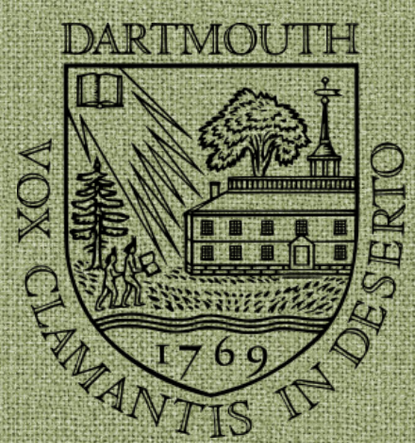
Overlooked Foundations: Exploits as Experiments and Constructive Proofs in the Science-of-Security

Sergey Bratus
(with Anna Shubina)



ISTS

Science-of-Security?

SoK: Science, Security, and the Elusive Goal of Security as a Scientific Pursuit

Cormac Herley
Microsoft Research, Redmond, WA, USA
cormac@microsoft.com

P.C. van Oorschot
Carleton University, Ottawa, ON, Canada
paulv@scs.carleton.ca

- Are programs & security a matter of **logical predicates**?
- Where is SoS on spectra of **pure** vs **applied**, "**cook-booky**" vs **basic**?
- Is security special w.r.t. to philosophical views of **falsifiability**, **deduction** vs **induction**, **predictive** power?

> At the risk of laboring the point, recall Einstein:
>
> *As far as the laws of Mathematics refer to reality they are not certain, and as far as they are certain they do not refer to reality.*

Exploits: ignored fundamentals of SoS

- Whatever Science of Security is, **exploitation** is fundamental to it:
  - **Precise** & **certain**, to math's standards
    - Up to best **evidentiary** standards of established natural sciences
- Yet had least "pure" research attention, w.r.t. "most **general theories**, covering the **widest** possible range of phenomena" (C.A.R. Hoare, 2009)

C.A.R. Hoare, 1968

An Axiomatic Basis for Computer Programming

5. Proofs of Program Correctness

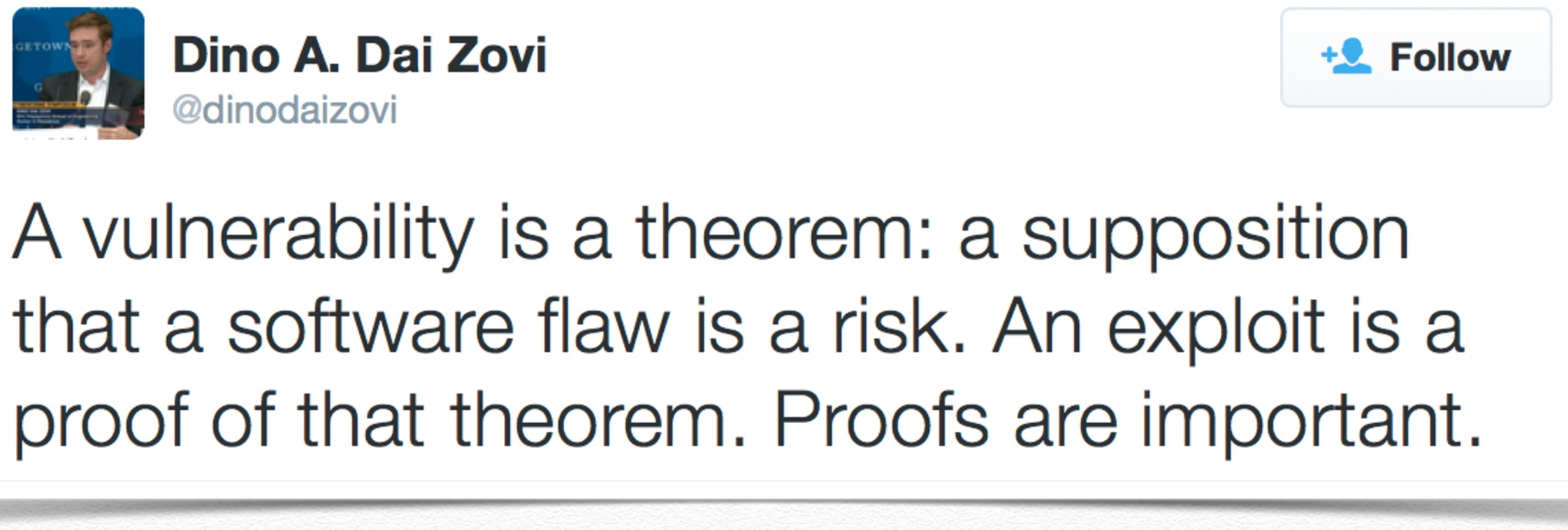
The most important property of a program is whether it accomplishes the intentions of its user. If these intentions can be described rigorously by making assertions about the values of variables at the end (or at intermediate points) of the execution of the program, then the techniques described in this paper may be used to prove the correctness of the program, provided that the implementation of the programming language conforms to the axioms and rules which have been used in the proof. This fact itself might also be established by deductive reasoning, using an axiom set which describes the logical properties of the hardware circuits. When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of the electronics.

Exploit is evidence of diverging user/developer **intent** and executable **reality**—and currently our only means of exploring the scope of this divergence.

Exploit (singular) vs Intentions

- Single exploit: **proof-by-construction** that **user intentions** are violated
  - For formalized intentions: constructive **counter-example** to the theorem that the system has intended security properties

Cf.:



Exploits (plural) vs Intentions

- Continual, multiple exploits despite fixes: evidence that user intentions are **hard** to **express** & **check**
  - Flawed program design?
    - Divergence between programmer's model & actual execution model/reality?
    - Development environment diverts programmer attention from intents?
- Either way, there's a **gap** between **intents** & **executable reality** that must be explored.

Ginnungagap!

Ginnungagap!

- There's is a gap between **intended** and **actual** computation
- **Exploitation** is a means of **empirically** exploring the gap for phenomena to generalize

User Intentions

Exploits!

Actual Computation

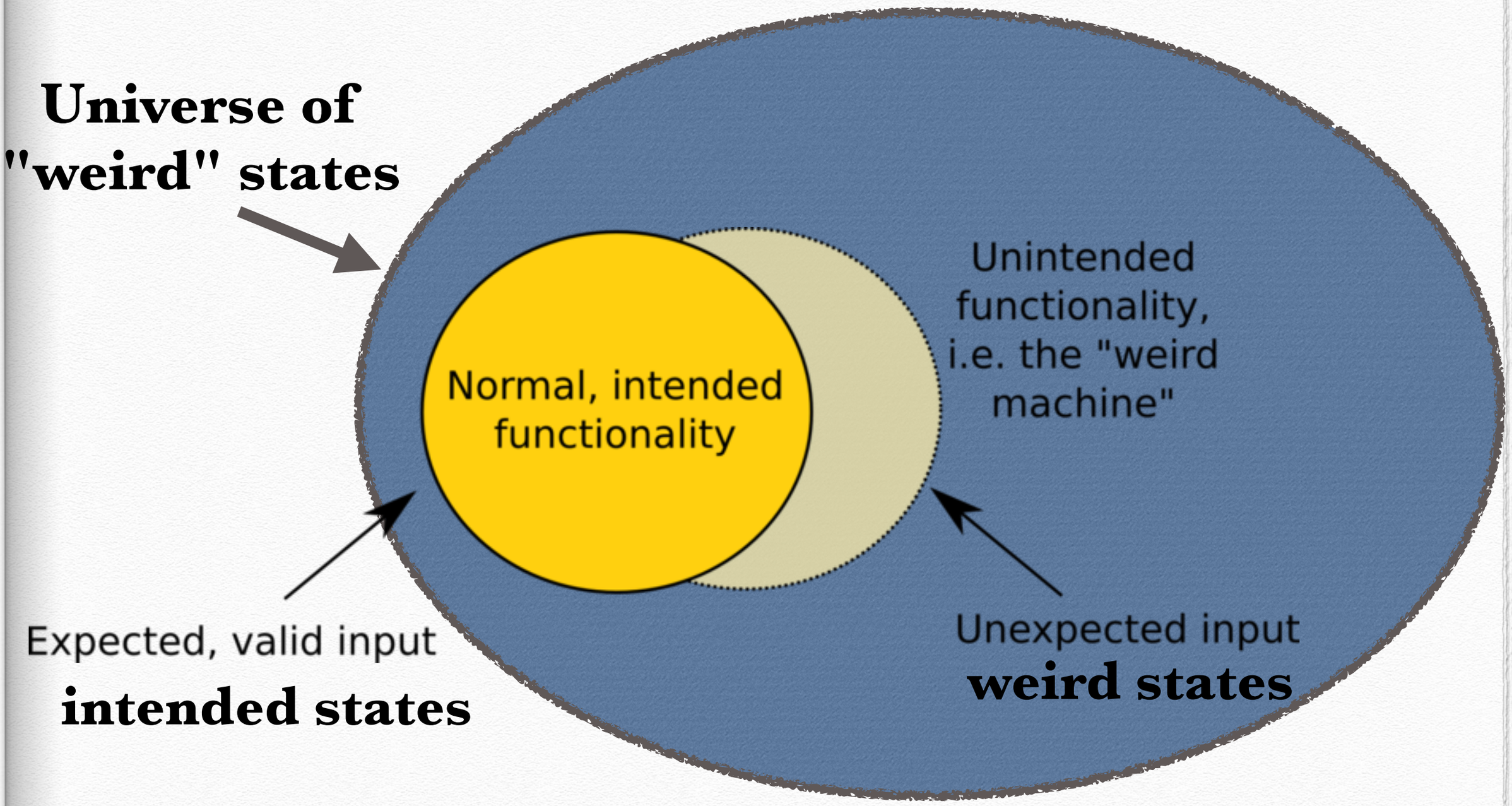
Exploit Revolution of 2000s

- From ad-hoc view of exploits to exploitation as a **generic** programming activity, **arbitrarily** expressive

- Unintended computation isn't a **serendipity**, but an **essential companion** of programming as we practice it

- Unintended computation has **stable**, **learnable** & **teachable** programming & execution **models**, akin to assembly with a “weird” ISA

Intended execution is **immersed** in a universe of unintended execution models



Example: data on stacks

- Standard function prologues & epilogues are an **automaton** distributed through code.
  - Data on stack (frame chains) are its **programs**
  - This automaton implements the Control Flow Graph
  - Aleph1 > Solar Designer > Tim Newsham > gera > Nergal > TylerDurden (Phrack, Bugtraq: **1997-2001**) return addr smashing -> assembling **arbitrary** programs
  - Return-oriented Programming (Hovav Shacham, 2007)

History in Bugraq & Phrack

- Solar Designer: "**Return into lib(c)**", 1997

- "Here's an overflow exploit [for the lpset bug in sol7 x86] that works on a non-exec stack on x86 boxes. It demonstrates how it is possible to **thread together** several libc calls." —Tim Newsham, May 2000

- Here I present a way to code **any program**, or almost any program, in a way such that it can be fetched into a buffer overflow [..]"—Gerardo 'gera' Richarte, October 2000

- Nergal "The advanced return-into-lib(c) exploits: PaX case study", December 2001 (bypassing pre-DEP NX defenses!)

Are all bugs shallow? Heck 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 **theorem** until we can use it in a **proof**
We don't understand a **bug** or a **feature** until we can use them in an **exploit**

Example: 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.

2008

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

Example: heap primitives

- Heap management code is a **machine**, heap metadata (pointers in chunk headers) are its **instructions**
  - "Once upon a free" (anon., Phrack 57:9),
    "Vudo malloc tricks" (MaXX, Phrack 57:8),
    "Advanced Doug lea's malloc exploits" (jp, Phrack 61:6),
    "Yet another free() exploitation technique" (huku, 66:6)
    "Malloc Des-Maleficarum" (blackngel, Phrack 66:10), and more!

- **aa4bmo** (write 4 bytes almost anywhere) in a double-free:

  `chunk->flink->blink = chunk->blink`

```
                    malloc                                  higher
vulnerability   ->  structures   ->  primitive   ->  level
                    corruption                              techniques
-------------------------------------------------------------------------
heap overflow          unlink()                                freeing the output
double free()   ->  technique    ->  aa4bmo      ->  hitting the output
...                                                 cushion chunk
                                                    ...
```

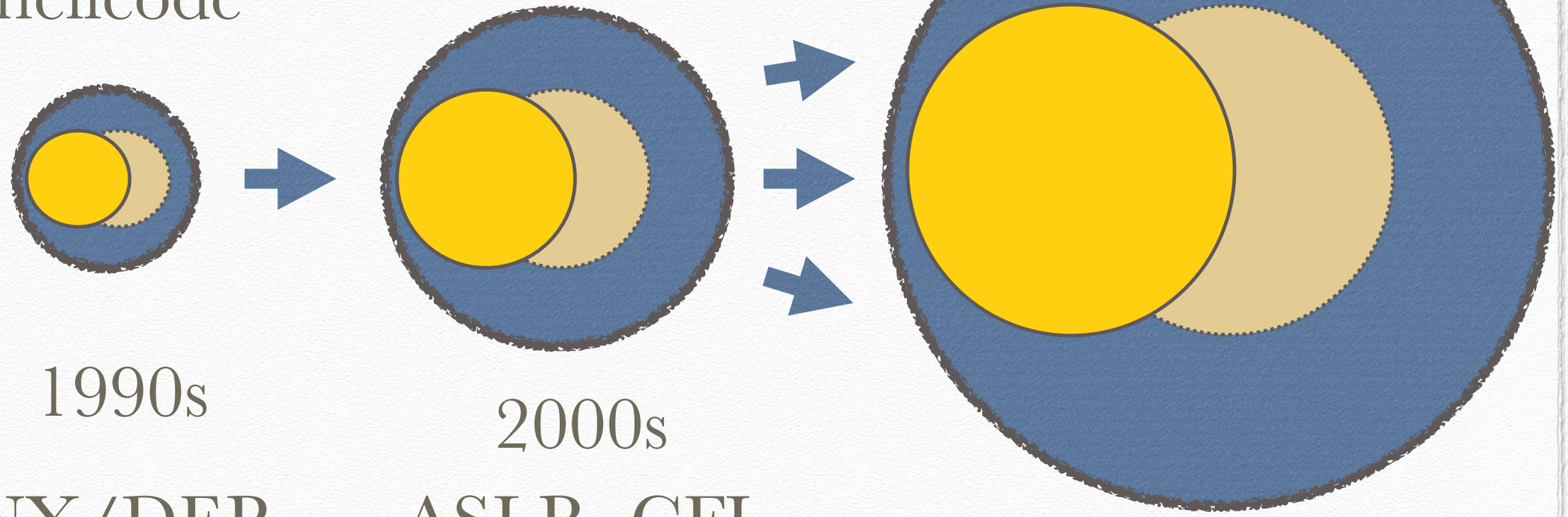
Example: heap machines

- **Heap management** is now a fully co-opted machine, configured via a series of mallocs:
  - "Heap Feng-shui" (Sotirov 2007), massaging the heap to create exploitable object layouts
  - Starvation-based machines (Gorenc et al. Recon.cx 2015, recent SMB-1/Samba exploits, ...)
  - Browser exploits that rely on context switches between higher-level JS & heap view of objects (Chris Rohlf, 2014--2016)

The Universe Expands!

Stack overflows, shellcode

ROP, heap ovf, malloc vudo, feng-shui, ...



1990s

NX/DEP, stack canaries

2000s

ASLR, CFI, Uderef

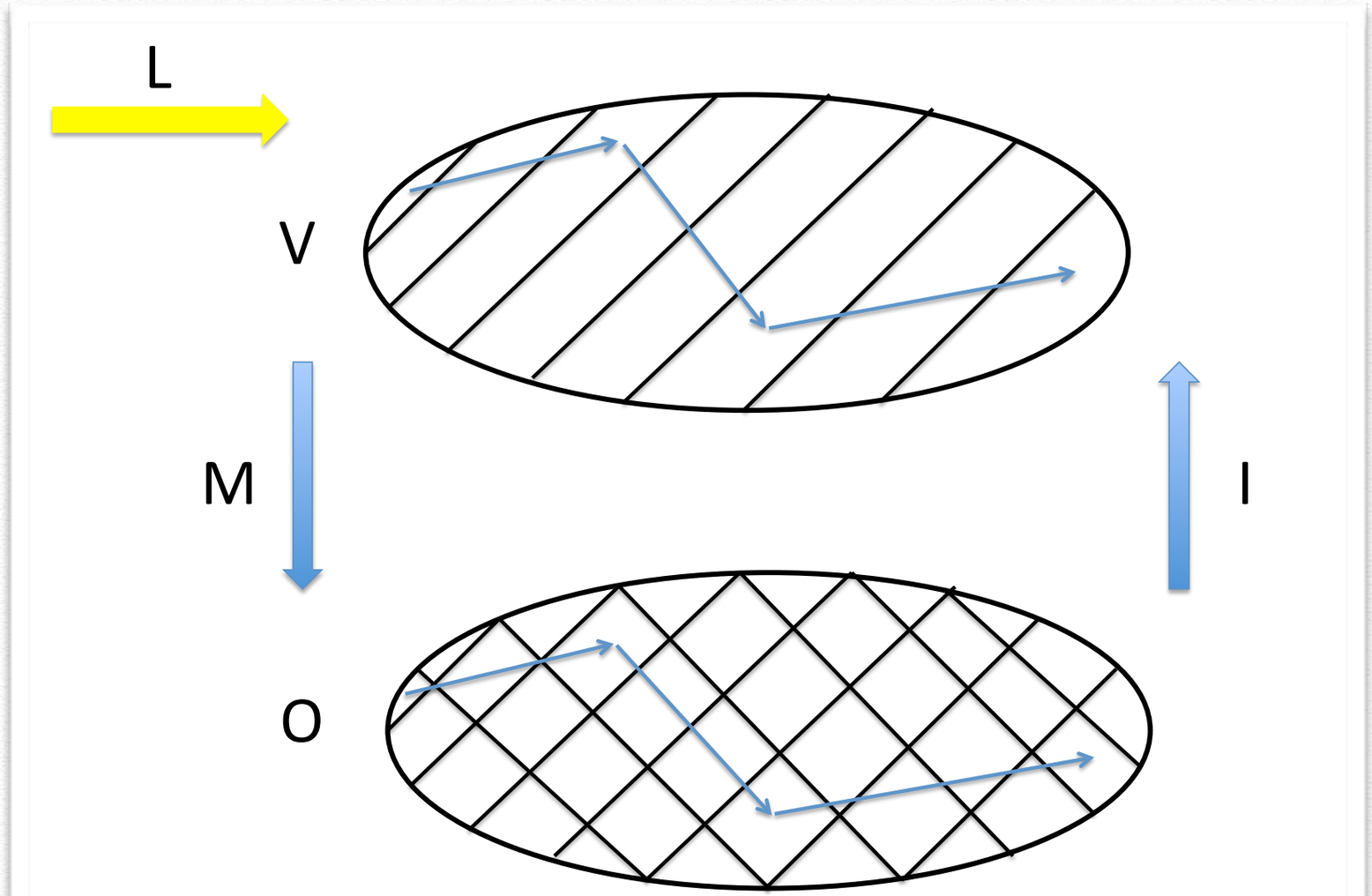
2010s

Unifying view of exploits

- How exploits are built: as **algorithms** out of **primitives**, conceptualized as "assembly" or ISA
  - In Phrack/Bugtraq some 8-10 years before academia got it with ROP/JOP/.*OP
- Primitives can be bugs, **features**, or any **actions**
  - **Features** can be more useful than bugs
  - Actions may include electric **glitching**, **light**, **heating**, well-timed **interrupts**, repeated **writes** (RowHammer), etc.

The **two**-abstraction view

- One abstraction/programmers model is **violated**, another is strictly **obeyed** & **relied on**



http://weirdmachines.org/papers/exploitation-as-code-reuse-position-paper.pdf

“Code reuse”

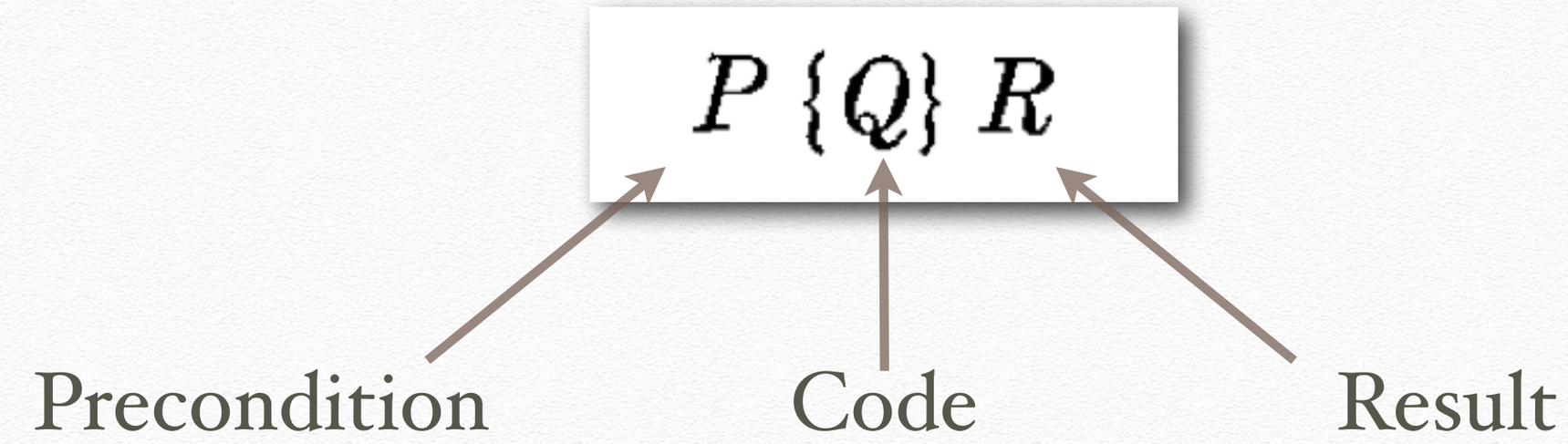
- "Code {re,ab}use" is unexpected computation by **benign**, **correct** code

- Classes of attacks are more: they are unexpected, emergent **programming models**

- These models are **essential companions** of our current programming. Why?

C.A.R. Hoare's verification formalism vs exploitation

"Programs are predicates"



- "Given P, code Q will (provably) result in the post-condition R" --C.A.R. Hoare, 1969

"Axiomatic Basis for Computer Programming"

…and then we chain them…

D1 Rules of Consequence

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

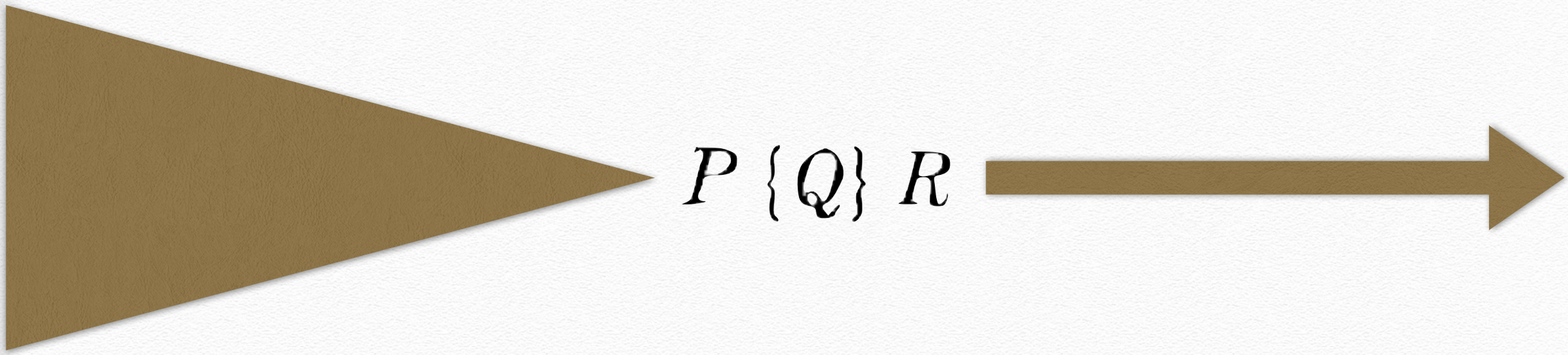
If $P\{Q\}R$ and $S \supset P$ then $S\{Q\}R$

D2 Rule of Composition

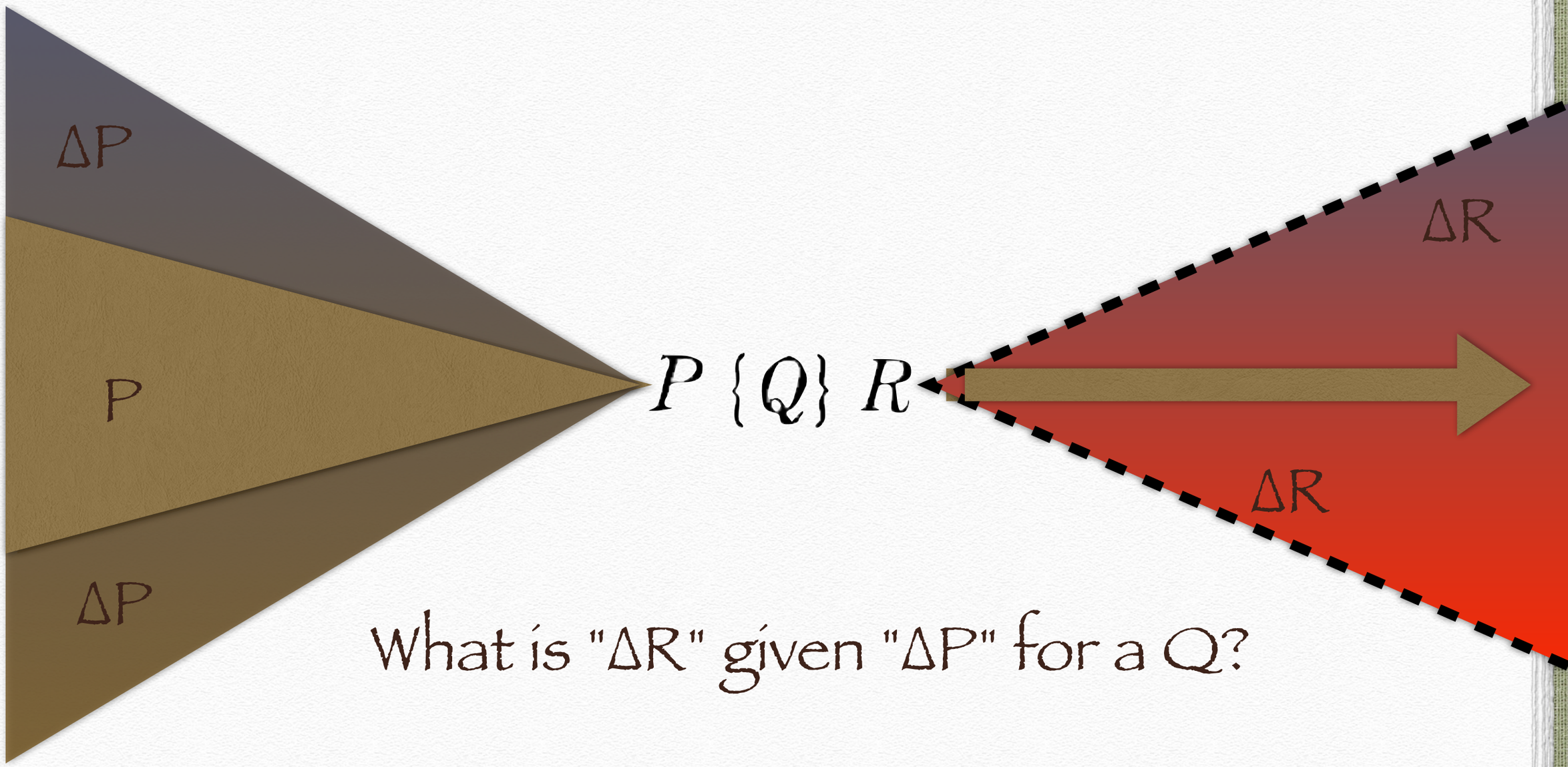
If $P\{Q_1\}R_1$ and $R_1\{Q_2\}R$ then $P\{(Q_1 ; Q_2)\}R$

- Long chains of intentions & expectations that must **all** match up at the joints, no exceptions

Formalism vs exploitation

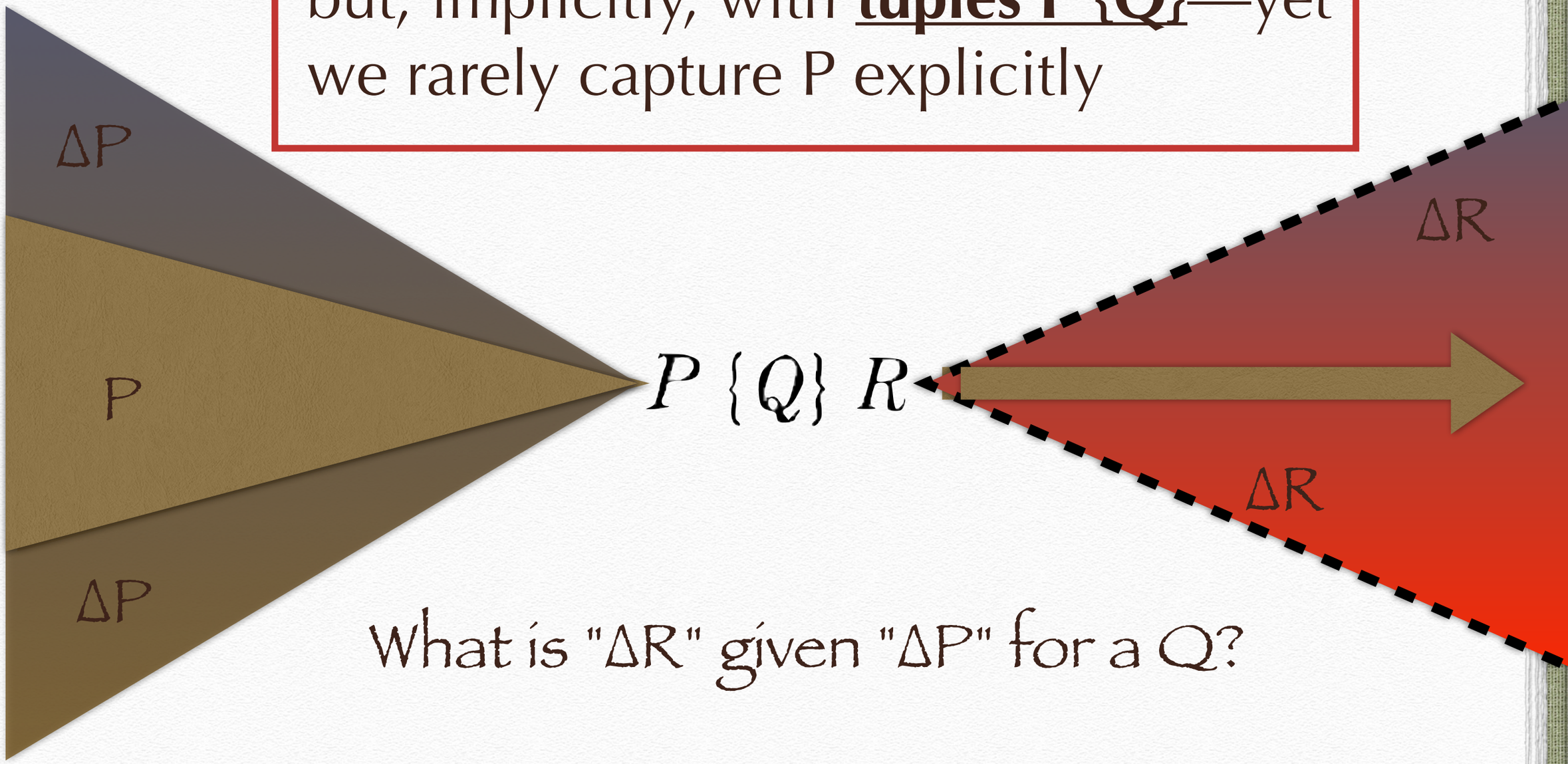


Formalism vs exploitation



Formalism vs exploitation

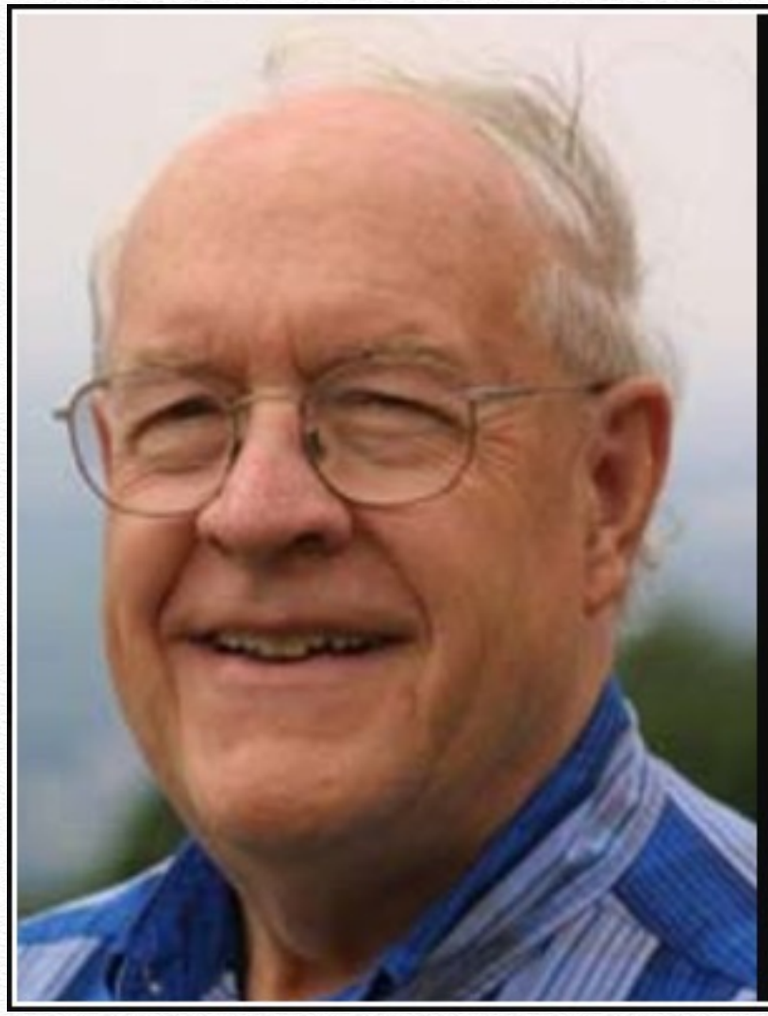
We program not with statements {Q}
but, implicitly, with **tuples P {Q}**—yet
we rarely capture P explicitly



What is "ΔR" given "ΔP" for a Q?

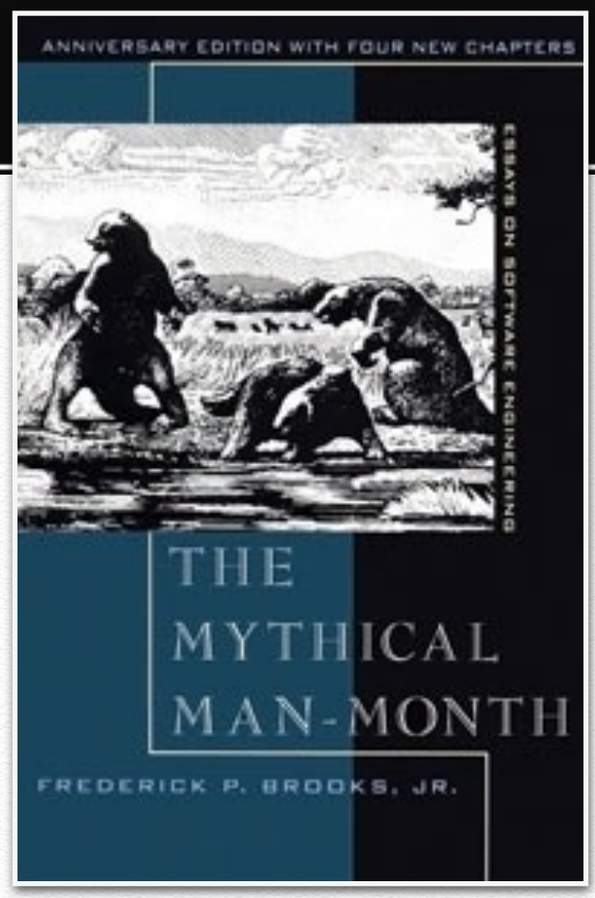
Engineering is about Stability. What's ours?

- Mathematical physics analyzes its artifacts for behaviors under perturbations (ODEs, PDEs, boundary problems, ...)
  - Physics & Mechanical Engineering build notions of **stability** on this analysis
- We have **no** notion or analysis of how properties of code Q change for **perturbed** pre-conditions P
  - Does code Q even keep its **class** in the **Chomsky hierarchy** of automata (or can it go full Turing-complete)?
  - **"What can a program compute on crafted inputs?"**
  - **This is what exploitation empirically explores**



The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures.

— *Fred Brooks* —



1975

CONSTRUCTING BELIEVABLE SOFTWARE

"One of the chief duties of the mathematician in acting as an adviser to scientists . . . is to discourage them from expecting too much of mathematics."

N. Wiener[+]

We can combine the arguments of previous sections with the intuition gathered from a century (or so) of experience in dealing with mathematics in a symbolic, formalistic way. Since "symbols" can be written and moved about with negligible expenditure of energy, it is tempting to leap to the conclusion that *anything* is possible in the symbolic realm. This is the lesson of computability theory (viz., solvable problems vs. unsolvable problems), and also the lesson of complexity theory (viz., solvable problems vs. feasibly solvable problems): physics does not suddenly break down at this level of human activity. It is no more possible to construct *symbolic* structures without using energy than it is possible to construct *material* structures for free.

"Social Processes and Proofs of Theorems and Programs"

DeMillo, Lipton, Perlis,
Yale TR-82, ***1979***

Engineering is working around physical impossibilities

solvable problems): physics does not suddenly break down at this level of human activity. It is no more possible to construct *symbolic* structures without using energy than it is possible to construct *material* structures for free. But if symbols and material objects are to be identified in this way, then we should perhaps pay special attention to the way material artifacts are *engineered*, since we might expect that, in principle, the same limitations apply.

Impossibilities (or, where security needs perpetual motion)

- Rice's Theorem: Algorithmically verifying *non-trivial* properties of programs is, *in general*, undecidable in T.-c.
  - What about security properties?
- **Undecidable** tasks abound, even in input handling:
  - **Safe input validation**/recognition: deciding if input causes benign or unexpected computation (for inputs that are T.-c. on receiving code)
    - Much code is T.-c. engine for data & metadata it consumes!
  - **Equivalence of recognizers**: will two input validators accept the same input language? (for automata beyond deterministic pushdown)

Empirical evidence

- Parser bugs still dominate other classes
  - **Complex formats** are clusters of trouble: e.g., **ASN.1**, **SSL/TLS**
  - **Similar** bugs across implementations (e.g., OpenSSL, GnuTLS, SecureChannel)
- Diverging interpretation of complex formats still an issue
  - X.509 certificate parsing diffs between CAs & clients, "**PKI Layer Cake**"
  - "**Android Master Key**" bugs, parsing diff between package crypto verifier and installer re ZIP structure

Hypothesis

- When validating a security property of code w.r.t. inputs is **undecidable**, automated testing whether code *accomplishes user intents* is **not feasible**
  - No meaningful **coverage** of "bad" inputs by tests; missed **classes** of "bad" inputs
  - Vulnerabilities will be "**dense**", not "sparse"

How do new execution models arise?

- Security-critical features start as power management, performance, admin/inventorying, etc. - **outside** of the **security** domain

- Attackers expose their **actual execution/programming models**

- Seeing these models, defenders understand the features' actual worth, and invest in (re)designing them properly

- New security models eventually result

Example: 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

Example: Symbol-related machines

- Dynamic linker (cf. Nergal's RTLD gadget, Phrack 58:4)
- Ld.so relocation (Shapiro et al, WOOT 2013; cf. LOCREATE in Uninformed 6:3)
- ELF relocation entries are T.-c. "bytecode"
- DWARF exception handler (helpfully a part of most processes) is T.-c. (Oakley et al, WOOT 2012)
- Diff. between execve() & ld.so:
  "All you need is GOT" (Bangert et al., 30c3)

  https://events.ccc.de/congress/2013/Fahrplan/system/attachments/2233/original/30c3-chain-of-trust.pdf

  Code & detailed explanation: "ELFs are Dorky, Elves are Cool", Bangert & Bratus, PoC||GTFO 0:3

Example: Newer machines

- Sigreturn-oriented programming (Bosman & Bos, S&P 2014)
  - "portable shellcode" via **`sigreturn`** structs
- Counterfeit **`vptr`** "OO-oriented" (COOP, 2015)
- "Control-flow bending" to defy CFI (e.g., printf as an execution engine that defies CFI, Usenix Sec 2015)
- "Interrupt-oriented programming" (Tan et al, ACSAC 2014)
  - a "bugdoor" via nesting MSP430 interrupts; fixed-entry, timed-exit "un-gadgets"

Example: co-opting caches & CPU instruction reordering

- L3 cache side-channel between VMs --Yarom & Falkner, 2014
  - "FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack"
    - VMs (and also processes) share L3 state (cf. T. Hornby, BH 2016)
- MFENCE to message between VMs --Sophia D'Antoine, 2015
  - VMs share the core's pipeline state
  - Reordering behavior is measurable, injecting MFENCE from one VM changes it for another

https://www.blackhat.com/us-15/briefings.html#exploiting-out-of-order-execution-for-covert-cross-vm-communication

Hypothesis

- Any non-trivial hardware or software mechanism that **rewrites** data and/or uses **indirection** based on data will be adopted by exploits
  - "mov is Turing-complete" (Dolan, 2013) & Movfuscator (Domas, Recon 2015)
  - Caches must become policy objects, just as page tables have

The weirdest machine (possibly)

- x86 MMU is Turing-complete on GDT+IDT+TSS+Page Tables (Bangert et al., WOOT 2013)
  - Arbitrary computation can be compiled in a combinations of these tables
  - No instruction is successfully dispatched
  - #PF & #DF alternate, acting as clock cycles

Will verification make the study of exploits irrelevant?

Equally spectacular (and to me unexpected) progress has been made in the automation of logical and mathematical proof. Part of this is due to Moore's Law. Since 1969, we have seen steady exponential improvements in computer capacity, speed, and cost, from megabytes to gigabytes, and from megahertz to gigahertz, and from megabucks to kilobucks. There has been also at least a thousand-fold increase in the efficiency of algorithms for proof discovery and counterexample (test case) generation. Crudely multiplying these factors, a trillion-fold improvement has brought us over a tipping point,

2009

Retrospective: An Axiomatic Basis for Computer Programming

C.A.R. Hoare revisits his past Communications *article on the axiomatic approach to programming and uses it as a touchstone for the future.*

"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.

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.

"Exploitation is a verification task"

CYBER GRAND_CHALLENGE

DARPA, 2015-2016

The Automated Exploitation Grand Challenge

Tales of Weird Machines

Julien Vanegue
julien.vanegue@gmail.com

H2HC conference, Sao Paulo, Brazil
October 2013

http://openwall.info/wiki/_media/people/jvanegue/files/aegc_vanegue.pdf

https://www.youtube.com/watch?v=AJKp0C5EHww

Weird machines, exploitability, and provable unexploitability

Thomas Dullien
thomas.dullien+wm@googlemail.com

http://www.dullien.net/thomas/weird-machines-exploitability.pdf

And yet...

> Verification technology can only work against errors that have been accurately specified, with as much accuracy and attention to detail as all other aspects of the programming task. There will always be a limit at which the engineer judges that the cost of such specification is greater than the benefit that could be obtained from it; and that testing will be adequate for the purpose, and cheaper. Finally, verification cannot protect against errors in the specification itself.

2009

Retrospective: An Axiomatic Basis for Computer Programming

C.A.R. Hoare revisits his past Communications *article on the axiomatic approach to programming and uses it as a touchstone for the future.*

Conclusion

For the foreseeable future, exploitation remains our **primary** and **fundamental** tool of experimentally exploring the space of unintended and emergent computation—unless and until we find principled ways of shrinking it.

As in other areas, reliability can be purchased only at the price of simplicity.

An Axiomatic Basis for Computer Programming

C. A. R. Hoare

Thank you!