Intrinsic Propensity for Vulnerability in Computers? Arbitrary Code Execution in the Universal Turing Machine

Pontus Johnson
KTH Royal Institute of Technology
Stockholm, Sweden
pontusj@kth.se

Abstract

The universal Turing machine is generally considered to be the simplest, most abstract model of a computer. This paper reports on the discovery of an accidental arbitrary code execution vulnerability in Marvin Minsky's 1967 implementation of the universal Turing machine. By submitting crafted data, the machine may be coerced into executing user-provided code. The article presents the discovered vulnerability in detail and discusses its potential implications. To the best of our knowledge, an arbitrary code execution vulnerability has not previously been reported for such a simple system.

1 Introduction

Arbitrary code execution holds a special position among malicious exploits. Most remarkable is the case when such code execution is effected through the submission of crafted data. In computing, the relationship between structure and behavior, between program and process, is perplexing in itself. That this relationship so often can be subverted, allowing an untrusted data provider to preternaturally gain control over program execution, is disquieting. Why is this a common phenomenon in computer systems? Is it the consequence of incidental but unfortunate decisions in the development history of those systems, or is it rather the result of some fundamental property of computing?

Commonly used to explore the foundational traits of computers and computing, the *universal Turing machine* is generally considered one of the most important ideas in computer science. Turing presented his universal machine in a paper in 1936 [14], where he promptly used it to solve one of the most pressing mathematical questions of the day, David Hilbert and Wilhelm Ackermann so called *Entscheidungsproblem* [7]. As expressed by Marvin Misky, "the universal machine quickly leads to some striking theorems bearing on what appears to be the ultimate futility of attempting to obtain effective criteria for effectiveness itself" [8] . But the universal Turing machine achieved more than that. As stated by Davis, Sigal and Weyuker in [4], "Turing's construction of a universal computer in 1936 provided reason to believe that, at least in principle, an all-purpose computer would be possible, and was thus an anticipation of the modern digital computer." Or, in the words of Stephen Wolfram [16], 'what launched the whole computer revolution is the remarkable fact that universal systems with fixed underlying rules can be built that can in effect perform any possible computation." Not only the universality, but also the simplicity of the universal Turing machine has attracted interest. In 1956, Claude Shannon explored some minimal forms of the universal Turing machine [13], and posed the challenge to find even smaller such machines. That exploration has continued to this day [16].

A common strategy for understanding a problem is to reduce it to its minimal form. In the field of computer security, we may ask the question: "What is the simplest system exploitable to arbitrary code execution?" In this article, we propose an answer to that question by reporting on the discovery that a well-established implementation [8] of the universal Turing machine is vulnerable to a both unintentional and non-trivial form of arbitrary code execution.

The article proceeds in the next section with a background to arbitrary code execution. Section 3 reviews universal Turing machines, and in particular the studied implementation. This is followed by a detailed analysis of the discovered vulnerability. In Section 5, we consider changes to the explored implementation that would mitigate the vulnerability. The paper concludes with a discussion on the significance of the findings, and some conclusions.

2 Arbitrary Code Execution

That a software user who is nominally only granted the possibility to provide some trivial data, such as her name, sometimes, by carefully crafting that seemingly inconsequential data, is able to take full control of the computer executing that software, is remarkable indeed. It is even more arresting that such arbitrary code execution vulnerabilities are quite frequently discovered in software systems. Arbitrary code

execution is not a fringe phenomenon, but a material class of vulnerabilities in modern computer systems. There are several specific types of vulnerabilities that may lead to arbitrary code execution. Among the 2019 CWE (Common Weakness Enumeration) Top 25 Most Dangerous Software Errors the following may lead to arbitrary code execution [9],

CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer
CWE-79	Improper Neutralization of Input During Web Page Generation
CWE-20	Improper Input Validation
CWE-89	Improper Neutralization of Special Elements used in an SQL Command
CWE-416	Use After Free
CWE-190	Integer Overflow or Wraparound
CWE-78	Improper Neutralization of Special Elements used in an OS Command
CWE-787	Out-of-bounds Write
CWE-476	NULL Pointer Dereference
CWE-434	Unrestricted Upload of File with Dangerous Type
CWE-94	Improper Control of Generation of Code
CWE-502	Deserialization of Untrusted Data

Table 1: Vulnerabilities that may lead to code execution

Those twelve constitute half of that top 25 list, highlighting the prevalence of this class of vulnerability. It is not, however, clear whether there is any common underlying cause to these vulnerabilities; is there any root explanation as to why they are so prevalent?

3 Universal Turing machine

As preparation for the presentation of the arbitrary code execution vulnerability in Section 4, we review the concept of the Turing machine, the universal Turing machine, and the Minsky implementation of that universal machine.

3.1 Turing machine

A *Turing machine*, T, is a *finite-state machine* operating on a *tape* by means of the machine's *head* (cf. Figure 1). The tape has the form of a sequence of squares onto one of which the head is positioned. The head can read and write symbols located in the currently scanned square. It can also move one square to the left or right.

The input to the finite-state machine is the currently scanned symbol, while the output is the printed symbol as well as the direction in which the head is to move. The finite-state machine, which thus controls the actions of the head, can therefore be represented as a quintuple,

$$Q_i S_i \; Q_{ij} S_{ij} D_{ij}$$

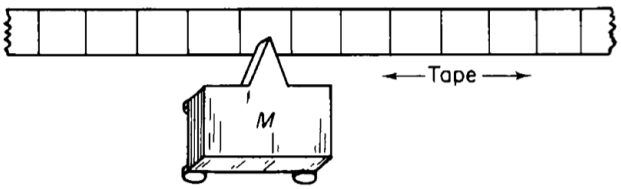

Figure 1: A Turing machine.

where Q_i represents the source state, S_i the scanned symbol, Q_{ij} the target state, S_{ij} the printed symbol and D_{ij} the direction in which the head is to move.

3.2 Universal Turing machine

A *universal* Turing machine, U, is a Turing machine that is capable of simulating any other Turing machine, T. There are multiple implementations of the universal Turing machine, the first one notably being the one proposed by Alan Turing himself in [14]. In this article, we consider the universal Turing machine proposed by Marvin Minsky in [8] . Our choice of the Minsky version is mainly based on (i) the ease with which it can be implemented, (ii) the ease with which it can be explained in a brief article, and (iii) it's solid place in computer science literature, presented by Marvin Minsky in his much-cited book *Computation: Finite and Infinite Machines* (1967). Turing's own universal machine is arguably more convoluted, and also contains a number of errors [3]. Other universal Turing machines include a set of minimally small universal Turing machines, counting the size of their alphabet and finite-machine state space [11] [17]. Those machines, however, add cognitive complexity by introducing an additional formalism (a *tag system*) in order to minimize the size of the machines.

3.2.1 Machine structure

In the words of Minsky himself, the universal machine, U,

> will be given just the necessary materials: a description, on its tape, of T and of [the initial configuration on Ts own, simulated tape] s_x; some working space; and the built-in capacity to interpret correctly the rules of operation as given in the description of T. Its behavior will be very simple. U will simulate the behavior of T one step at a time. It will be told by a marker `M` at what point on its tape T begins, and then it will keep a complete account of what T's tape looks like at each moment. It will remember what state T is supposed to be in, and it can see what T would read on the 'simulated' tape. Then U will simply look at the description of T to see what T is next supposed to do, and do it! This really involves no more than looking up, in a table of

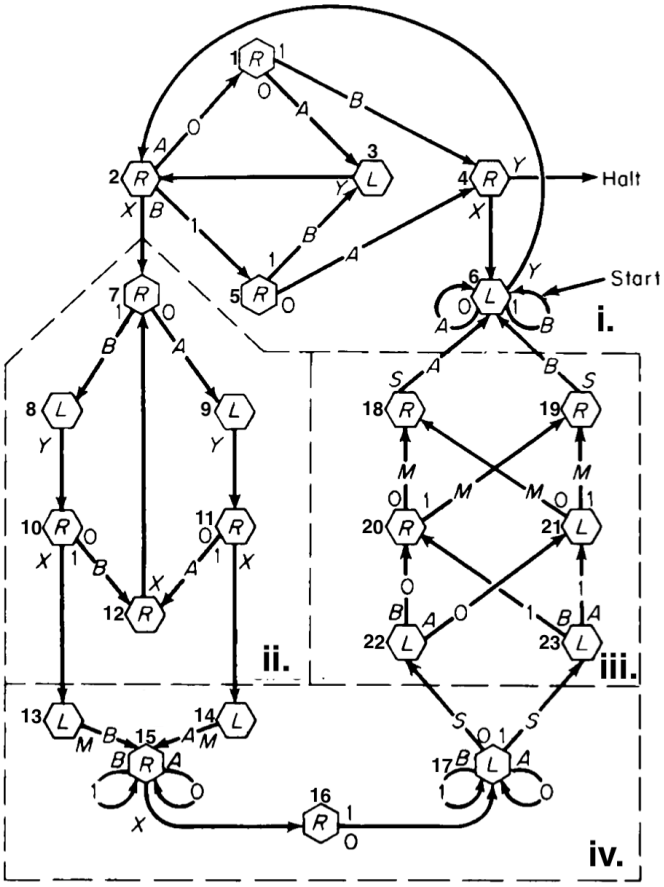

Figure 2: Finite-state machine of Marvin Minsky's universal Turing machine.

> quintuples, to find out what symbol to write, which way to move, and what new state to go into. We will assume that T has a tape which is infinite only to the left, and that it is a binary (2-symbol) machine. These restrictions are inessential, but make matters much simpler.

Concretely, Us tape is divided into four regions. The infinite region to the left will be the tape of the simulated machine T. The second region, $q(t)$, contains the name of the current state of T. The third region, $s(t)$ stores the value of the symbol under T's head. Together, we denote $q(t)s(t)$ the *machine condition*. The fourth region, d_T will contain the machine description of T, i.e. the program.

If U is to simulate the binary counter represented in Figure 3, U's tape may initially be configured as follows,

```
...00000M000Y001X0000001X0010110X0100011X0110100Y00...
                ↑
```

where the arrow points to the location of U's head, the `M` marks the location of T's head, the leftmost `Y` separates T's tape from $q(t)$, and the leftmost `X` identifies the start of the machine description, d_T. Within d_T, quintuples are separated by `X`s, and the rightmost `Y` marks the end of the machine description.

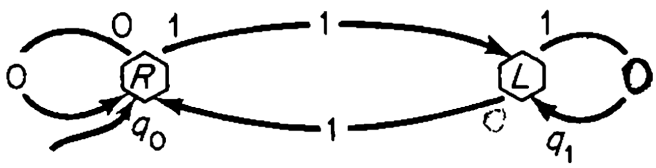

Figure 3: The state machine of a binary counter Turing machine.

The machine description of T is constituted of a set of quintuples, $Q_i S_i Q_{ij} S_{ij} D_{ij}$, recorded in binary format. An example would be `0010110`, where $Q_i =$ `00`, $S_i =$ `1`, $Q_{ij} =$ `01`, $S_{ij} =$ `1`, and $D_{ij} =$ `0`. Any number of binary digits may be used to represent the machine state, Q. We adopt the convention that $D_{ij} =$ `0` indicates a shift of the head to the left, while $D_{ij} =$ `1` shifts the head to the right.

3.2.2 Machine execution

U's finite-state machine, presented in Figure 2, is constituted of four distinct phases, marked as i-iv in the figure. The first phase uses T's state, $q(t)$, and the symbol under T's head, $s(t)$, to identify the next quintuple to execute. A recurring approach for marking positions is to recast `0`s and `1`s into `A`s and `B`s. The example tape presented above would by the first phase be modified to

```
...00000M000Y001XAAAAAAABXAAB0110X0100011X0110100Y00...
                ↑
```

where the transition from `A`s and `B`s into `0`s and `1`s specifies the position of the quintuple (`0010110`, in the example) matching $q(t)$.

From the identified quintuple, the second phase copies the target state, Q_{ij} (`01`), and the symbol to be written, S_{ij} (`1`), to $q(t)$ and $s(t)$, and remembers the direction D_{ij} by entering into the appropriate state, (state 13 or 14 in Figure 2).

```
...00000M000YABBXAAAAAAABXAABABBAX0100011X0110100Y00...
                ↑
```

The third phase records that direction by replacing T's head symbol `M` with an `A` or `B` (`A` in the example), performs some clean-up, replaces the symbol to be written, S_{ij}, stored in $s(t)$, with an `S`, and instead remembers S_{ij}.

```
...00000A000Y01SX0000001X0010110X0100011X0110100Y00...
                ↑
```

The fourth and final phase performs the actual operations of T: it prints S_{ij} (`1`) in the appropriate location on T's tape,

places an M to the left or right of that symbol depending on D_{ij}, and performs some final clean-up.

```
...0000M1000Y01AX0000001X0010110X0100011X0110100Y00...
              ↑
```

At this point, the first execution cycle is complete, and the second cycle begins.

4 Exploiting the Universal Turing Machine

Users of computer systems typically provide the input to, or argument of the computations, and are provided the results. From the point of view of computer security, it is typically undesirable to allow the user to subvert the functionality of the program performing the function. A malicious actor may, however, attempt to do so. A particularly serious security vulnerability is when it is possible for the end user to provide maliciously crafted data that effectively allows the execution of arbitrary code. In this section, we demonstrate that Marvin Minsky's universal Turing Machine suffers from an arbitrary code execution vulnerability.

4.1 Trust boundary

There is one obvious trust boundary in a universal Turing machine, U: the initial string on the tape of the simulated Turing machine, T. That string corresponds to the user-provided data of an ordinary computer program. Because the potential users may be unknown to the developers and administrators of the computer and its programs, it is common to view this data as untrusted. In our explorations of the universal Turing machine, we will make the same assumption. Therefore, if it were possible to execute arbitrary code without manipulating the program of T, but only by providing crafted data on T's simulated tape, that would constitute a vulnerability.

4.2 Requirements on the machine description

Nearly all possible machine descriptions of T appear to be vulnerable to arbitrary code execution. We consider the case when the first executed quintuple is of the form $Q_0S_0Q_{01}S_{01}0$. The final symbol is thus fixed to a 0, indicating that the direction of the head must thus not shift right in the first execution cycle. This is arguably in line with Minsky, p.138, [8]: "We will assume that T has a tape which is infinite only to the left [...]". Because exploitation occurs already in the first executed quintuple, additional quintuples will not affect the outcome.

4.3 The exploit

The following crafted input data will achieve arbitrary code execution by injecting a new Turing machine I, and coercing U into simulating it:

$$\Delta \mathtt{Y} q(i) \mathtt{AX} Q_0S_0Q_{0x}S_{0x}D_{0x}\mathtt{X}Q_1S_1Q_{1y}S_{1y}D_{1y}\ldots\mathtt{S}$$

where Δ represents the input data provided to the injected machine, I, $q(i)$ denotes the name of I's current state, and $Q_xS_xQ_{xy}S_{xy}D_{xy}$ are the quintuples of I. All variable values need to be coded as Bs and As instead of 1s and 0s. $q(i) = \mathtt{BA}$ if $S_{xy} = 1$ of the first executed quintuple and $q(i) = \mathtt{AA}$ if $S_{xy} = 0$.

4.4 Example exploit

We explain the exploitation mechanism by an example, where T's machine description, d_T, consists of a simple program acting as a binary counter, according to Figure 3,

$$d_T = \mathtt{0000001X0010110X0100011X0110100}$$

U's initial tape is laid out as follows:

```
M000Y001X0000001X0010110X0100011X0110100Y00
```

The injected machine, I, will aim to wipe the tape clean of user input, thus writing a 0 whenever encountering either a 0 or a 1, and then shifting left. This can be accomplished with two quintuples:

$$d_I = \mathtt{0000000X0010000}$$

According to the previous subsection, the crafted input will take the form

```
1111YBAAXAAAAAAAXAABAAAAS
```

where 1111 is the data on which the injected machine, I will operate, $q(i) = \mathtt{BA}$ is the injected machine state, $s(i) = \mathtt{A}$ is the currently scanned symbol, and AAAAAAAXAABAAAA encodes I's machine description, d_I, thus representing the wiper program.

4.4.1 First execution cycle

At the start of execution, U's tape has the following appearance:

```
...001111YBAAXAAAAAAAXAABAAAASM000Y001X0000001X0010110X0...
                                  ↑
```

with U's head positioned on the X between T's currently scanned symbol, $s(t)$, and machine description, d_T. The first three phases of U follow the description in Section 3, finding the identity of the quintuple, 0010110 stored in the machine condition, $q(t)s(t) = 001$, and replacing that machine condition with the action part of the identified quintuple, $Q_TS_T = 011$.

```
...001111YBAAXAAAAAAAAXAABAAAASA000Y01SX0000001X0010110X0...
                                  ↑
```

In the fourth phase, U aims to perform the action on T's tape as specified by the retrieved quintuple. It does write a `1` at the expected location, but then, however, the crafted input, consisting of `A`s and `B`s instead of the expected `0`s and `1`s, causes U's head to shift far left into the user-provided data, placing the marker, `M`, representing T's head, at an unexpected location.

```
...00111MYBAAXAAAAAAAAXAABAAAAB1000Y01SX0000001X0010110X0...
                               ↑
```

At the end of U's first execution cycle, not only T's, but also U head comes to rest further to the left than expected. Importantly, U's head is located to the left of the symbol `Y` indicating the end of T's tape.

4.4.2 Second execution cycle

Because U's head is located in the attacker-controlled segment of the tape, in it's attempt to identify the next quintuple to execute, the first phase of U's second execution cycle mistakenly refers to the injected machine condition, $q(i)s(i) = \texttt{BAA}$. Looking for `0`s and `1`s rather than `A`s and `B`s, it won't find anything in the injected machine description, d_I. Instead, it encounters the first match, `100` at a rather random location, just before the `Y` representing the end of T's tape. The initial `1` is the result of T's first and successful print operation. The ensuing `00` are simply a part of a buffer between T's tape and machine condition, $Q_T S_T$, as introduced by Minsky in [8].

```
...00111MY100XAAAAAAAAXAABAAAABBAA0Y01SX0000001X0010110X0...
              ↑
```

In the second phase, attempting to collect the action part of the identified quintuple, U will find the four digits closest to the right of its head. While these were supposed to constitute the tail end of a quintuple, they are instead are pieces of the aforementioned buffer, of T's machine condition, $q(t)s(t)$, and T's first quintuple, jointly creating the string `0010`, which is thus interpreted as $Q_{ij}S_{ij}D_{ij}$. Furthermore, in the middle of the attempt to copy the first three digits to the injected machine condition, $q(i)s(i)$, U slips back to the right of the `Y` indicating the start of T's machine condition, $q(t)s(T)$. The end result is that the first digit is copied to $q(i)$ while the remaining part is copied to $q(t)s(t)$. At the end of this phase, the complete tape has the following layout:

```
...00111MYA00XAAAAAAAAXAABAAAABBAAAYAASXA000001X0010110X0...
              ↑
```

The third phase reverts the `A`s and `B`s to `0`s and `1`s, and replaces T's head, `M`, with a symbol indicating the direction of the next shift. U's head is once again positioned far into the untrusted, user-provided data.

```
...00111AY00SX0000000X00100001100OY00SX0000001X0010110X0...
           ↑
```

In the fourth and final phase of the second execution cycle, U shifts T's head one step and records in I's machine condition, $q(i)s(i)$, the symbol under M.

```
...0011M0Y00BX0000000X00100001100OY00SX0000001X0010110X0...
          ↑
```

At this point, the compromise is complete, as the injected machine, I, is syntactically correct, and the head of U is located in the injected machine condition, $q(i)s(i)$, rather than in the originally intended one, $q(t)s(t)$. The head will never again traverse the `Y` denoting the end of T's tape, interpreting it instead as the end of I's machine description, d_I.

4.4.3 Subsequent execution cycles

The following execution cycles will faithfully execute the injected machine, I, wiping the contents of the inputs provided I.

```
...001M00Y00BX0000000X00100001100OY00SX0000001X0010110X0...
...00M000Y00BX0000000X00100001100OY00SX0000001X0010110X0...
...0M0000Y00AX0000000X00100001100OY00SX0000001X0010110X0...
...M00000Y00AX0000000X00100001100OY00SX0000001X0010110X0...
```

5 Mitigations

It is possible to improve on the Minsky implementation in order to mitigate the presented vulnerability. As a first mitigation strategy, we propose to validate inputs. This could be performed by introducing a preprocessing phase validating that the simulated machine's tape only consists of the expected `0`s and `1`s.

Secondly, we can restrict the execution space by reducing the number of defined quintuples. To simplify the finite-state description captured by Figure 2, Minsky declares many quintuples implicitly: "The most common quintuples, of the form (qi, sj, qi, sj, dij) are simply omitted [in the diagram]." [8]. This is convenient, because explicitly adding all necessary such quintuples to the diagram would require close to 70 arrows in addition to the 45 currently in the diagram. However, because the implicit definition creates approximately twice as many quintuples as the 70 required, this strategy allows the machine to accept many tape symbols that are not necessary for the proper functioning of the machine. Of the 70 unnecessary quintuples, close to 30 were required in order to allow the exploitation demonstrated in the previous section. This mitigation does require some effort, as the 70 required quintuples must be specified.

Thirdly, we could fortify the division between program and data. In Minsky's implementation, only 17 of the 184

quintuples defining U are supposed to operate on T's tape. By, for instance, using special symbols on T's tape, and ensuring that only the privileged 17 quintuples read and write on that alphabet, additional barriers to exploitation would be established.

6 Discussion

We return to the puzzling prevalence of arbitrary code execution vulnerabilities. Are these the result of some fundamental property of computing, or are they rather the consequence of coincidental, unfortunate decisions during the development of those systems?

It is interesting to note that, as was the case for Minsky's universal Turing machine, arbitrary code execution vulnerabilities can be accidentally introduced even in the simplest computer model. Minsky obviously attempted to design neither a secure nor a vulnerable system, but despite his indifference, he happened to design a vulnerable machine. That would suggest that vulnerability is a property that is not unlikely to arise in universal Turing machines. The volume of vulnerabilities discovered in computer systems in recent decades would further support such a proposition. Is it then the case that computers are intrinsically brittle - that they at their very core have a propensity to arbitrary code execution vulnerabilities?

Considering the exploitation of the universal Turing machine in the previous section, we may speculate about reasons for such a potential propensity. One suggested root cause of computer insecurity is *complexity* [12]. While there is surely truth to that statement, the insecurity of Minsky's minimally complex computing machine would appear to indicate a propensity to vulnerability even in the absence of complexity.

A related theory points the finger at the human factor [6]; poor decision-making by people is the cause of insecurity. But also this hypothesis insufficiently explains the problems of Minsky's universal Turing machine. One could possibly argue that Marvin Minsky suffered from a lack of security awareness, but that would not alone explain the demonstrated possibility for the user to achieve code execution. There is also something inherent to the machine that makes it not only theoretically possible, but oftentimes also actually the case.

Another proposed theory blames John von Neumann's *stored program concept* [15] for the woes of arbitrary code execution; the fact that data and program in computers are stored on the same storage medium may allow an attacker to illegitimately modify the program rather than the intended data [10]. Considering the universal Turing machine, this does indeed seem to have something to do with its vulnerability. The exploit demonstrated in the previous section is striking in the manner the machine head so unquestioningly saunters from program to data. This does intimate that the answer is at least partially related to the flimsy border between program and data. However, it can only be a partial answer to the question. While the co-location of program and data in the same memory unit might be a cause for the vulnerability of the universal Turing machine, of the general prevalence of buffer overflows (CWE-119) and use-after-free vulnerabilities (CWE-416), it is less obvious how it would constitute a root cause of SQL injection vulnerabilities (CWE-89) or operating system command injection vulnerabilities (CWE-78). Return-oriented programming [2] is another argument against the stored-program hypothesis: in such an attack, the program is never modified. Instead the program *flow* is deftly controlled by the attacker, cherry picking among assembly statements in the unmodified original program text to concoct attacker-controlled behavior. Return-oriented programming can almost always substitute modification of the actual program. Even if programs are located in a memory separate from data, they typically need to be modifiable by some means - if they are hard coded, the infinitely flexible universal Turing machine reduces to yet another specific Turing machine. And if the programs are indeed located in a separate but writeable memory (e.g. as in the Modified Harvard Architecture), then it would appear that the vulnerabilities of modifiable program code are back, e.g. as demonstrated in [5].

A final theory that may be on the cusp of explaining computers' propensity to vulnerability is suggested by Bratus et al. in the ;login: article *Exploit Programming* [1], considering the *weird* and oftentimes surprisingly potent *machines* that may appear when unexpected, crafted input is provided to a computer program. A remarkable prevalence of such machines, is, I believe, at the heart of the problem.

7 Conclusion

This paper presents the discovery of an arbitrary code execution vulnerability in Marvin Minsky's 1967 universal Turing machine implementation. By submitting crafted input data, an attacker can coerce the machine into executing arbitrary instructions. While this vulnerability has no real-world implications, we discuss whether it indicates an intrinsic propensity for arbitrary code execution vulnerabilities in computers in general.

Availability

A Python program implementing and exploiting the Minsky Turing machine considered in this paper is available on GitHub at https://github.com/intrinsic-propensity/turing-machine.

References

[1] Sergey Bratus, Michael E Locasto, Meredith L Patterson, Len Sassaman, and Anna Shubina. Exploit programming: From buffer overflows to weird machines and theory of computation. *USENIX; login*, 36(6), 2011.

[2] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572, 2010.

[3] B Jack Copeland. *The essential turing*. Clarendon Press, 2004.

[4] Martin Davis, Ron Sigal, and Elaine J Weyuker. *Computability, complexity, and languages: fundamentals of theoretical computer science*. Elsevier, 1994.

[5] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26, 2008.

[6] Jose J Gonzalez and Agata Sawicka. A framework for human factors in information security. In *Wseas international conference on information security, Rio de Janeiro*, pages 448–187, 2002.

[7] David Hilbert and Wilhelm Ackermann. Grundzüge der theoretischen logik, berlin 1928. *Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen mit besonderer Berücksichtigung der Anwendungsgebiete*, 27, 1938.

[8] Marvin Lee Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall Englewood Cliffs, 1967.

[9] Mitre. 2019 cwe top 25 most dangerous software errors. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html, 2019. Accessed: 2020-02-07.

[10] G Paul. Method and apparatus for providing a security system for a computer, July 3 1973. US Patent 3,744,034.

[11] Yurii Rogozhin. Small universal turing machines. *Theoretical Computer Science*, 168(2):215–240, 1996.

[12] Bruce Schneier. A plea for simplicity: You can't secure what you don't understand. *Information Security*, 1999.

[13] Claude E Shannon. A universal turing machine with two internal states. *Automata studies*, 34:157–165, 1956.

[14] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1936.

[15] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[16] Stephen Wolfram. *A new kind of science*, volume 5. Wolfram media Champaign, IL, 2002.

[17] Damien Woods and Turlough Neary. The complexity of small universal turing machines: A survey. *Theoretical Computer Science*, 410(4-5):443–450, 2009.