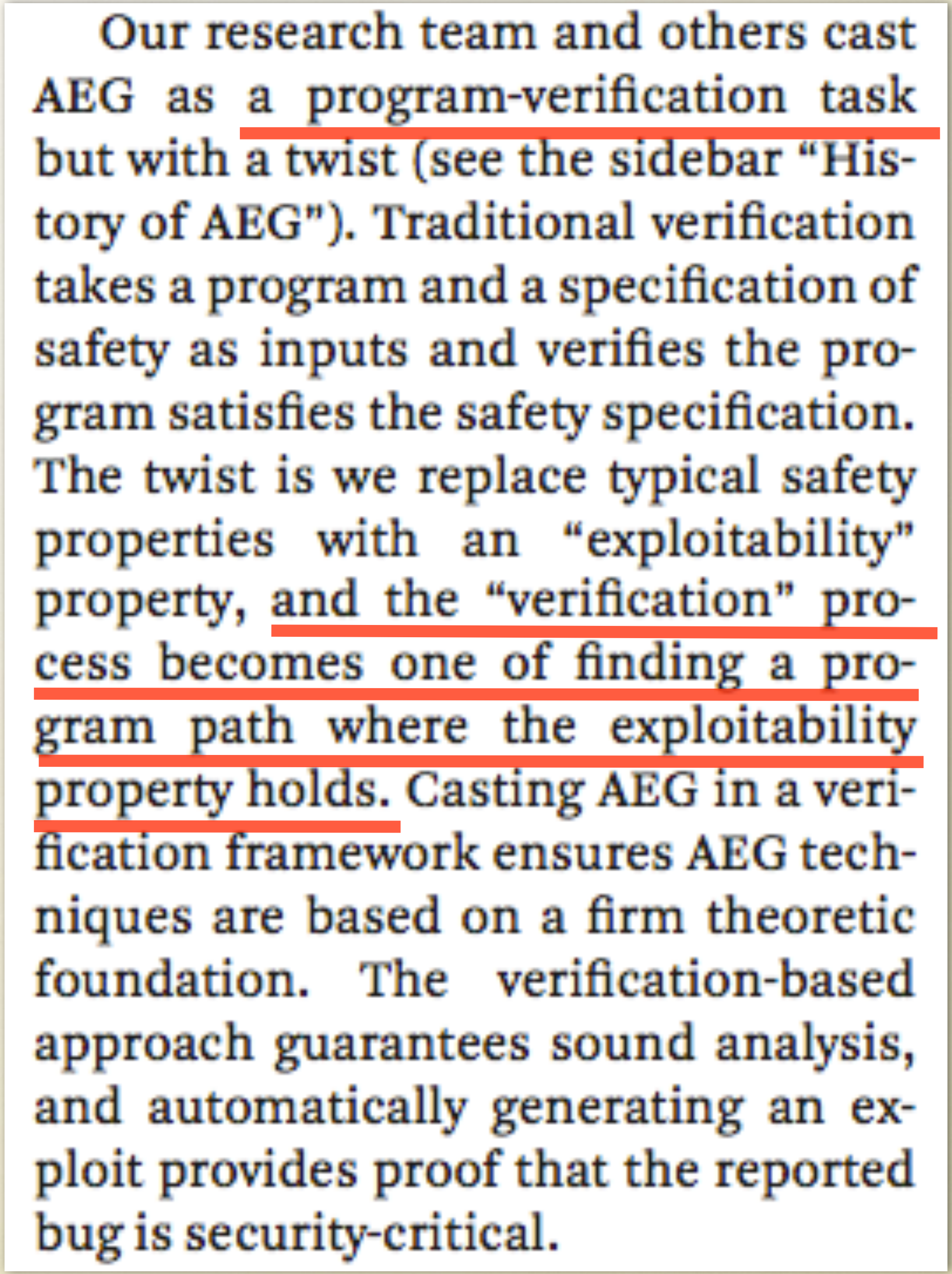
Automatic Exploit Generation



AEG is far from being solved. Scalability will always be an open and interesting problem. As of February 2013, AEG tools typically scale to finding buffer overflow exploits in programs the size of common Linux utilities.



74 COMMUNICATIONS OF THE ACM | FEBRUARY 2014 | VOL. 57 | NO. 2



Our research team and others cast AEG as a program-verification task but with a twist (see the sidebar "History of AEG"). Traditional verification takes a program and a specification of safety as inputs and verifies the program satisfies the safety specification. The twist is we replace typical safety properties with an "exploitability" property, and the "verification" process becomes one of finding a program path where the exploitability property holds. Casting AEG in a verification framework ensures AEG techniques are based on a firm theoretic foundation. The verification-based approach guarantees sound analysis, and automatically generating an exploit provides proof that the reported bug is security-critical.

Evolution?

Evolution?

Weapons control!

Weapons control!

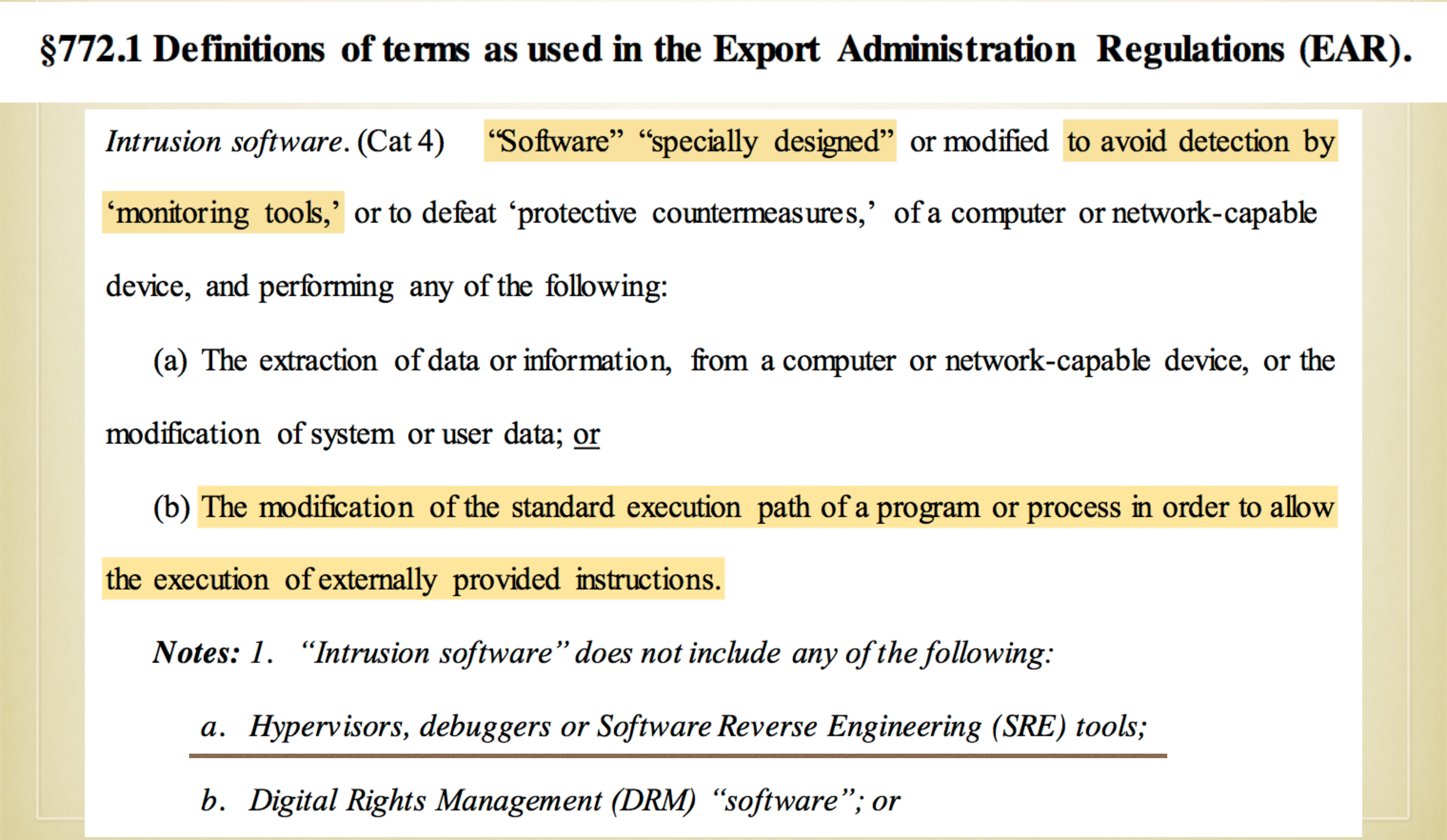
What's "Wassenaar"?

Credit: FX, "**Information Security War Room**", invited talk USENIX Security 2014

WASSENAAR ARRANGEMENT

- Wassenaar Arrangement addendum was signed Dec. 2013
    - An **arms-control** agreement (nuclear, chemical, ...)
- WA defines "***intrusion software***":
    - "...The **modification** of the **standard execution path** of a **program** or **process** in order to allow the execution of externally provided instructions..."
    - Controls means of **generating**, **developing**, **delivering**, **communicating with** "intrusion software"

"Intrusion Software"



§772.1 Definitions of terms as used in the Export Administration Regulations (EAR).

Intrusion software. (Cat 4) "Software" "specially designed" or modified to avoid detection by 'monitoring tools,' or to defeat 'protective countermeasures,' of a computer or network-capable device, and performing any of the following:

(a) The extraction of data or information, from a computer or network-capable device, or the modification of system or user data; or

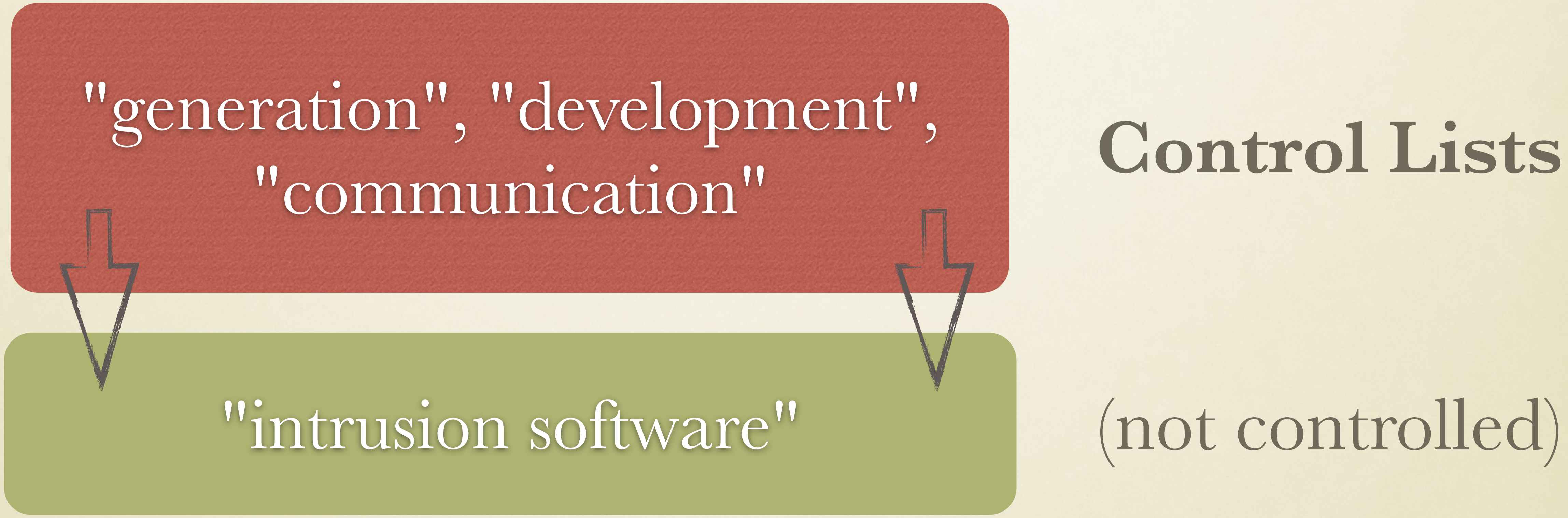
(b) The modification of the standard execution path of a program or process in order to allow the execution of externally provided instructions.

Notes: *1. "Intrusion software" does not include any of the following:*

a. Hypervisors, debuggers or Software Reverse Engineering (SRE) tools;

b. Digital Rights Management (DRM) "software"; or

WA TWO-TIER CONTROL

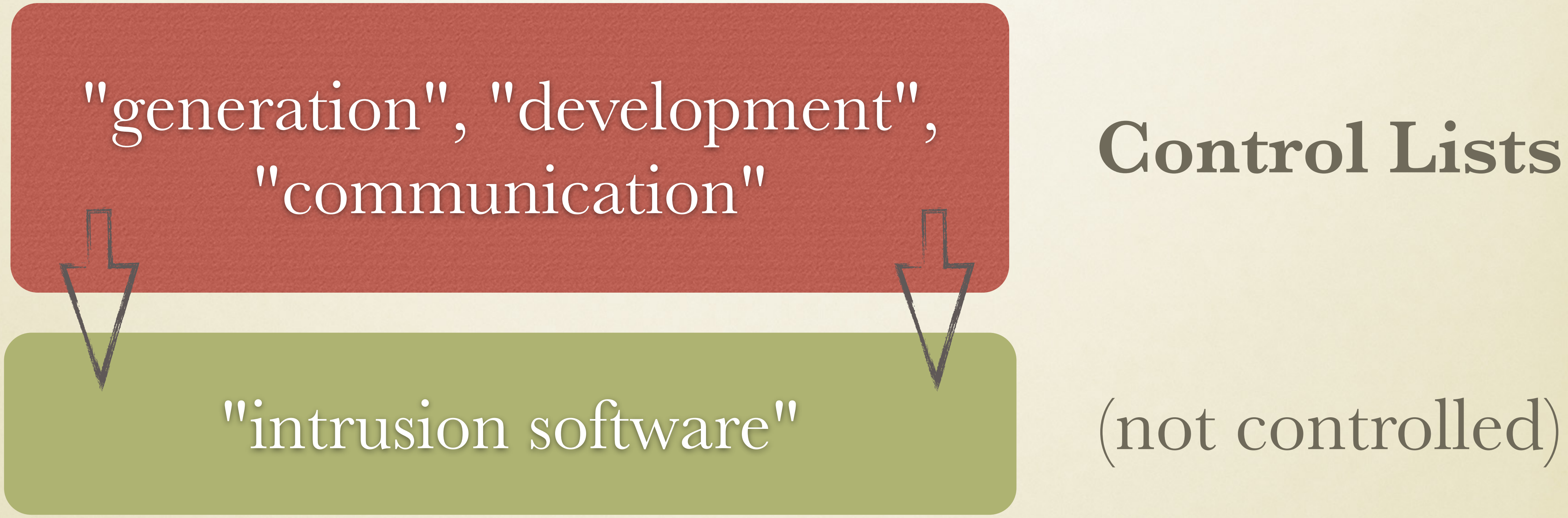


"malware samples", proof-of-concept

Exceptions: "debuggers, RE tools, hypervisors"

WA TWO-TIER CONTROL

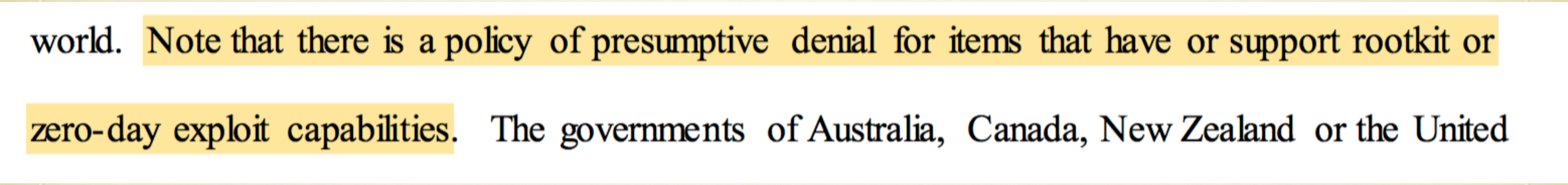
Fuzzers, frameworks, generators, AEG?
Compilers, instrumentation, analyzers?



"malware samples", proof-of-concept

Exceptions: "debuggers, RE tools, hypervisors"

New in BIS rules: "Default deny"

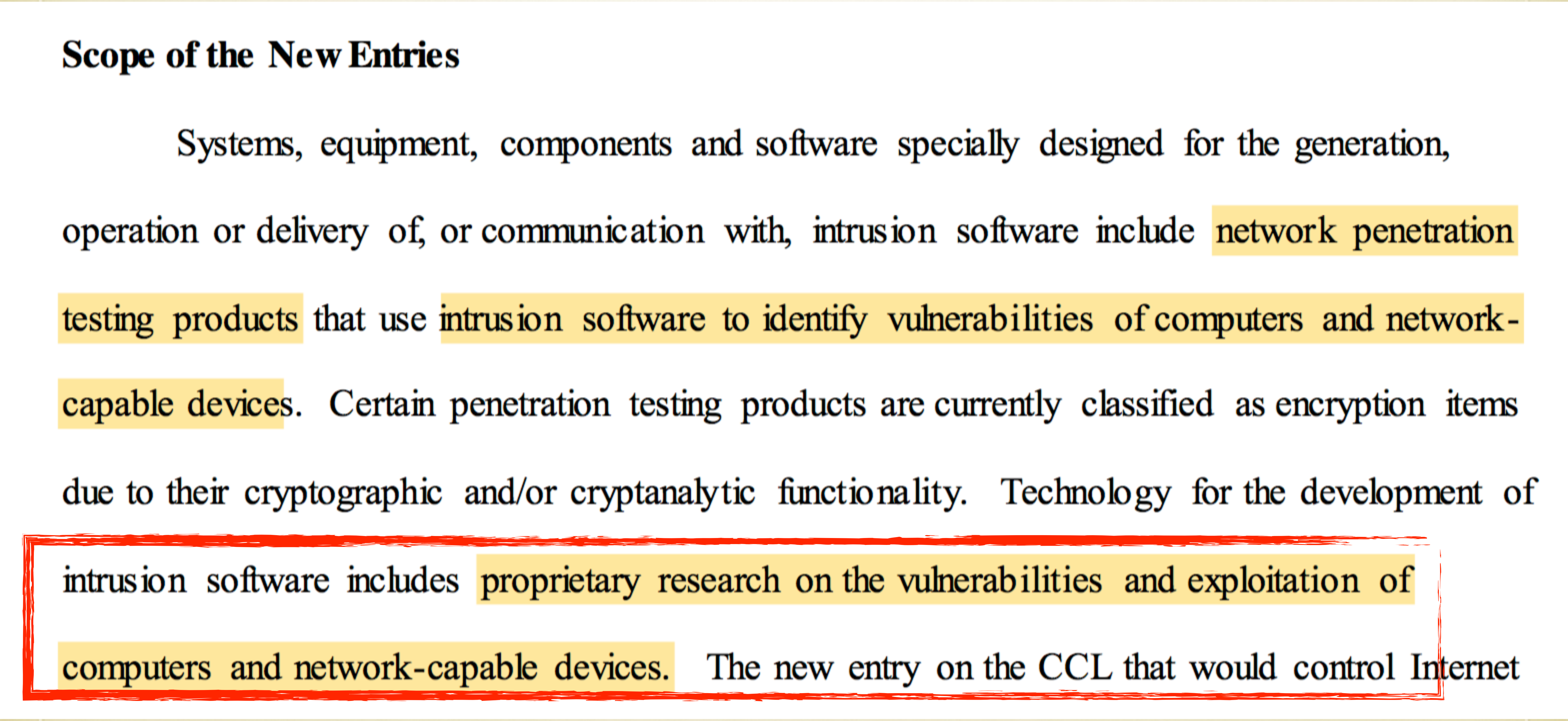


world. Note that there is a policy of presumptive denial for items that have or support rootkit or zero-day exploit capabilities. The governments of Australia, Canada, New Zealand or the United

BIS so far did not define how "zero-day exploit" is different from just an "exploit"!

BIS so far did not define "rootkit"

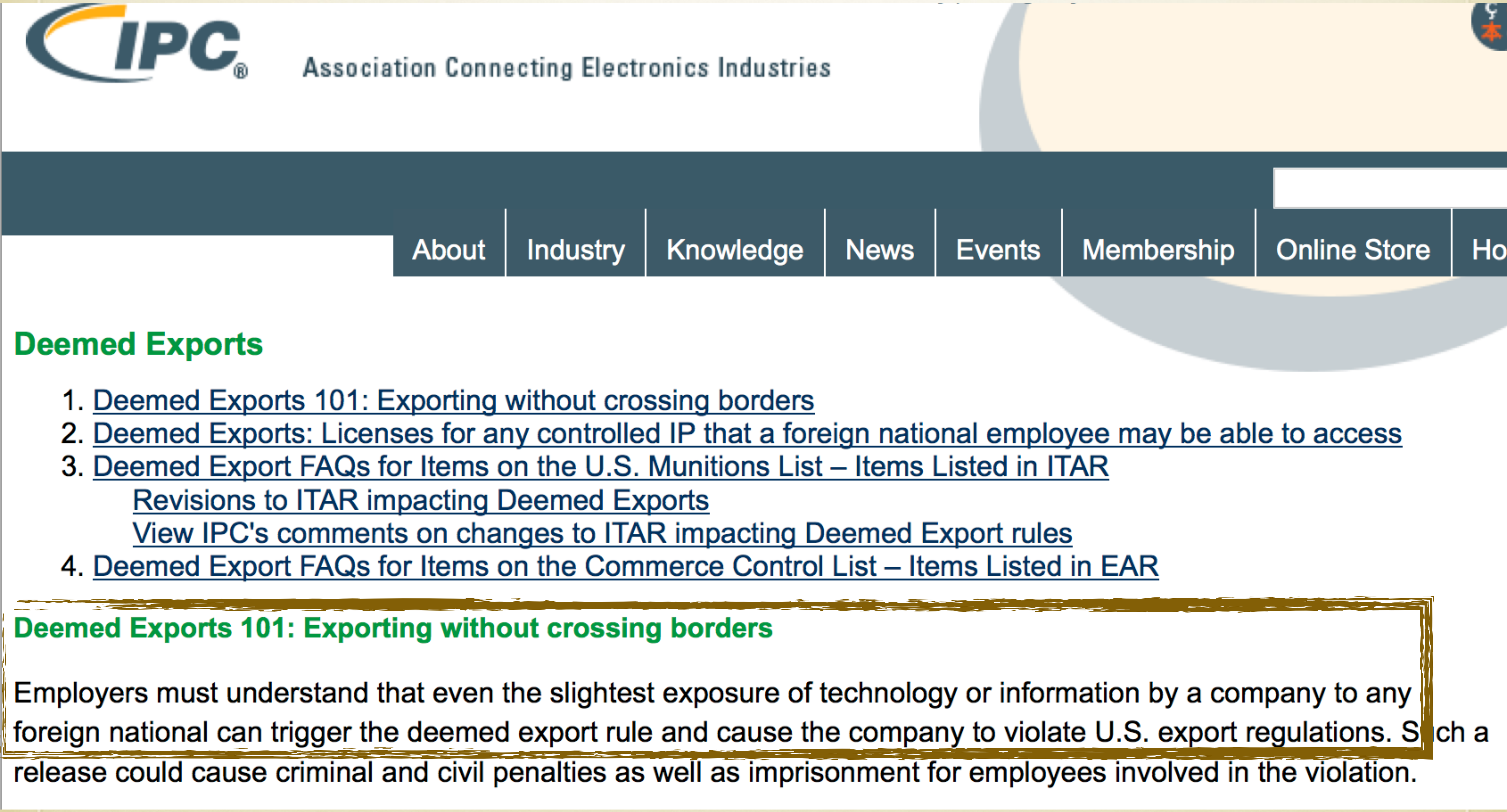
NEW IN BIS RULES: "PROPRIETARY RESEARCH"



Scope of the New Entries

Systems, equipment, components and software specially designed for the generation, operation or delivery of, or communication with, intrusion software include network penetration testing products that use intrusion software to identify vulnerabilities of computers and network-capable devices. Certain penetration testing products are currently classified as encryption items due to their cryptographic and/or cryptanalytic functionality. Technology for the development of intrusion software includes proprietary research on the vulnerabilities and exploitation of computers and network-capable devices. The new entry on the CCL that would control Internet

No Conversations => no Research



Deemed Exports

1. Deemed Exports 101: Exporting without crossing borders
2. Deemed Exports: Licenses for any controlled IP that a foreign national employee may be able to access
3. Deemed Export FAQs for Items on the U.S. Munitions List – Items Listed in ITAR
   - Revisions to ITAR impacting Deemed Exports
   - View IPC's comments on changes to ITAR impacting Deemed Export rules
4. Deemed Export FAQs for Items on the Commerce Control List – Items Listed in EAR

Deemed Exports 101: Exporting without crossing borders

Employers must understand that even the slightest exposure of technology or information by a company to any foreign national can trigger the deemed export rule and cause the company to violate U.S. export regulations. Such a release could cause criminal and civil penalties as well as imprisonment for employees involved in the violation.

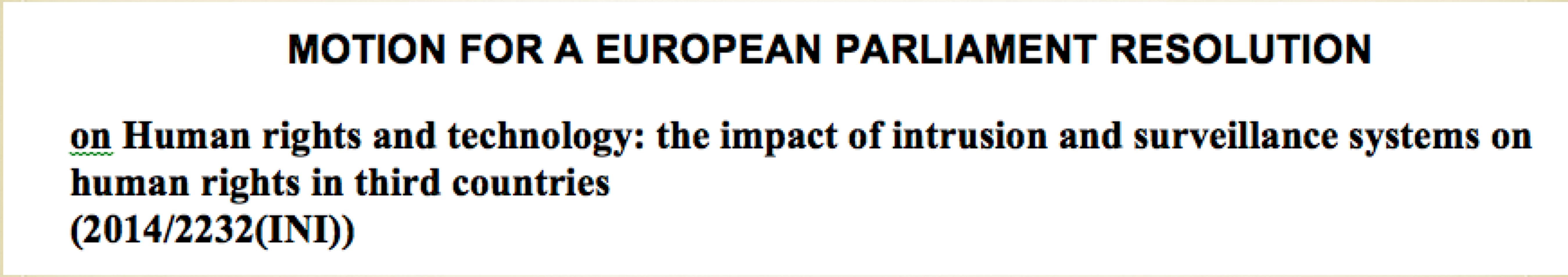
Public Comment: There's Still Time

- We have till **July 20**
- Please submit a **comment** on how this may or will affect you!
- If you are not sure how, please contact me: sergey@cs.dartmouth.edu, @sergeybratus

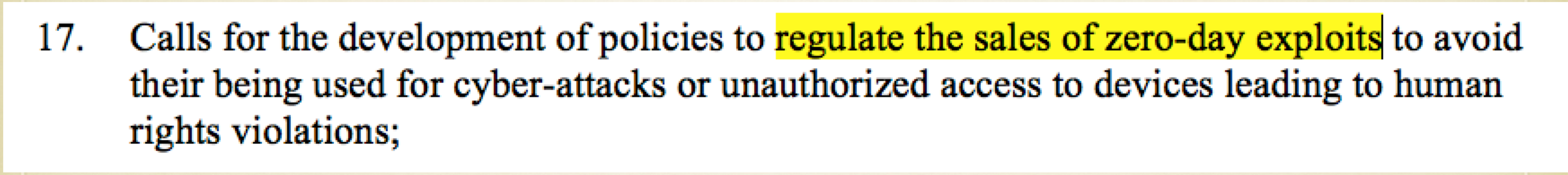
Chilling Automation

- We progress by writing programs that **write programs**
  - Binary > Asm > Compilers > Templates > Analyzers > ...
- **Automating** program analysis is rapidly developing
  - SAT/SMT solvers, abstract interpretation, symbolic execution can & should become mainstream
- WA's **generation** & **development** clauses will **chill** this path significantly

WHAT IS "ZERO-DAY"?

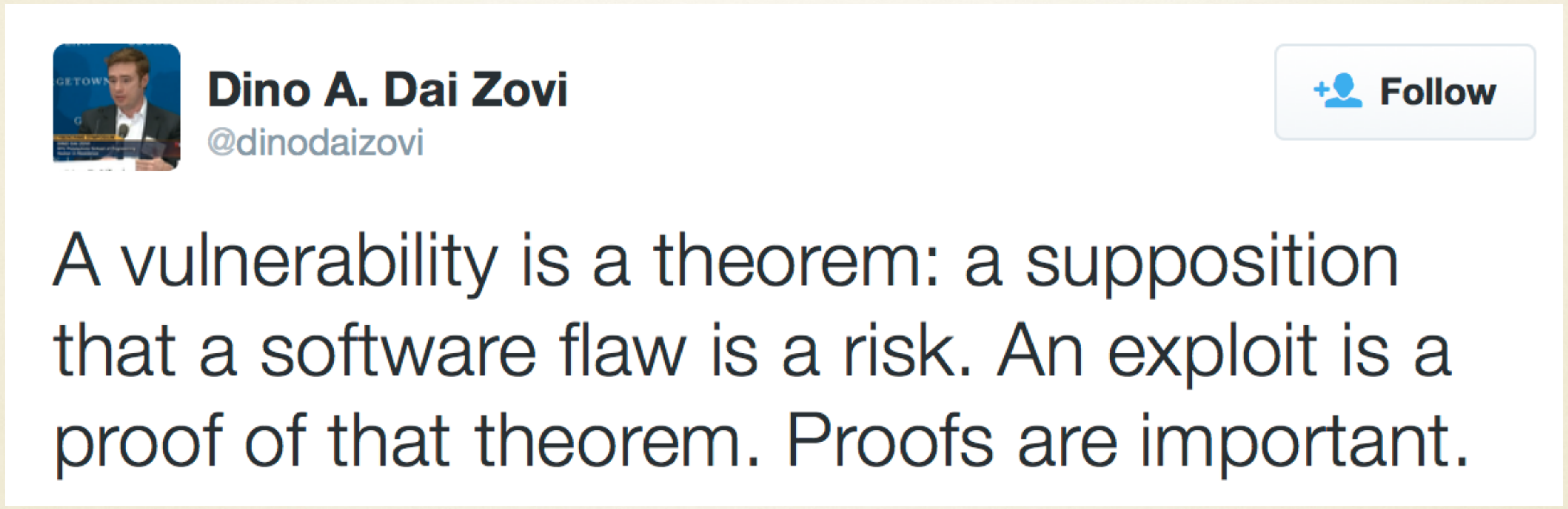

MOTION FOR A EUROPEAN PARLIAMENT RESOLUTION

on Human rights and technology: the impact of intrusion and surveillance systems on human rights in third countries (2014/2232(INI))


17. Calls for the development of policies to regulate the sales of zero-day exploits to avoid their being used for cyber-attacks or unauthorized access to devices leading to human rights violations;

- "Zero-day" means new, novel. Exploit means proof.
- In science, only "zero-day" results are worth pursuing & publishing. **All science is about "zero-day"!**

Exploit = experimental evidence


Dino A. Dai Zovi
@dinodaizovi

A vulnerability is a theorem: a supposition that a software flaw is a risk. An exploit is a proof of that theorem. Proofs are important.

- In absence of hard evidence, hypothetical risks are either ignored or everything is "critical" (= no priority)
- Exploit is evidence that **a phenomenon is real**, same as a physical experiment in Physics, Chemistry, ...
  - Would Physics or any technology succeed without **ubiquitous** lab experiments?

Thank you!