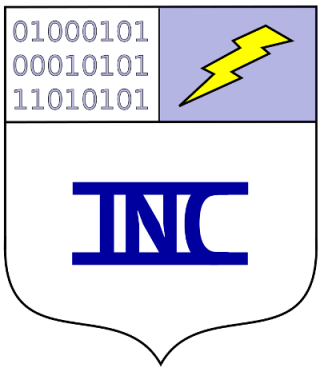
Computer Security is Algorithmically Intractable

Dr. Brent Kirkpatrick*

Aug. 17, 2018

Intrepid Net Computing



www.intrepidnetcomputing.com

*bbkirk@intrepidnetcomputing.com

Abstract

Computer security, as a problem, is difficult to define, and when we do invent formal definitions, most of those definitions appear to be undecidable. Building on the work of Alan Turning, where he proved that the halting problem is undecidable, we demonstrate that three definitions related to computer security are undecidable.

Practical computer security must be pursued, regardless of algorithmic intractability. Computer scientists can discover and innovate new tools for computer security that constitute breakthroughs.

Operating system security and network security hinge on skilled scientists securing small code-bases.

Brent Kirkpatrick

1 Introduction

How is computer security defined? Why is computer security so difficult?

Formal mathematical definitions relating to computer security have been demonstrated to be intractable. Defining computer security itself is a nebulous concept. Various attempts have been made, leading all the way back to the original proof of undecidability and the definition of the halting problem [1].

Since then other attempts include the definition of formal verification, the Post Correspondence Problem, and other problems [2]. In all these cases, mathematical proofs of

undecidability demonstrate that these problems have no algorithmic solutions. This means that there are no computer programs that can produce answers to these problems.

This manuscript introduces three new formalisms for computer security:

1. The Digital Forensics Problem,

2. The Hacking Detection Problem, and

3. The Buffer Over-Run Detection Problem.

The manuscript gives proofs that these problems are undecidable. This leaves open the question of whether computer security can be formalized in a way that is tractable or which special cases of computer security are tractable. For now, we discuss how to obtain computer security in the face of algorithmic intractability.

2 Background

Computer science is split into two subfields: theory and systems. The subfield of theory is devoted to understanding computational complexity theory, algorithms, game theory, quantum computing, computational biology, randomized algorithms, and machine learning. The subfield of systems is devoted to writing the computer programs for operating systems, networks, and databases. There are few scientists that span the divide between theory and systems.

The theory of computing is the study of algorithmic tractability and formal language theory [2]. The papers that founded this field where the Turing-Church thesis of 1936 [3, 1]. This established a formal definition of an algorithm. On that foundation, the original proof of undecidability was done by Alan Turing [1].

Computer systems builds on a foundation of good programming skills. These skills are built through experience in programming and through the use of formal methods. Formal

methods are rigorous tools from mathematics that establish guarantees about the operation of aspects of a computer program. Systems programmers write programs that work in practice.

Computer security is a subfield of computer systems that aims to secure computer systems against adversaries called hackers. From examining the work of hackers, we know that buffer over-runs are at the heart of most vulnerabilities [4, 5]. This is well known. Buffer over-runs are prevalent, and a number tools exist that attempt to detect buffer over-runs either statically [6, 7] or dynamically [8, 9]. All of these tools have false positives and false negatives in detection. None of these tools constitute formal methods, and this manuscript will elucidate why.

3 Results

Here we discuss new theoretical results for problems pertaining to computer security. Before we can give the theorems, we must formalize the problems. The main theorems are proven by reduction to the halting problem and by contradiction. We will also discuss the algorithmic intuition that leads to these reductions.

Dealing with one problem at a time, first we formalize digital forensics.

Problem 1 (Digital Forensics)**.** *Given a clean operating system installation and a hacked operating system installation (where all the legitimate operating system code is identical in both installations), identify every byte of foreign machine code that executes.*

Why is this problem mathematically rigorous in statement? Each operating system installation is represented as a string of bytes. All the bytes that pertain to the operating system are identical in both strings. The only executed instructions that will differ are the instructions for the foreign machine code that the hacker(s) introduced. This definition allows for these strings to contain flat data files that are never executed. This definition

even allows for those flat data files (i.e. configuration files) to differ on the two installations, although this is not mathematically necessary.

Because this problem is not as intuitive as we want, we will now define an equivalent-by-complexity problem:

Problem 2 (Hacking Detection)**.** *Given a clean operating system installation and a hacked operating system installation (where all the legitimate operating system code is identical in both installations), identify the* first *byte of foreign machine code that executes.*

This formalism is interesting because this first byte of foreign machine code has a name: *the hook*. Hacker's use the hook to load their machine code into memory for execution. This problem definition seeks to answer one question: does foreign machine code get executed? Oddly enough, in order to detect hacking, there must be at least one byte of foreign machine code that executes. This is equivalent to the previous definition by complexity. Meaning, if you are to find the first byte of foreign machine code, this is just as hard as finding all the bytes of foreign machine code. As we will see shortly.

The algorithmic intuition that explains the intractability of this problem to us is the following. In order to detect the first byte of foreign machine code, we would have to execute the two versions of the OS side by side, marking each common byte as it is executed. The first time we encounter a byte for execution that differs, we would flag that byte as foreign machine code. This intuition suggests that the problem is undecidable. We will do a formal reduction to the halting problem, next.

Our approach now is to reduce this problem to the halting problem. If we have a Turing Machine (TM) that can decide this Hacking Detection Problem, then we will use that TM to create a new TM that can decide the halting problem. This gives a contradiction and proves our claim of undecidability.

Theorem 1. *The Hacking Detection Problem is undecidable.*

Proof. Assume that we have a Turing Machine $hack_{TM}$ that decides the Hacking Detection Problem. We will prove by contradiction that no such TM can exist. We need to name several objects. Let I_{TM} be an arbitrary input Turing Machine. Now we will define $halt_{TM}$. This is designed to be a TM that decides the halting problem.

$halt_{TM}$ takes the arbitrary input I_{TM}, makes a modification called J_{TM}, and then runs $hack_{TM}$ on the clean input I_{TM} and the hacked input J_{TM}. $halt_{TM}$ returns true if and only if $hack_{TM}$ reports that the J_{TM} has foreign machine code.

The modified input J_{TM} is obtained from I_{TM} by adding a single print statement "this is hacked" to the end of the program. Alternatively, if there are multiple exit statements from the program then each of them must consist of a jump statement to the end of the I_{TM} program, after which the new statement is added.

J_{TM} will print "this is hacked" if and only if I_{TM} halts. This means that our definition of $halt_{TM}$ decides the halting problem. This is a contradiction to our assumption. Therefore we have proven that no Turing Machine can decide the Hacking Detection Problem. □

Corollary 1. *The Digital Forensics Problem is undecidable.*

Proof. Use the same construction, except include all the bytes of foreign machine code at the end of the modified TM. □

Next we discuss the problem of detecting buffer over-runs. The existing literature claims that buffer over-runs are a significant source of vulnerabilities, and so detecting them would improve computer security. Formally, the problem is as follows:

Problem 3 (Buffer Over-Run Detection)**.** *Every time a memory buffer (array) is written, check that the length of the input written to the buffer is within the bounds of the allocated buffer size.*

This is a formal definition, because the buffer is allocated (declared) once with a fixed

size. It can be written many times, but each time it is written, we must check that the length of data written to the buffer is within the limit declared during allocation.

The algorithmic intuition that explains the intractability of this problem to us is the following. In order to detect a buffer over-run, we might need to explore all the dynamic executions paths of the program (backward) in order to discover that there was a mistaken bound used to check the length of the input written to the array. As soon as we start exploring all the execution paths, we are begging to decide whether the program ever halts.

Before we can prove the formal reduction, we need to establish a result that seems counter-intuitive at first. The buffer over-run appears to be an artifact of finite memory machines.

Theorem 2. *Every Turing Machine can be written without buffer over-runs.*

Proof. Recall that the Turing Machine has an infinite tape and a customized alphabet. Therefore, we can simply declare the end of a buffer with a special symbol. Whenever we run into the end of a buffer, we can copy the whole stack down to a fresh portion of the tape, allocate new space, copy everything and then continue writing. Since the tape of a TM is infinite the memory for buffer allocations is effectively infinite. □

Once we understand this, then we know that any TM can be written without buffer over-runs. Now we can do the formal proof, by contradiction, that no algorithm exists for detecting buffer over-runs.

Theorem 3. *The Buffer Over-Run Detection Problem is undecidable.*

Proof. Assume that we have a Turing Machine (TM) that decides the Buffer Over-Run Detection Problem, called B_{TM}. Let us be given an arbitrary TM as input, I_{TM}. We will take this Turing Machine and modify it so that we can use B_{TM} to decide if I_{TM} ever halts.

Let our modified TM be called J_{TM}, and it will be defined as I_{TM} with two modifications:

1. Allocate a buffer b of length 0 at the beginning of J_{TM}

2. Add a buffer b write operation of length 2 to the very end of J_{TM}

Check also that every exit command in I_{TM} redirect to the end of I_{TM}, before those two modifications are made.

By Theorem 2, we can assume that the input I_{TM} has no buffer over-runs. Now, our decider B_{TM} applied to the input J_{TM} will detect a buffer over-run, because we wrote one in. This means that we detect the buffer over-run in J_{TM} if and only if I_{TM} halts. This is a contradiction to the undecidability of the halting problem. Therefore there is no Turing Machine that decides the Buffer Over-Run Detection Problem. □

We have discussed three proofs of undecidability for three formal problems in computer security. All three of these problems are motivated by practical applications in computer security. We are now faced with the challenge of discovering computer security without the direct aid of computer algorithms.

4 Practical Computer Security

These undecidability results for formal problems in computer security would seem to suggest that computer security is impossible. However, that is not true. An area of math that is algorithmically intractable is logic itself. Yet, every day productive mathematicians make useful contributions to our understanding of logic. The idea of undecidability simply suggests that a machine cannot solve the problem. Our professionalism as computer scientists is what drives us forward to do computer security with human intuition and human innovation. Operating system security and network security hinge on skilled scientists securing small code-bases.

Digital forensics is one task in computer security that is undecidable. There are numerous

proprietary and public methods used to do digital forensics. There are no guarantees that the digital forensics scientists will find what is there to find. However, good scientists routinely find interesting machine code and uncover critical vulnerabilities.

Incident response is a task in computer security that usually relies on digital forensics for its rigor. Incident response is the task of confronting ongoing cyberattack in real-time in order to secure a network. Incident response is done, accurately enough, by many IT security practitioners. In this case 'accurately enough' means that they are able to reduce the risk of hacking substantially enough that the hacking victim experiences relief.

Code reviews are done every day by skilled software engineers who find vulnerabilities in computer code. Code reviews look for buffer over-runs, for instances where user supplied code is executed, and for other tricky vulnerabilities. Code reviews are conducted of both closed and open source software systems. Sometimes sophisticated static or dynamic analysis tools are employed during code reviews. Sometimes weaknesses in the hardware are considered.

Security testing is performed by skilled testers on their own software. This is done with the idea of demonstrating that a vulnerability can be exploited. Testing, including security testing, will always be a mainstay of software engineering.

Writing less code is something some very sophisticated software companies do. It is clear that the attack surface of a computer program increases with the number of conditional statements. When very skilled programmers work to write very short programs and to prove guarantees about those programs, we have better security. These people are algorithms experts.

All of these tools can be used to do practical computer security. Practical computer security is an engineering goal that can be assessed with principles from software engineering. A degree of engineering rigor can be employed that makes computer security practical.

It is our task as professionals to continue delivering computer security. Even proofs of undecidability cannot dissuade us from our responsibilities. We need to look for solvable

formalisms and produce tools that help scientists do the hard work of securing operating systems and networks.

Biography. Dr. Kirkpatrick was a professor of computer science at the University of Miami before taking the opportunity to participate in computer security. He is a scientific researcher with expertise in systems, algorithms, data science, and artificial intelligence.

References

[1] Alan Turing. The undecidable, basic papers on undecidable propositions, unsolvable problems and computable functions. New York: Raven Press. *Davis, edited 1965*, 1936.

[2] Michael Sipser. *Introduction to the Theory of Computation.* Cengage Learning, Boston, 3rd edition, 2013.

[3] Alonzo Church. The undecidable, basic papers on undecidable propositions, unsolvable problems and computable functions. New York: Raven Press. *Davis, edited 1965*, 1934.

[4] S. Zubair and A. Zamani. SigFree: A signature-free buffer overrun and overflow attack blocker. *International Journal of Advanced Research in Computer Science and Software Engineering*, 4(3), 2014.

[5] E. Leontie, G. Bloom, O. Gelbart, B. Narahari, and R. Semha. A compiler-hardware technique for protecting against buffer overflow attacks. *Journal of Information Assurance and Security*, 5, 2010.

[6] D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. *NDSS*, pages 2000–02, 2000.

[7] D. Wagner and R. Dean. Intrusion detection via static analysis. *Security and Privacy*, 2001.

[8] O. Ruwase and M.S. Lam. A practical dynamic buffer overflow detector. *NDSS*, 2004.

[9] J. Newsome and D.X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. *NDSS*, 2005.