Detecting Traditional Packers, Decisively

Denis Bueno, Kevin J. Compton, Karem A. Sakallah, and Michael Bailey

Electrical Engineering and Computer Science Department
University of Michigan
{dlbueno,kjc,karem,mibailey}@umich.edu

Abstract. Many important decidability results in malware analysis are based on Turing machine models of computation. We exhibit computational models that use more realistic assumptions about machine and attacker resources. While seminal results such as [1–5] remain true for Turing machines, we show that under more realistic assumptions important tasks are decidable instead of undecidable. Specifically, we show that detecting traditional malware unpacking behavior – in which a payload is decompressed or decrypted and subsequently executed – is decidable under our assumptions. We then examine the issue of dealing with complex but decidable problems, and look for lessons from the hardware verification community, which has been striving to meet the challenge of intractable problems for the past three decades.

1 Introduction

In recent years, malware researchers have seen incoming malware rates multiply by an order of magnitude [6]. By the numbers alone, manual analysis, which takes a couple of hours per sample, will never be able to keep up. Thus, there is a critical need to develop scalable, automated analysis techniques. Currently, a wide variety of automated methods exist for unpacking, for malicious code detection, for clustering related malware samples, and for reverse engineering. Unfortunately, the *possibility* of complete, automated analysis has long been impeded by theoretical results in Computer Science: we simply can't design algorithms clever enough to solve *undecidable* problems.

Although there are a variety of important malware analysis problems, *packing* is one which typifies the analysis challenges. In order to evade anti-virus detection, malware authors obfuscate their code; packers are software programs that automate obfuscation [7]. When the packed binary is executed, it unpacks its original code and then executes that. Packers are indeed effective at avoiding signature-based detection: signatures must be manually created, while packed versions are produced automatically.

Recent papers on practical topics in malware analysis have included some discouraging decidability results. For example, Christodorescu *et al.* [2] describe a technique for matching malware samples against hand-constructed templates of malicious behavior. A program matches a template if and only if the program contains an instruction sequence that contains the behavior specified by the template. Christodorescu *et al.* prove this matching problem is undecidable: the proof exhibits a template that, if matched, solves the halting problem for Turing machines.

This paper examines the standard approach to decidability and complexity in the context of malware analysis. Specifically, we make the following contributions:

- We critically analyze theoretical models used to prove prominent undecidability results. We thoroughly examine the widely-held assumptions [1–5] behind these results, and find that the assumptions about time and space constraints are unrealistic.
- We introduce a new theoretical model for malware analysis, based on the existing concept of *RASP machines* [8]. In the general case, RASP machines have the computational power of Turing machines. As an example of our approach, we use RASPs to formalize the problem of detecting traditional unpacking behavior. We prove that under certain very loose and realistic time and space assumptions, detecting unpacking is not only decidable, but NP-complete.
- We acknowledge that NP-complete does not mean tractable. For inspiration in dealing with intractable problems, we look to the three-decade-long effort in hardware verification.

2 Motivation

> There isn't (and never will be) a general test to decide whether a piece of software contains malicious code.
>
> — IEEE Security & Privacy magazine, 2005 [3]

The mantra that malicious code detection is undecidable has pervaded the community's consciousness, as the quote above indicates. The article even explains the halting problem reduction that is typically used to prove undecidability results.

Indeed, we find the literature littered with claims that various malware tasks are undecidable. We give several examples. The purpose of these examples is not to point out errors in the proofs (most results are claimed without proof) but to illustrate how widespread the opinion is.

Jang *et al.* aver that "malware analysis often relies on undecidable questions" [9]. Moser *et al.* describe several attacks against static analyzers; they motivate this work by claiming that "[static] detection faces the challenge that the problem of deciding whether a certain piece of code exhibits a certain behavior is undecidable in the general case" [10].

The MetaAware paper describes a static analysis for recognizing metamorphic variants of malware [11]. The authors claim that "determining whether a program will exhibit a certain behavior is undecidable" and that the task of checking whether a virus is a polymorphic variant of another virus is undecidable.

In the context of botnet analysis, Brumley *et al.* have examined "trigger-based behavior" – code paths that are triggered by environmental conditions, such as the occurrence of a particular date. They make similar claims: "deciding whether a piece of code contains trigger-based behavior is undecidable" [4]. Newsome *et al.* consider the problem of replaying executions, which requires searching for inputs satisfying a program's control flow. According to them, "finding a satisfying input can be reduced to deciding the halting problem" [5]. Sharif *et al.* describe a system for analyzing virtualization obfuscators; they claim that "theoretically, precisely and completely identifying an emulators bytecode language is undecidable" [12].

The PolyUnpack paper, by Royal *et al.*, describes an automated unpacker which works by comparing any executed code against the executable's static code model [1]. Appendix A in that paper proves that detecting unpacking behavior is undecidable by giving a formal reduction from the halting problem for Turing Machines. Many later papers cite PolyUnpack for exactly this decidability result [13–18]. We formally examine packed code analysis in the next section.

We emphasize that we do not mean that the respective authors are wrong in their claims. We cite them to support the assertion that undecidability results are a common thread in the automated malware analysis literature, common enough to state without proof. They are part of the community's collective consciousness and thus potentially influence the work we pursue.

A ray of hope. Alongside the malware analysis community some decidability results have slipped by. A small article appeared in 2003 that proved that a bounded variant of Cohen's decidability question is NP-complete [19]. Subsequently, another paper showed that detecting whether a program P is a metamorphic variant of Q is NP-complete, under a certain kind of metamorphic transformation [20]. While their assumptions are somewhat restrictive, these proofs should give us *some* hope – if, under suitable restrictions, these tasks are decidable, can we use similar restrictions to obtain decidability for other questions?

We believe so and exhibit proofs in this paper. Our key insight is that Turing machines are *too generous* – they allow programs to use potentially infinite amounts of time and space. But digital computers are not abstract; they are limited along these most basic dimensions. We offer an example for comparison. In the cryptographic literature, standard assumptions are much more realistic than in most of the malware analysis literature. The attacker, Eve, is allowed *probabilistic polynomial time* to accomplish her nefariousness [21]. By analogy, we might consider malware models in which the malware is allowed polynomial time to accomplish its malicious behavior.[1]

[1] Some malware is persistent, so we might amend this analogy to say that the malware is allowed polynomial time to accomplish its *first* malicious behavior.

3 RASP Model and Decidability Results

Proof roadmap. The following sections have a somewhat complex structure, which we now explain.

3.1 We begin with a review of related work, including foundational models in the theoretical malware analysis literature.
3.2 We introduce a Random Access Stored Program (RASP) machine that draws heavily from prior work in algorithmic analysis [8, 22–24]. The RASP has the same computational power as a Turing machine, but is more convenient for formalizing unpacking behavior.
3.3 We introduce a novel element, the RASP interpreter. The interpreter is a RASP program that interprets other RASP programs. It models a dynamic analyzer and plays an important role in our reduction proofs.
3.4 We formalize the malware unpacking problem in terms of the RASP interpreter. We prove that detecting unpacking is undecidable for RASPs – complementing decidability results for Turing machines [1].
3.5 We show that if we restrict the space a RASP program is allowed, detecting unpacking is *decidable* for RASPs.
3.6 We show that if we restrict the time a RASP program is allowed, detecting unpacking is not only decidable, but NP-complete.

3.1 Related Work

The earliest decidability results for malware are found in Cohen's classic work on viruses [25, 26]. His work formalizes "viral sets," pairs (M, V) where M is a Turing machine and for all v in V, there is a v' in V that M can produce when executed on v. Viral sets are clearly inspired by biological virus evolution. Cohen proves a variety of theorems about viral sets. He proves, for instance, that viral set detection is undecidable (Theorem 6), and that viruses are at least as powerful as Turing machines as a means of computation (Theorem 7).

Shortly thereafter, Adleman's work formalizes aspects of viruses and infection using total recursive functions and Gödel numbering [27]. He shows that the virus problems he considers are Π_2-complete. Two years later, Thimbleby *et al.* [28] describe a general mathematical framework for Trojans, also using recursion theory; they show that Trojan detection is undecidable.

Chess and White [29] give an extension of Cohen's Theorem 6; they show that some viruses have no error-free detectors. They draw the conclusion that it is not possible to create a precise detector for a virus even if you reverse engineer and completely understand it. Filiol *et al.* [30] give a statistical variant of Cohen's result using his definitions. They show that the false positive probability of a series of statistical tests can never go to 0, and thus that one can never write a detector without some false positives.

3.2 RASP Machine

Elgot and Robinson [22] developed the RASP out of a desire to have a model of computation more like a real computer than is a Turing machine, but with the same computational power. Hartmanis [23] and Cook and Reckhow [8] proved a number of fundamental results concerning RASPs. Aho *et al.* [24], in an early influential book on algorithms, promoted the RASP as a basic model for algorithm analysis. Our treatment most closely follows Hartmanis [23].

The RASP is a von Neumann machine. It has an addressable memory that stores programs and data, an instruction pointer ip that stores the address of the current instruction, and a simple arithmetical unit – the accumulator register ac. Our version of the RASP also has simple input/output operations.[2]

RASPs differ from real computers in two ways: they have infinitely many memory locations $M[i]$, where the addresses i are elements of $\mathbb{N} = \{0, 1, 2, \ldots\}$, and each $M[i]$ stores an arbitrary integer from $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. There is no fixed word size. The RASP models program behavior in a natural way, by reference to addresses and instructions. Unlike universal Turing machines, which must execute a large number of decoding instructions when they emulate other Turing machines (particularly ones with a large tape alphabet), a RASP interpreter emulates other RASPs in a straightforward manner (in fact, in a manner similar to the operation of virtualization obfuscators [31]).

With RASPs it is easy to describe decidability and complexity results in terms of asymptotic behavior as input size grows. In contrast, models of computation with a fixed bound on memory size become obsolete when technology changes because memory storage grows with each successive generation of digital computers. Sometimes word size also grows. Models of computation with a fixed word size also require complicated (and usually irrelevant) multi-precision arithmetic algorithms as input size increases. RASPs strike a balance between a realistic model of computation and models suitable for asymptotic analysis.

In our instruction set architecture (ISA), an instruction consists of an opcode and an operand. Opcodes are integers in the range $0 \leq r < 16$. To interpret any integer n uniquely as an instruction, we write $n = 16j + r$, where r is the opcode and j is the operand. Table 1 in Appendix A specifies a simple assembly language for the 16 RASP instructions. The opcode associated with a particular assembly language instruction is expressed as a mnemonic (such as load, stor, etc.) and the addressing mode – either *immediate*, *direct*, or *indirect* addressing – indicated by writing the operand j without brackets (j), within single angle brackets ($\langle j \rangle$), or within double angle brackets ($\langle\langle j \rangle\rangle$), respectively. For example, the integer 39, viewed as an instruction, is $2 \cdot 16 + 7$: its operand is 2 and its opcode is 7. Its assembly language representation is add $\langle 2 \rangle$. Thus, this is a direct add instruction. We consult the operational semantics column in Table 1 to see what should happen when this instruction executes. The table tells us that we

[2] The RASP model we use differs from those in the works cited in one inessential respect: program instructions take one word of memory rather than two; that is, an instruction is a single integer. This design choice results in somewhat simpler definitions of malware behavior.

must determine the r-value (denoted rval) of the operand. We find this in Table 2 (in Appendix A). Since j is 2, the rvalue of $\langle j \rangle$ is the value $M[2]$. The RASP updates ac to be the value stored in $M[2]$ plus the value in the ac register and then increments the value in the ip register.

The Tables in Appendix A also specify the time cost for each instruction in terms of the function $l(i)$ defined by:

$$l(i) = \begin{cases} \lfloor \lg |i| \rfloor + 1, & \text{if } i \neq 0 \\ 1, & \text{if } i = 0. \end{cases} \tag{1}$$

This is the approximate number of bits needed to represent i. Since the RASP does not have a fixed word size, $l(i)$ is roughly proportional to the time required to process i during an instruction execution.

Continuing with our example, suppose that at some time during the execution of a program, ac contains 128, ip contains 16, $M[2]$ contains -8, and $M[16]$ contains 39. Since ip contains 16, the instruction stored in $M[16]$ (viz., 39) is executed. We have seen that this instruction is $\mathsf{add}\ \langle 2 \rangle$. Its execution causes rval (in this case, the value -8 at $M[2]$) to be added to ac, changing the value stored there from 128 to 120. Finally, ip is incremented and its new value is 17. Table 1 tells us that the cost of executing this instruction is $l(\mathsf{ip}) + l(\mathsf{ac}) + \mathsf{rcost}$. Table 2 tells us that rcost is $l(2) + l(M[2])$. Therefore, the cost of executing the instruction is:

$$l(16) + l(128) + l(2) + l(-8) = 19.$$

We will say that the execution of an instruction takes one *step*, but this example illustrates that the cost of an instruction step is variable.

The read instruction gets successive values from an input stream in and the write instruction puts successive values into an output stream out. If the machine reads and no input is available, it reads a 0.

A RASP *program* P is a pair (I, D), where I, the instruction set, is a partial function $I : \mathbb{N} \rightharpoonup \mathbb{Z}$ with finite domain $\operatorname{dom}(I)$, and D, the data set, is a partial function $D : \mathbb{N} \rightharpoonup \mathbb{Z}$ with finite domain $\operatorname{dom}(D)$. We also require that $\operatorname{dom}(I) \cap \operatorname{dom}(D) = \emptyset$.

To begin executing a RASP program $P = (I, D)$, the program is "loaded" and the RASP initialized by setting M, ip and ac thusly:[3]

$$\mathsf{ip} = 0 \qquad \mathsf{ac} = 0 \qquad M[i] = \begin{cases} I(i) & i \in \operatorname{dom}(I) \\ D(i) & i \in \operatorname{dom}(D) \\ 0 & \text{otherwise} \end{cases}$$

Executing P proceeds in a straightforward way. After loading, the RASP enters a loop that fetches the next instruction $M[\mathsf{ip}]$ then decodes the instruction and executes as specified in Tables 1 and 2. The machine halts if it reaches a

[3] A more realistic initial value for M would not require zero content at locations outside $\operatorname{dom}(I) \cup \operatorname{dom}(D)$ since a real computer typically runs many processes concurrently, but this will suffice for our analysis.

halt instruction or if any memory operand references a negative address during execution.

We may view a RASP program's dynamic behavior as computing a partial function that maps an input stream to an output stream. Alternatively, we may think of a RASP program with read instructions as a nondeterministic machine. Whenever a read instruction loads a value from the input stream in to a memory location, we view this as a nondeterministic choice. This nondeterministic interpretation is apt if P is malware that initiates an undesirable computation when it receives the appropriate external trigger.

RASP machines are equivalent in computational power to classical Turing machines [22, 8]. This shows, in particular, that the halting problem for RASP machines is undecidable. This will be important later.

Definition 1 (Time and space). *The* time *for the execution of a RASP program $P = (I, D)$ on a particular input stream* in *is the sum of the costs of all the instructions' steps, or ∞ if the program does not halt.*

The definition of space for an execution is slightly more subtle because we do not include the space required for in *or for $dom(I) \cup dom(D)$, unless one of these locations is referenced.*[4] *At any given step t of the execution, let $A(t)$ be the set of addresses that have been referenced by a* stor *or* read *instruction up to step t. The space used at step t is:*

$$s(t) = l(\mathsf{ip}) + l(\mathsf{ac}) + \sum_{i \in A} (l(i) + l(M[i])).$$

The space *for the execution is the maximum value of $s(t)$ taken over all steps t of the execution. It is not difficult to show that the space for an execution is always bounded above by the time of that execution.*

Careful readers will have noted that space is determined in terms of time cost. This is done because our ISA uses simple operations (addition and subtraction) that run quickly relative to the input size. If we had chosen more complex operations, our time and space characterization would change.

3.3 RASP Program Interpreter

In order to formulate our main results, we require a RASP interpreter, which we dub Rasputin. Rasputin is a RASP program $(I_{\mathcal{R}}, D_{\mathcal{R}})$ that reads an integer sequence $\langle P, w \rangle$ encoding a RASP program $P = (I, D)$ and a finite input w for P, then emulates P's execution on input w. Recall that if P were loaded directly into a RASP, location j_0 gets $I(j_0)$, j_1 gets $I(j_1)$, and so on; and location k_0 gets $D(k_0)$, k_1 gets $D(k_1)$, and so on. $\langle P, w \rangle$ is simply a sequence of these pairs; specifically, it is a listing of the pairs in the graph[5] of I,

$$j_0, I(j_0), j_1, I(j_1), \ldots, j_r, I(j_r)$$

[4] This allows us to consider sublinear space bounds.

[5] The graph of a function is the set of all of the pairs that define it.

followed by a delimiter -1. That is then followed by a listing of the pairs in the graph of D,

$$k_0, D(k_0), k_1, D(k_1), \dots, k_s, D(k_s)$$

followed by a delimiter -1, followed by a listing of the integers $w_0, w_1, \dots, w_u$ in w.

Rasputin uses three special memory locations in $\mathrm{dom}(D_{\mathcal{R}})$: sip, the *stored instruction pointer* address; sac, the *stored accumulator* address; and sopr, the *stored operand* address. The data values are $D_{\mathcal{R}}(\mathsf{sip}) = b$, $D_{\mathcal{R}}(\mathsf{sac}) = 0$, and $D_{\mathcal{R}}(\mathsf{sopr}) = 0$, where b is a base offset larger than any address in $\mathrm{dom}(I_{\mathcal{R}}) \cup \mathrm{dom}(D_{\mathcal{R}})$.

We describe Rasputin's instructions in English, but they are straightforward to implement as a RASP program. Rasputin first reads the initial part of $\langle P, w\rangle$, specifying the graph pairs of I and D. As it reads, it stores them relative to its *base address* b: thus, $M[b+j] \leftarrow I(j)$ for every $j \in \mathrm{dom}(I)$ and $M[b+j] \leftarrow D(j)$ for every $j \in \mathrm{dom}(D)$.

Next, Rasputin enters a fetch-decode-execute loop. During each cycle, it transfers the instruction j whose address is in sip to the accumulator. It then decodes j into an opcode r and operand q, where $j = 16q + r$, and stores these values in the accumulator and sopr.[6] Next, by alternately executing bpa instructions and decrementing the value in the accumulator, Rasputin finds the section in its program that will execute instruction j. At this point, it carries out the operational semantics in Tables 1 and 2, with sip and sac substituted for ip and ac, and with offset addresses whenever they are needed. It then repeats the cycle.

We offer this drawn-out description to emphasize that Rasputin is a well-behaved program. Whatever the input $\langle P, w\rangle$ may be, Rasputin will not execute an instruction outside of those in $I_{\mathcal{R}}$ or modify any of the instructions inside $I_{\mathcal{R}}$. Rasputin is not malware.

Below, we use Rasputin to represent a dynamic analyzer. Rasputin observes RASP code as it executes, and it may modify its behavior in response to what it sees.

3.4 Formalizing Unpacking Behavior

We begin by using the RASP model to exhibit a version of the undecidability result of the PolyUnpack paper [1]. Our proof improves upon previous work by providing a precise and intuitive characterization of unpacking behavior (Definition 2). It also justifies the fact that our model is just as general as a Turing machine. The basic fact we need is the undecidability of the following problem.

Theorem 1 (Halting Problem for RASPs). ***Given:*** *RASP program $P = (I, D)$ and finite input sequence x.* ***Question:*** *Does P halt when it executes with input x?*

[6] The RASP code required to do this when j is positive involves generating powers of 2 by repeated doubling until one at least as large as j is generated, using these powers of 2 to determine the binary representation of j, and then from this computing r and q. The procedure when j is negative is similar.

Proof. We have immediately that this problem is undecidable by the Elgot-Robinson [22] result giving an effective transformation from Turing machines into equivalent RASP programs and from the undecidability of the Halting Problem for Turing Machines. □

Now we come to the main definition of this section.

Definition 2 (Unpacking Behavior). *Let $P = (I, D)$ be a program and x a sequence of inputs. P is said to exhibit* unpacking behavior *(or to* unpack*) on x if, at some point during execution, $\mathsf{ip} \notin dom(I)$ (data-execution) or P stores to an address in $dom(I)$ (self-modification).*

From this, we formalize the problem of detecting unpacking. We demonstrate two independent results. Theorem 2 mirrors Royal *et al.* [1]. Theorem 3 is the general case of the problem of greatest import.

Definition 3 (Special Unpacking Problem). ***Given:*** *RASP program $P = (I, D)$ and finite input sequence x.* ***Question:*** *Does P unpack on input x?*

Theorem 2. *The Special Unpacking Problem is undecidable.*

Proof. Reduce the Halting Problem for RASP machines (Theorem 1) to the Special Unpacking Problem.

First, we describe a modification of Rasputin we will call Evil Rasputin. Evil Rasputin is a RASP program $(I_{\mathcal{E}}, D_{\mathcal{E}})$ obtained from Rasputin by replacing Rasputin's halt conditions (*viz.* emulation of a halt instruction or an attempt by the emulated program to reference a negative address) with a jmp instruction to an address not in $\mathrm{dom}(I_{\mathcal{R}})$. (This involves inserting checks for negative addresses and branches at appropriate points in $I_{\mathcal{R}}$.)

Now, P halts on input w if and only if Evil Rasputin unpacks on input $x = \langle P, w\rangle$. This reduces the Halting Problem for RASPs to the Special Unpacking Problem. If there were a decision algorithm for the latter problem, there would be one for the former problem, as well. This would be a contradiction to Theorem 1. □

Definition 4 (Unpacking Problem). ***Given:*** *RASP program $P = (I, D)$.* ***Question:*** *Is there a finite input sequence x such that P unpacks on x?*

Theorem 3. *The Unpacking Problem is undecidable.*

Proof. The proof is very similar to the proof of Theorem 2. Reduce the Halting Problem for RASPs to the Unpacking Problem.

Let P be a RASP program and x an input (*i.e.*, a finite integer sequence) for P. We describe a modified version of Evil Rasputin called Evil Rasputin$_{P,x}$, which has no read instructions. Instead, P and x are preloaded in the data section section, $D_{\mathcal{E}}$. Rather than reading $\langle P, x\rangle$ from an input stream, Evil Rasputin$_{P,x}$ transfers values from its data section to the appropriate locations. In all other respects, it behaves in the same way as Evil Rasputin. In particular, Evil Rasputin$_{P,x}$ unpacks (irrespective of its input since it has no reads) if and only if P

halts on input x. Thus, the mapping from $\langle P, x\rangle$ to $\mathsf{Evil\ Rasputin}_{P,x}$ is a reduction from the Halting Problem for RASPs to the Unpacking Problem. If there were a decision algorithm for the latter problem, there would be one for the former problem, as well. Again, this would be a contradiction. □

3.5 Space bounded RASP

The undecidability results of the previous section do not address the real issue of malware detection because *no real machine looks like our RASP*. Real machines cannot store arbitrary sized integers in every memory location. Real machines do not have an infinite set of memory registers. *Real machines have fixed resources.* We therefore present a restriction of the RASP model by bounding space in terms of input size. This is analogous to the restriction used for Linear Bounded Automata [32].

Definition 5 (Space bounded RASP). *A* Γ-space bounded RASP program *is a RASP program that uses space at most* $\Gamma(n)$ *on all inputs of size* n. *A* Γ-space bounded RASP *is one that executes only* Γ*-space bounded programs. It executes programs in exactly the same way as a RASP, except that on inputs of size* n*, if a program ever attempts to use more than space* $\Gamma(n)$*, the* Γ*-space bounded RASP will halt.*

The following problem is a step toward formulating a more realistic goal for static malware detection.

Definition 6 (Space Bounded Unpacking Problem). ***Given:*** *Computable function* Γ*,* Γ*-space bounded RASP program* P*, and integer* $k > 0$*.* ***Question:*** *Is there an input* x *with* $l(x) \leq k$ *such that* P *unpacks on input* x*?*

Theorem 4. *The Space Bounded Unpacking Problem is decidable.*

Proof. We describe an algorithm to decide the Space Bounded Unpacking Problem. Let the *size* of a finite integer sequence w be $l(w) = \sum_{i \in w} l(i)$.

First, consider a specific x with $n = l(x) \leq k$. P is restricted to space at most $\Gamma(n) = s$ on input x. A *configuration* for P at any step of its execution is a list of all of the information needed to determine future actions of P. More precisely, the configuration at a given step is a list of the following:

1. the contents of ac;
2. the contents of ip;
3. a list of all of the addresses that have been referenced up to this step, and their contents; and
4. the number of integers in the input sequence x that remain to be read.

From this, we will determine an upper bound for the total possible number of configurations.

First, note that there are precisely 2^s nonnegative integers i with $l(i) \leq s$, *viz.*, the integers in $A = \{0, 1, \ldots, 2^s - 1\}$. Also, there are precisely $2^{s+1} - 1$ integers

i with $l(i) \leq s$, *viz.*, the integers in $B = \{-(2^s - 1), -(2^s - 2), \ldots, 2^s - 1\}$. The contents of ip must be from A. The contents of ac must be from B. When P executes on input x, every address in A has either never been referenced, or its contents are in B. Moreover, only addresses in A could possibly have been referenced. Thus, for item 1 above, there are at most $2^{s+1} - 1$ possibilities; for item 2, there are at most 2^s possibilities; for item 3, there at most $(2^{s+1})^{2^s}$ possibilities; and for item 4, there are at most $n + 1$ possibilities. Therefore, there are at most the following possible configurations:

$$b(n) = (2^{s+1}) \cdot 2^s \cdot 2^{(s+1)2^s} \cdot (n+1)$$

Now to see if P unpacks on a given x, use an augmented Rasputin to emulate P's execution on x. After each step, check to see if P has unpacked, and if it has, report the result. If, at some point, P halts and no unpacking behavior has occurred, report that result. Keep a tally of the number of emulated steps. When the tally exceeds $b(n)$, we know that we are in an infinite loop, so if no unpacking behavior has been observed up to that point, it never will be. Report that result.

Apply the algorithm outlined above for every x such that $l(x) \leq k$. There are only finitely many such x's, so we can decide if unpacking behavior ever occurs. □

Real computers are all space bounded, in fact, constant space bounded. Therefore, *detecting unpacking behavior for real computers is decidable.* Unfortunately, for real computers the algorithm given in the proof above has an execution time many orders of magnitude greater than the lifetime of the universe, so the result appears to be of only theoretical interest. But all is not lost. Researchers in areas of computer security, such as cryptography, have long recognized that even malevolent adversaries must have bounded computational resources, particularly time resources.

3.6 Time Bounded RASP

Our formalization is similar to the space bounded case.

Definition 7 (Time bounded RASP). *Let $\Delta : \mathbb{N} \to \mathbb{N}$ be a computable function. A Δ-time bounded RASP program is a RASP program that uses time at most $\Delta(n)$ on all inputs of size n. A Δ-time bounded RASP is one that executes only Δ-time bounded programs. It executes programs in exactly the same way as a RASP except that on inputs of size n, if a program ever attempts to use more than time $\Delta(n)$, the Δ-time bounded RASP will halt.*

Definition 8 (Time Bounded Unpacking Problem.). ***Given:*** *Δ-time bounded RASP program P and integer $k > 0$.* ***Question:*** *Is there an input x with $l(x) \leq k$ such that P unpacks on input x?*

Theorem 5. *The Time Bounded Unpacking Problem is decidable.*

Proof. The proof is completely trivial. P is always guaranteed to halt within time $\Delta(n)$ for all x of size $n = l(x) \leq k$. Run P on all such x's to see if it exhibits unpacking behavior. $\square$

Why should we bother to include such an obvious result? The reason is that the restricted version of *this very question is the one that the malware analysis community should be considering.*

So far we have shown that, when suitably restricted, detecting unpacking for RASP machines is decidable. The restrictions we imposed are realistic: in reality, the attacker has a finite amount of space or time to do damage.

It is difficult to grasp how these results can be applied. Malware does not come with a computable function Δ and it would be time consuming to express the cost of each instruction on a real architecture, such as the x86. We also do not generally know the input size. Therefore, we formulate a restricted version of Theorem 5 that is in terms of the number of steps (*i.e.*, machine instructions) used.

Here t is an integer, rather than a function of the input size. It is customary in complexity theory to express results of this type using unary notation for the bound. That is, the integer t is represented as

$$\underbrace{11\cdots1}_{t \text{ times}}$$

or, more succinctly, 1^t. The reason we use this is so that algorithms of polynomial time complexity in t are expressed asymptotically as $O(t^k)$ instead of $O((\lg t)^k)$ for some $k > 0$.

Definition 9 (Time Guarantee Unpacking Problem). ***Given:*** *RASP program P and unary integer 1^t.* ***Question:*** *Is there an input x with such that P unpacks on input x within time t?*

Notice that we may also assume that $l(x) \leq t$ in this problem since input cost is one of the terms summed to derive execution time for P.

Theorem 6. *The Time Guarantee Unpacking Problem is NP-complete.*

Proof. The proof has two steps: we show that the bounded unpacking problem is in NP and exhibit a reduction from 3-SAT to it.

Bounded unpacking behavior is in NP. We simply execute P under Rasputin for up to time t. Whenever Rasputin requires an input integer, we nondeterministically generate an integer j with $l(j) \leq t$. After each step of the emulation, we check for unpacking behavior. This is a nondeterministic polynomial time algorithm.

Bounded unpacking behavior is NP-hard. We reduce (in polynomial time) 3-SAT to the Time Guarantee Unpacking Problem. 3-SAT is the problem of deciding if a given 3-CNF Boolean formula φ is satisfiable. A conjunctive normal form (CNF) *formula* is a conjunction of clauses; a *clause* is a disjunction of literals; a *literal* is a Boolean variable x or its negation $\neg x$. In a 3-CNF formula, each clause has exactly three disjuncts.

In order to satisfy a 3-SAT formula φ, we need an assignment. An *assignment* α is a function from φ's variables into $\{0, 1\}$. A negative literal $\neg x$ is satisfied if $\alpha(x) = 0$, and unsatisfied otherwise. A positive literal x is satisfied if $\alpha(x) = 1$, and unsatisfied otherwise. A clause is satisfied if *any* of its literals are satisfied. And a formula is satisfied if *all* of its clauses are satisfied. For example, a satisfying assignment of the following Boolean formula is $\alpha(x_1) = 1, \alpha(x_2) = 0, \alpha(x_3) = 0, \alpha(x_4) = 1$.

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee x_3 \vee x_4) \tag{2}$$

Let φ be an arbitrary 3-SAT formula whose variables are $x_1, x_2, \ldots, x_n$. We can encode φ as follows:

- Each variable x_i is represented as a positive integer i.
- Each negated variable $\neg x_i$ is represented as a negative integer $-i$.
- A 3-CNF formula is represented by a sequence of integers representing its literals in the order they occur in the list of clauses, followed by a terminating 0. For example, formula 2 above is represented as

  $$1, 2, -3, -1, 3, 4, 2, 3, 4, 0.$$

  Since each clause has exactly three literals, this is an unambiguous representation.

Now it is a fairly simple task to write a polynomial time RASP program, which we dub Raspberry, to check satisfiability of 3-CNF formulas. Raspberry takes as input a sequence $\langle \varphi, w \rangle$ consisting of the representation of φ, followed by a 0, followed by a sequence of n 0s and 1s representing an assignment to the Boolean variables $x_1, x_2, \ldots, x_n$, followed by a -1. Raspberry stores these integers in consecutive memory locations and then cycles through the integers representing φ to verify that in each clause at least one literal is satisfied.

Just as we turned Rasputin to the dark side by transforming it into Evil Rasputin, we transform Raspberry, an innocent program, into Wild Raspberry, a program that unpacks if it determines that w is a satisfying truth assignment for φ. Finally, for each Boolean formula φ, we create a RASP program Wild Raspberry$_\varphi$, where φ is hard coded into the data set. The mapping from φ to Wild Raspberry$_\varphi$ is polynomial time computable, and φ is satisfiable if and only if Wild Raspberry$_\varphi$ exhibits unpacking behavior within time t, where t is determined by the polynomial time bound for Raspberry. This is a reduction from an NP-complete problem 3-SAT to the Time Guarantee Unpacking Problem, thus proving NP-completeness of that problem. □

We have shown that the bounded unpacking problem is not only decidable, but NP-complete. A natural reaction to these results is, “Undecidable, NP-complete – doesn’t matter. Either way we can’t solve it!” The next section challenges this idea by reviewing approaches to intractable problems from other disciplines.

4 Approaching the Intractable

Intractable problems are encountered in many disciplines; we might therefore expect a large diversity of approaches to solving these problems. Indeed, there are many different algorithms and models, but effective approaches exploit a combination of optimization and parallelism. Important recent breakthroughs in computer science and computational science are made possible by exactly these techniques:

– Special-purpose hardware was built for Anton, a molecular dynamics simulation machine [33].
– Stevens *et al.* demonstrate chosen-prefix collisions in the MD5 cryptographic hash algorithm, computed in 6 months with thousands of machines [34].

Problems from many disciplines have been proven NP-complete [35]. In the particular domain of hardware verification, NP-complete problems have been a central topic of investigation for the past three decades. The focus of much of the work has been in increasingly clever search strategies. In the following section, we examine this field in depth in order to gather some lessons learned.

4.1 Formal Hardware Verification and the Intractable

Formal modeling and verification of complex hardware and software systems has advanced significantly over the past three decades, and formal techniques are increasingly seen as a critical complement to traditional verification approaches, such as simulation and emulation. The foundational work was established in the early 1980s with the introduction of model checking (MC) as a framework for reasoning about the properties of transition systems [36, 37]. A model checker’s fundamental goal is to prove that states that violate a given specification f cannot be reached from M’s initial (reset) states or to provide a counterexample trace (a state sequence) that serves as a witness for how f can be violated. Computationally, to verify the query “does M satisfy f” a model checker needs to perform some sort of (direct or indirect) reachability analysis in the state space of M. Since a transition system with n state elements (e.g., flip-flops) has 2^n states, model checkers have had to cope with the so-called state explosion problem, and much of the research in MC over the past thirty years has been primarily focused on attacking this problem [38]. MC for these properties (e.g., “X is true in all states” or “we shall not reach state Y”) is NP-complete [39]. The next few paragraphs review some significant milestones along this journey.

The EMC model checker [40], developed in the early 1980s, was based on an explicit representation of the state transition system. This system was able to handle up to about 10^5 states or roughly 16 flip-flops. The system was based on a naive *enumeration of each state.*

Subsequent checkers leveraged the key insight of implicit state representations. The use of binary decision diagrams (BDDs) to represent sets of states by characteristic Boolean functions enabled MC to scale to about 10^{20} states or about 66 flip-flops [41]. The key insight here was to reason about *sets of related states as a unit*, rather than as individuals.

The development of modern conflict-driven clause-learning (CDCL) Boolean satisfiability (SAT) solvers in the mid 1990s [42–44] provided another opportunity to scale model checkers to larger design sizes. This use of SAT solvers to perform MC was dubbed bounded model checking (BMC) [45] to contrast it with the unbounded BDD-based MC and it proved extremely useful for finding "shallow bugs." BMC extended the range of designs that could be handled to those containing several hundred flip-flops and relatively short counterexamples (10 steps or less) [46]. The key insight of this approach was to trade *completeness* (it would miss bugs) for *scalability* (it would find shallow bugs quickly).

An orthogonal attack on complexity was based on abstracting the underlying transition system. Abstraction methods create an over-approximation of the transition relation with the hope of making it more tractable for analysis. The technique was popularized by Clarke *et al.* [47, 48] who showed its effectiveness in scaling symbolic MC by verifying a hardware design containing about 500 flip-flops. The key insight was *a system absent of some of its details was sometimes sufficient for proving the properties of interest.*

The latest development to address the state explosion problem in MC is a clever deployment of incremental SAT solving to check the property f without the need to unroll the transition relation. The original idea was described by Bradley *et al.* [49, 50] and implemented in the IC3 tool. IC3 is able to solve systems with around 5,000 flip-flops. The key insight here was to *summarize important facts about program state transitions on demand* as the search progresses.

We have seen a variety of clever search strategies that help increase the design sizes for which we can prove properties. Implicit and over-approximate state representations, more intelligent underlying solvers, and on-demand characterization of important facts all contributed to current methods that can precisely analyze systems with thousands of flip-flops.

5 Malware analysis, Reprise

We have shown, under realistic assumptions about victim machines and attacker resources, that several important malware analysis questions are decidable rather than undecidable, as previously thought. The above example in hardware verification highlights a sequence of approaches for dealing with intractabile problems.

In general, when optimization [51] and parallelization [52] reach their limits, we employ a variety of approaches to coping with intractability [53]:

- Finding good average case algorithms rather than worse case algorthims (i.e., those algorithms which are fast most of the time);
- Using approximate algorithms (i.e., algorithms that provide bounds on quality and speed, but are not optimal);
- Qualitatively changing the amount of computation available (i.e, using FPGAs and GPUs or more radically, and more speculatively, quantum computing);
- Examining parameterization of the problems for which solutions are possible (i.e., acknowledging that an algorithm may not need to work on all inputs);
- The use of heuristics (i.e., algorithms that find solutions which are "good enough").

An important consequence of our results is the ability to derive *ground truth* for the community. Even if precise systems do not scale to realistic malware rates (tens of thousands per day), they still can be used to evaluate more scalable techniques by providing ground truth. It should be possible to construct a system where, if malware A and B are variants of one another, the system *always* tells you so. It might take an inordinate amount of time to do so, but, when it finally does, one has very high confidence in the result. We are investigating exactly this question.

Limitations. It is important to note that we do not address virtualization obfuscators [31, 54]; we only address traditional unpacking mechanisms. We have not found a crisp definition of what it means for a program to be virtualization-obfuscated that does not depend on the particular details of the obfuscation mechanism. If we address a particular virtualization obfuscator, we may be able to formulate detection problems that are decidable under assumptions similar to those presented here.

Conclusion. We have shown that by either restricting the space or the time that a program is allowed, we can decide whether a program unpacks; indeed, it is NP-complete. A natural question to ask is: for how many steps should we execute? While we do not yet have a definitive answer for the question, we instead offer the following vision of the future. Imagine a world where you download an untrusted executable and your personal anti-virus (AV) product performs a combined static and dynamic analysis on your laptop. In a minute or two, the AV product says, "Program `this-is-definitely-not-a-virus.exe` will not unpack, nor does it evolve into a known virus for the next 6 months." This would be a fantastic guarantee!

Although this situation seems far from reality, it is not out of the question. If – with a combination of abstraction, refinement, clever search strategies, and perhaps even special purpose hardware – we can produce time-based guarantees of (a lack of) malicious behavior, we will have reached an important milestone in the automated analysis of malicious software.

References

1. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In: Annual Computer Security Applications Conference, IEEE Computer Society (2006) 289–300
2. Christodorescu, M., Jha, S., Seshia, S.A., Song, D.X., Bryant, R.E.: Semantics-aware malware detection. In: Security and Privacy, IEEE Computer Society (2005) 32–46
3. Oppliger, R., Rytz, R.: Does trusted computing remedy computer security problems? Security Privacy, IEEE **3**(2) (2005) 16–19
4. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D.X., Yin, H.: Automatically identifying trigger-based behavior in malware. In: Botnet Detection. Springer (2008) 65–88
5. Newsome, J., Brumley, D., Franklin, J., Song, D.: Replayer: automatic protocol replay by binary analysis. In: ACM conference on Computer and Communications Security. CCS '06, New York, NY, USA, ACM (2006) 311–321
6. Bayer, U., Kirda, E., Kruegel, C.: Improving the efficiency of dynamic malware analysis. In: Proceedings of the 2010 ACM Symposium on Applied Computing, ACM (2010) 1871–1878
7. Guo, F., Ferrie, P., Chiueh, T.C.: A study of the packer problem and its solutions. In: Recent Advances in Intrusion Detection, Springer (2008) 98–115
8. Cook, S.A., Reckhow, R.A.: Time bounded random access machines. J. Comput. Syst. Sci. **7**(4) (1973) 354–375
9. Jang, J., Brumley, D., Venkataraman, S.: Bitshred: feature hashing malware for scalable triage and semantic analysis. In: ACM Conference on Computer and Communications Security. CCS '11, New York, NY, USA, ACM (2011) 309–320
10. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Computer Security Applications Conference. (2007) 421–430
11. Zhang, Q., Reeves, D.S.: MetaAware: Identifying metamorphic malware. In: Annual Computer Security Applications Conference, IEEE Computer Society (2007) 411–420
12. Sharif, M.I., Lanzi, A., Giffin, J.T., Lee, W.: Automatic reverse engineering of malware emulators. In: Security and Privacy, IEEE Computer Society (2009) 94–109
13. Kang, M.G., Poosankam, P., Yin, H.: Renovo: A hidden code extractor for packed executables. In: WORM, ACM (November 2007)
14. Martignoni, L., Christodorescu, M., Jha, S.: OmniUnpack: Fast, generic, and safe unpacking of malware. In: Annual Computer Security Applications Conference, IEEE Computer Society (2007) 431–441
15. Yin, H., Song, D.: Hidden code extraction. In: Automatic Malware Analysis. SpringerBriefs in Computer Science. Springer New York (2013) 17–26
16. Liu, L., Ming, J., Wang, Z., Gao, D., Jia, C.: Denial-of-service attacks on host-based generic unpackers. In Qing, S., Mitchell, C., Wang, G., eds.: Information and Communications Security. Volume 5927 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 241–253
17. Xie, P.D., Li, M.J., Wang, Y.J., Su, J.S., Lu, X.C.: Unpacking techniques and tools in malware analysis. Applied Mechanics and Materials **198–199** (2012) 343–350
18. Perdisci, R., Lanzi, A., Lee, W.: Classification of packed executables for accurate computer virus detection. Pattern Recognition Letters **29**(14) (2008) 1941–1946

19. Spinellis, D.: Reliable identification of bounded-length viruses is NP-complete. Information Theory, IEEE Transactions on **49**(1) (2003) 280–284
20. Borello, J.M., Mé, L.: Code obfuscation techniques for metamorphic viruses. Journal in Computer Virology **4**(3) (2008) 211–220
21. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman & Hall (2008)
22. Elgot, C.C., Robinson, A.: Random-access stored-program machines, an approach to programming languages. J. ACM **11**(4) (1964) 365–399
23. Hartmanis, J.: Computational complexity of random access stored program machines. Mathematical Systems Theory **5**(3) (1971) 232–245
24. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
25. Cohen, F.: Computer Viruses. PhD thesis, University of Southern California (1986)
26. Cohen, F.: Computational aspects of computer viruses. Computers & Security **8**(4) (1989) 297–298
27. Adleman, L.M.: An abstract theory of computer viruses (invited talk). In: Proceedings on Advances in cryptology. CRYPTO '88, New York, NY, USA, Springer-Verlag New York, Inc. (1990) 354–374
28. Thimbleby, H., Anderson, S., Cairns, P.: A framework for modelling trojans and computer virus infection. The Computer Journal **41**(7) (1998) 444–458
29. Chess, D.M., White, S.R.: An undetectable computer virus. In: Proceedings of Virus Bulletin Conference. Volume 5. (2000)
30. Filiol, E., Josse, S.: A statistical model for undecidable viral detection. Journal in Computer Virology **3**(2) (2007) 65–74
31. Oreans Technologies. http://www.oreans.com/themida.php
32. Sipser, M.: Introduction to the Theory of Computation. Volume 27. Thomson Course Technology Boston, MA (2006)
33. Shaw, D.E., Deneroff, M.M., Dror, R.O., Kuskin, J.S., Larson, R.H., Salmon, J.K., Young, C., Batson, B., Bowers, K.J., Chao, J.C., et al.: Anton, a special-purpose machine for molecular dynamics simulation. In: ACM SIGARCH Computer Architecture News. Volume 35., ACM (2007) 1–12
34. Stevens, M., Lenstra, A., Weger, B.: Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities. In Naor, M., ed.: Advances in Cryptology - EUROCRYPT 2007. Volume 4515 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 1–22
35. Garey, M.R., Johnson, D.S.: Computers and Intractability. Volume 174. Freeman New York (1979)
36. Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Logic of Programs. (1981) 52–71
37. Queille, J.P., Sifakis, J.: Specification and Verification of Concurrent Systems in CESAR. In: Symposium on Programming. (1982) 337–351
38. Clarke, E.M.: The Birth of Model Checking. In: 25 Years of Model Checking. (2008) 1–26
39. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. J. ACM **32**(3) (July 1985) 733–749
40. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. In: POPL. (1983) 117–126
41. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: LICS. (1990) 428–439

42. Marques-Silva, J.a.P., Sakallah, K.A.: GRASP-A New Search Algorithm for Satisfiability. In: Digest of IEEE International Conference on Computer-Aided Design (ICCAD), San Jose, California (November 1996) 220–227
43. Marques-Silva, J.a.P., Sakallah, K.A.: GRASP: A Search Algorithm for Propositional Satisfiability. IEEE Transactions on Computers **48**(5) (May 1999) 506–521
44. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC. (2001) 530–535
45. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems. TACAS '99, London, UK, Springer-Verlag (1999) 193–207
46. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded Model Checking Using Satisfiability Solving. Form. Methods Syst. Des. **19** (July 2001) 7–34
47. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In Emerson, E., Sistla, A., eds.: Computer Aided Verification. Volume 1855 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2000) 154–169 10.1007/10722167_15.
48. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. J. ACM **50** (September 2003) 752–794
49. Bradley, A.R., Manna, Z.: Checking Safety by Inductive Generalization of Counterexamples to Induction. In: Formal Methods in Computer Aided Design (FMCAD'07). (nov. 2007) 173 –180
50. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation. VMCAI'11, Berlin, Heidelberg, Springer-Verlag (2011) 70–87
51. Knuth, D.E.: Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition). 3 edn. Addison-Wesley Professional (July 1997)
52. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, spring joint computer conference. AFIPS '67 (Spring), New York, NY, USA, ACM (1967) 483–485
53. Downey, R.G., Fellows, M.R., Stege, U.: Computational tractability: The view from mars. In: Bulletin of the European Association of Theoretical Computer Science. 73–97
54. VMProtect Software. http://vmpsoft.com/

A Appendix – RASP Tables

Table 1. Operational semantics and time cost for the sixteen RASP instructions. Most mnemonics are obvious; one that isn't is bpa, which stands for "branch on positive accumulator." Instructions have several addressing modes. The instruction cost depends on the addressing mode; see Table 2 for details. (The definition of $l(\cdot)$ is equation 1 on page 6.) This ISA allows direct formalization of unpacking behavior.

Mnemonic	*Operand*	*Opcode*	*Operational Semantics*	*Time Cost*
halt		0	halt	1
load	j	1	ac $\leftarrow$ rval; ip++;	l(ip) + rcost
	$\langle j \rangle$	2		
	$\langle\!\langle j \rangle\!\rangle$	3		
stor	$\langle j \rangle$	4	M[lval] $\leftarrow$ ac; ip++;	l(ip) + l(ac) + lcost
	$\langle\!\langle j \rangle\!\rangle$	5		
add	j	6	ac $\leftarrow$ ac + rval; ip++;	l(ip) + l(ac) + rcost
	$\langle j \rangle$	7		
sub	j	8	ac $\leftarrow$ ac − rval; ip++;	l(ip) + l(ac) + rcost
	$\langle j \rangle$	9		
jmp	j	10	ip $\leftarrow$ rval;	rcost
	$\langle j \rangle$	11		
bpa	j	12	if (ac > 0) then ip $\leftarrow$ rval;	l(ip) + l(ac) + rcost
	$\langle j \rangle$	13	else ip++;	
read	$\langle j \rangle$	14	M[lval] $\leftarrow$ in; ip++;	l(ip) + l(in) + lcost
write	$\langle j \rangle$	15	out $\leftarrow$ rval; ip++;	l(ip) + rcost

Table 2. Values and costs for the three addressing modes. The costs allow us to analyze asymptotic behavior as machine word and input size grow, and allow us to formulate the restrictions on time and space crucial for our decidability results.

Mode	*Operand*	rval	rcost	lval	lcost
immediate	j	j	$l(j)$		
direct	$\langle j \rangle$	$M[j]$	$l(j) + l(M[j])$	j	$l(j)$
indirect	$\langle\!\langle j \rangle\!\rangle$	$M[M[j]]$	$l(j) + l(M[j]) + l(M[M[j]])$	$M[j]$	$l(j) + l(M[j])$