Carnegie Mellon University
Software Engineering Institute

AN ANALYSIS OF HOW MANY UNDISCOVERED VULNERABILITIES REMAIN IN INFORMATION SYSTEMS

Jonathan M Spring

February 2022

1 Introduction

This paper will bring computing theory and security operations into conversation to answer the question “How many undiscovered vulnerabilities are there in a piece of software?” The answers to this question influence how both theoretical computer science and security operations should behave. The answer this paper’s arguments support is that there are always more undiscovered vulnerabilities in any modern deployed software. I will also provide some possible directions for economic and research reactions to this situation.

Eminent security writers such as Dan Geer [14] and Bruce Schneier [26] have weighed in on these questions. Both reach similar conclusions that are differently worded. Geer states, “I believe that vulns are scarce enough for [cornering the global vulnerability market] to work.” Schneier states, “But while vulnerabilities are plentiful, they’re not uniformly distributed. . . [P]ractices that eliminate many easy-to-find ones greatly improve software security.” Dale Peterson and Josh Corman state that vulnerabilities are dense in medical devices, and mostly want to get on to discussing what to do about it [22].

The claim throughout all these assessments is, roughly, that good practices and smart policy can lead to software that is secure enough that it is fit for purpose. The statements in Peterson and Corman [22] assess vulnerabilities to be dense because the devices are not created carefully. This implies the position that enough secure coding practice could be the end of vulnerabilites in software. Schneier and Geer take more pessimistic positions, but only slightly. All three positions leave the current software engineering and production paradigms intact and imply that the community can get to a place where software is secure.

Other writers have also analyzed the question of software vulnerabilities. There is a discipline of measuring defect density in software engineering that was already well established in the late 1990s [13]. This type of empirical analysis has been applied by security researchers since the mid-2000s (see for example [21]). This paper will suggest that such metrics do not pose well-formed questions because—unlike the unresolved stances of Geer, Schneier, and Peterson—it appears to be clear from the following analysis that there are always more vulnerabilities to find (in Geer’s terms, vulnerabilities are dense).

This paper will relate vulnerabilities to the fundamental computer science understanding of the halting problem and Turing machines. In so doing, we can understand what vulnerabilities being densely located in computer code actually means for security, software engineering, and program verification, and what it would actually take to eliminate vulnerabilities from software. Section 2 introduces the theoretical concepts of Turing machines and some light mathematical background. Section 3 introduces the information security concepts of vulnerability management, program verification, and fuzzing. Section 4 begins the contact between these two areas with a discussion of how vulnerabilities in software could meet the mathematical definition of density. Section 5 continues by discussing how program verification communities have interacted with the halting problem and what that means for pragmatic vulnerability discovery. Section 7 examines the ramifications of Sections 4 and 5 on the cost of vulnerability discovery and defense.

2 Primer on Theoretical Concepts

Computing theory starts with a precise definition of computability. This definition is still useful for practical security managers because a computer can only calculate answers to questions that are computable. Computability is the study of what questions have computable answers. The practice of computer science has been to continually make computation more efficient. This improvement has made calculating complex computable answers practical. However, no amount of practical improvement can make an uncomputable question answerable by a computer. Security managers should be familiar with these concepts to avoid asking uncomputable questions. Section 2.1 introduces this definition of computability. Section 2.2 provides some mathematical background to give two important examples of questions that are uncomputable. Section 2.3 completes the mathematical background by introducing the formal concept of “density” that some writers in information security use as a metaphor in discussions about vulnerabilities.

2.1 Turing Machines

The concept of a Turing machine was first introduced by Turing [30] as a tool in philosophy of mathematics to answer the question of whether there are mathematical objects that a person can define but nonetheless cannot be calculated. Turing [30] called his conceptual invention a “computing machine,” after the job title (“computer”) of a person paid to follow a set of instructions to calculate values for accounting, war planning, and other similar large, social projects. The cleverness of the construction is in its simplicity. The computing machine has two conceptual parts, which can interact in only one of four actions at a time. There is a strip of paper, the *tape*, marked into squares that are either blank or not. The other part is the head, which can read, write with a pen, or erase. The head takes up exactly one square on the tape and, after reading the current square, can do one of (1) move one square left; (2) move one square right; (3) erase the current square and make it blank; or (4) write a stroke in the current square to make it not blank.

Understanding this simple conceptual computer is important because within four years of writing this concept down, Turing was building machines with these two parts that could actually conduct these four actions. In truth, similar but more simple machines existed in 1936: mechanized looms for weaving patterned cloth used a card with punched-out holes to represent the patterns. The loom would read the current square in the punch card and, based on whether it was blank or punched out, change its behavior. But the allies won World War II in no small part due to Turing’s physical computing machines. Any number can be represented on this tape. Zero is one blank space, One is one stroke, Two is a stroke followed by a blank, Three is two strokes, and Four is a stroke followed by two blanks. This is why modern computers use binary notation.

The head part of the Turing machine is conceptually “designed in such a way that at each stage of the computation it is in one of a finite number of internal *states*" [7, p. 25]. Each state is paired to an instruction. In this state, take this action and move to some other specific state. For example, a Turing machine that writes three strokes (a binary-encoded seven) can be described by a program with three states. In the initial state (q_1) the instructions are that if the tape is blank, right a stroke. If the tape contains a stroke, move left one square and move to state 2 (q_2). State 2 is the same as q_1, either write a stroke in a blank space or move left if there is a stroke in the square. State 3 (q_3) will also add a stroke to a blank tape square, but in q_3 if it reads a stroke, the program *halts* and the machine head stops taking any further instructions [7, p. 27].

We can construct more interesting Turing machines if we have some states that transition back to prior states under some conditions. For example, a machine that doubles the number of strokes on the tape can be written in 12 states. In brief, the machine can start out by moving all the way to the left of the set of strokes, noting and moving over a blank square that marks the leftmost edge; writing two strokes; moving all the way to the right side of the original set of strokes, erasing one stroke; and repeating the process of writing two strokes and erasing one stroke until it is trying to erase the blank space we noted on the left side of the original set of strokes at the beginning, at which state the computation halts. See [7, p. 28] for the specific description of states for this example. Similarly, Turing machines can be constructed that multiply two numbers together (the starting set up on the tape is two sets of strokes representing the numbers).

Up to this point, making any of these into physical machines probably would not have been worth it. Like the punch-card loom, they can compute only one very specific calculation, even if there are important ease-of-configuration gains in being able to read in some different patterns or initial conditions. Turing's innovation here was to realize that he could construct a Universal Turing machine (UTM) where the initial state of the tape was a definition of states of a Turing machine.

So a Universal Turing machine for the first example (writing seven on a blank tape) starts off with an ordered quadruple of strokes and blanks that define each state. The four actions (write a blank, write a stroke, move left, and move right) can easily be represented as $(1, 2, 3, 4)$, where 1 is write a blank, etc. The first element in the quadruple is the number of the action to take if the UTM reads a blank on the tape in this state. The second element is the number of the state to transition to after reading a blank, and the number is defined by the order of the quadruple in the list. The third and fourth elements of the quadruple are the action and state to move to after reading a stroke. So for the first state of the example, it would be 2,1,3,2.

The central processing unit (CPU) in the device you are using to read this paper is a UTM. When you loaded the software to read the paper, the CPU read in a set of instructions about a list of states and transitions that define the program. That is, it loaded the Turing machine represented by the program. Once that was done, it started the program at the head of the tape with strokes and blanks (this file's representation in binary). The Turing machine then calculated what color each pixel on your screen should turn (represented by three numbers between 0 and 255 for the amount of red, green, and blue, respectively). The values are between 0 and 255 because that is how many values eight squares on the Turing machine tape can represent. Since Universal Turing machines are extremely flexible, it is convenient to fix them into silicon and plastic as chips and reconfigure them millions of times during their useful life to perform different calculations.

The problem for vulnerability management with this is the theorem that Turing was able to prove about Turing machines. This is known as the "halting problem" because it is about whether a given

Turing machine m with initial tape input n reaches its final halted (stopped) state or not [7, p. 38]. Section 2.2 will describe the background on this question. Section 5 will explain why this presents a problem for vulnerability managers.

2.2 Halting Problem and Countable Infinity

Several important questions in computing theory, such as the halting problem, depend on the concept of *infinity*. A simple understanding of infinity is that it describes something without bounds, or where there is always another; that is, something which is not finite. However, the tricky aspect of the concept of infinity is that it is not one concept. This section will differentiate two concepts: an infinitely large set that can be counted or enumerated and what it means for a set to be larger than that. This definition will be important because an uncountably large set is also uncomputable by a Turing machine.

The counting numbers are familiar to most school children. An *enumerable* or *countable* set is just one whose members can be enumerated. That is, “arranged in a single list with a first entry, a second entry, and so on, so that every member of the set appears sooner or later on the list” [7, p. 3]. The set of positive integers P is named by the list $1, 2, 3, 4, 5, \ldots$, going on as long as you like.

An *enumerably infinite* set is any set whose members can be shown to be equivalent to, or a function of, the counting numbers P. While these lists in principle go on forever, for any specific n^{th} element we can list it. The tricky thing is that sets that intuitively seem smaller or larger than P contain the same number of elements, since these sets all contain enumerably infinitely many elements.

Consider the positive even integers $2, 4, 6, 8, 10, \ldots$. We can calculate this list as a function f of the first list P. Namely, $f(n) = 2n$ will produce this list, mapping every element n of P to an even number. Therefore, the set of positive integers is the same size as the set of positive even integers (the official term is that they have the same *cardinality*).

Consider the whole set of integers $\ldots, -2, -1, 0, 1, 2, \ldots$. This representation is not a proper list because it does not have a specific beginning. However, we can rearrange it as $0, 1, -1, 2, -2, \ldots$ and see that the set of integers is the same size as the set of positive integers. The function here is has two cases, $f(n) = \frac{n}{2}$ if n is even and $f(n) = -\frac{n-1}{2}$ if n is odd (0 is traditionally considered neither positive nor negative, so -0 is written as 0).

We can push further to find more sets that intuitively seem larger than the positive integers but are actually the same size. For example, the set of all ordered pairs of positive integers is enumerable. This list starts

$$(1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1), (1, 4), (2, 3), \ldots.$$

The list first enumerates every pair where the sum of the two numbers is 2, then 3, then 4, and so on. This pattern will enumerate every pair (m, n). We can calculate at which item the pair (m, n) will appear as $f(m, n) = \frac{m^2+2mn+n^2-m-3n+2}{2}$ [7, p. 9].

A rational number is any number that can be written as one positive integer divided by another, such as $\frac{3}{10}$ or $\frac{12}{7}$. The positive rationals can be enumerated as the list $\frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{1}{3}, \frac{2}{2}, \frac{3}{1}, \frac{1}{4}, \frac{2}{3}, \ldots$, which follows the same strategy as the list of ordered pairs.

The complete list of ordered triples (p, q, r) can be enumerated using the list of ordered pairs. From the original list of pairs (m, n) we can generate the triples by replacing the number n by the ordered pair that is the n^{th} element in the list of pairs. So the fourth element in the list of pairs $(1, 3)$ becomes

$(1,(2,1))$ or the triple $(1,2,1)$. We can use the list of pairs and triples to enumerate the list of all quadruples of positive integers. Replace n in the pair (m,n) by the n^{th} element in the list of triples. This method works for any ordered tuple of a fixed size. Quintuples can be enumerated from the list of quadruples and 10-tuples enumerated from 9-tuples.

While there are many mathematical objects that are enumerable, everything is not. How can we show that a set is *not* countable? We must demonstrate that there are always elements that are not in its listing. This question is important because computers work on the equivalent of one enumerable list.

An example of an uncountably infinite list is the set of all sets of positive integers $P*$. The set $P*$ contains the set of positive integers P; other infinite sets, such as the even positive integers; and finite sets, such as $\{1,2,3\}$ and $\{100,547,1,3,900000\}$. We can convince ourselves this set $P*$ is not countable by showing a way to construct a genuine element of $P*$ that will never be in an enumerated list of the elements of $P*$. This method does not find the same element each time, but rather it is a way to always find a genuine element of $P*$ not in the list of elements of $P*$. Since there will always be elements not in the list, the list is not enumerable or countable.

If we have a list L of sets of positive integers, *diagonalization* constructs a new set S^L_Δ that is not in the list L but is a member of $P*$. The list of sets L is made up of sets $S_1, S_2, S_3, S_4, \ldots$. We construct the set S^L_Δ by taking every positive integer n as a member if n is not in set S_n. So, if S_1 is the set of even positive integers, then 1 is in S^L_Δ. If S_1 is $\{1,2,3\}$ then 1 is not in S^L_Δ. The set S^L_Δ has to be different from every set in L because S^L_Δ contains an element not in each set in L. Even if we add S^L_Δ to L to get L', we can construct a new set $S^{L'}_\Delta$ which was not in L'. We can construct and add these diagonal sets forever, and we can always diagonalize the new, bigger list to construct another element of $P*$ not on the list.

Understanding whether each Turing machine will complete its computation and halt is equivalent to enumerating $P*$. Section 2.1 demonstrated that a Turing machine can be defined as a list of ordered quadruples. The list of all possible Turing machines is countable. To make this clear, we can assign each machine a single unique positive integer calculated from its list of ordered quadruples. For example, given the machine $(4,3,1,2),(4,1,2,4),(2,4,1,2)$, we can encode the sequence of numbers as the exponents to the list of prime numbers:

$$2^4 \cdot 3^3 \cdot 5^1 \cdot 7^2 \cdot 11^4 \cdot 13^1 \cdot 17^2 \cdot 19^4 \cdot 23^2 \cdot 29^4 \cdot 31^1 \cdot 37^2 = \\ 120472792249125444328648833318480$$

Since every integer has a unique prime factorization, this guarantees a unique number to label the Turing machine [7, p. 36].

Turing machines describe a method for computing properties or functions of the positive integers. But Turing machines are also positive integers. Conceptual constructions, such as numbers or logic, cannot prove properties about themselves [25]. A demonstration of this for Turing machines is a construction of a diagonal function on the numeral representation of Turing machines [7, p. 37]. An intuitive reason for this is that, while there are countably many Turing machines, each one represents a process going through possibly countably infinitely many states. This situation results in equivalent cardinality to $P*$: countably infinite number of list elements, each of which may itself have countably infinite elements.

Knowing whether a Turing machine will go through a finite number of state transitions is the same as knowing whether it will halt. The most efficient method for computing this fact is to run the Turing machine through its computations and see if it halts. To be able to calculate ahead of time whether it halts would mean defining a function $h(m, n)$ over the set of integers that represent Turing machines (m) and the input tape (n) such that $h(m, n) = 1$ if m eventually halts and 2 otherwise. If it were possible to construct a Turing machine H to compute h, then we can construct a Turing machine H' with a couple additions to H that would *both* halt *and* not halt when given its own number as input [7, p. 39]. Therefore, the machine H cannot exist; it is not possible to compute whether a Turing machine will halt in general better than by running that Turing machine and waiting to see. That is, given a Turing machine, it is not possible to know a priori what the result of its computation will be.

Turing machines are powerful conceptual machines, and the electronic computers designed to make them real can do an almost magical amount of work. We want to know whether listing all the vulnerabilities in a piece of software is equivalent to a countable or uncountable list. Programs are Turing machines and vulnerable states are a subset of resulting states. Section 5 will continue working with these conceptual tools and evaluate the extent they matter to practical vulnerability management. However, some claims about vulnerabilities are not whether they are countable or not, but whether they are *dense* or not. This concept is related to infinite lists, but distinct, and Section 2.3 introduces it.

2.3 Dense and Sparse Mathematical Spaces

In computational logic, a set of ordered elements is *dense* if there is always another element in between any two given elements. Formally, a linear order is dense [7, p. 152] if it is a model of

$$\forall x \forall y \, (x < y \rightarrow \exists z \, (x < z \wedge z < y))$$

and *sparse* otherwise. The positive integers are sparse because there is no integer between 2 and 3. The rational numbers are dense because there is always some value in between. For example, $\frac{1}{2}$ between 0 and 1, $\frac{1}{3}$ between 0 and $\frac{1}{2}$, and so on. The positive integers are sparse and the rational numbers are dense even though Section 2.2 demonstrated they are both the same size: countably infinite.

3 Primer on Security Concepts

The relevant security concepts for discussing this problem are the practices around management of vulnerabilities, especially discovery and response. Discovery usually refers to discovery of previously unknown vulnerabilities in a software product. Program verification and fuzzing are two categories of technology for discovering vulnerabilities in software. Response includes the detection of instances of known vulnerabilities as well as response actions to reduce risk.

3.1 Defining “Vulnerability”

What counts as a vulnerability is fairly broad. In practice, vulnerability managers may focus on vulnerabilities that are assigned a Common Vulnerabilities and Exposures (CVE) identifier (CVE-ID). But the definition covers many more things besides CVE-IDs. Namely, a *vulnerability* is

> “a set of conditions or behaviors that allows the violation of an explicit or implicit security policy. Vulnerabilities can be caused by software defects, configuration or design

> decisions, unexpected interactions between systems, or environmental changes. Successful exploitation of a vulnerability has technical and risk impacts" [15].

The phrase "explicit or implicit security policy" is important. A security policy is "a set of policy rules (or principles) that direct how a system (or an organization) provides security services to protect sensitive and critical system resources" [27]. A crucial feature of security policies is that they are specific to systems and organizations. Security policies change over time as well. Thus, whether a system contains a vulnerability depends on the context of which security policy applies. This changeability is not simply because the system owner did not think of something. An *implicit* security policy captures the scenario where a previously unthought side channel circumvents explicit security policy. Different organizations and systems have legitimately different security policies. The difference is not merely on the surface due to lack of explicitness or lack of awareness of side channels. There are many cases with broad consensus on the security policy and, therefore, identification of vulnerabilities. For example, if key management software has a flaw that discloses private/secret keys, the implicit security policy of the software design makes it clear this is a vulnerability. However, whether it is permissible for anyone on the Internet to query a DNS server depends on whether the software is deployed as an open recursive resolver (it is permissible) or a local resolver (not permissible).

There are several different senses of the word vulnerability that are often clear to a security practitioner by context [28]. For example, a software product such as a word processor has a vulnerability. Every installation of that word processor version contains an instance of the product vulnerability. The product vulnerability usually fits into a category of vulnerabilities, such as buffer overflows, as captured by Common Weakness Enumeration (CWE). Discovery usually attends to product vulnerabilities whereas response usually attends to instances of vulnerabilities.

Vulnerability management "includes services related to the discovery, analysis, and handling of new or reported security vulnerabilities in information systems" as well as"services related to the detection of and response to known vulnerabilities in order to prevent them from being exploited" [4]. Knowing how many undiscovered vulnerabilities remain in a piece of software would inform discovery, analysis, handling, detection, and response. Our assessment will focus on discovery and response. Section 3.2 introduces the two most common discovery methods. The remainder of this section introduces vulnerability response.

3.2 Vulnerability Discovery

Vulnerability discovery is about learning about vulnerabilities previously unknown to oneself. Many vulnerability managers accomplish this through mailing lists, Internet searchers, or other search methods. While these interpersonal methods are perhaps the most common, there would be nothing to discuss if analysis of programs and software could not discover new vulnerabilities. This section introduces the two most common methods of analysis to discover vulnerabilities in programs and software: program verification and fuzzing.

Program verification is about assessing the properties of programs. If the properties a verification system is assessing are security properties, then program verification finds vulnerabilities by analyzing whether the program violates those properties. In software engineering practice, program verification techniques are best used as part of unit testing or other analysis at build time [20].

Program verification was a mature field of study even in 1980 [2]. For at least as long, its practitioners have debated the nature of what program verification achieves [12]. One perspective is that programs are Turing machines about which one can conclusively prove properties based on appropriate logical methods, which is essentially treating programming as a branch of pure mathematics. An alternative perspective is that programs are an implementation on a physical machine, which one must conjecture properties to study via modeling and induction like a pond or a star system. An altogether separate view is that program verification may not be about proving program properties but rather checking for compliance with a specification. On this specification view, logics are for clearly specifying and demonstrating properties of the specification of what the program is intended to achieve [16].

Practical program verification contains aspects of each of these three perspectives. After 70 years, Turing's thesis continues to hold; there is no general-purpose method for proving properties of programs generally. What we have are myriad specific tools that are good at identifying specific properties in specific types of systems. One way to describe the situation is that program verification works best when the logical tools are designed and built to represent the relevant properties of the real-world system being verified [24]. So, for example, analysts have analyzed the Transport Layer Security (TLS) protocol, with specific tools to check for specific security properties under specific assumptions [5]. To check whether some implementation of the TLS version 1.3 protocol accurately and completely implements the protocol is a totally separate set of verification goals and tools. An analysis of whether some specific security properties are the best or fairest properties for the Internet community to adopt for an important protocol such as TLS is mostly outside the scope of program verification.

Program verification is easiest on source code, as opposed to a compiled program. Therefore, software developers tend to be the main users of program verification techniques. Since program verification is mostly used at build time by developers, most of the vulnerabilities discovered this way are fixed before the software is deployed. One goal of this paper is to understand the plausibility of using program verification to find all vulnerabilities before software release.

Fuzzing, introduced in the 1990s, is a process of "repeatedly running a program with generated inputs that may be syntactically or semantically malformed" [18, p. 1]. Fuzzing is prominently used by researchers or adversaries after software is released because it is a kind of testing that only requires the evaluator to generate program inputs. Due to its relative simplicity, it is used extensively by software developers before release as well.

A good fuzzer can efficiently explore the program's input space to find inputs that cause unexpected program behavior. The fuzzing community has developed a variety of methods for accomplishing this, such as developing models of commonly used expected input types (for example, building a fuzzer for a PDF reader that only creates valid PDFs as input), predicting what groups or types of input will produce the same or similar output, or introducing some program verification techniques to build a model of the program's execution.

Whether a systematic approach, such as program verification, or a random input mutation approach, such as fuzzing, is better or faster depends on the specifics of computation time [6]. Computational efficiency aside, if we follow Pym et al. [24] and accept that specific program verification tools are suited to assessment of specific properties in specific systems, then there will always be a complementary place for fuzzing. Fuzzing makes fewer assumptions about what properties to search for and tries to find inputs that lead to unstable or undesired program states. For a defensive standpoint, fuzzing is particularly useful after program verification since we can search for unstable states outside the assumptions of the verification systems.

One common problem for fuzzers is differentiating an input that causes a crash from an input that causes an *exploitable* crash [18]. A related challenge is when two exploitable crashing test cases should be called the same vulnerability or not, which often relates to whether one fix would eliminate both. But that definition pushes the problem on to defining whether changes to multiple source code lines or characters is one or two fixes. While this is of practical importance, for this paper we take definition assumptions most favorable to the paradigm of sparse vulnerabilities: We only consider as vulnerabilities those that are exploitable and consider all "equivalent" exploitable crashing states as one vulnerability.

A practical problem for analyzing fuzzing results is generalization. Suppose we run a fuzzer against a program for 300 hours, and it discovers no vulnerabilities. It is unclear what we should believe about how many vulnerabilities remain undiscovered in the program. In ecology, a researcher can answer similar questions by capturing two samples of fish in a lake and measuring how many fish were caught both times [23]. But such statistical measures do not work for vulnerabilities from fuzzing campaigns because (1) it is not obvious when two crashing test cases are in fact the same vulnerability, and (2) the statistics require an assumption of statistical independence between the two capture events that program outputs do not meet.

Fuzzing, like program verification, is a useful practical tool for vulnerability discovery. However, also like program verification, it provides little to no empirical evidence about what vulnerabilities may yet continue to be undiscovered in a program. Section 5 returns to the question of undiscovered vulnerabilities after Section 3.3 discusses actions in response to known vulnerabilities. Section 4 clarifies the definition of a vulnerability relative to the necessary mathematical concepts.

3.3 Vulnerability Response

Vulnerability response is a part of security operations. Security operations are the practical administration of information systems that support a security architecture and delivery or recovery of security services [27]. Specifically, vulnerability response includes detection of deployed vulnerable systems and actions to mitigate or remediate risk due to detected systems [4, §7.6].

Vulnerability detection occurs in at least three ways: scanning, penetration testing, and intrusion detection system (IDS) logs. Scanning, as usually done, requires a database of software versions and what vulnerabilities affect each version. The scan is usually a software fingerprinting and asset management effort followed by a database lookup of which vulnerabilities affect each software version. This kind of effort is dependent on a good database of vulnerabilities. The National Vulnerability Database (NVD) is often used in practice, which contains all the CVE-IDs. Scanning, therefore, often focuses on vulnerabilities which have been issued a CVE-ID.

Penetration testing is when "evaluators attempt to circumvent the security features of a system," often within some constraints or fixed starting conditions [27]. Some penetration testing tools, such as Metasploit, function as a vulnerability scanner (as above) connected to exploit modules for the associated vulnerability. The evaluator often works from a different set of vulnerabilities than the NVD and tries to pivot to different scanning points to test systems that might not appear in a fixed external scan. The evaluator may also search for common weaknesses in servers. Overall, a vulnerable system is any system on which the evaluator could circumvent security features.

IDS logs may detect adversary scans or indicators of successful attacks. Adversary scans may detect vulnerable systems, and the responses that indicate system vulnerability should also be captured in the logs. Indicators of compromised systems are not specific vulnerability detectors, but they indicate systems that likely have some vulnerability in them since they are compromised. There is a perhaps definition-based debate as to whether a weak or reused password that allows an adversary access is a “vulnerability.” But regardless of how that definition is split, a compromised system manifests some condition or behavior which calls for a response action.

Actions to reduce risk from vulnerabilities fall into two categories: mitigation and remediation [19]. *Mitigation* is reducing the impact of a vulnerability without eliminating it. Examples include restricting network access to vulnerable components or limiting access to a vulnerable component through software configuration changes. *Remediation* is an action that eliminates or removes the vulnerability. Examples include applying a patch or decommissioning the vulnerable system.

4 Definition of Dense or Sparse Vulnerabilities

Before we can analyze whether vulnerabilities are dense, we have to make this metaphorical connection to number theory a bit more explicit. Security practitioners define a vulnerable state as a set of conditions or behaviors. In the language of Turing machines, a single vulnerability (say, as identified by a CVE-ID) is one or more states of the Turing machine that lead to those conditions or behaviors. Vulnerabilities per se do not constitute an ordered set, and so the term *dense* does not properly apply. But we can take the colloquial meaning of “there is always another element” and define what that might mean for vulnerable states of a Turing machine. For a conceptual Universal Turing machine, I will argue that enumerating all vulnerabilities is equivalent to the halting problem.

For vulnerabilities being sparse or dense to be a meaningful question, we need a measure by which states generally are dense and vulnerable states might not be. We can model each state s_n of the Turing machine as a rational number n such that $0 \leq n \leq 1$, where n models how well the state of the program adheres to the relevant security policy. This model creates a preference order over the states of the program, where 1 is most preferred because it is in alignment with the security policy and 0 is least preferred because it is completely unaligned with the security policy.

However, this mapping to a subset of the rational numbers is not particularly interesting. Since any subset of a dense linear order is also dense, if there are any vulnerabilities at all, then there are dense vulnerabilities. That is, either every state is completely in line with the security policy ($n = 1$), or there is always another state with a smaller preference ranking (more vulnerable).

One vulnerability identified by a CVE-ID is usually not one single state of the program. A CVE-ID usually identifies a set of states that are functionally equivalent in violating a security policy. Therefore, the question is whether the set of such sets is dense, not the set of states themselves.

Therefore, we should prefer a measure over sets of states, rather than of individual states. We can simplify the values for s_n to be a binary $f(s_n) \in \{0, 1\}$ that represents whether the state satisfies the security policy (1) or not (0). A listing of all the states with value 1 is essentially the definition of the security policy. If we had that complete list, then any actual set of states of the program could receive a score based on the proportion of states that meet the policy versus the total states in question. There are several aspects of this modeling choice that would be difficult in practice, such as creating the infinite list of states and assigning each a value. The ramifications of this difficulty are discussed further in Section 5. However, for now, let us say that we can score a set of states (S

with $s_n \in S$) as a rational number a where $a = \frac{|s_n \in S : f(s_n)=1|}{|S|}$, with $|S|$ denoting the cardinality (size) of a set.

This definition puts the security score a of a set of states on the rational interval $[0, 1]$. However, it's not clear how this is useful in modeling sets of states of the program. If there is a provably finite set of states where $a = 1$, which is something that program verification tools can regularly produce, then it does not inform about the question of whether there are other sets of states with a lower value of a (that is, a set of states where some are insecure).

Alternatively, we could try to reason about the set of all states of the program. However, unless we can prove that the program will halt, we must assume an infinite number of states. If $|S| = \infty$, then $a = 0$ due to the convention that any integer divided by ∞ is 0.

If we take subsets of the sets of program states $S_0, S_1, \ldots, S_i, \ldots$, then we end up analyzing all possible combinations of sets of states, which is the power set of S, notated $\mathcal{P}(S)$. The values of a over each set $S \in \mathcal{P}(S)$ will be dense if there are infinitely many vulnerable states. Therefore, the answer to whether this measure is dense depends on the question of whether vulnerabilities are dense, so it is not useful in answering our question of interest. If vulnerabilities were not dense, then there would be a maximum non-1 value of a in $\mathcal{P}(S)$ at $\frac{v}{v+1}$ where v is the number of vulnerable states, as long as there is at least one secure state.

Given these modeling results, perhaps it does not make sense to try to formally equate the metaphor of dense vulnerabilities to a mathematical concept of a dense linear order. If vulnerable states do not meet the definition of *dense* in Section 2.3, there still may "always be another one." After all, there are countably infinite positive integers P, even though the positive integers are not dense. Unfortunately, this means we cannot rely on structured reasoning rules that an ordered space would supply.

5 Halting and Vulnerabilities

Given these problems with numerical representation of vulnerable states, the problem should be modeled more generally. This section will argue that a solution that is appropriately general to avoid the problems described in Section 4 will instead be subject to the halting problem.

Let's take a model of states of the Turing machine where the states are labeled by a natural number i, but there is no sequence or other meaning to i, it is just a label. Let us hypothesize that there is a function v where $v(m, n, i) = 1$ if i is a secure state in Turing machine m started with input n, otherwise $v(m, n, i) = 2$. This modeling choice provides space to ask the relevant question: For any given Turing machine m with input n, is the number of vulnerable states finite or infinite?

If vulnerabilities represent sets of states, then the set of these sets is uncountably infinite if both (A) the size of the sets of states is unbounded and (B) the number of sets of states is unbounded. If one of these two conditions holds, then vulnerabilities are countably infinite. From this conceptual lens, countably infinite might be an acceptable security outcome because it should be possible to create a program that lists all the vulnerable states. Uncountably many vulnerabilities would indicate there are always more and the best one could do, a priori, is to estimate some vulnerabilities in a Turing machine. This situation would mean that formal verification of a Turing machine could never, in principle, discover all the vulnerable states.

The output of the function v would be a series of 1s and 2s, for example:

$$1, 1, 1, 2, 2, 1, 1, 1, 1, 2, 2, 2, 2, \ldots$$

The first challenge with analyzing v would be enumerating the input. To answer part (A), the question amounts to asking whether, given a 2, if the next value will be a 2. But to know whether i is a state of m, one must enumerate all the states of m, which intuitively appears to be the halting problem again. Since all Turing machines are bound by the halting problem, for the general case we have to treat the sets of states of the machine as unbounded. Section 6 will constrain this analysis, since computers that humans can build so far have bounded (rather than limitless) memory.

This function $v(m, n, i)$ takes similar inputs as the proposed halting function $h(m, n)$. We can make an argument from contradiction that v does not exist. Assume the function v exists. For v to be well defined, we must know the range of i for which there are states of the machine m for input n. Either there is a finite integer value K for the largest i, or $i \in \mathbb{N}$ and i is countably infinite. If K exists, it is the halting state of m with input n. If $i \in \mathbb{N}$, then m does not halt with input n. That is, if K is finite, then the function $h(m, n) = 1$, and if $i \in \mathbb{N}$, then $h(m, n) = 2$. As described in Section 2.2, h is not computable. But if we could compute v, then we could compute h. Therefore, v must not be computable.

Let us make a less general attempt. Rather than one function that determines whether any Turing machine is vulnerable, what if there are different functions for some specific machine m with specific input n. That is, $v_m^n(i)$ might be possible for some m and not others. In practice, program verification works better when it is specific to a type of system and type of vulnerability [24]. Requiring a different function for each machine m is not exactly the same as focusing on a type of system, but it is driving at the same idea of specialization.

Let us imagine a machine m where it happens to halt, and we know this because we have run it and observed it to halt with input n at state K. Then it would be possible construct the function $v_m^n(i)$ for $i \in K$. "Possible" means there is no mathematical definitional contradiction. When we declare the definition of $v_m^n(i)$, we are essentially declaring a security policy for the Turing machine. At this point, there is another important feature of the function to consider: uniqueness.

The information about the input state i is useful if there is exactly one output, 1 or 2, vulnerable or not vulnerable. If $v_m^n(i)$ is a function, as we've defined, and $v_m^n(i)$ is unique, then this is true. However, vulnerabilities are relative to a given security policy (see Section 3). Therefore, if a different principal asserts a different security policy for machine m with input n, we might have ${v_m^n}'(i)$ where for some values of i it is the case that $v_m^n(i) \neq {v_m^n}'(i)$. This multiplicity is not a problem when declaring function definitions (that is, security policies), but it is a problem with algorithms that search for a valid function definition as if there is just one unique valid function to find when in fact there can be more than one.

This section has demonstrated that in the general sense of Turing machines, there is not a general method to discover (compute) vulnerabilities. Specialized mathematical functions for specific Turing machines with specific inputs amount to declaring a security policy. Therefore, an important question is how pragmatic it might be to find a "good enough" approach between these two extremes, which will be analyzed in Section 7. First, Section 6 argues that this section's conclusions from the general modeling choice of Turing machines is useful and applicable to real modern systems.

6 Applicability to Modern Systems

A vulnerability is a set of conditions that lead to a security policy violation. One thing many of these sets of conditions have in common is mismanagement of some system resource. The resource may be allocations of physical memory, filesystem pointers, memory pointer variables, network bandwidth, etc. Although there is a diverse set of resources that a computer system manages, one might question why resource management and allocation is so hard.

One answer to this question is that the system cannot know a priori when a program (that is, a Turing machine) will stop needing resources, because the system cannot know when it will stop. Computer engineers have created a beautiful, clever, and diverse web of heuristics to estimate how many resources a program should be allocated. Shared physical memory is one such. While 250 programs may be running at any given time in memory, it is unlikely that any given one of them will need $\frac{1}{250}$ of available memory. Memory could, hypothetically, be allocated statically based on a maximum number of programs, such as 250, with firm boundaries where programs could not reference memory locations outside their allotment. But it has proven much more useful to dynamically allocate memory in a general-purpose way because, normally, a program can be trusted to just ask for more memory if it needs it. This usually works. But the trust sometimes fails—a program is allowed to read memory that was not rightfully allocated to it, and we get vulnerabilities like "Heartbleed" [11].

So as long as our computers are subject to the halting problem, we should expect to find more resource mismanagement vulnerabilities. One natural question, then, is to what extent are our current computers actually subject to the halting problem? The Turing machine model requires an infinite tape to read from and write to. Our computers are not infinite, so it seems like the halting problem should not apply to the software Turing machines running on it.

However, the Turing machine model fits modern software better than one might think for two reasons. The first reason is that software with a network connection effectively has an infinite tape. The network may have a bandwidth limitation, but this only limits how fast the tape can be used, not how large it is. So while we could get into discussions about how the program is likely to stop by the time the sun turns into a red giant and consumes the earth, that is not constructive. Since a network connection to the modern Internet allows for indefinite reads and writes interacting with arbitrary other processes (including physical and human processes), the best model for a network-connected program is an infinite-tape Turing machine.

Secondly, even for non-networked programs, the number of states the Turing machine represented by the program can enter is unmanageably large. This is the case even if we limit the analysis to physical RAM, and disallow mapping of virtual memory on disk, and ignore reads and writes from memory to disk in general. With a modest 4 GB of RAM, available on any mid-tier 2020 smart phone, most of the memory is available for any program to use. If we say a program might be allowed to use 3.5 GB, there are $\left(2^8\right)^{3.5\cdot10^9} = 256^{3,500,000,000}$ possible states. Since $log_{10}256 = 2.408$, this is equivalent to $10^{8,428,839,878}$ states. 32 bytes ($2^{256} \approx 10^{77}$) is a cryptographically secure search space [3, p. 59]. So unless a program's memory footprint can be reduced to less than the equivalent of 14 bytes, it will not be practical to check through all of them ahead of time to ensure all the configurations are allowed by the security policy.

Reducing program states to larger equivalence classes in order to do some logical verification is one interpretation of the practice of program verification. For example, a common tactic in program verification is to express a `while` loop as a logical sentence with a loop invariant where that sentence formally entails that the loop will terminate. This tactic creates an equivalence class among all the memory states the `while` loop can create and captures them in the rules of entailment and the loop invariant. With a 1 kB numerical array in a loop, the 2^{8192} possible memory states could be reduced to one statement expressible in a couple dozen bytes (characters). This kind of reduction in state to search is quite powerful.

Program verification is quite useful at finding specific kinds of flaws, especially when the verification system is tailored specifically to one narrow kind of flaw [24]. Therefore, program verification works well in specific contexts, and has been demonstrated to be useful for preventing a specific class of program flaw. However, verification cannot solve the whole problem, even with myriad different specialized verification modules, because many of the insecure memory states are not code writing errors. Many are design errors. There is some work in verification of the properties of protocols, such as TLS. However, most software architecture plans are not verified. A verified implementation of a bad design will have vulnerabilities that are outside the scope of what the verification can find. Program verification is improving and is in practical use to make software better, but it is not within its scope to eliminate all vulnerabilities.

In our hypothetical 4 GB of memory, we might think of a successful program verification system as reducing all the possible states of, say, 1 kB of memory to a single verifiable statement. This reduction would be quite successful if we can find one statement for each 1 kb of memory. Maybe the 1 kB represents a loop over the integers, and we can reduce that loop to a Hoare triple representing inputs and outputs of the code segment. What the correct statement is requires some computation, so it is not free to find, but once we have it there is a great complexity reduction. In the case where each memory chunk is treated as independent, it is as if the problem has been reduced from searching $10^{8,428,839,878}$ states to searching the $10^{1,028,912}$ possible interactions between statements. Both of these numbers are so much larger than 10^{77} that the distinction makes no difference. If the logic statements are not independent, then it is easier to get a conclusive result from the statements but harder to find a set of statements that are both valid and accurately models what the program actually does.

Some program verification techniques are *composable*, which means changes to one part of memory demonstrably do not affect other regions of memory. This property helps dramatically with scaling the computation of proofs by parallelizing computation and reanalysis of a slightly modified program. Such systems can make pragmatic claims about specific memory safety properties of programs [9]. Composable logics for program verification are clearly a good engineering choice for practical program verification, but they need to be specialized enough that we arrive back at the problem of what properties should be verified. In order to reduce the complexity of the problem, logic tools—based on, for example, temporal logic [17] or separation logic [9]—do not search for security flaws in general. They also do a complex amount of syntactic and semantic manipulation such that the validation failures are often not actually flaws in the program, even in good systems [8]. While it is pragmatic and useful to be confident an analysis has identified any potential null pointer dereferences, this accomplishment is far from analyzing all possible program states for all possible vulnerabilities.

Although vulnerabilities are not technically infinite in a 4GB system without a network interface, it is not practically feasible to exhaustively check every possible state of a moderately complicated program for whether it violates every security policy. Program verification can usefully check many program states about some specific security policy [24]. And multiple program verification techniques can be used to check about different specific security policies. But the search space of possible states is too large to exhaustively check a Turing-Complete program, even with the help of fuzzing and program verification.

There is a practical question that remains open: Can the effort an attacker must spend to find another vulnerable state be increased to the point that it costs the attacker more time or money than the knowledge of the vulnerable state is worth?

7 Cost of Vulnerability Discovery

As introduced in Section 3.2, the effort to find previously unknown vulnerable states of a program is *vulnerability discovery* [4]. Both attackers and defenders conduct vulnerability discovery. Defenders use the information to make the vulnerable states unreachable, while attackers use it to force the program into vulnerable states. The overall goal of a secure software engineering practice would be to find and fix every vulnerable state before an attacker does. If vulnerabilities are dense or never-ending, then the software engineer cannot close all the vulnerabilities before finishing the software. If vulnerable states are so numerous as to be effectively infinite, then the result is the same. As Section 5 argued, there is good reason to believe vulnerabilities are effectively unlimited.

In a world where software engineers cannot close all vulnerabilities in a program, how much effort should they put into closing some of them? This question blossoms into a complex set of interdependencies. The development team should expect some vulnerabilities to be found by external parties; therefore, they should have a vulnerability management function that can receive reports, triage them, and publish updates [4]. Prioritization during triage is a complex topic in its own right [29]. And the market for software expresses some economic inefficiencies that often push suppliers of software to rush development rather than invest in secure code [1]. The economic aspects of information asymmetry and network effects explain why software developers often rush to market rather than focus on pre-release vulnerability discovery [1]. However, this criticism of software developers presupposes a somewhat naïve solution—exhaustive prerelease vulnerability discovery. Section 5 provides some good reasons to believe there will always be another vulnerability to discover. If this is true, the optimal system for minimizing costs is less clear.

Vulnerabilities may be discovered prerelease or postrelease of the software. Prerelease discovery can be done only by the software developer. The prerelease vulnerability discovery costs we are primarily concerned with are delays to release and software developer time and attention. Postrelease discovery can be conducted by anyone. The postrelease vulnerability discovery costs we are primarily concerned with are developing remediations and mitigations (such as software updates), deploying remediations and mitigations, and incidents caused by attackers exploiting the vulnerability. While developers bear all the prerelease costs, their customers bear the postrelease costs of deploying patches and any incidents.

Whether the person doing vulnerability discovery is an attacker or defender, they have access to and use essentially the same tools: program verification, fuzzing, and manual code review. Program verification is often easier with source code rather than compiled machine code, possibly giving close-source developers a small advantage. But in general, if there is an openly available program verification or fuzzing technique, then a software developer should use it because some attacker will likely use any readily available techniques to find flaws.

Attackers have one additional method for creating exploits besides vulnerability discovery—reverse engineering security updates. Reverse engineering security updates does not discover new vulnerabilities, but it changes information flows. Attackers can often reverse a patch and start attacking the old version long before every deployed system has applied the patch or upgrade. Because of this lag between patch publication and patch application, there are expected incident costs to defenders even for post-release vulnerabilities discovered by *defenders*.

Whatever methods attackers use to discover vulnerabilities, an important question is how the defense community might incentivize them to stop. In other words, make the value of some other behavior exceed the value of vulnerability discovery. Vulnerability discovery methods such as program verification and fuzzing require compute time. If the economic return on that compute time when directed at vulnerability discovery is lower than that of mining bitcoin, then we expect attackers to stop spending their limited compute resources on vulnerability discovery. Perhaps reverse engineering security patches would remain economically viable when searching for new vulnerabilities is not, but let us put that aside for a moment. As fewer people search for previously unknown vulnerabilities, basic supply and demand predicts the value of finding them goes up as long as demand remains constant. This reasoning suggests that if there is a market for vulnerabilities there will always be people who can extract value by finding them. Specifically, as the supply goes down, the price will rise to a point where it is worth some attacker's time to discovery vulnerabilities.

The question of whether there are always more undiscovered vulnerabilities can be viewed as a market dynamics question. Economics primarily studies elasticity in regard to demand. Inelastic demand refers to products whose consumption does not vary (much) as price increases, such as fuel. If vulnerabilities are "dense," then this might be modeled as inelastic *supply*. Land is a good example of a good with inelastic supply, as generally regardless of the price for land more cannot be produced. Vulnerabilities may have a strange place in this model because Section 4 suggests that *fewer* vulnerabilities cannot be produced. Policies that discourage discovery by attackers, such as criminalization, would not reduce supply. However, such policies may reduce some specific attacker's inventory. While such policies may drive prices higher to compensate for the increased risk of punishment, they would not be able to quash supply.

Costs borne by defenders, both system owners and suppliers, are also important. There are several aspects to cost in vulnerability management given a known vulnerability: vulnerability risk, change risk, labor costs, and operational downtime costs [10]. Vulnerability risk is the negative consequence of not mitigating or remediating the vulnerability. Change risk is any unforeseen negative consequence of applying the mitigation or remediation. Both of these are difficult to reason about because most vulnerabilities and most changes are low impact. Most vulnerabilities are not exploited, and most changes are managed properly via regression testing, incremental roll outs, etc. such that they do not break an organization's infrastructure. But some vulnerabilities let in ransomware that freezes the oil pipeline for four days, and some updates flag critical Windows DLLs as malware and render the machine inoperable. Labor and downtime costs are more predictable, but not negligible. If the cost of creating or applying a patch is greater than the cost of not creating or applying the patch, then good software management practices will not happen.

An efficient software security update delivery methodology will remain important if there will always be another vulnerability. Efficiency is important at all levels, from speed of development through to low cost to the end user for quick deployment. This suggests some software designs and development methods will be better suited to deliver efficient software security updates than others. These should be identified and preferred. As the SolarWinds incident demonstrated, software updates are also a risk as an attack vector. Due care is therefore required.

Finally, how to prioritize use of limited resources for vulnerability management remains an open and important question. If there will always be more vulnerabilities, then the technical details of each specific vulnerability matter less and the context of which vulnerabilities are actively being exploited by attackers to cause material harm matters more. This preference is reflected in some vulnerability prioritization systems, such as the Stakeholder-Specific Vulnerability Categorization (SSVC) [29].

8 Conclusions

Vulnerabilities appear to be dense, not in the proper mathematical sense but in the sense that there will always be more vulnerabilities in a given piece of software. While vulnerabilities can be reduced, and best practices for secure software development need to continue, under the modern computing paradigm a pragmatically useful piece of software cannot be produced without vulnerabilities. This conclusion suggests three conclusions: adjust defender expectations, adjust defender practice, and consider drastic adjustments to the modern computing paradigm.

Defenders should expect there to continually be new vulnerabilities. When there is a new vulnerability in an important system, the first reaction should not be surprise or denial. This change in expectation does not mean we stop doing all the best practices to limit vulnerabilities in software. The efforts to reduce and limit vulnerabilities in software have made important and valuable improvements. And these practices not only reduce vulnerabilities but also increase system reliability and fit to intended purpose. Nonetheless, leadership at organizations should not be surprised when software contains a previously unknown vulnerability. This ongoing risk should be surfaced so that decisions can account for it and an organization can own the risk in its software.

Defender practice should shift towards resiliency and efficient response. This advice is not new, but defenders still, to some extent, practice a castle mentality. Of course, defenders should maintain a border and patch CVE-IDs. However, defenders always need to segment and build resilient networks because the defender would be justified in believing the systems contain unknown vulnerabilities. Ongoing response to vulnerabilities actively being exploited by adversaries should become routine so that broader strategic resilience and defense can be built in.

One more drastic conclusion the community could reach is that Turing machines are actually too powerful. There are other abstract computation machines that are demonstrably less powerful than a Turing machine. While they are also less flexible, because they are less powerful it would also be possible to fully verify their operations. Current computer chips are designed such that arbitrary computation and arbitrary transitions are possible. This flexibility means that the chip designer does not need to know what software will be loaded and run. An alternative would be to make computer chips more restrictive, following a computational model of a finite state machine. The drawback is that the community would likely have to standardize what states and transitions are possible, and there will not be worldwide consensus on this. So different chips would likely become necessary for different purposes. Where flexibility remains important, one could use a field programmable gate array for customization of the hardware by the end consumer. Such a change would have tremendous ramifications on the current design, deployment, and maintenance of information systems. However, the drastic step of reducing computing devices to implementations of finite state machines may be the only way to overcome the theoretical barriers to eliminating vulnerabilities in Turing machines. Whether this shift would be economically justified would be a valuable topic for future research.

References

[1] Ross J. Anderson. Why information security is hard: an economic perspective. In *Computer Security Applications Conference*, pages 358–365, New Orleans, LA, December 2001. IEEE.

[2] Krzysztof R. Apt. Ten years of Hoare's logic: a survey—Part I. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):431–483, 1981.

[3] Elaine Barker. Recommendation for key management: Part 1 – general. Technical Report SP 800-57r5, US Dept of Commerce, National Institute of Standards and Technology, Gaithersburg, MD, May 2020.

[4] Vilius Benetis, Olivier Caleff, Cristine Hoepers, Angela Horneman, Allen Householder, Klaus-Peter Kossakowski, Art Manion, Amanda Mullens, Samuel Perl, Daniel Roethlisberger, Sigitas Rokas, Mary Rossell, Robin M. Ruefle, D'esir'ee Sacher, Krassimir T. Tzvetanov, and Mark Zajicek. Computer security incident response team (CSIRT) services framework. Technical Report ver. 2, FIRST, Cary, NC, USA, July 2019.

[5] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *Symposium on Security and Privacy*, pages 483–502. IEEE, May 2017.

[6] Marcel Böhme and Soumya Paul. A probabilistic analysis of the efficiency of automated software testing. *Transactions on Software Engineering*, 42(4):345–360, 2015.

[7] George S. Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and logic*. Cambridge University Press, Cambridge, 4th edition, 2002.

[8] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods*, number 9058 in LNCS, pages 3–11. Springer, 2015.

[9] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Logic in Computer Science*, pages 366–378. IEEE, 2007.

[10] James L Cebula and Lisa R Young. A taxonomy of operational cyber security risks. Technical Report CMU/SEI-2010-TN-028, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2010.

[11] CISA. OpenSSL "Heartbleed" vulnerability (cve-2014-0160). Technical Report TA14-098A, US Cybersecurity and Infrastructure Security Agency, April 2014.

[12] James H Fetzer. Program verification: the very idea. *Communications of the ACM*, 31(9):1048–1063, 1988.

[13] William Frakes and Carol Terry. Software reuse: metrics and models. *ACM Computing Surveys (CSUR)*, 28(2):415–435, 1996.

[14] Dan Geer. Cybersecurity as realpolitik. In *Black Hat USA 2014*, Las Vegas, Nevada. UBM.

[15] Allen D. Householder, Garret Wassermann, Art Manion, and Christopher King. The CERT® guide to coordinated vulnerability disclosure. Technical Report CMU/SEI-2017-TR-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2020.

[16] Leslie Lamport. What good is temporal logic? In R.E.A. Mason, editor, *IFIP Congress*, pages 657–668. Elsevier, 1983.

[17] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley, Boston, MA, USA, 2002.

[18] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[19] Office of the DoD Chief Information Officer. DoD vulnerability management. Technical Report 8531.01, US Department of Defense, Washington, DC, 2020.

[20] Peter W. O'Hearn. From Categorical Logic to Facebook Engineering. In *Logic in Computer Science (LICS)*, pages 17–20. IEEE, 2015.

[21] Andy Ozment and Stuart E Schechter. Milk or wine: does software security improve with age? In *USENIX Security Symposium*, volume 15, pages 93–104, Vancouver, B.C., Canada, 2006.

[22] Dale Peterson. Medical cybersecurity & dense vulnerabilities, feb 2018.

[23] Kenneth H Pollock. Review papers: modeling capture, recapture, and removal statistics for estimation of demographic parameters for fish and wildlife populations: past, present, and future. *Journal of the American Statistical Association*, 86(413):225–238, 1991.

[24] David Pym, Jonathan M Spring, and Peter O'Hearn. Why separation logic works. *Philosophy & Technology*, 32(3):483–516, 2018.

[25] Panu Raatikainen. Gödel's incompleteness theorems. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2015 edition, 2015.

[26] Bruce Schneier. Should U.S. hackers fix cybersecurity holes or exploit them? *The Atlantic*, May 14, 2014.

[27] R. Shirey. Internet Security Glossary, Version 2. RFC 4949 (Informational), August 2007.

[28] Jonathan M Spring, April Galyardt, Allen D Householder, and Nathan VanHoudnos. On managing vulnerabilities in AI/ML systems. In *New Security Paradigms Workshop*, Virtual conference, October 2020. ACM.

[29] Jonathan M Spring, Eric Hatleback, Allen D. Householder, Art Manion, and Deana Shick. Prioritizing vulnerability response: A stakeholder-specific vulnerability categorization. In *Workshop on the Economics of Information Security*, Brussels, Belgium, December 2020.

[30] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230–265, November 1936.

Contact Us

Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA 15213-2612
Phone: 412.268.5800 | 888.201.4479
Web: www.sei.cmu.edu
Email: info@sei.cmu.edu