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

Sergey Bratus, Anna Shubina

"The most important property of a program is whether it accomplishes the intentions of the user." [1] With these words, C.A.R. Hoare opened up his research programme, which we now recognize as a key part to the science of security—a still-nascent science, as Herley and Van Oorschot persuasively argued in [3]. Too many parts of that science are too well described by the Einstein quote about Mathematics in [3], "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"—largely due to the difficulties of mathematically describing the user intentions and their deviations from reality.

Yet there is a part of security that is *both* precise and certain, and answers to the best standards of experimental and evidentiary approaches of established sciences. It is the part dealing with exploitation of computing systems.

A single successful exploit for a program is a proof-by-construction that user intents of that program are violated, and that unintended computation is possible. If the intended properties are defined formally and are thus a theorem, then the exploit is the constructive counterexample, proving that the theorem doesn't hold.

A multitude of exploits—evidence of multiple vulnerabilities or recurrent exploitability despite fixes—suggests more. Its existence may impugn the design of the program, by suggesting that the original intents are hard for maintaining programmers to read and, therefore, to fix. It may also suggest that the programmer's model of the system and its actual properties dangerously diverge, so that the programmer has no clue when vulnerabilities are introduced (despite the habitual exhortations to program securely and be careful). It may suggest that the chosen language or the rest of the development environment is inherently error-prone, diverting programmer attention away from specifying intents succinctly.

In any case, repeated exploitability is empirical evidence of a significant gap between user intents and the actual computational properties of a program. As usual, statistical evidence is harder than interpreting a formalized logical proposition—but it opens the door to understanding broader classes of phenomena.

Exploits have been the primary—and, predominantly, the only—means of exploring the phenomenon of diverging intents and executable reality in computing. And yet exploitability and exploit construction is where we tend to find the least amount of pure research—constructing "the most general theories, covering the widest possible range of phenomena"[1]. In this note we seek to draw attention to this imbalance in security science and to provide perspective on the role of exploitation as the experimental vehicle of empirical security.

Exploitation as exploring the universe of computation. Early programs and systems tended to be designed for specific, limited purposes. Exploitation of these systems—that is, causing them to deviate from the intended purpose—was conceived of as ad-hoc and opportunistic, stemming from accidental and preventable errors or oversights; essentially, a serendipity for the attacker that had no general laws or principles governing it. As such, exploitation phenomena were not seen as generalizable (or worthy of generalization).

At the same time, developers of exploits (e.g., authors of Phrack, Bugtraq contributors, and others) converged to a very different view, as early as late 1990s–early 2000s.[2] To them, exploitation was generic programming akin to assembly programming in an unusual ISA, where side-effects of features or bugs served as "weird" but eminently usable assembly instructions. Just as with a "normal" ISA, the flexibility of programming was unlimited—even though a root shell could usually be obtained with just a few chained instructions, there was no limit to what could be programmed.

[1] Quoting from C.A.R. Hoare's 40-year retrospective of the original 1968 paper [2].

[2] For a historical sketch of this development, see, e.g., [4].

The implication of this world view was revolutionary. Exploits were not merely isolated careless accidents. Instead, the set of system's intended computations was seen as immersed in a larger universe of possible (and unintended) computation paths, which could be carried out by supplying the system with unexpected inputs.

This view, when finally rediscovered by academia, was expanded and made precise by proofs that certain programs or parts thereof would deliver no less than Turing completeness (TC) to whoever controlled their data (e.g., ROP [7, 6], SROP [8], COOP [9], ELF loading [10], DWARF exception handling [11], and even the x86 MMU without successfully dispatching any instructions [12]). Indeed, TC being achievable via very limited means such as control of certain data structures, a limitation to just a few ISA instructions like `mov` [13, 14], or even via crafted `printf` format strings [15] became the focus of many academic publications. TC was shown to be achievable even under restrictions placed on the existing code's execution path by schemes such as CFI [15].

However, despite being a nifty theorem that may have drawn disproportionate attention for being the one hard thing one could prove about an exploit mechanism, TC is secondary to the more general concept: that of intended computation being embedded in a space of unintended ones, just waiting to be unleashed by a crafted series of inputs or other external influences. Moreover, the practice of exploitation empirically shows that this embedding is inherently hard to avoid, and that the gap between the intended and the actual possible operation of a program is no accidental phenomenon but rather a fundamental property of computing systems.

At this point, we should note that this view of immersion does not contradict the C.A.R.Hoare concept of a verified program, and is indeed complementary to it. Hoare's formalism is constructed of theorems $P\{Q\}R$ asserting code post-conditions R given pre-conditions P and a decomposition of code Q into some elementary sequence of operations.

However, this formalism sheds no light on the behavior of Q under conditions other than P—in fact, it does not even let us distinguish if, under different inputs, Q would remain in the same class of automata as when supplied with inputs compliant with P, or could be elevated into an entirely different class, up to full TC.

Many areas of Mathematics analyze their artifacts with respect to how they behave when their conditions are perturbed; for example, stability with respect to perturbations plays a huge role in classical mechanics, as well as in the theory of ordinary and partial differential equations, underlying the different notions of stability used in physics.

For the C.A.R. Hoare's formalism, however, no notion of stability for the proven properties of code Q in the face of perturbing the pre-conditions P exists, to the best of our knowledge—not even with respect to the broad classes of computation theory such as the Chomsky-Schuetzenberger hierarchy.

We should, therefore, acknowledge that the practice of exploitation was the first to identify and explore the phenomena of such computation power escalation and lack of stability. It is due to exploitation that we now have experimental proof that the gap between the intended operation and the actual capability can be this wide, and is typically so.

This gap, then, cannot be accidental. What might be the CS fundamentals that underlie it?

Forgotten fundamentals? Frederick Brooks famously said in his "The Mythical Man-Month",

> 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... [5]

Yet castles in the air eventually start exhibiting behaviors not unlike those of physical materials: brittleness, unexpected behaviors under stresses, and strains, limited strength as in limited ability to support further structures. Just as a steel beam "flows" under a load, and beam frames lose their intended rigidity, so structural elements of air castles lose their intended behaviors and develop emergent ones.

Unexpected behaviors multiply and increase in richness. Deviations from programmer expectations become complex—occasionally more complex than the intended behaviors!

It is this difference between reality and intent and the spectrum of these emergent behaviors that exploits empirically explore.

Are these unexpected behaviors accidental or inherent? DeMillo, Lipton, and Perlis discussed this question back in 1979 when the potential of "castles in the air" and, relatedly, of improving efficiency of most human pursuits via mathematical (symbolic) modeling seemed unbounded:

> 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. 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. [16]

In other words, programmers should pay as much attention to natural limitations of their craft as material engineers pay to the natural impossibilities of theirs, such as conservation laws. Engineering in a physical world is arguably defined by its impossibilities, which engineers would readily name, perhaps starting with perpetual motion; in the realm of programming outside of cryptography, programmers may be hard pressed to enumerate theirs.

Indeed, many problems of Internet protocol insecurity are due to designs that would require the ability to solve undecidable problems [17, 18]. Persistent insecurity may be the result of the impossibility to automate testing for the desired properties to any meaningful extent, the state space being too large and devoid of a convenient asymptotic approximation (unlike, say, NP-complete problems that admit asymptotic solutions provably only n times worse than the actual optimum). As it is, we lack a simple means of assuredly generating the right amount of tests to convince us of any given degree of coverage for our programs' intents.

Whatever the mechanisms behind recurrent exploitability, exploits are currently the only available form of mapping out the underlying phenomena. With time, we may discover more general ways of predicting the topology of the unintended computation space in which our programs are embedded.

Exploits and program verification. Program verification has made great strides, owing, in part, to Moore's law and in part to an orders of magnitude increase in the efficiency of algorithms generating proofs and counterexamples [2]. Indeed, generation of exploits itself has been conceptualized as a verification problem [19], and the DARPA Cyber Grand Challenge demonstrated the fruitfulness of this approach, despite the numerous special challenges that it presents [20].

Will then verification eventually edge out exploits or render their study irrelevant? Not likely.

The relationship between formal program specification (when undertaken) and exploits is subtle. As Hoare noted, the primary contribution of the axiomatic approach was

> "$\cdots$ a simple and flexible technique for leaving certain aspects of a language *undefined* [$\cdots$]. This is absolutely essential for standardization purposes, since otherwise the language will be impossible to implement efficiently on differing hardware designs."

Exploitation, on the other hand, often exists in the space where behavior is undefined, either by design (as a necessity of standardization) or unwittingly. As a result, emergent execution models can co-exist even with systems designed with verification in mind, such as proof-carrying code [21]. As with other systems, exploits demonstrate the very existence of these models, as proofs by construction.

Quoting further from Hoare's retrospective [2],

> "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. All these limits can be freely acknowledged by the scientist, with no reduction in enthusiasm for pushing back the limits as far as they will go."

Thus, for the foreseeable future, exploitation remains our primary and fundamental ways of experimentally exploring the space of unintended and emergent computation—unless and until we find principled ways of shrinking it. To finish with the quote from the original C.A.R. Hoare paper [1], "As in other areas, reliability can be purchased only at the price of simplicity."

References

[1] "An Axiomatic Basis for Computer Programming", C.A.R. Hoare, Communications of the ACM, Volume 12 Issue 10, Oct. 1969, Pages 576-580.

[2] "Retrospective: An Axiomatic Basis for Computer Programming", C.A.R. Hoare, Communications of the ACM, Vol. 52 No. 10, Pages 30–32.

[3] "SoK: Science, Security and the Elusive Goal of Security as a Scientific Pursuit", Cormac Herley and P. C. van Oorschot, 2017 IEEE Symposium on Security and Privacy.

[4] "Exploit Programming: from Buffer Overflows to Weird Machines and Theory of Computation", Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, Anna Shubina, USENIX ;login: 2011, `http://langsec.org/papers/Bratus.pdf`

[5] "The Mythical Man-Month: Essays on Software Engineering", Frederick Brooks, (1975, 1995), p. 7

[6] "Return-Oriented Programming: Systems, Languages, and Applications", Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage, ACM Transactions on Information and System Security (TISSEC), Volume 15 Issue 1, March 2012.

[7] "A gentle introduction to return-oriented programming", Tim Kornau, `https://blog.zynamics.com/2010/03/12/a-gentle-introduction-to-return-oriented-programming/`, March 2013.

[8] "Framing Signals—A Return to Portable Shellcode", Erik Bosman and Herbert Bos, 2014 IEEE Symposium on Security and Privacy, Pages 243–258.

[9] "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications", Felix Schuster, Thomas Tendyck, Christopher Liebchen Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, 2015 IEEE Symposium on Security and Privacy, Pages 745–762.

[10] "Weird Machines in ELF: A Spotlight on the Underappreciated Metadata", Rebecca Shapiro et al., USENIX WOOT 2013.

[11] "Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code", James Oakley et al., USENIX WOOT 2011.

[12] "The Page-Fault Weird Machine: Lessons in Instruction-less Computation", Bangert et al., USENIX WOOT 2013.

[13] "`mov` is Turing-complete", Stephen Dolan, `http://www.cl.cam.ac.uk/~sd601/papers/mov.pdf`, 2013.

[14] "MOVfuscator", Christopher Domas, Recon 2015, `https://github.com/xoreaxeaxeax/movfuscator/`

[15] "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity", Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross, 24th Usenix Security Symposium, 2015.

[16] "Social Processes and Proofs of Theorems and Programs", Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis, Yale TR-82, 1979, p. 7, `http://www.cs.yale.edu/publications/techreports/tr82.pdf`,

[17] "Security Applications of Formal Language Theory", Len Sassaman, Meredith L. Patterson, Sergey Bratus, Michael E. Locasto, IEEE Systems Journal, Volume 7, Issue 3, Sept. 2013.

[18] "The Halting Problems of Network Stack Insecurity", Len Sassaman, Meredith L. Patterson, Sergey Bratus, Anna Shubina, USENIX ;login:, 2011, `http://langsec.org/papers/Sassaman.pdf`.

[19] "Automatic exploit generation", Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, David Brumley, Communications of the ACM, Volume 57 Issue 2, February 2014, Pages 74–84.

[20] "The Automated Exploitation Grand Challenge", Julien Vanegue, H2HC conference, Sao Paulo, Brazil, October 2013, `http://openwall.info/wiki/_media/people/jvanegue/files/aegc_vanegue.pdf`

[21] "Weird Machines in Proof-Carrying Code", Julien Vanegue, 1st IEEE Language-theoretic Security & Provacy Workshop, 2014, `http://spw14.langsec.org/papers/jvanegue-pcc-wms.pdf`