Chapter 5
Theoretical Foundation

What computers can and cannot do has been a long-standing topic in the foundation of computer science. Some of the pioneers of the field had a strong background in mathematics and, in the early days of computing, worked on the mathematical formulation of the limits of computation. The work led to the notion of decidability. Informally speaking, a question that can be answered by either yes or no is decidable if a computer can compute the correct answer in a finite amount of time.

The relation that the notion of decidability has to our problem of vendor trust should be obvious. If the question of whether an executable program performs malicious acts is decidable, we can hope to devise a program to check the code made by an untrusted vendor. If it is known to be undecidable, this conclusion should impact on where to invest our efforts. In this chapter, we review and explain some of the key results on decidability and explain how these results impact the problem of untrusted equipment vendors.

5.1 Gödel and the Liar's Paradox

The easiest accessible pathway into decidability is through the *liar's paradox*. Although we intuitively think that a statement is either true or false, it is possible to make an expression that is inconsistent if it is true and equally absurd if it is false. The liar's paradox is such an example: consider the statement, 'This statement is false.' If it is true, then it has to be false and, if it is false, then it has to be true; thus it can be neither true nor false.

The liar's paradox has been subject to long philosophical discussions throughout history. Its first application of relevance to our case was by the logician and mathematician Kurt Gödel [6]. Gödel used a slightly modified version of the liar's paradox to prove his first incompleteness theorem. This theorem states that no theory with a countable number of theorems is able to prove all truths about the relation of

O. Lysne, *The Huawei and Snowden Questions*, Simula SpringerBriefs on Computing 4, https://doi.org/10.1007/978-3-319-74950-1_5

natural numbers. Roughly, what Gödel did was replace the statement 'This statement is false' with 'This statement is not provable'. Clearly, if the latter statement is false, it has to be provable, meaning that it has to be true. This situation again implies that the statement has to be true but not provably so.

The incompleteness theorem of Gödel is not of direct interest to us, but his proof technique is. At the core of it lies the observation that there are limits to what a statement can say about itself without becoming an absurdity. Our interest in this question is as follows: we would like to understand what a program can say about a program or, more precisely, if a program can decide whether another program will behave maliciously.

5.2 Turing and the Halting Problem

Before we dive into the limits of what a computer can do, we need to have a firm understanding of what a computer is. The most common way to model the concept of a computer is through an abstract device described by Alan Turing [10].

There are several manifestations of this device. What they have in common is that they consist of a finite state machine, a read/writable tape, and a read/write head that is located over a position on the tape. The device operates by reading the position on the tape, changing the state of the state machine depending on what it read, writing a symbol to the tape depending on the state and what it read, and moving the tape forward or backward depending on the state and what it read. A more formal definition, similar to the definition given by Cohen [5], is as follows:

- Σ is a finite set of states for the Turing machine.
- Γ is a finite set of symbols that can be read from and written to the tape.
- Ω is a function $\Sigma \times \Gamma \to \Gamma$ that decides the symbol to be written to tape in each computation step.
- Δ is a function $\Sigma \times \Gamma \to \{-1, 0, 1\}$ that decides in which direction the tape should move after having written a symbol to the tape.
- Π is a function $\Sigma \times \Gamma \to \Sigma$ that decides the state that the Turing machine enters after having written a symbol to the tape.

The Turing machine starts with a tape with symbols on it and executes its operation by performing the functions defined by Ω, Δ, and Π in each execution step. In this definition, the computation of the machine halts when a computation step changes neither the symbol on the tape, the position of the tape, nor the state of the state machine (Fig. 5.1).

A famous postulate is that every function that is computable by any machine is computable by a Turing machine. Although this postulate has not been proven, it is widely believed to be true. In the years since its formulation, there has been no reasonable description of a machine that has been proven able to do computations that cannot be simulated by a Turing machine. Consequently, the strength of programming languages and computational concepts is often measured by their ability to simulate a

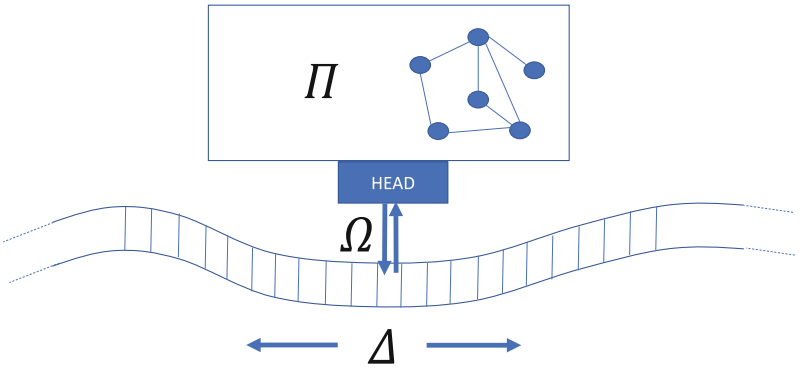

Fig. 5.1 A Turing machine. Based on the state of the state machine and the symbol on the tape, Ω decides the symbol to be written, Δ decides how the tape should be moved, and Π decides the new state of the machine

Turing machine. If they are able to, then we can assume that anything programmable can be programmed by the language or the concept.

The definition of a Turing machine and the postulation that it can simulate any other computing machine is interesting in itself. Still, its most important property, from our point of view, is that it can help us understand the limits of what a computer can do. Turing himself was the first to consider the limitations of computations, in that he proved that a computer is unable to decide if a given Turing machine terminates by reaching a halt. This is famously known as the halting problem and, to prove it, Turing used the liar's paradox in much the same way as Gödel did.

In a modernized form, we can state the proof as follows: assume that we have a programmed function P that, for any program U, is able to answer if U halts for all inputs. This would mean that $P(U)$ returns a value of *true* if U halts for all inputs and *false* if there is an input for which U does not halt. Then we could write the following program Q:

$$Q : \text{if } P(Q) \text{ then loop forever; else exit}$$

This program is a manifestation of the liar's paradox. If Q halts, then it will loop forever and, if it does not halt, then it halts. The only possible explanation for this is that our assumption that P exists was wrong. Therefore, no P can exist that is able to decide the halting problem.

5.3 Decidability of Malicious Behaviour

The importance of understanding the work of Gödel and Turing is that it forms the basis for a series of results on the analysis of what a piece of code actually does. This

basis was exploited by Cohen's [5] Ph.D. thesis from 1985. Cohen first defined the main characteristic of a computer virus to be its ability to spread. Then the author assumed the existence of a programmed function P with the ability to decide for any program U whether it spreads. Using the liar's paradox in the same way as Gödel and Turing, Cohen found that the following example code constituted an absurdity:

$$Q : \text{if } P(Q) \text{ then exit; else spread}$$

The reasoning is the same. If Q spreads, then it exits without spreading. If Q does not spread, then it spreads. Again, the only consistent explanation is that P does not exist.

Cohen's work was a breakthrough in the understanding of the limits of looking for malicious code. Subsequently, many developments extended undecidability into other areas of computer security. The most important basis for this development was the insight that the automatic detection of malicious machine code requires code with the ability to analyse code. By using the liar's paradox, we can easily generate absurdities similar to those described above. A general variant would be the following: Assume that P is a program that detects *any* specific behaviour B specified in any program code. Then we can write the following program:

$$Q : \text{ if } P(Q) \text{ then exit; else behave according to } B$$

In the same way as seen before, this is a manifestation of the liar's paradox and the only conclusion we can draw is that P does not exist.

Note that the latter example is entirely general, in that it does not specify what behaviour B is. This means that, by using different instantiations of B, we can conclude the following:

- It is undecidable if computer code contains a Trojan [4] (B defines the behaviour of a Trojan).
- It is undecidable if computer code contains the unpack–execute functionality often found in Trojans [8] (B defines the behaviour of an unpacker).
- It is undecidable if computer code contains trigger-based malware, that is, malicious computer code that will execute only through an external stimulus [2] (B defines the behaviour of trigger-based code).
- It is undecidable if computer code is an obfuscated version of some well-known malware behaviour [1] (B contains a specification of how the well-known malware behaves).

We will return to these bullet points in Chaps. 7 and 8, where the craft of malware detection is discussed in more detail.

5.4 Is There Still Hope?

The theoretical observations that we have elaborated on above appear to extinguish all hope that we can solve security problems in computer code. This is, of course, not true, since there is a lively and vibrant computer security industry out there accomplishing valuable work. In the remainder of this chapter, we take a look at where the rays of light are that are exploited by this industry. First, however, we make things even darker by correcting a common misunderstanding regarding the undecidability proofs above.

All the proofs above were based on the liar's paradox, meaning that a central part of the proof is that it is impossible to decide which branch of the following program will actually be followed:

$$Q: \text{ if } P(Q) \text{ then exit; else do something bad}$$

This situation has led to the very common misunderstanding that the only undecidable question is whether the code that does something bad is actually executed. Clearly, any detection method that found badly behaving code – regardless of whether it would be executed – would be of great help.

Unfortunately, identifying badly behaving subparts of code is also undecidable and the proof is very similar to those we cited above. A common misunderstanding stems from mixing up two proof concepts: proof by counterexample and *reductio ad absurdum*. The program Q above is not a counterexample in the sense that it is a program for which all P will have to give the wrong answer. Rather, Q is an absurdity that is implied by the existence of P and, consequently, P cannot exist. The proof itself sheds no light on for what subparts or subsets of programs P would actually exist.

A ray of light is to be found in the definition of the Turing machine itself. This is a theoretically defined machine that is not constrained by the realities of the physical world. Where a real-world machine can only work on a limited amount of memory, a Turing machine can have an infinitely long tape. Based on this insight, we have seen some limited results that identify decidable versions of the problems studied above. The existence of certain malware unpacking behaviour in code is shown to be decidable and NP-complete [3]. A similar result exists, where, under some restrictions, the decision of whether some code is an obfuscated version of another is proven to be NP-complete [1]. Additionally, for the detection of viruses, a restricted version of the problem is NP-complete [9].

However, NP-completeness is still a major problem. It does imply the existence of a system that, in finite time, can produce the right answer, but the amount of time needed rises very steeply with the size of the investigated program. For real-life program sizes, 'finite time' means 'finite but not less than a thousand years'. The practical difference between undecidable and NP-complete for our problem is therefore not significant. Still, the future may have further developments along

this axis that will help. Work on identifying decidable formulations of our problem therefore remains important.

A more promising possibility is to be found in the absoluteness in the definition of decidability. The proofs based on the liar's paradox hold as long as we require that P have no false negatives or false positives. In particular, when looking for malware injected by a vendor, we may be able to accept a relative high ratio of false positives, as long as there are few false negatives. Most would be happy to require a purchased system to be designed so that it tested negatively, if one could assume that deliberately building a system for a false negative was hard.

5.5 Where Does This Lead Us?

It has been stated [7] that there will never be a general test to decide whether a piece of software performs malicious acts. Above we have gone through the reasoning that substantiates this claim, which urges us to ask what options remain. The answer lies in the observation at the end of the previous section. We must aim to organize processes, mechanisms, and human expertise for investigating equipment such that deliberately building equipment that would generate a false negative in the investigation is hard. In other words, any vendor that deliberately inserts unwanted malicious functionality into its products should run a high risk of being caught.

This places our problem in the same category as most other sciences related to security. It is usually impossible to guarantee that security will never be breached, but one can make it difficult to the extent that it rarely happens. This is the case in aviation, in finance, and in traditional computer security. We must therefore understand what it means to make it difficult to build malicious functionality into a system without being caught.

Several fields of research have the potential to help. First and perhaps most obvious, we have all the research that has been done in malware detection. Although most of the work in that field is based on the assumption that the perpetrator is a third party and not the vendor, the problem it addresses is close to ours. We study the applicability of malware detection techniques to our problem in Chap. 7. Then we study how developments in formal methods can help us. The aim of formal methods is to build formal proofs of the properties of computer systems and we thus consider how that can help us in Chap. 9. Adding to this, Chap. 8 examines how the systematic testing of a computer system can help us make it difficult to include malicious behaviour into a system without being caught. Looking for machine code that can solve the problem once and for all is, unfortunately, futile.

References

1. Borello, J.M., Mé, L.: Code obfuscation techniques for metamorphic viruses. J. Comput. Virol. **4**(3), 211–220 (2008)
2. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Automatically identifying trigger-based behavior in malware. Botnet Detection, pp. 65–88. Springer, New York (2008)
3. Bueno, D., Compton, K.J., Sakallah, K.A., Bailey, M.: Detecting traditional packers, decisively. Research in Attacks, Intrusions, and Defenses, pp. 184–203. Springer, Berlin (2013)
4. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: 2005 IEEE Symposium on Security and Privacy, pp. 32–46. IEEE (2005)
5. Cohen, F.: Computer viruses. Ph.D. thesis, University of Southern California (1985)
6. Gödel, K.: Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. Monatshefte für mathematik und physik **38**(1), 173–198 (1931)
7. Oppliger, R., Rytz, R.: Does trusted computing remedy computer security problems? IEEE Secur. Priv. **2**, 16–19 (2005)
8. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: Polyunpack: automating the hidden-code extraction of unpack-executing malware. In: Null, pp. 289–300. IEEE (2006)
9. Spinellis, D.: Reliable identification of bounded-length viruses is np-complete. IEEE Trans. Inf. Theory **49**(1), 280–284 (2003)
10. Turing, A.: On computable numbers, with an application to the entscheidungs problem. Proc. Lond. Math. Soc. **42**, 230–265 (1936–1937)