On Protection by Layout Randomization*

Martín Abadi
Microsoft Research, Silicon Valley,
University of California, Santa Cruz, and
Collège de France

Gordon Plotkin
Microsoft Research, Silicon Valley and
LFCS, University of Edinburgh

Abstract

Layout randomization is a powerful, popular technique for software protection. We present it and study it in programming-language terms. More specifically, we consider layout randomization as part of an implementation for a high-level programming language; the implementation translates this language to a lower-level language in which memory addresses are numbers. We analyze this implementation, by relating low-level attacks against the implementation to contexts in the high-level programming language, and by establishing full abstraction results.

1 Introduction

Several techniques for protection are based on randomization (e.g., [7, 9–11, 13, 14, 16, 21, 28, 29, 36]). The randomization may concern the layout of data and code within an address space, data representations, or the underlying instruction set. In all cases, the randomization introduces artificial diversity that can serve for impeding attacks. In particular, layout randomization can thwart attacks that rely on knowledge of the location of particular data and functions (such as system libraries). In addition, randomization can obfuscate program logic, against reverse engineering.

Other techniques for protection address similar goals. For example, methods that ensure the integrity of control flow and data flow, statically or dynamically, can also regulate the use of system libraries (e.g., [3, 14, 22, 27]). The static methods may be based on types or other static analyses. The dynamic methods often rely on reference monitors, whether implemented in hardware or software, at the boundaries of address spaces or inline. In addition to the diversity of

*A preliminary version of this work was presented at the 23rd IEEE Computer Security Foundations Symposium (July 2010).

their mechanisms, protection techniques vary in their goals and the underlying attack models. Some aim to offer precise, general guarantees, while others stop only some specific attack that can be easily modified to overcome the protection. They also vary in the difficulty of deploying them and in their costs. No single protection technique appears to be always superior to all others. In this paper we focus on layout randomization because it is in widespread use [20, 31], it has been subject to practical attacks (e.g., [5, 33, 34]), and it has hardly been studied rigorously.

We present layout randomization as part of an implementation for a high-level programming language. The language that we consider includes higher-order functions and mutable variables that hold natural numbers, which we call locations. Some of the locations are designated as public while others are designated as private, with the intent that an attacker should not have direct access to the latter. For instance, a program may store secret quantities in a private location; it may publish a function that internally uses the private location, and this function may be invoked by untrusted pieces of code. The implementation translates the high-level language to a lower-level language in which memory addresses are natural numbers; layout randomization consists in mapping the private locations to random addresses in data memory. If the data memory is large enough and the randomization good enough, then even an attacker with access to all of data memory cannot find the private locations efficiently with high probability. (Otherwise, the attacker may succeed, as demonstrated in actual exploits, e.g., [33].) We derive that the security properties that hold against attackers that cannot access the private locations directly continue to hold in this implementation, in a probabilistic sense and against resource-bounded adversaries.

Thus, our work takes place in a programming-language setting, and it draws on a line of research on protection in programming languages, and more broadly on ideas and techniques from programming-language theory (e.g., [1, 26]). These include the use of contexts for representing attackers, and of contextual equivalence and similar relations for expressing security properties. Remarkably, though, this line of research has said little on randomization; a notable exception is the work of Pucella and Schneider [32], which we describe further below. In addition, our probabilistic results are analogous to computational-soundness theorems in the analysis of security protocols (e.g., [4, 6, 12]). These theorems relate symbolic proofs of protocol security, in which keys and ciphertexts are formal expressions, to proofs in a computational model in which keys and ciphertexts are bitstrings subject to complexity-theoretic assumptions. Unlike security protocols, however, the systems that we consider neither include concurrency nor rely on cryptography, but they do include higher-order functions. Despite these important differences, we hope that our work will enrich the study of computational soundness, in particular by showing that some of its themes and methods are applicable beyond security protocols.

The next section (Section 2) discusses our results in more detail but still informally. Section 3 contains preliminary technical material. Sections 4 and 5 are the core of the paper; they treat models in which errors are fatal and recoverable (but costly), respectively. Section 6 concludes.

2 Discussion of results

Layout randomization can be applied in a variety of systems contexts. In some (in particular, in kernel mode), accesses to unmapped memory addresses may be fatal violations that result in immediate termination. In others (often in user mode), erroneous accesses may take place repeatedly without causing execution to abort; a program that performs an erroneous access may often recognize that it has done so.

This distinction leads to two models for what happens when an attacker accesses an unused address in data memory (rather than an address that houses a private location). In one model, such accesses are fatal violations; in the other, such accesses are not fatal and can be detected.

In both cases, our main results concern translations between the high-level language with locations and a lower-level language with natural-number addresses. In the high-level language, there is a distinct type of locations `loc` and, assuming that the expression M has this type, one can write expressions like $!_{\texttt{loc}}M$ and $M :=_{\texttt{loc}} M'$ for reading from and storing into a location. In contrast, in the low-level language, if M has type `nat` then one can write $!_{\texttt{nat}}M$ and $M :=_{\texttt{nat}} M'$ for reading from and writing to a natural-number address, which may be obtained as the result of arbitrary numerical computations in M.

In order to study the security of these translations, we represent high- and low-level attackers as contexts. More precisely, for a program M of type σ, we take attackers to be expressions C of type $\sigma \to \texttt{bool}$. The boolean output is standard in programming-language theory, and intuitively it could be seen as indicating whether the attacker is “happy”, but technically we could as well use the type $\sigma \to \texttt{nat}$, for example. Informally, we think of C as interacting with M and possibly trying to obtain information about the contents of private locations or to tamper with them. Attackers must not have direct access to the private locations, so we consider only public attackers C, which are those containing no occurrences of any private locations. (Public low-level attackers do have access to all of memory, nevertheless, but via natural-number addresses rather than via locations.)

This representation of the attacker as a context amounts to a threat model, which allows rich interactions between the program being protected and its attacker. Both theoretical work and practical attacks often employ more limited threat models, in which, for example, the attacker provides only one input or a small number of inputs. On the other hand, this representation excludes power-analysis attacks, timing-analysis attacks, and the like, as well as any attacks that subvert the underlying execution platform. Realistically, layout randomization may not withstand such attacks anyway.

In general, a low-level attacker could exploit the operations $!_{\texttt{nat}}$ and $:=_{\texttt{nat}}$ for crafting attacks that would be impossible in the high-level language. In an extreme case, when erroneous accesses are not fatal, an attacker could iterate over all addresses.

Nevertheless, we show that the attacks possible in the low-level language are no worse than those that are possible in the high-level language, in a probabilistic sense and, if erroneous accesses are not fatal, within some number of such memory accesses that serves as a bound on the complexity of the attacks. More precisely, we map each high-level program M to a low-level program $M^{\downarrow}$, and consider the behavior of $M^{\downarrow}$ in an arbitrary low-level context C. We construct a corresponding high-level context $C^{\uparrow}$ which does not directly access M's private locations and is such that M in $C^{\uparrow}$ exhibits the same behavior as $M^{\downarrow}$ does in C. For example, in the model where erroneous accesses are fatal, if $C^{\uparrow}M$ returns a given boolean then, with high probability, so does $CM^{\downarrow}$ (see Corollary 4.15). In the model where erroneous accesses are not fatal, this implication holds too, but only with an assumption on the number of memory accesses being bounded (Theorem 5.16).

Some of our results are phrased as full abstraction theorems for translations between the high-level language with locations and the lower-level language with natural-number addresses (Theorems 4.16 and 5.17). These theorems say, roughly, that two programs are equivalent in the high-level language if, and only if, their translations are equivalent (in a probabilistic sense) in the low-level language. (Computational soundness is the implication from the high-level equivalence to the low-level one.) The equivalences capture indistinguishability in the presence of an arbitrary attacker, represented as the context of the programs. Thus, the equivalences can express both secrecy and integrity properties. Therefore, the theorems imply the preservation of those secrecy and integrity properties.

This use of equivalences is similar to that in information-flow security. There, equivalences often relate two versions of the same program with different secret inputs, and preservation results often focus on equivalences that can be established using particular logics or type systems (see, e.g., [2, Section 7] and [**?**, 8, 18]). Those restrictions are however not universal (e.g., [17]), and we do not make them.

For instance, assuming that l is a private location and c and c' are two natural-number constants, the following expressions M and M':

$$\begin{array}{lcl} M & = & l := c \\ M' & = & l := c' \end{array}$$

are two trivial programs in the high-level language that differ only by the value they store in l. (Here and in other examples, we omit the subscript `loc` on memory operations, for brevity.) The programs M and M' are equivalent with respect to contexts that cannot access private locations. This property captures a secrecy guarantee. Similarly, if l' is a public location, the following expressions

M and M':

$$\begin{array}{lcl} M & = & \lambda f{:}\texttt{nat} \to \texttt{unit}. \\ & & \quad l := c; \\ & & \quad f(c); \\ & & \quad \texttt{if}\, !l = c\, \texttt{then}\, l' := c\, \texttt{else}\, l' := c' \\ M' & = & \lambda f{:}\texttt{nat} \to \texttt{unit}. \\ & & \quad l := c; \\ & & \quad f(c); \\ & & \quad l' := c \end{array}$$

are equivalent with respect to contexts that cannot access private locations, because the command f (supplied by the context) cannot tamper with l. This property captures an integrity guarantee. In an implementation in which l is housed in a random address in data memory, an attacker should find it hard to read or write the contents of l, so the secrecy and integrity guarantees should be preserved. We prove that this is indeed the case.

Such a result may seem obvious. However, as we discuss, some other "equally obvious" results do not hold, and some variants and extensions appear problematic. We illustrate this point with the following small example. Writing Ω for a nonterminating program and $*$ for the value of type `unit`, we consider the programs:

$$\begin{array}{lcl} M & = & \lambda f{:}\texttt{unit} \to \texttt{unit}.\, \Omega \\ M' & = & \lambda f{:}\texttt{unit} \to \texttt{unit}.\, \texttt{let}\, x\, \texttt{be}\, f(*)\, \texttt{in}\, \Omega \end{array}$$

The implementations of M and M' can be distinguished by a context that passes a function f that always produces a fatal error. Such a function can easily be expressed in the model where erroneous accesses are fatal. On the other hand, M and M' will be equivalent in the high-level language unless this language too includes constructs that force immediate termination. Therefore, full abstraction fails without such constructs. Although of mostly theoretical interest, this small example is reminiscent of some actual attacks in which the distinction between error and nontermination leaks important information [35].

Thus, our work demonstrates that layout randomization can offer some delicate but strong guarantees. Layout randomization is not just an ad hoc mitigation, or "security by obscurity".

Nevertheless, our results have substantial limitations. They provide an incomplete account of software protection, ignoring most of the complications of practical implementations.

Many of the limitations directly correspond to limitations of the languages that we consider. For instance, these languages do not include the storage of functions in the heap, which our results do not treat; so we do not study whether an attacker can call a piece of code by guessing where in memory it resides, as in "jump-to-`libc`" attacks [14].

Another limitation of our results is that they do not all apply to programs that receive or send "raw" locations—-although they do apply to higher-order programs that receive or send functions for manipulating locations. We deliberately define our languages with locations as first-class values of a type `loc`.

While this generality leads to an extra hypothesis in some of our theorems, it also enables us to discuss the difficulties that arise with locations as first-class values:

- Suppose that we allow `loc` to occur in contravariant positions in the types of the programs that we are protecting. In the implementations of those programs, locations correspond to natural numbers, but in general this correspondence is not surjective. So a low-level attacker may attempt to poison the programs by providing a number that does not represent a location instead of one that does represent a location, and might gain information from the resulting errors. Consider for instance the programs:

  $$\begin{array}{lcl} M & = & \lambda x\!:\!\texttt{loc}.\,\Omega \\ M' & = & \lambda x\!:\!\texttt{loc}.\,\texttt{let}\,y\,\texttt{be}\,!x\,\texttt{in}\,\Omega \end{array}$$

  While a high-level attacker cannot distinguish these two programs, a low-level attacker may attempt to distinguish them, with high probability, by passing a number that does not represent a public location: the naive implementation of M will diverge, that of M' will produce an error. Such examples might be addressed by an implementation strategy in which incoming numbers that should represent locations are tested. These tests are reminiscent of how pointers are treated with suspicion when they cross trust boundaries in operating systems and other software systems.

- Suppose that we allow `loc` to occur in covariant positions in the types of the programs that we are protecting. Then a low-level attacker may store the numbers that represent the locations that it receives, and use them later, while analogous storage is not possible for a high-level attacker—simply because locations cannot hold locations in our high-level model. Letting l_1 and l_2 be private locations, consider for instance the programs:

  $$\begin{array}{lcl} M & = & \lambda f\!:\!\texttt{loc} \rightarrow \texttt{unit}. \\ & & \quad \texttt{if}\,!l_2 = 0\,\texttt{then}\,l_2 := 1; f(l_1)\,\texttt{else}\,\Omega; \\ & & \quad l_1 := 0 \\ M' & = & \lambda f\!:\!\texttt{loc} \rightarrow \texttt{unit}. \\ & & \quad \texttt{if}\,!l_2 = 0\,\texttt{then}\,l_2 := 1; f(l_1)\,\texttt{else}\,\Omega; \\ & & \quad l_1 := 1 \end{array}$$

  They differ only in whether they store 0 or 1 in l_1. Both of these leak l_1 to an argument function f, then set l_1. They do the leaking at most once: this linearity is enforced by the flag l_2. A low-level context can store the number that represents l_1, then use it for reading what is stored in l_1, and thereby could distinguish the implementations of M and M' if no additional precautions are taken. This counterexample is reminiscent of some that arise in the study of cryptographic protocols, most notably a counterexample to forward secrecy [1]. It could perhaps be addressed by some of the techniques developed in such contexts. Unfortunately, those techniques may not result in attractive, realistic implementation strategies

> for a programming language such as ours, or for its obvious extensions where locations can hold other locations. Such extensions can bring up further problems, which it would be interesting to investigate in future research.

The significance of this limitation remains open to debate: one could argue that programs should never receive or send "raw" locations, that it is too hard to make this safe, and that exchanging functions (or objects with public methods and private fields) provides more flexibility.

These arguments are particularly sensible in the context of implementations where attackers have information on the offsets between private locations (much as in [33]). For instance, a practical implementation may well store several private locations near one another, in a randomly placed block of memory chosen for this purpose. Then an attacker that learns where l_1 is housed may also be able to infer that l_2 is nearby. Such dependencies can weaken security.

3 Technical preliminaries

This section presents basic technical material on which both Sections 5 and 4 rely. It describes high- and low-level memory models and the common components of the languages considered in this paper.

3.1 Memory models

We begin with a discussion of our memory models. We need two: an abstract one, for the high-level language, and a more concrete one, for the low-level language.

For the abstract model we assume a finite set Loc of *locations*, ranged over by l, and further assumed to be the disjoint union of two sets, PubLoc and PriLoc, of *public* and *private* locations. *Stores*, ranged over by s, are maps

$$s : \mathrm{Loc} \longrightarrow \mathbb{N}$$

sending locations to natural numbers. We write Store for the set of stores.

For the concrete model we take the memory as having addresses $0, \ldots, c$, for a given $c \geq 0$, which we think of as logical or virtual addresses rather than physical addresses; we assume that $|\mathrm{Loc}| \leq c+1$. *Memories*, ranged over by m, are maps

$$m : \{0, \ldots, c\} \longrightarrow \mathbb{N} + \mathbb{1}$$

where $\mathbb{1}$ is the set $\{*\}$, $+$ is disjoint union, and $m(a) = *$ indicates that a is an unused address. Storing natural numbers rather than words is an idealization, as is the view of natural numbers as atomic entities that all occupy the same space. With a little more effort we could use an alternative model where words are stored and arithmetic operations can be performed on them.

Memory layouts, ranged over by w, are 1-1 maps

$$w : \mathrm{Loc} \hookrightarrow \{0, \ldots, c\}$$

They are used to connect the abstract and concrete memory models. We consider only those memory layouts extending a given *public layout* $w_p : \mathrm{PubLoc} \hookrightarrow \{0, \ldots, c\}$. For any store s and a memory layout w, there is a corresponding memory s_w defined by:

$$s_w(a) = \begin{cases} s(l) & \text{if } w(l) = a \\ * & \text{if } a \notin \mathrm{Ran}(w) \end{cases}$$

where $\mathrm{Ran}(w)$ is the range of w. Note that the map $s \mapsto s_w$ is 1-1, but not onto; we say that *m has the form s_w* if it equals s_w for some w.

In order to make probabilistic assertions, we need a distribution over the layouts extending w_p: we take this to be the uniform distribution. That can be generated by fixing an ordering of Loc and then selecting, one-by-one, a non-repeating sequence of elements randomly from $\{0, \ldots, c\} \setminus \mathrm{Ran}(w_p)$, choosing uniformly at each point from the remaining elements. When $\varphi(w)$ is a statement, we write $\mathrm{P}(\varphi(w))$ for the probability that it holds with respect to this distribution.

For any $A \subseteq \mathbb{N}$ we define δ_A to be $\mathrm{P}(w \# A)$ where we write $w \# A$ to mean that $A \cap (\mathrm{Ran}(w) \setminus \mathrm{Ran}(w_p)) = \emptyset$. As we are using the uniform distribution, this depends only on the cardinality of A, if $A \subseteq \{0, \ldots, c\} \setminus \mathrm{Ran}(w_p)$; so we can set $\delta_{|A|} =_{\mathrm{def}} \delta_A$, having chosen such an A. Intuitively, δ_n is the probability that n distinct probes do not hit any of the private locations. Note that the notation δ_n makes sense only when $n \leq c + 1 - |\mathrm{PubLoc}|$, and that $\delta_n > 0$ if, and only if, $n \leq c + 1 - |\mathrm{Loc}|$. Then we can give δ_n by:

$$\delta_n = \Pi_{i=0}^{n-1} \left(1 - \frac{|\mathrm{PriLoc}|}{c + 1 - |\mathrm{PubLoc}| - i} \right)$$

or, in closed form (in terms of binomial coefficients), by:

$$\delta_n = \binom{c + 1 - n - |\mathrm{PubLoc}|}{|\mathrm{PriLoc}|} \Big/ \binom{c + 1 - |\mathrm{PubLoc}|}{|\mathrm{PriLoc}|}$$

These two forms are equivalent, because:

$$\begin{array}{cl} \multicolumn{2}{l}{\tbinom{c+1-n-|\mathrm{PubLoc}|}{|\mathrm{PriLoc}|} \big/ \tbinom{c+1-|\mathrm{PubLoc}|}{|\mathrm{PriLoc}|}} \\ = & \frac{(c+1-n-|\mathrm{PubLoc}|)!|\mathrm{PriLoc}|!(c+1-|\mathrm{Loc}|)!}{|\mathrm{PriLoc}|!(c+1-n-|\mathrm{Loc}|)!(c+1-|\mathrm{PubLoc}|)!} \\ = & \frac{(c+1-n-|\mathrm{PubLoc}|)!(c+1-|\mathrm{Loc}|)!}{(c+1-n-|\mathrm{Loc}|)!(c+1-|\mathrm{PubLoc}|)!} \\ = & \frac{\Pi_{i=0}^{n-1}(c+1-|\mathrm{Loc}|-i)}{\Pi_{i=0}^{n-1}(c+1-|\mathrm{PubLoc}|-i)} \\ = & \Pi_{i=0}^{n-1} \frac{c+1-|\mathrm{Loc}|-i}{c+1-|\mathrm{PubLoc}|-i} \\ = & \Pi_{i=0}^{n-1} \left(1 - \frac{|\mathrm{Loc}|-|\mathrm{PubLoc}|}{c+1-|\mathrm{PubLoc}|-i}\right) \\ = & \Pi_{i=0}^{n-1} \left(1 - \frac{|\mathrm{PriLoc}|}{c+1-|\mathrm{PubLoc}|-i}\right) \end{array}$$

Thus, δ_n tends to 1 as c increases while PriLoc and PubLoc remain fixed. Intuitively, this fact means that, if one looks for private locations in a large enough memory, getting n tries, one is almost certain to miss if the memory is large enough. In the special case $n = 0$, δ_0 is always simply 1.

Using the above closed-form expression, we have the following formula for δ_n:

$$\delta_n = \frac{|\{w \mid w \text{ extends } w_p \text{ and } \mathrm{Ran}(w) \cap A = \emptyset\}|}{|\{w \mid w \text{ extends } w_p\}|}$$

where A is any subset of $\{0, \ldots, c\} \setminus \mathrm{Ran}(w_p)$ of size n; it does not matter which such subset is chosen. That is, δ_n is the fraction of all memory layouts that extend w_p but do not meet any of a fixed choice of n non-public memory addresses.

3.2 Languages

A number of quite similar languages are considered in this paper. They are all versions of Moggi's (call-by-value) computational λ-calculus, or λ_c-calculus, [24, 25] with natural number and, possibly, location types, and with memory-access operations at natural-number or location types. They all also have sum types, which represent disjoint or discriminated unions [23], and recursion [19].

The types of such a language are given as follows:

$$\sigma ::= b \mid \mathtt{unit} \mid \sigma \times \sigma \mid \sigma + \sigma \mid \sigma \to \sigma$$

where b ranges over a given set of *basic* types which always includes a natural-number type `nat` and may also include a location type `loc`. We write `bool` to abbreviate $\mathtt{unit} + \mathtt{unit}$.

The terms of such a language are ranged over by M and N, and given by:

$$\begin{array}{lll} M & ::= & x \mid c \mid * \mid (M, M) \mid \mathtt{fst}\, M \mid \mathtt{snd}\, M \mid \\ & & \mathtt{inl}_{\sigma,\sigma}\, M \mid \mathtt{inr}_{\sigma,\sigma}\, M \mid \\ & & \mathtt{cases}\, M\, \mathtt{inl}\, x\!:\!\sigma.\, M\, \mathtt{inr}\, x\!:\!\sigma.\, M \mid \\ & & \lambda x\!:\!\sigma.\, M \mid MM \mid \mathtt{rec}(f\!:\!\sigma \to \tau, x\!:\!\sigma).\, M \end{array}$$

where $c : \sigma$ ranges over a given set of constants c of given unique types σ. These always include the natural numbers $n \in \mathbb{N}$, together with a supply of constants for the usual arithmetic operations and relations, such as addition $+ : \mathtt{nat} \times \mathtt{nat} \to \mathtt{nat}$ and equality $=_{\mathtt{nat}} : \mathtt{nat} \times \mathtt{nat} \to \mathtt{bool}$. They may also include constants for memory access, for example $:=_{\mathtt{nat}} : \mathtt{loc} \times \mathtt{nat} \to \mathtt{unit}$ for assignment. The recursion construction $\mathtt{rec}(f : \sigma \to \tau, x : \sigma).\, M$ should be thought of as defining a function $f\!:\!\sigma \to \tau$ such that $f(x) = M$.

There are standard notions of free and bound variables, of closed terms, and of the capture-avoiding substitution $M[N/x]$ of a term N for all free occurrences of a variable x in a term M. There are also standard typing rules for judgements $\Gamma \vdash M\!:\!\sigma$, that a term M has type σ in the context Γ, where contexts have the form $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$. Here are two examples:

$$\frac{\Gamma \vdash M\!:\!\sigma}{\Gamma \vdash \mathtt{inl}_{\sigma,\tau}\, M\!:\!\sigma + \tau}$$

$$\frac{\Gamma, f\!:\!\sigma \to \tau, x\!:\!\sigma \vdash M\!:\!\tau}{\Gamma \vdash \mathtt{rec}(f\!:\!\sigma \to \tau, x\!:\!\sigma).\, M\!:\!\sigma \to \tau}$$

We write $M : \sigma$ for $\vdash M : \sigma$ and then say that M is well-typed (when it is necessarily closed). Unique typing holds: a term has at most one type relative to a given environment.

We may omit type subscripts when that should not cause confusion; for example we write $\mathtt{inl}\, M$ instead of $\mathtt{inl}_{\sigma,\tau}\, M$. We also write $\mathtt{let}\, x\!:\!\sigma\; \mathtt{be}\, M\, \mathtt{in}\, N$ for $(\lambda x\!:\!\sigma.\, N)M$, and we adopt standard infix notations, e.g., writing $M := N$ for $:= (M, N)$, if that improves readability. For the booleans, we write $\mathtt{true}$ and $\mathtt{false}$ for $\mathtt{inl}\, *$ and $\mathtt{inr}\, *$, respectively, and we write $\mathtt{if}\, B\, \mathtt{then}\, M\, \mathtt{else}\, N$ for $\mathtt{cases}\, B\, \mathtt{inl}\, x\!:\!\mathtt{unit}.\, M\, \mathtt{inr}\, x\!:\!\mathtt{unit}.\, N$, where x occurs free in neither M nor N. To make the usual connection between applicative and imperative programs, we may write $\mathtt{com}$ (which stands for "command") for $\mathtt{unit}$, $\mathtt{skip}$ for $*$, and $M; N$ for $\mathtt{let}\, x\!:\!\mathtt{unit}\, \mathtt{be}\, M\, \mathtt{in}\, N$ (where x is not free in N).

Throughout this paper, we define the operational semantics of such a language in the style of Felleisen and Friedman [15], beginning by defining *values* V, *evaluation contexts* E, and *redexes* R. We classify each constant as a value or a redex; in particular the numerals and the constants for the assumed arithmetic operations and relations are always values. Values are terms which can be thought of as (syntax for) completed computations; they are ranged over by V and defined by:

$$\begin{array}{rcl} V & ::= & c \quad (\text{if } c \text{ is classified as a value}) \mid \\ & & * \mid (V, V) \mid \mathtt{inl}\, V \mid \mathtt{inr}\, V \mid \lambda x\!:\!\sigma.\, M \end{array}$$

Evaluation contexts are ranged over by E and are defined by:

$$\begin{array}{rcl} E & ::= & [-] \mid \\ & & (E, M) \mid (V, E) \mid \mathtt{fst}\, E \mid \mathtt{snd}\, E \mid \\ & & \mathtt{inl}\, E \mid \mathtt{inr}\, E \mid \mathtt{cases}\, E\, \mathtt{inl}\, x\!:\!\sigma.\, M\, \mathtt{inr}\, x\!:\!\sigma.\, M \mid \\ & & EM \mid VE \end{array}$$

We write $E[M]$ for the term obtained by replacing the "hole" $[-]$ in an evaluation context E by a term M. The computational thought behind evaluation contexts is that, in a term of the form $E[M]$, the first computational step arises within M. The redexes R include:

$$c \quad (\text{if } c \text{ is classified as a redex})$$

$$\mathtt{fst}\, (V, V) \qquad \mathtt{snd}\, (V, V)$$

$$\mathtt{cases}\, \mathtt{inl}\, V\, \mathtt{inl}\, x\!:\!\sigma.\, M\, \mathtt{inr}\, x\!:\!\sigma.\, M$$

$$\mathtt{cases}\, \mathtt{inr}\, V\, \mathtt{inl}\, x\!:\!\sigma.\, M\, \mathtt{inr}\, x\!:\!\sigma.\, M$$

$$(\lambda x\!:\!\sigma.\, M)V \qquad \mathtt{rec}(f\!:\!\sigma \to \tau, x\!:\!\sigma).\, M$$

together with specified other redexes involving the various constants, including evident arithmetic redexes for the assumed arithmetic operations and relations, for example $i + j$ and $i =_{\mathtt{nat}} j$.

For every well-typed term M, one of the following two mutually exclusive possibilities holds:

- M is a value, or
- M can be analyzed uniquely in the form $E[R]$, with R a well-typed redex.

However, this has to be verified separately for each language.

The operational semantics itself involves various relations and properties, and there is quite a bit of variation between the different languages. In all cases, however, a relation $R \longrightarrow M$ between the above redexes and terms proves useful. It is defined as follows:

$$\begin{gathered}
\texttt{fst}\,(V, V') \longrightarrow V \qquad \texttt{snd}\,(V, V') \longrightarrow V' \\
(\lambda x\!:\!\sigma.\, M)V \longrightarrow M[V/x] \\
\texttt{rec}(f\!:\!\sigma \to \tau, x\!:\!\sigma).\, M \longrightarrow \lambda x\!:\!\sigma.\, M[\texttt{rec}(f\!:\!\sigma \to \tau, x\!:\!\sigma).\, M/f] \\
\cdots
\end{gathered}$$

where the ellipses indicate evident missing arithmetic redex transitions, such as:

$$i =_{\texttt{nat}} i \;\longrightarrow\; \texttt{true} \qquad \text{and} \qquad i + j \;\longrightarrow\; k$$

where k is the sum of i and j.

4 The fatal-error model

In this model, an erroneous low-level memory access gives rise to an irrecoverable error. A corresponding construct to raise such errors is needed at high-level, as discussed in Section 2. Section 4.1 presents the high-level language; since it has an error-raising construction, its operational semantics has an error predicate, as well as the usual transition relation. Section 4.2 presents the low-level language; its operational semantics additionally has an error predicate which is labelled by a natural number; this instrumentation is used in proofs and is not regarded as part of program behavior.

To mediate between the two languages, an instrumented high-level language, extending the high-level language, is given in Section 4.3. This language has facilities for memory access at both location and natural-number types, with non-public natural-number accesses always raising an instrumented error. We have a behavior-preserving translation to the high-level language and a translation to the low-level language which is additionally sensitive to the instrumentation.

With these tools, we prove our main results for the fatal-error model in Sections 4.4 and 4.5, to the effect that high- and low-level attackers have essentially equal power, modulo the translation from the high-level to the low-level language, and that this translation preserves and reflects suitable notions of contextual public equivalence. In both this and the following section, we often omit, or abbreviate, routine proofs.

4.1 The high-level language

The high-level language employs the abstract notion of location. The basic types are $\texttt{nat}$ and $\texttt{loc}$. The constants are the arithmetic constants, together with error-raising constants:

$$\texttt{raise\_error}_\sigma : \sigma \qquad \text{(for every } \sigma\text{)}$$

and constants for accessing and updating locations:

$$\begin{gathered} l_{\texttt{loc}} : \texttt{loc} \quad (l \in \mathrm{Loc}) \\ !_{\texttt{loc}} : \texttt{loc} \to \texttt{nat} \\ :=_{\texttt{loc}} : \texttt{loc} \times \texttt{nat} \to \texttt{com} \end{gathered}$$

Of these, the arithmetic constants and the constants for accesses and updating locations are values, and the error-raising constant is a redex. As well as the redexes specified by the general framework, there are the following two kinds:

$$!_{\texttt{loc}} V \qquad V :=_{\texttt{loc}} V$$

For the semantics of the high-level language we define a *configuration* to be a pair (s, M) with s a store and M a well-typed term. The semantics then consists of a transition relation and an error property:

$$(s, M) \longrightarrow (s', M') \qquad\qquad (s, M) \downarrow_{\text{error}}$$

The transition relation and the error property are obtained from the special case of redexes by two rules:

$$\frac{(s, R) \longrightarrow (s', M')}{(s, E[R]) \longrightarrow (s', E[M'])} \qquad\qquad \frac{(s, R) \downarrow_{\text{error}}}{(s, E[R]) \downarrow_{\text{error}}}$$

For redexes we take the transitions to be given by:

$$(s, !_{\texttt{loc}} l_{\texttt{loc}}) \longrightarrow (s, n) \quad (s(l) = n)$$

$$(s, l_{\texttt{loc}} :=_{\texttt{loc}} n) \longrightarrow (s[l \mapsto n], \texttt{skip})$$

and a rule:

$$\frac{R \longrightarrow M'}{(s, R) \longrightarrow (s, M')}$$

The error property is given by:

$$(s, \texttt{raise\_error}_\sigma) \downarrow_{\text{error}}$$

We have the following subject-reduction theorem for the high-level semantics:

Lemma 4.1 *For any configuration (s, M), with $M : \sigma$, one of the following three mutually exclusive statements holds:*

- *M is a value,*
- *$(s, M) \longrightarrow (s', M')$ for some uniquely determined s' and $M' : \sigma$, or*
- *$(s, M) \downarrow_{\mathrm{error}}$.*

The operational semantics is "small-step"; one can define a corresponding "big-step" semantics by:

$$\begin{array}{lcl} (s, M) \Longrightarrow (s', V) & \Longleftrightarrow & (s, M) \rightarrow^* (s', V) \\ (s, M) \Downarrow_{\mathrm{error}} & \Longleftrightarrow & \exists s', M'.\, (s, M) \rightarrow^* (s', M') \downarrow_{\mathrm{error}} \\ (s, M) \Uparrow & \Longleftrightarrow & \forall n.\, \exists s', M'.\, (s, M) \rightarrow^n (s', M') \end{array}$$

Note that these relations and properties are mutually exclusive. Lemma 4.1 yields the following big-step subject-reduction theorem:

Lemma 4.2 *For any configuration (s, M), with $M : \sigma$, one of the following three mutually exclusive statements holds:*

- *$(s, M) \Longrightarrow (s', V)$ for a unique s' and $V : \sigma$,*
- *$(s, M) \Downarrow_{\mathrm{error}}$, or*
- *$(s, M) \Uparrow$.*

4.2 The low-level language

In the low-level language all memory accesses are made via natural numbers. Consequently we take the only basic type to be `nat`. (A possible variant would be to have a separate memory-address type.) As well as the arithmetic constants, the low-level language has error-raising constants:

$$\mathtt{raise\_error}_\sigma \qquad \text{(for every } \sigma\text{)}$$

and memory-access constants:

$$\begin{array}{c} l_{\mathtt{nat}} : \mathtt{nat} \quad (l \in \mathrm{Loc}) \\ !_{\mathtt{nat}} : \mathtt{nat} \rightarrow \mathtt{nat} \\ :=_{\mathtt{nat}} : \mathtt{nat} \times \mathtt{nat} \rightarrow \mathtt{com} \end{array}$$

Note that `loc` cannot occur in σ in $\mathtt{raise\_error}_\sigma$, and that there are constants $l_{\mathtt{nat}}$ for all the locations, not just the public ones. We say that a term is *public* if every $l_{\mathtt{nat}}$ that occurs in it has $l \in \mathrm{PubLoc}$. We take $!_{\mathtt{nat}}$ and $:=_{\mathtt{nat}}$ to be values, and $\mathtt{raise\_error}_\sigma$ and $l_{\mathtt{nat}}$ (with $l \in \mathrm{Loc}$) to be redexes. The other redexes are those specified by the general framework, together with:

$$!_{\mathtt{nat}} V \qquad V :=_{\mathtt{nat}} V$$

Configurations in the low-level operational semantics are pairs (m, M) of a memory m and a well-typed term M. The semantics is defined relative to a choice of a memory layout. It consists of a transition relation and two error properties, all relative to the memory layout chosen:

$$w \models (m, M) \longrightarrow (m', M')$$

$$w \models (m, M) \downarrow_{\mathrm{error}} \qquad w \models (m, M) \downarrow^{a}_{\mathrm{error}}$$

These are obtained from the special case of redexes much as before:

$$\frac{w \models (m, R) \longrightarrow (m', M')}{w \models (m, E[R]) \longrightarrow (m', E[M'])}$$

$$\frac{w \models (m, R) \downarrow_{\mathrm{error}}}{w \models (m, E[R]) \downarrow_{\mathrm{error}}} \qquad \frac{w \models (m, R) \downarrow^{a}_{\mathrm{error}}}{w \models (m, E[R]) \downarrow^{a}_{\mathrm{error}}}$$

For the redexes we take the transitions and error properties to be given by the rule:

$$\frac{R \longrightarrow M'}{w \models (m, R) \longrightarrow (m, M')}$$

together with:

$$\begin{array}{l} w \models (m, \texttt{raise\_error}_{\sigma}) \downarrow_{\mathrm{error}} \\ w \models (m, l_{\texttt{nat}}) \longrightarrow (m, w(l)) \ \ (l \in \mathrm{Loc}) \end{array}$$

and:

$$\begin{array}{ll} w \models (m, !_{\texttt{nat}} a) \longrightarrow (m, n) & (\text{if } a \in \{0, \ldots, c\} \text{ and } m(a) = n) \\ w \models (m, !_{\texttt{nat}} a) \downarrow^{a}_{\mathrm{error}} & (\text{if } a \notin \{0, \ldots, c\} \text{ or } m(a) = *) \end{array}$$

and:

$$\begin{array}{ll} w \models (m, a :=_{\texttt{nat}} n) \longrightarrow (m[a \mapsto n], \texttt{skip}) & (\text{if } a \in \{0, \ldots, c\} \text{ and } m(a) \neq *) \\ w \models (m, a :=_{\texttt{nat}} n) \downarrow^{a}_{\mathrm{error}} & (\text{if } a \notin \{0, \ldots, c\} \text{ or } m(a) = * \ ) \end{array}$$

We have the following subject-reduction theorem for the low-level semantics:

Lemma 4.3 *For any memory layout w and configuration (m, M), with $M : \sigma$, one of the following four mutually exclusive statements holds:*

- *M is a value,*
- *$w \models (m, M) \longrightarrow (m', M')$ for some uniquely determined m' and $M' : \sigma$, and if m has the form s_w, so does m',*
- *$w \models (m, M) \downarrow_{\mathrm{error}}$, or*
- *$w \models (m, M) \downarrow^{a}_{\mathrm{error}}$ for some uniquely determined a.*

The low-level big-step operational semantics is defined by:

$$\begin{array}{lcl} w \models (m, M) \Longrightarrow (m', V) & \Longleftrightarrow & w \models (m, M) \rightarrow^{*} (m', V) \\ w \models (m, M) \Downarrow^{a}_{\mathrm{error}} & \Longleftrightarrow & \exists m', M'. w \models (m, M) \rightarrow^{*} (m', M') \downarrow^{a}_{\mathrm{error}} \\ w \models (m, M) \Downarrow_{\mathrm{error}} & \Longleftrightarrow & \exists m', M'. w \models (m, M) \rightarrow^{*} (m', M') \downarrow_{\mathrm{error}} \\ w \models (m, M) \Uparrow & \Longleftrightarrow & \forall n. \exists m', M'. w \models (m, M) \rightarrow^{n} (m', M') \end{array}$$

As is the case at the high-level, these relations and properties are mutually exclusive. Note that if $w \models (m, M) \downarrow^a_{\mathrm{error}}$ or $w \models (m, M) \Downarrow^a_{\mathrm{error}}$, and m has the form s_w, then $a \notin \mathrm{Ran}(w_p)$.

Lemma 4.3 yields a big-step subject-reduction theorem:

Lemma 4.4 *For any memory layout w and configuration (m, M), with $M : \sigma$, one of the following four mutually exclusive statements holds:*

- $w \models (m, M) \Longrightarrow (m', V)$ *for a unique m' and $V : \sigma$, and if m has the form s_w, so does m',*
- $w \models (m, M) \Downarrow_{\mathrm{error}}$*,*
- $w \models (m, M) \Downarrow^a_{\mathrm{error}}$*, for some uniquely determined a, or*
- $w \models (m, M) \Uparrow$.

4.3 The instrumented high-level language

In order to relate the high-level semantics uniformly to the low-level language we instrument it by adding some constants for accessing the store at type `nat`; in the final analysis, these will be translated away. Thus, the instrumented high-level language serves as a stepping stone, with semantics that resembles that of the high-level language but with a syntax that includes low-level constructs.

The instrumented high-level language has the same basic types as the high-level language and its constants are those of the high-level language together with:

$$\begin{gathered} l_{\mathtt{nat}} : \mathtt{nat} \quad (l \in \mathrm{PubLoc}) \\ !_{\mathtt{nat}} : \mathtt{nat} \to \mathtt{nat} \\ :=_{\mathtt{nat}} : \mathtt{nat} \times \mathtt{nat} \to \mathtt{com} \end{gathered}$$

We take $l_{\mathtt{nat}}$ to be a redex (for $l \in \mathrm{PubLoc}$), and $!_{\mathtt{nat}}$ and $:=_{\mathtt{nat}}$ to be values, and classify the other constants as in the case of the high-level language. The other redexes are those specified by the general framework, together with the following ones:

$$\begin{array}{ll} !_{\mathtt{nat}} V & V :=_{\mathtt{nat}} V \\ !_{\mathtt{loc}} V & V :=_{\mathtt{loc}} V \end{array}$$

the latter two kinds being inherited from the high-level language.

For the operational semantics, configurations are defined as for the high-level language, but we add an instrumented error property:

$$(s, M) \downarrow^a_{\mathrm{error}} \quad (a \in \mathbb{N})$$

We then proceed as for the high-level language, adding a rule for the instrumented error property:

$$\frac{(s, R) \downarrow^a_{\mathrm{error}}}{(s, E[R]) \downarrow^a_{\mathrm{error}}}$$

redex transitions:

$$\begin{array}{lll}(s, l_{\texttt{nat}}) \longrightarrow (s, w_p(l)) & (l \in \text{PubLoc}) \\ (s, !_{\texttt{nat}} a) \longrightarrow (s, s(l)) & (l \in \text{PubLoc}, a = w_p(l)) \\ (s, a :=_{\texttt{nat}} n) \longrightarrow (s[l \mapsto n], \texttt{skip}) & (l \in \text{PubLoc}, a = w_p(l))\end{array}$$

and instrumented error properties:

$$(s, !_{\texttt{nat}} a) \downarrow^a_{\text{error}} \qquad (s, a :=_{\texttt{nat}} n) \downarrow^a_{\text{error}} \qquad (a \notin \text{Ran}(w_{p)})$$

For the analogue to Lemma 4.1, one adds one more possibility to the list of mutually exclusive possibilities:

- $(s, M) \downarrow^a_{\text{error}}$ for some uniquely determined a.

For the big-step semantics one defines one more predicate:

$$(s, M) \Downarrow^a_{\text{error}} \quad \Longleftrightarrow \quad \exists s', M'.(s, M) \rightarrow^* (s', M') \downarrow^a_{\text{error}}$$

and then, for the analogue of Lemma 4.2 one adds the following possibility to the list of mutually exclusive possibilities:

- $(s, M) \Downarrow^a_{\text{error}}$ for some unique a.

Note that if $(s, M) \downarrow^a_{\text{error}}$ or $(s, M) \Downarrow^a_{\text{error}}$ then $a \notin \text{Ran}(w_p)$. Note too that the operational semantics of the instrumented high-level language is conservative over that of the high-level language. For the small-step operational semantics, a transition $(s, M) \longrightarrow (s', M')$ holds in the high-level language if, and only if, it does in the instrumented high-level language, and the same holds for a property $(s, M) \downarrow_{\text{error}}$. Conservativity then follows for the big-step operational semantics.

4.3.1 Translating instrumented high-level to high-level

Every term $M : \sigma$ of the instrumented high-level language can be translated to a term $M^{\uparrow} : \sigma$ of the high-level language. First we need a function to convert addresses of public locations to the locations themselves. Let $l^{(1)}, \ldots, l^{(p)}$ be a listing without repetitions of PubLoc, and set $a_i =_{\text{def}} w_p(l^{(i)})$, for $i = 1, p$. Define the high-level term $G : \texttt{nat} \rightarrow \texttt{loc}$ to be:

$$\begin{array}{ll}\lambda x{:}\texttt{nat}. & \texttt{if}\ x = a_1\ \texttt{then}\ (l^{(1)})_{\texttt{loc}} \\ & \texttt{elseif}\ x = a_2\ \texttt{then}\ (l^{(2)})_{\texttt{loc}} \\ & \qquad\qquad \vdots \\ & \texttt{elseif}\ x = a_p\ \texttt{then}\ (l^{(p)})_{\texttt{loc}} \\ & \texttt{else raise\_error}_{\texttt{loc}}\end{array}$$

with the evident understanding of the multiple conditional.

Then the translation is given by replacing the additional constants of the instrumented high-level language as follows:

$$\begin{array}{lcl} l^{\uparrow}_{\mathtt{nat}} & = & w_p(l) \quad (l \in \mathrm{PubLoc}) \\ !^{\uparrow}_{\mathtt{nat}} & = & \lambda x\!:\!\mathtt{nat}.\, !_{\mathtt{loc}} Gx \\ :=^{\uparrow}_{\mathtt{nat}} & = & \lambda x\!:\!\mathtt{nat} \times \mathtt{nat}.\, G(\mathtt{fst}\, x) :=_{\mathtt{loc}} (\mathtt{snd}\, x) \end{array}$$

and leaving the other constants fixed.

Define $(s, A) \downarrow^o_{\mathrm{error}}$ to hold if, and only if, either $(s, A) \downarrow_{\mathrm{error}}$ holds or $(s, A) \downarrow^a_{\mathrm{error}}$ does, for some a. Then the translation is correct in the following sense:

Lemma 4.5 *Let M be a well-typed term of the instrumented high-level language. Then:*

1. *If M is a value then so is $M^{\uparrow}$.*
2. *If $(s, M) \longrightarrow (s', M')$ then $(s, M^{\uparrow}) \longrightarrow^* (s', (M')^{\uparrow})$.*
3. *If $(s, M) \downarrow^o_{\mathrm{error}}$ then $(s, M^{\uparrow}) \longrightarrow^* (s, M') \downarrow_{\mathrm{error}}$, for some M'.*

Proof: Part 1 follows by inspection. For part 2, one shows first that, for any redex R, if $(s, R) \longrightarrow (s', M')$ then $(s, R^{\uparrow}) \longrightarrow^* (s', (M')^{\uparrow})$. One shows next that, if E is an evaluation context, then $E^{\uparrow}$ is too (taking $[-]^{\uparrow} = [-]$, etc.) and $E[M]^{\uparrow} = E^{\uparrow}[M^{\uparrow}]$. Part 2 then follows. For part 3, one shows first that, for any redex R, if $(s, R) \downarrow^o_{\mathrm{error}}$ then $(s, R^{\uparrow}) \longrightarrow^* (s, M') \downarrow_{\mathrm{error}}$, for some M', and then applies the previous remark on evaluation contexts. ■

Define $(s, A) \Downarrow^o_{\mathrm{error}}$ to hold if, and only if, either $(s, A) \Downarrow_{\mathrm{error}}$ holds or $(s, A) \Downarrow^a_{\mathrm{error}}$ does, for some a. Then in terms of big-step relations and properties, using Lemma 4.5 we obtain:

Proposition 4.6 *Let M be a well-typed term of the instrumented high-level language. Then:*

1. *If $(s, M) \Longrightarrow (s', V)$ then $(s, M^{\uparrow}) \Longrightarrow (s', V^{\uparrow})$.*
2. *If $(s, M) \Downarrow^o_{\mathrm{error}}$ then $(s, M^{\uparrow}) \Downarrow_{\mathrm{error}}$.*
3. *If $(s, M) \Uparrow$ then $(s, M^{\uparrow}) \Uparrow$.*

4.3.2 Translating instrumented high-level to low-level

We translate types σ and terms $M : \sigma$ of the instrumented high-level language to types $\sigma^{\downarrow}$ and terms $M^{\downarrow} : \sigma^{\downarrow}$ of the low-level language. For types we replace all occurrences of `loc` by `nat`. For terms we replace each occurrence of a type σ by one of $\sigma^{\downarrow}$, and translate the constants as follows:

$$\begin{array}{lcl} (l_{\mathtt{loc}})^{\downarrow} & = & l_{\mathtt{nat}} \\ (!_{\mathtt{loc}})^{\downarrow} & = & !_{\mathtt{nat}} \\ (:=_{\mathtt{loc}})^{\downarrow} & = & :=_{\mathtt{nat}} \\ (\mathtt{raise\_error}_{\sigma})^{\downarrow} & = & \mathtt{raise\_error}_{\sigma^{\downarrow}} \end{array}$$

taking the translation to act as the identity on the other constants, viz.: $l_{\mathtt{nat}}$ ($l \in \mathrm{PubLoc}$), $!_{\mathtt{nat}}$ and $:=_{\mathtt{loc}}$.

The translation is correct with respect to the low-level semantics, in the sense, roughly, that $M^{\downarrow}$ simulates M. However there is a small problem in that the translation of a location value is not a natural-number value but, rather, is a natural-number redex. For that reason a translation of a term of a given type, e.g., $l_{\mathtt{loc}}$, can make a transition to a term which is not itself a translation of a term of that type.

In order to track this correspondence between high-level and low-level terms, we define a simulation relation $M \searrow_w N$ between terms of the instrumented high-level language and the low-level language, parameterized on a memory layout w. We take this relation to be the least relation between terms of the instrumented high-level language and the low-level language which includes:

$$c \searrow_w c^{\downarrow} \qquad l_{\mathtt{loc}} \searrow_w w(l)$$

and which is closed under the other language constructs, meaning that, for example:

- if $M_1 \searrow_w N_1$ and $M_2 \searrow_w N_2$ then $M_1 M_2 \searrow_w N_1 N_2$, and
- if $M \searrow_w N$ then $\lambda x\!:\!\sigma.\, M \searrow_w \lambda x\!:\!\sigma^{\downarrow}.\, N$.

For any term M of the instrumented high-level language we have $M \searrow_w M^{\downarrow}$; further, if $M\!:\!\sigma$ and $M \searrow_w N$ then $N\!:\!\sigma^{\downarrow}$.

We now prove a series of lemmas, leading to our main simulation lemma. The first lemma concerns values.

Lemma 4.7 *Suppose that $V \searrow_w N$ for a well-typed value V. Then for some V' with $V \searrow_w V'$ we have, for any m, $w \models (m, N) \longrightarrow^* (m, V')$.*

The second lemma concerns redexes.

Lemma 4.8 *Suppose that $R \searrow_w N$ and that $(s, R) \longrightarrow (s', M')$. Then for some N' with $M' \searrow_w N'$ we have $w \models (s_w, N) \longrightarrow^* (s'_w, N')$.*

Proof: The proof is straightforward. The case where $R \longrightarrow M'$ makes use of Lemma 4.7. For example, suppose that R is $\mathtt{fst}\,(V, V')$. Then N has the form $\mathtt{fst}\,(N_1, N'_1)$ where $V \searrow_w N_1$ and $V' \searrow_w N'_1$. Applying Lemma 4.7 twice, we then have that:

$$(s_w, \mathtt{fst}\,(N_1, N'_1)) \longrightarrow (s_w, \mathtt{fst}\,(V_1, V'_1)) \longrightarrow (s_w, V_1)$$

for some values V_1, V'_1 with $V \searrow_w V_1$ and $V' \searrow_w V'_1$. ■

The third lemma concerns evaluation contexts. The simulation relation is extended in an evident way to evaluation contexts, taking, $[-] \searrow_w [-]$, etc. One easily sees that if $E \searrow_w E'$ and $M \searrow_w N$ then $E[M] \searrow_w E'[N]$.

Lemma 4.9 *Suppose that $E[R] \searrow_w N$. Then N has the form $E'[N_1]$ where $E \searrow_w E'$ and $R \searrow_w N_1$.*

We can now give the anticipated simulation lemma:

Lemma 4.10 *Suppose that $M \searrow_w N$ for a well-typed terms M of the instrumented high-level language and N of the low-level language. Then:*

1. *If M is a value, then there is a V', with $M \searrow_w V'$, such that for any m, $w \models (m, N) \longrightarrow^* (m, V')$.*
2. *If $(s, M) \longrightarrow (s', M')$, then there is an N' with $M' \searrow_w N'$ such that $w \models (s_w, N) \longrightarrow^* (s'_w, N')$.*
3. *If $(s, M) \downarrow_{\text{error}}$ then $w \models (s_w, N) \downarrow_{\text{error}}$.*
4. *If $(s, M) \downarrow^a_{\text{error}}$ then, if $a \notin \text{Ran}(w)$, $w \models (s_w, N) \downarrow^a_{\text{error}}$.*

Proof: The first part is an immediate consequence of Lemma 4.7, as is the second part of Lemmas 4.8 and 4.9. For the third and fourth parts, one verifies the assertions when M is a redex, and then applies Lemma 4.9. ■

The fourth case is particularly important as it enables one to use the instrumented high-level language to track memory accesses in the low-level language largely independently of memory layout. In terms of big-step relations and properties, using the previous lemma, we have:

Proposition 4.11 *Suppose that $M \searrow_w N$ for well-typed terms M of the instrumented high-level language and N of the low-level language. Then:*

1. *If $(s, M) \Longrightarrow (s', V)$ then $w \models (s_w, N) \Longrightarrow (s'_w, V')$ for some V' with $V \searrow_w V'$.*
2. *If $(s, M) \Downarrow_{\text{error}}$ then $w \models (s_w, N) \Downarrow_{\text{error}}$.*
3. *If $(s, M) \Downarrow^a_{\text{error}}$ then, if $a \notin \text{Ran}(w)$, $w \models (s_w, N) \Downarrow^a_{\text{error}}$.*
4. *If $(s, M) \Uparrow$ then, for any w, $w \models (s_w, N) \Uparrow$.*

4.4 High- and low-level attackers

We are now in a position to formulate our theorems for the fatal-error case. The general idea is to show that a program (taken to be a closed term) executed in the abstract memory model is equally secure if executed in the concrete one. In terms of our typed programming language we wish to show that a high-level term $M : \sigma$ is as secure as its low-level counterpart $M^\downarrow : \sigma^\downarrow$. We will prove that this holds if σ is `loc`-free, i.e., if $\sigma^\downarrow = \sigma$. (It does not hold more generally—see the discussion in Section 2.)

In this section, we study the relation between high- and low-level attackers. In Section 4.5, we consider equivalences.

Say that an instrumented high-level term (low-level term) is *public* if it contains no occurrence of any $l_{\texttt{loc}}$ (respectively $l_{\texttt{nat}}$) with $l \in \mathrm{PriLoc}$. We would like to show that attackers gain no advantage by attacking at low-level rather than at high-level. They certainly lose none, as, for any public high-level term $C : \sigma \to \texttt{bool}$, the low-level term $C^{\downarrow}$ is of equal attacking power:

Proposition 4.12 *Let $M : \sigma$ be a high-level term and let $C : \sigma \to \texttt{bool}$ be a public high-level term. Then:*

1. *If $(s, CM) \Longrightarrow (s', V)$ then, for any w, $w \models (s_w, C^{\downarrow}M^{\downarrow}) \Longrightarrow (s'_w, V)$.*
2. *If $(s, CM) \Downarrow_{\mathrm{error}}$ then, for any w, $w \models (s_w, C^{\downarrow}M^{\downarrow}) \Downarrow_{\mathrm{error}}$.*
3. *If $(s, CM) \Uparrow$ then, for any w, $w \models (s_w, C^{\downarrow}M^{\downarrow}) \Uparrow$.*

These exhaust all possibilities for the big-step semantics of CM.

Proof: That these are all the possibilities is simply because CM is a high-level term. That the statements concerning these possibilities hold is an immediate consequence of Proposition 4.11. (Note that, for any value $V : \texttt{bool}$, if $V \searrow_w V'$ then $V' = V$.) ■

We can restate this in terms of a convenient notion of evaluation function. For any store s and term $M : \sigma$ of the instrumented high-level language, and so also of the high-level language, define their *behavior* $\mathrm{Eval}(M, s)$ by:

$$\mathrm{Eval}(M, s) = \begin{cases} (s', V) & \text{if } (s, M) \Longrightarrow (s', V) \\ \mathrm{error} & \text{if } (s, M) \Downarrow^{o}_{\mathrm{error}} \\ \Omega & \text{if } (s, M) \Uparrow \end{cases}$$

Here Ω and error are tokens indicating, respectively, nontermination and the raising of an error. Note that we do not distinguish between ordinary and instrumented errors when defining behavior.

Similarly, for any low-level term $M : \sigma$, memory m, and layout w define their *behavior* $\mathrm{Eval}_w(M, m)$ by:

$$\mathrm{Eval}_w(M, m) = \begin{cases} (m', V) & \text{if } w \models (m, M) \Longrightarrow (m', V) \\ \mathrm{error} & \text{if } w \models (m, M) \Downarrow^{o}_{\mathrm{error}} \\ \Omega & \text{if } w \models (m, M) \Uparrow \end{cases}$$

where $w \models (m, A) \Downarrow^{o}_{\mathrm{error}}$ is defined to hold if, and only if, either $w \models (m, A) \Downarrow_{\mathrm{error}}$ holds or $w \models (m, A) \Downarrow^{a}_{\mathrm{error}}$ does, for some a.

We write x_w to mean (s_w, M) when x is (s, M) and x when x is error or Ω. Proposition 4.12 then yields:

Corollary 4.13 *Let $M : \sigma$ be a high-level term and let $C : \sigma \to \texttt{bool}$ be a public high-level term. Then:*

$$\mathrm{Eval}(CM, s)_w = \mathrm{Eval}_w(C^{\downarrow}M^{\downarrow}, s_w)$$

for any store s and memory layout w.

For a converse, suppose now that $C : \sigma \to \mathtt{bool}$ is a public low-level term (so, as before, σ is `loc`-free). Then C is also a public instrumented high-level term of the same type, and we would would like to show that the public high-level term $C^\uparrow : \sigma \to \mathtt{bool}$ is an attacker of equal power. This will be true in a probabilistic sense:

Theorem 4.14 *Suppose that $M : \sigma$ is a high-level term and $C : \sigma \to \mathtt{bool}$ is a public low-level term. Then one of the following three mutually exclusive statements holds for any store s:*

- $\exists s', V.\, \forall w.\, w \models (s_w, CM^\downarrow) \Longrightarrow (s'_w, V) \wedge (s, C^\uparrow M) \Longrightarrow (s', V)$*,*
- $\mathrm{P}(w \models (s_w, CM^\downarrow) \Downarrow^o_{\text{error}}) \geq \delta_1 \wedge (s, C^\uparrow M) \Downarrow_{\text{error}}$*, or*
- $\forall w.\, w \models (s_w, CM^\downarrow) \Uparrow \wedge (s, C^\uparrow M) \Uparrow$*.*

Proof: First note that $(CM)^\uparrow = C^\uparrow M$ (as $M^\uparrow = M$) and that $(CM)^\downarrow = CM^\downarrow$ (as $C^\downarrow = C$).

The proof now proceeds by considering the big-step behavior of (s, CM). There are four mutually exclusive possibilities: we consider each of them in turn.

1. In the first case we have $(s, CM) \Longrightarrow (s', V)$ for some s' and V. We then have $(s, C^\uparrow M) \Longrightarrow (s', V)$, by part 1 of Proposition 4.6. We also have, for any w that $w \models (s_w, C^\downarrow M) \Longrightarrow (s'_w, V)$, using part 1 of Proposition 4.11.
2. In the second case we have $(s, CM) \Downarrow_{\text{error}}$, and so, arguing as before but now using the second parts of the propositions, $(s, C^\uparrow M) \Downarrow_{\text{error}}$ and, for any w, $w \models (s, C^\downarrow, M) \Downarrow_{\text{error}}$, and so $\mathrm{P}(w \models (s_w, CM^\downarrow) \Downarrow^o_{\text{error}}) = 1 \geq \delta_1$.
3. In the third case we have $(s, CM) \Downarrow^a_{\text{error}}$, for some $a \geq 0$ with $a \notin \mathrm{Ran}(w_p)$. So, again arguing as before, but now using the second and third parts of the propositions, respectively, we have that $(s, C^\uparrow M) \Downarrow^a_{\text{error}}$, and that, for any w with $a \notin \mathrm{Ran}(w)$, $w \models (s_w, C^\downarrow M) \Downarrow_{\text{error}}$. It follows that:
$$\mathrm{P}(w \models (s_w, CM^\downarrow) \Downarrow^a_{\text{error}}) \geq \mathrm{P}(w \# \{a\}) \geq \delta_1$$
4. The fourth case is similar to the first case, but uses the third and fourth parts of the respective propositions.

■

Note the proof strategy used for the first half of the theorem: the behavior of a term CM of the instrumented high-level language is used to coordinate the behaviors of terms $C^\uparrow M$ and $CM^\downarrow$ of the high- and low-level languages, respectively.

Using the evaluation function we obtain a weaker but more memorable statement:

Corollary 4.15 *Suppose that $M : \sigma$ is a high-level term and $C : \sigma \to \mathtt{bool}$ is a public low-level term. Then, for any store s, we have:*

$$\mathrm{P}(\mathrm{Eval}(C^\uparrow M, s)_w = \mathrm{Eval}_w(CM^\downarrow, s_w)) \geq \delta_1$$

4.5 Equivalences

There is a natural relation of *public (contextual) operational (high-level) equivalence*, refining the standard relation of operational equivalence. It is defined by setting, for any two high-level terms, M, N of type σ:

$$M \approx_{h,p} N \iff \forall C\!:\!\sigma \rightarrow \texttt{bool}.\, CM \sim_{h,p} CN$$

where the quantification over C ranges over public high-level terms, and where, for high-level terms $A, B\!:\!\texttt{bool}$, we define:

$$A \sim_{h,p} B \iff \forall s.\, \mathrm{Eval}(A, s) =_p \mathrm{Eval}(B, s)$$

where $x =_p y$ holds if, and only if, either x and y have the forms (s, V) and (s', V'), and $s \upharpoonright \mathrm{PubLoc} = s' \upharpoonright \mathrm{PubLoc}$ and $V = V'$, or else $x = y = \mathrm{error}$, or else $x = y = \Omega$.

At low-level, for any low-level terms $A, B\!:\!\texttt{bool}$ say that $A \sim_{l,p} B$ holds if, and only if, for every store s one of the following three possibilities holds:

- $\exists s', s'', V.\, \forall w.\, w \models (s_w, A) \Longrightarrow (s'_w, V) \,\wedge\, w \models (s_w, B) \Longrightarrow (s''_{w'}, V) \,\wedge$ $s' \upharpoonright \mathrm{PubLoc} = s'' \upharpoonright \mathrm{PubLoc}$,
- $\mathrm{P}(w \models (s_w, A) \Downarrow^o_{\mathrm{error}}) \geq \delta_1 \,\wedge\, \mathrm{P}(w \models (s_w, B) \Downarrow^o_{\mathrm{error}}) \geq \delta_1$, or
- $\forall w.\, w \models (s_w, A) \Uparrow \wedge\, w \models (s_w, B) \Uparrow$.

This relation is a partial equivalence. (Reflexivity fails, in general.) Note that we quantify over memories that are layouts of stores, not all memories. Now we define *public (contextual) operational (low-level) partial equivalence*, by putting, for low-level terms M, N of type σ:

$$M \approx_{l,p} N \iff \forall C\!:\!\sigma \rightarrow \texttt{bool}.\, CM \sim_{l,p} CN$$

where the C are restricted to be public low-level terms.

Theorem 4.16 *Let $M, N : \sigma$ be high-level terms. Then, if σ is* `loc`*-free and $M \approx_{h,p} N$, then $M^{\downarrow} \approx_{l,p} N^{\downarrow}$. The converse holds without restriction on σ if $\delta_1 > 0$.*

Proof: In one direction, we assume that σ is `loc`-free, so that $\sigma^{\downarrow} = \sigma$, and $M \approx_{h,p} N$, and then consider a public low-level term $C\!:\!\sigma \rightarrow \texttt{bool}$ in order to show $CM^{\downarrow} \sim_{l,p} CN^{\downarrow}$. Choose a store s. From the assumption that $M \approx_{h,p} N$ we then obtain $\mathrm{Eval}(C^{\uparrow}M, s) =_p \mathrm{Eval}(C^{\uparrow}N, s)$, and three cases arise.

In the first case we have that $(s, C^{\uparrow}M) \Rightarrow (s', V)$, $(s, C^{\uparrow}N) \Rightarrow (s'', V)$, and $s' \upharpoonright \mathrm{PubLoc} = s'' \upharpoonright \mathrm{PubLoc}$, for some s', s'' and V. Applying Theorem 4.14, we obtain that $w \models (s_w, CM^{\downarrow}) \Rightarrow (s'_w, V)$ and $w \models (s_w, CM^{\downarrow}) \Rightarrow (s''_w, V)$, for any w, which concludes this case.

In the second case we have $(s, C^{\uparrow}M) \Downarrow_{\mathrm{error}}$ and $(s, C^{\uparrow}N) \Downarrow_{\mathrm{error}}$. We again apply Theorem 4.14, and obtain that $\mathrm{P}(w \models (s_w, C^{\uparrow}M) \Downarrow^o_{\mathrm{error}}) \geq \delta_1$ and that $\mathrm{P}(w \models (s_w, C^{\uparrow}N) \Downarrow^o_{\mathrm{error}}) \geq \delta_1$. The third case is similar to the first two.

For the converse, we assume $M^{\downarrow} \approx_{l,p} N^{\downarrow}$ and consider a public high-level term $C : \sigma \rightarrow \texttt{bool}$ in order to show that $\mathrm{Eval}(CM, s) =_p \mathrm{Eval}(CN, s)$, for any given store s. We know that $C^{\downarrow}M^{\downarrow} \sim_{l,p} C^{\downarrow}N^{\downarrow}$, and, by Corollary 4.13, that $\mathrm{Eval}(CM, s)_w = \mathrm{Eval}_w(C^{\downarrow}M^{\downarrow}, s_w)$ and $\mathrm{Eval}(CN, s)_w = \mathrm{Eval}_w(C^{\downarrow}N^{\downarrow}, s_w)$, for all w.

The definition of $\sim_{l,p}$ then yields three cases, of which the first and third are immediate. For the second, as $\delta_1 > 0$, there are w_1 and w_2 such that $w_1 \models (s_{w_1}, C^{\downarrow}M^{\downarrow}) \Downarrow^o_{\mathrm{error}}$ and $w_2 \models (s_{w_2}, C^{\downarrow}N^{\downarrow}) \Downarrow^o_{\mathrm{error}}$, and the conclusion follows. ■

5 The recoverable-error model

In this model, an erroneous low-level memory access gives rise to a recoverable error, and a local recovery mechanism is available for handling such errors. Sections 5.1 and 5.2 present the high-level language and the low-level language, respectively. As in the fatal-error case, we employ an instrumented high-level language in order to mediate between these two languages. This language, defined in Section 5.3, has facilities for memory access at both location and natural-number types, with an instrumented operational semantics that records natural-number memory accesses.

With these tools, we prove our main results for the recoverable-error model in Sections 5.4 and 5.5, to the effect that high- and low-level attackers have essentially equal power, modulo the translation from the high-level to the low-level language, and that this translation preserves and reflects suitable notions of contextual public equivalence. Thus, our main results are analogous to those for the fatal-error model. However, the recoverable-error model requires a more delicate analysis of probabilities, and also upper bounds on numbers of memory accesses.

5.1 The high-level language

The high-level language employs the abstract notion of location. The basic types are `nat` and `loc`, and the constants are the arithmetic constants, together with constants for accessing and updating locations:

$$\begin{gathered} l_{\texttt{loc}} : \texttt{loc} \quad (l \in \mathrm{Loc}) \\ !_{\texttt{loc}} : \texttt{loc} \rightarrow \texttt{nat} \\ :=_{\texttt{loc}} : \texttt{loc} \times \texttt{nat} \rightarrow \texttt{com} \end{gathered}$$

Note that, unlike in the fatal-error case, there is no constant for raising an error. All the constants are values, and as well as the redexes specified by the general framework, there are the following two:

$$!_{\texttt{loc}} V \qquad V :=_{\texttt{loc}} V$$

For the semantics of the high-level language we define a *configuration* to be a pair (s, M) with s a store and M a well-typed term. The semantics then consists of a transition relation: $(s, M) \longrightarrow (s', M')$ which is obtained from the special case of redexes:

$$\frac{(s, R) \longrightarrow (s', M')}{(s, E[R]) \longrightarrow (s', E[M'])}$$

For redexes we take the transitions to be given by:

$$\begin{array}{l} (s, !_{\mathtt{loc}} l_{\mathtt{loc}}) \longrightarrow (s, n) \quad (s(l) = n) \\ (s, l_{\mathtt{loc}} :=_{\mathtt{loc}} n) \longrightarrow (s[l \mapsto n], \mathtt{skip}) \end{array}$$

and the rule:

$$\frac{R \longrightarrow M'}{(s, R) \longrightarrow (s, M')}$$

The small-step subject-reduction theorem is:

Lemma 5.1 *For any configuration (s, M), with $M : \sigma$, one of the following two mutually exclusive statements holds:*

- *M is a value, or*
- *$(s, M) \longrightarrow (s', M')$ for some uniquely determined s' and $M' : \sigma$.*

The big-step semantics is defined by:

$$\begin{array}{lcl} (s, M) \Longrightarrow (s', V) & \iff & (s, M) \rightarrow^{*} (s', V) \\ (s, M) \Uparrow & \iff & \forall n. \exists s', M'. (s, M) \rightarrow^{n} (s', M') \end{array}$$

The relation and property are mutually exclusive. Lemma 5.1 yields the big-step subject-reduction theorem:

Lemma 5.2 *For any configuration (s, M), with $M : \sigma$, one of the following two mutually exclusive statements holds:*

- *$(s, M) \Longrightarrow (s', V)$ for a unique s' and $V : \sigma$, or*
- *$(s, M) \Uparrow$.*

5.2 The low-level language

In the low-level language all memory accesses are made via natural numbers, just as in the fatal-error case. Consequently we take the only basic type to be `nat`. As well as the arithmetic constants, the low-level language has memory-access constants:

$$\begin{array}{c} l_{\mathtt{nat}} : \mathtt{nat} \quad (l \in \mathrm{Loc}) \\ !_{\mathtt{nat}} : \mathtt{nat} \rightarrow \mathtt{nat}^{e} \\ :=_{\mathtt{nat}} : \mathtt{nat} \times \mathtt{nat} \rightarrow \mathtt{com}^{e} \end{array}$$

where, for any type σ, we write σ^e for $\sigma + \mathtt{unit}$. Note that there are constants for all the locations, not just the public ones. We take $!_{\mathtt{nat}}$ and $:=_{\mathtt{nat}}$ to be values, and $l_{\texttt{nat}}$ to be a redex, for each $l \in \mathrm{Loc}$. The other redexes are those specified by the general framework, together with:

$$!_{\texttt{nat}}V \qquad V :=_{\texttt{nat}} V$$

Configurations in the low-level operational semantics are pairs (m, M) of a memory m and a well-typed term M. The semantics is defined relative to a choice of a memory layout. It consists of two transition relations, both relative to the memory layout chosen:

$$\begin{array}{c} w \models (m, M) \longrightarrow (m', M') \\ w \models (m, M) \xrightarrow{a} (m', M') \quad (a \in \mathbb{N}) \end{array}$$

These are obtained from the special case of redexes in the usual way:

$$\frac{w \models (m, R) \longrightarrow (m', M')}{w \models (m, E[R]) \longrightarrow (m', E[M'])}$$

$$\frac{w \models (m, R) \xrightarrow{a} (m', M')}{w \models (m, E[R]) \xrightarrow{a} (m', E[M'])}$$

For the redexes we take the transition relations to be given by the rule:

$$\frac{R \longrightarrow M'}{w \models (m, R) \longrightarrow (m, M')}$$

together with:

$$w \models (m, l_{\texttt{nat}}) \longrightarrow (m, w(l)) \;\; (l \in \mathrm{Loc})$$

and:

$$\begin{array}{lll} w \models (m, !_{\texttt{nat}}a) \longrightarrow (m, \texttt{inl}\, n) & & (\text{if } a \in \{0, \ldots, c\} \text{ and } m(a) = n) \\ w \models (m, !_{\texttt{nat}}a) \xrightarrow{a} (m, \texttt{error}) & & (\text{if } a \notin \{0, \ldots, c\} \text{ or } m(a) = *) \end{array}$$

and:

$$\begin{array}{ll} w \models (m, a :=_{\texttt{nat}} n) \longrightarrow (m[a \mapsto n], \texttt{inl}\,\texttt{skip}) & (\text{if } a \in \{0, \ldots, c\} \text{ and } m(a) \neq *) \\ w \models (m, a :=_{\texttt{nat}} n) \xrightarrow{a} (m, \texttt{error}) & (\text{if } a \notin \{0, \ldots, c\} \text{ or } m(a) = * \,) \end{array}$$

where we write `error` for $\texttt{inr}\, * : \texttt{nat}^e$. Notice that when an erroneous access is made then a non-fatal error arises, modeled using the term `error`.

The small-step subject-reduction theorem is:

Lemma 5.3 *For any memory layout w and configuration (m, M), with $M : \sigma$, one of the following three mutually exclusive statements holds:*

- *M is a value,*
- *$w \models (m, M) \longrightarrow (m', M')$ for some uniquely determined m' and $M' : \sigma$, and if m has the form s_w, so does m', or*

- *$w \models (m, M) \xrightarrow{a} (m', M')$ for some uniquely determined a, m', and $M' \!:\! \sigma$, and if m has the form s_w, so does m'.*

For the low-level big-step semantics one needs to keep track of sets of erroneous accesses. Accordingly, for $A \subseteq \mathbb{N}$, define

$$w \models (m, M) \xrightarrow{A} (m', M')$$

to hold if either $A = \emptyset$ and $w \models (m, M) \longrightarrow (m', M')$, or else $A = \{a\}$ and $w \models (m, M) \xrightarrow{a} (m', M')$. Then define

$$w \models (m, M) \overset{A}{\Longrightarrow} (m', M')$$

to hold if there is a sequence:

$$(m, M) = (m_0, M_0), \ldots, (m_n, M_n) = (m', M')$$

and sets $A_i \subseteq \mathbb{N}$, for $i = 1, n$, such that

$$w \models (m_{i-1}, M_{i-1}) \xrightarrow{A_i} (m_i, M_i)$$

for $i = 1, n$, and $A = \bigcup_{i=1}^{n} A_i$. Finally, define $(m, M) \Uparrow^A$ to hold if there is an infinite sequence:

$$(m, M) = (m_0, M_0), \ldots, (m_i, M_i), \ldots$$

and sets $A_i \subseteq \mathbb{N}$, for $i \geq 1$, such that

$$w \models (m_{i-1}, M_{i-1}) \xrightarrow{A_i} (m_i, M_i)$$

for $i \geq 1$, and $A = \bigcup_{i=1}^{\infty} A_i$.

Lemma 5.3 then yields the big-step subject-reduction theorem:

Lemma 5.4 *For any configuration (m, M), with $M : \sigma$, one of the following two mutually exclusive statements holds:*

- *$w \models (m, M) \overset{A}{\Longrightarrow} (m', V)$ for a unique m', $V \!:\! \sigma$, and $A \subseteq \mathbb{N}$, and if m has the form s_w, so does m', or*
- *$w \models (m, M) \Uparrow^A$ for a unique $A \subseteq \mathbb{N}$.*

5.3 The instrumented high-level language

In order to relate the high-level semantics uniformly to the low-level language we instrument it by adding some constants for accessing the store at type `nat`. In the instrumented high-level language, accesses to the natural-number addresses of private locations will simply result in errors. In contrast, these accesses may work in the low-level language.

The instrumented high-level language has the same basic types as the high-level language and its constants are those of the high-level language together with:

$$\begin{array}{c} l_{\mathtt{nat}}\!:\!\mathtt{nat} \quad (l \in \mathrm{PubLoc}) \\ !_{\mathtt{nat}}\!:\!\mathtt{nat} \to \mathtt{nat}^e \\ :=_{\mathtt{nat}}\!:\!\mathtt{nat} \times \mathtt{nat} \to \mathtt{com}^e \end{array}$$

We take $l_{\mathtt{nat}}$ to be a redex (for $l \in \mathrm{PubLoc}$), and $!_{\mathtt{nat}}$ and $:=_{\mathtt{nat}}$ to be values, and classify the other constants as in the case of the high-level language. The other redexes are those specified by the general framework, together with the following ones:

$$\begin{array}{cc} !_{\mathtt{nat}}V & V :=_{\mathtt{nat}} V \\ !_{\mathtt{loc}}V & V :=_{\mathtt{loc}} V \end{array}$$

the latter two kinds being inherited from the high-level language.

For the operational semantics, configurations are defined as for the high-level language, but we add an instrumented transition relation:

$$(s, M) \xrightarrow{a} (s', M') \qquad (a \in \mathbb{N})$$

We then proceed as for the high-level language, adding a rule for the instrumented transition relation:

$$\frac{(s, R) \xrightarrow{a} (s', M')}{(s, E[R]) \xrightarrow{a} (s', E[M'])}$$

together with:

$$(s, l_{\mathtt{nat}}) \longrightarrow (s, w_p(l)) \quad (l \in \mathrm{PubLoc})$$

and:

$$\begin{array}{ll} (s, !_{\mathtt{nat}}a) \longrightarrow (s, \mathtt{inl}\, s(l)) & (l \in \mathrm{PubLoc}, a = w_p(l)) \\ (s, a :=_{\mathtt{nat}} n) \longrightarrow (s[l \mapsto n], \mathtt{inl}\, \mathtt{skip}) & (l \in \mathrm{PubLoc}, a = w_p(l)) \\ (s, !_{\mathtt{nat}}a) \xrightarrow{a} (s, \mathtt{error}) & (a \notin \mathrm{Ran}(w_p)) \\ (s, a :=_{\mathtt{nat}} n) \xrightarrow{a} (s, \mathtt{error}) & (a \notin \mathrm{Ran}(w_p)) \end{array}$$

For the analogue to Lemma 5.1, one adds one more possibility to the list of mutually exclusive possibilities:

- $(s, M) \xrightarrow{a} (s', M')$ for some unique a, s', and M'.

For the big-step semantics one again needs to keep track of sets of non-public memory accesses. Accordingly define $(s, M) \xrightarrow{A} (s', M')$, where $A \subseteq \mathbb{N}$, to hold if either

$$A = \emptyset \text{ and } (s, M) \longrightarrow (s', M')$$

or else

$$A = \{a\} \text{ and } (s, M) \xrightarrow{a} (s', M')$$

Then define $(s, M) \overset{A}{\Longrightarrow} (s', M')$, where $A \subseteq \mathbb{N}$, to hold if there is a sequence:

$$(s, M) = (s_0, M_0) \xrightarrow{A_1} \dots \xrightarrow{A_n} (s_n, M_n) = (s', M')$$

with $n \geq 0$, such that $A = \bigcup_{i=1}^{n} A_i$ and define $(s, M) \Uparrow^A$, where $A \subseteq \mathbb{N}$, to hold if there is an infinite sequence:

$$(s, M) = (s_0, M_0) \xrightarrow{A_1} \dots \xrightarrow{A_i} (s_i, M_i) \xrightarrow{A_{i+1}} \dots$$

such that $A = \bigcup_{i=1}^{\infty} A_i$. The big-step subject-reduction theorem is then:

Lemma 5.5 *For any configuration (s, M), with $M : \sigma$, one of the following two mutually exclusive statements holds:*

- *$(s, M) \overset{A}{\Longrightarrow} (s', V)$ for a unique s', $V : \sigma$, and $A \subseteq \mathbb{N}$, or*
- *$(s, M) \Uparrow^A$ for a unique $A \subseteq \mathbb{N}$.*

Note that the small-step semantics of the instrumented high-level language is a conservative extension of that of the high-level language. That is, a transition $(s, M) \longrightarrow (s', M')$ holds in the high-level language if, and only if, it does in the instrumented high-level language. For the big-step semantics, we have, for any terms M, M' of the high-level language:

$$\begin{array}{lcl} (s, M) \Longrightarrow (s', M') & \Longleftrightarrow & (s, M) \overset{\emptyset}{\Longrightarrow} (s', M') \\ (s, M) \Uparrow & \Longleftrightarrow & (s, M) \Uparrow^{\emptyset} \end{array}$$

5.3.1 Translating instrumented high-level to high-level

Every term $M : \sigma$ of the instrumented high-level language can be translated to a term $M^{\uparrow} : \sigma$ of the high-level language. First, as in the fatal-error case, we need a function to convert addresses of public locations to the locations themselves. Define the high-level term

$$G_\sigma : \mathtt{nat} \to (\mathtt{loc} \to \sigma) \to (\mathtt{unit} \to \sigma) \to \sigma$$

to be:

$$\begin{array}{l} \lambda x : \mathtt{nat}. \lambda f : \mathtt{loc} \to \sigma. \lambda g : \mathtt{unit} \to \sigma. \\ \quad \mathtt{if}\ x = a_1\ \mathtt{then} f((l^{(1)})_{\mathtt{loc}}) \\ \quad \mathtt{elseif}\ x = a_2\ \mathtt{then} f((l^{(2)})_{\mathtt{loc}}) \\ \qquad \vdots \\ \quad \mathtt{elseif}\ x = a_p\ \mathtt{then} f((l^{(p)})_{\mathtt{loc}}) \\ \quad \mathtt{else}\ g(*) \end{array}$$

where we make use of the enumeration of PubLoc and the definition of the a_i given in Section 4.3.1. Then the translation replaces the additional constants as shown in Figure 1, leaving the others fixed.

The translation is correct in the following sense:

$$\begin{array}{lcl}
l^{\uparrow}_{\mathtt{nat}} & = & w_p(l) \quad (l \in \mathrm{PubLoc}) \\
!^{\uparrow}_{\mathtt{nat}} & = & \lambda x\!:\!\mathtt{nat}.\, Gx(\lambda y\!:\!\mathtt{loc}.\,\mathtt{inl}\,(!_{\mathtt{loc}}y))(\lambda y\!:\!\mathtt{unit}.\,\mathtt{error}) \\
:=^{\uparrow}_{\mathtt{nat}} & = & \lambda x\!:\!\mathtt{nat}\times\mathtt{nat}.\, G(\mathtt{fst}\,x)(\lambda y\!:\!\mathtt{loc}.\,\mathtt{inl}\,(y :=_{\mathtt{loc}} \mathtt{snd}\,x))(\lambda y\!:\!\mathtt{unit}.\,\mathtt{error})
\end{array}$$

Figure 1: Replacement of constants for translation from instrumented high-level to high-level languages.

Lemma 5.6 *Let M be a well-typed term of the instrumented high-level language. Then:*

1. *If M is a value then so is $M^{\uparrow}$.*

2. *If $(s, M) \xrightarrow{A} (s', M')$ then $(s, M^{\uparrow}) \longrightarrow^{*} (s', (M')^{\uparrow})$.*

Proof: Part 1 follows by inspection. For part 2, one shows first that, for any redex R, if $(s, R) \xrightarrow{A} (s', M')$ then $(s, R^{\uparrow}) \longrightarrow^{*} (s', (M')^{\uparrow})$. One shows next that if E is an evaluation context, then $E^{\uparrow}$ is too (taking $[-]^{\uparrow} = [-]$, etc.) and that $E[M]^{\uparrow} = E^{\uparrow}[M^{\uparrow}]$. Part 2 then follows. ■

In terms of big-step relations and properties we have:

Proposition 5.7 *Let M be a well-typed term of the instrumented high-level language. Then:*

1. *If M is a value then so is $M^{\uparrow}$.*

2. *If $(s, M) \overset{A}{\Longrightarrow} (s', V)$ then $(s, M^{\uparrow}) \Longrightarrow (s', V^{\uparrow})$.*

3. *If $(s, M) \Uparrow^{A}$ then $(s, M^{\uparrow}) \Uparrow$.*

A small variation on this translation will also prove useful. For any $a \in \mathbb{N}$ define a translation $M^{\uparrow}_{a}$ by the following alternative replacement of the additional constants.

$$\begin{array}{lcl}
(l_{\mathtt{nat}})^{\uparrow}_{a} & = & (l_{\mathtt{nat}})^{\uparrow} \\
(!_{\mathtt{nat}})^{\uparrow}_{a} & = & \lambda x\!:\!\mathtt{nat}.\ \mathtt{if}\ x = a\ \mathtt{then}\ \Omega\ \mathtt{else}\ !^{\uparrow}_{\mathtt{nat}}x \\
(:=_{\mathtt{nat}})^{\uparrow}_{a} & = & \lambda x\!:\!\mathtt{nat}\times\mathtt{nat}.\ \mathtt{if}\ \mathtt{fst}\,x = a\ \mathtt{then}\ \Omega\ \mathtt{else}\ :=^{\uparrow}_{\mathtt{nat}}\ x
\end{array}$$

Proposition 5.8 *Let M be a well-typed term of the instrumented high-level language. Then, if $a \notin \mathrm{Ran}(w_p)$:*

1. *If M is a value then so is $M^{\uparrow}_{a}$.*

2. *If $(s, M) \overset{A}{\Longrightarrow} (s', V)$ then $(s, M^{\uparrow}_{a}) \Longrightarrow (s', V^{\uparrow}_{a})$, if $a \notin A$.*

3. *If $(s, M) \overset{A}{\Longrightarrow} (s', V)$ then $(s, M^{\uparrow}_{a}) \Uparrow$, if $a \in A$.*

4. *If $(s, M) \Uparrow^{A}$ then $(s, M^{\uparrow}_{a}) \Uparrow$.*

5.3.2 Translating instrumented high-level to low-level

We can translate types σ and terms $M:\sigma$ of the instrumented high-level language to types $\sigma^{\downarrow}$ and terms $M^{\downarrow}:\sigma^{\downarrow}$ of the low-level language. We obtain the translation $\sigma^{\downarrow}$ of a type σ by replacing all occurrences of `loc` by `nat`. For terms we replace each occurrence of a type σ by one of $\sigma^{\downarrow}$ and we replace the missing constants as follows:

$$\begin{array}{lcl}
(l_{\mathtt{loc}})^{\downarrow} & = & l_{\mathtt{nat}} \\
(!_{\mathtt{loc}})^{\downarrow} & = & \lambda x\!:\!\mathtt{nat}.\,\mathtt{cases}\,!_{\mathtt{nat}}x\,\mathtt{inl}\,y.\,y\,\mathtt{inr}\,z.\,0 \\
(:=_{\mathtt{loc}})^{\downarrow} & = & \lambda x\!:\!\mathtt{nat}\times\mathtt{nat}.\,\mathtt{cases}\ :=_{\mathtt{nat}}x\,\mathtt{inl}\,y.\,y\,\mathtt{inr}\,z.\,\mathtt{skip}
\end{array}$$

and take the translation to act as the identity on the other constants, viz.: $l_{\mathtt{nat}}$ ($l \in \mathrm{PubLoc}$), $!_{\mathtt{nat}}$ and $:=_{\mathtt{loc}}$.

The translation is correct with respect to the low-level semantics, in the sense, roughly, that $M^{\downarrow}$ simulates M. As before we employ a simulation relation $M \searrow_w N$ between terms of the instrumented high-level language and terms of the low-level language, parameterized on a memory layout w.

We take this relation to be the least relation between terms of the instrumented high-level language and terms of the the low-level language which includes:

$$c \searrow_w c^{\downarrow} \qquad l_{\mathtt{loc}} \searrow_w w(l)$$

and which is closed under the other language constructs. For any term M of the instrumented high-level language we have $M \searrow_w M^{\downarrow}$; further, if $M:\sigma$ and $M \searrow_w N$ then $N:\sigma^{\downarrow}$.

We now prove a series of lemmas, leading to our main simulation lemma. The first lemma concerns values.

Lemma 5.9 *Suppose that $V \searrow_w N$ for a well-typed value V. Then for some V' with $V \searrow_w V'$ we have, for any m, $w \models (m, N) \longrightarrow^{*} (m, V')$.*

The second lemma concerns redexes.

Lemma 5.10 1. *Suppose that $R \searrow_w N$ and that $(s, R) \longrightarrow (s', M')$. Then we have $w \models (s_w, N) \longrightarrow^{*} (s'_w, N')$ for some N' with $M' \searrow_w N'$.*

2. *Suppose that $R \searrow_w N$ and that $(s, R) \xrightarrow{a} (s', M')$ for some a not in $\mathrm{Ran}(w)\backslash\mathrm{Ran}(w_p)$. Then we have $w \models (s_w, N) \xrightarrow{a} (s'_w, N')$ for some N' with $M' \searrow_w N'$.*

The third lemma concerns evaluation contexts. The simulation relation is extended in an evident way to evaluation contexts, taking $[-] \searrow_w [-]$, etc. One easily sees that if $E \searrow_w E'$ and $M \searrow_w N$ then $E[M] \searrow_w E'[N]$.

Lemma 5.11 *Suppose that $E[R] \searrow_w N$. Then N has the form $E'[N_1]$ where $E \searrow_w E'$ and $R \searrow_w N_1$.*

We then have the small-step simulation lemma:

Lemma 5.12 *Suppose that $M \searrow_w N$ for well-typed terms M of the instrumented high-level language and N of the low-level language. Then:*

1. *If M is a value, then there is a V', with $M \searrow_w V'$, such that, for any m, $w \models (m, N) \longrightarrow^* (m, V')$.*
2. *If $(s, M) \longrightarrow (s', M')$, then there is an N' with $M' \searrow_w N'$ such that $w \models (s_w, N) \longrightarrow^* (s'_w, N')$.*
3. *If $(s, M) \xrightarrow{a} (s', M')$ and a is not in $\mathrm{Ran}(w) \backslash \mathrm{Ran}(w_p)$, then we have $w \models (s_w, N) \xrightarrow{a} (s'_w, N')$ for some N' such that $M' \searrow_w N'$.*

The third case is particularly important as it enables one to find the memory access largely independently of the memory layout. In terms of big-step relations and properties we have:

Proposition 5.13 *Suppose that $M \searrow_w N$ for well-typed terms M of the instrumented high-level language and N of the low-level language. Then:*

1. *If $(s, M) \overset{A}{\Longrightarrow} (s', V)$, then, if $w \# A$, there is a V' with $V \searrow_w V'$ such that $w \models (s_w, N) \overset{A}{\Longrightarrow} (s'_w, V')$.*
2. *If $(s, M) \Uparrow^A$ then, if $w \# A$, $w \models (s_w, N) \Uparrow^A$.*

5.4 High- and low-level attackers

We are now in a position to formulate our theorems for the recoverable-error case. Much as in the fatal-error case, we wish to show that a high-level term $M : \sigma$ is as secure as its low-level counterpart $M^{\downarrow} : \sigma^{\downarrow}$. We will prove that this holds if σ is `loc`-free, i.e., if $\sigma^{\downarrow} = \sigma$. (As in the fatal-error case, it does not hold more generally.)

Say that an instrumented high-level term (low-level term) is *public* if it contains no occurrence of any $l_{\texttt{loc}}$ (respectively $l_{\texttt{nat}}$) with $l \in \mathrm{PriLoc}$. We would again like to show that attackers gain no advantage by attacking at low-level rather than at high-level. As before, they certainly lose none since, for any public high-level term $C : \sigma \rightarrow$ `bool`, the low-level term $C^{\downarrow}$ is of equal attacking power. The following proposition follows immediately from Proposition 5.13:

Proposition 5.14 *Let $M : \sigma$ be a high-level term and let $C : \sigma \rightarrow$ `bool` be a public high-level term. Then:*

1. *If $(s, CM) \Longrightarrow (s', V)$ then, for any w, $w \models (s_w, C^{\downarrow} M^{\downarrow}) \overset{\emptyset}{\Longrightarrow} (s'_w, V)$.*
2. *If $(s, CM) \Uparrow$ then, for any w, $w \models (s_w, C^{\downarrow} M^{\downarrow}) \Uparrow^{\emptyset}$.*

These exhaust all the possibilities for the big-step semantics of CM.

We restate this using evaluation functions. For any store s and term $M:\sigma$ of the instrumented high-level language (and so also any term of the high-level language) define their *behavior* $\mathrm{Eval}(M,s)$ by:

$$\mathrm{Eval}(M,s) = \begin{cases} (s',V) & \text{if } (s,M) \overset{A}{\Longrightarrow} (s',V) \\ \Omega & \text{if } (s,M) \Uparrow^A \end{cases}$$

Here Ω is a token that indicates nontermination. Note that we forget the A, regarding that as part of the instrumentation rather than the actual behavior. However, it also proves useful to define $\mathrm{Acc}(M,s)$ to be $A \cap \{0,\ldots,c\}$ when $(s,M) \overset{A}{\Longrightarrow} (s',V)$ or $(s,M) \Uparrow^A$; $\mathrm{Acc}(M,s)$ records the accesses made to non-public addresses.

Similarly, for any low-level term $M:\sigma$, memory m, and layout w define their *behavior* $\mathrm{Eval}_w(M,m)$ by:

$$\mathrm{Eval}_w(M,m) = \begin{cases} (m',V) & \text{if } w \models (m,M) \overset{A}{\Longrightarrow} (m',V) \\ \Omega & \text{if } w \models (m,M) \Uparrow^A \end{cases}$$

It also proves useful to define $\mathrm{Acc}_w(M,m)$ to be $(A \cap \{0,\ldots,c\}) \backslash \mathrm{Ran}(w_p)$ when $w \models (m,M) \overset{A}{\Longrightarrow} (m',V)$ or $w \models (m,M) \Uparrow^A$; $|\mathrm{Acc}_w(M,m)|$ measures the number of "relevant" memory accesses made by M, starting from m, meaning those erroneous accesses within memory bounds.

We write x_w to mean (s_w,M) when x is (s,M) and Ω when x is Ω.

Corollary 5.15 *Let $M:\sigma$ be a high-level term and let $C:\sigma \to \texttt{bool}$ be a public high-level term. Then:*

$$\mathrm{Eval}(CM,s)_w = \mathrm{Eval}_w(C^{\downarrow}M^{\downarrow}, s_w)$$

for any s and w.

For a converse, suppose now that $C:\sigma \to \texttt{bool}$ is a public low-level term (so σ is `loc`-free). Then C is also a public instrumented high-level term of the same type, and we would like to show that the public high-level term $C^{\uparrow}:\sigma \to \texttt{bool}$ is an attacker of equal power. This will be true in a probabilistic sense.

The following theorem gives a lower bound on the probability that high- and low-level semantics (for $C^{\uparrow}M$ and $CM^{\downarrow}$, respectively) coincide, where the layout w is allowed to vary according to its distribution and the store s is arbitrary. The theorem requires an assumption on the number b of erroneous accesses: without a bound on b, an attacker could explore all of memory. For small b, the high- and low-level semantics coincide the most, with probability close to 1 when c is sufficiently large.

Theorem 5.16 *Let $M:\sigma$ be a high-level term and let $C:\sigma \to \texttt{bool}$ be a public low-level term. Then, for any store s, and $0 \leq b \leq c - |\mathrm{Loc}|$, one of the following holds:*

1. $P(|\mathrm{Acc}_w(CM^{\downarrow}, s_w)| > b) \geq \delta_{b+1}$, *or*

2. $P(|\mathrm{Acc}_w(CM^{\downarrow}, s_w)| \leq b \ \wedge\ \mathrm{Eval}(C^{\uparrow}M, s)_w = \mathrm{Eval}_w(CM^{\downarrow}, s_w)) \geq \delta_b$.

These alternatives are mutually exclusive if $\delta_{b+1} > 1/2$.

Proof: Fix M, C, and s. The proof proceeds by cases on whether or not $|\mathrm{Acc}(CM, s)| \leq b$.

Suppose first that $|\mathrm{Acc}(CM, s)| \leq b$. Take a w such that $w \# \mathrm{Acc}(CM, s)$. Then by Proposition 5.13 we have that $\mathrm{Acc}(CM, s) = \mathrm{Acc}_w(CM^{\downarrow}, s_w)$ and $\mathrm{Eval}(CM, s)_w = \mathrm{Eval}_w(CM^{\downarrow}, s_w)$, as $CM \searrow_w (CM)^{\downarrow} = CM^{\downarrow}$. By Proposition 5.7 we also have that $\mathrm{Eval}(C^{\uparrow}M, s) = \mathrm{Eval}(CM, s)$.

We therefore have:

$$\begin{array}{rcl} \delta_b & \leq & P(w \# \mathrm{Acc}(CM, s)) \\ & \leq & P(|\mathrm{Acc}_w(CM^{\downarrow}, s_w)| \leq b \ \wedge\ \mathrm{Eval}(C^{\uparrow}M, s)_w = \mathrm{Eval}_w(CM^{\downarrow}, s_w)) \end{array}$$

which is the second alternative.

Otherwise we have $|\mathrm{Acc}(CM, s)| > b$. So, as $(s, CM) \overset{A}{\Longrightarrow} (s', M')$ for some s', M', and A with $\mathrm{Acc}(CM, s) = A \cap \{0, \ldots, c\}$, we have $(s, CM) \overset{A'}{\Longrightarrow} (s'', M'')$ for some s'', M'', and A', where, setting $A'' = A' \cap \{0, \ldots, c\}$, $|A''| = b + 1$.

Now, take a w such that $w \# A''$. Then, as $CM \searrow_w CM^{\downarrow}$, Lemma 5.12 implies that $w \models (s_w, CM^{\downarrow}) \overset{A'}{\Longrightarrow} (s''_w, N)$, for some N, and so $\mathrm{Acc}_w(CM^{\downarrow}, s_w) \supseteq A''$ and $|\mathrm{Acc}_w(CM^{\downarrow}, s_w)| > b$. We therefore have:

$$\begin{array}{rcl} \delta_{b+1} & = & P(w \# A'') \\ & \leq & P(|\mathrm{Acc}_w(CM^{\downarrow}, s_w)| > b) \end{array}$$

which is the first alternative. ■

We remark that, following its proof, the first of the alternatives of Theorem 5.16 holds if $|\mathrm{Acc}(CM, s)| > b$ and the second if $|\mathrm{Acc}(CM, s)| \leq b$. In the special case $b = 0$, the theorem implies that, for all s,

- either $P(|\mathrm{Acc}_w(CM^{\downarrow}, s_w)| > 0) \geq \delta_1$ or,
- for all w, $|\mathrm{Acc}_w(CM^{\downarrow}, s_w)| = 0$ and $\mathrm{Eval}(C^{\uparrow}M, s)_w = \mathrm{Eval}_w(CM^{\downarrow}, s_w)$.

In other words, either an erroneous access to memory is probable, with probability at least δ_1, or there is no such access and the high- and low-level semantics coincide.

It is natural to wonder if the probability bound δ_{b+1} could be improved to δ_b in Theorem 5.16. The reason for the δ_{b+1} bound is that $|\mathrm{Acc}_w(CM^{\downarrow}, s_w)|$ counts only erroneous accesses; what seems needed for a δ_b bound is a way of counting attacker guesses, including successful ones.

5.5 Equivalences

We define a relation of *(high level) public (contextual) operational equivalence*, refining the standard relation of operational equivalence. For any two high-level terms, M, N of type σ, set:

$$M \approx_{h,p} N \iff \forall C{:}\sigma \to \texttt{bool}.\, CM \sim_{h,p} CN$$

where the quantification over C ranges over public high-level terms, and where, for high-level terms $A, B{:}\texttt{bool}$, we define:

$$A \sim_{h,p} B \iff \forall s.\, \mathrm{Eval}(A, s) =_p \mathrm{Eval}(B, s)$$

where the relation $x =_p y$ holds if, and only if, either x and y have the forms (s, V) and (s', V'), and $s \restriction \mathrm{PubLoc} = s' \restriction \mathrm{PubLoc}$ and $V = V'$, or else $x = y = \Omega$. (As usual, if f is a function and S is a set then $f \restriction S$ is the restriction of f to S.)

In order to define a corresponding low-level relation, we first define a modified version of the low-level evaluation function that yields nontermination if there are more than b erroneous accesses. For any $b \geq 0$, set:

$$\mathrm{Eval}_w^b(M, m) = \begin{cases} \mathrm{Eval}_w(M, m) & \text{if } |\mathrm{Acc}_w(M, m)| \leq b \\ \Omega & \text{otherwise} \end{cases}$$

Next, for any b such that $0 \leq b \leq c - |\mathrm{Loc}|$ and $\delta_{b+1} > 1/2$ we define a relation $\sim_{l,p}^b$ between low-level terms $A, B{:}\texttt{bool}$, by taking $A \sim_{l,p}^b B$ to hold if, and only if, for every store s one of the following (mutually exclusive) alternatives holds:

- for some $s' \restriction \mathrm{PubLoc} = s'' \restriction \mathrm{PubLoc}$ and V,

  $$\mathrm{P}(\mathrm{Eval}_w^b(A, s_w) = (s'_w, V)) \geq \delta_b$$

  and

  $$\mathrm{P}(\mathrm{Eval}_w^b(B, s_w) = (s''_w, V)) \geq \delta_b$$

or

- $$\mathrm{P}(\mathrm{Eval}_w^b(A, s_w) = \Omega) \geq \delta_{b+1}$$

  and

  $$\mathrm{P}(\mathrm{Eval}_w^b(B, s_w) = \Omega) \geq \delta_{b+1}$$

Note that we quantify over memories that are layouts of stores, not all memories. This relation is a partial equivalence: symmetry is evident, and transitivity follows from the assumption that $\delta_{b+1} > 1/2$, which ensures that the two possibilities are mutually exclusive. (Reflexivity fails, in general.) Now we define *(low-level) public (contextual) operational partial equivalence*, by setting, for any two low-level terms M, N of type σ:

$$M \approx_{l,p}^b N \iff \forall C{:}\sigma \to \texttt{bool}.\, CM \sim_{l,p}^b CN$$

where the contexts C are restricted to be public low-level terms.

The following theorem says, roughly, that two programs are publicly equivalent in the high-level language if, and only if, their translations are publicly equivalent in the low-level language, with the caveat that the low-level equivalence is probabilistic and conditioned on a bound b on the number of erroneous accesses.

Theorem 5.17 *Let $M, N : \sigma$ be high-level terms. Then, assuming that σ is* `loc`*-free, $0 \leq b \leq c - |\mathrm{Loc}|$, and $\delta_{b+1} > 1/2$, we have:*

$$M \approx_{h,p} N \quad \textit{iff} \quad M^{\downarrow} \approx^{b}_{l,p} N^{\downarrow}$$

Proof: In one direction, we assume that $M \approx_{h,p} N$, and then consider a public low-level term $C : \sigma \rightarrow$ `bool` in order to show that $CM^{\downarrow} \sim^{b}_{l,p} CN^{\downarrow}$. Choose a store s. Applying Theorem 5.16 to M and N four cases arise.

1. In the first case we have:

$$P(|\mathrm{Acc}_w(CM^{\downarrow}, s_w)| > b) \geq \delta_{b+1}$$

   and

$$P(|\mathrm{Acc}_w(CN^{\downarrow}, s_w)| > b) \geq \delta_{b+1}$$

   But then

$$P(\mathrm{Eval}^{b}_{w}(CM^{\downarrow}, s_w) = \Omega) \geq \delta_{b+1}$$

   and

$$P(\mathrm{Eval}^{b}_{w}(CN^{\downarrow}, s_w) = \Omega) \geq \delta_{b+1}$$

   concluding this case.

2. In the second case we have:

$$P\left(\begin{array}{l} |\mathrm{Acc}_w(CM^{\downarrow}, s_w)| \leq b \quad \wedge \\ \mathrm{Eval}(C^{\uparrow}M, s)_w = \mathrm{Eval}_w(CM^{\downarrow}, s_w) \end{array} \right) \geq \delta_b$$

   and

$$P\left(\begin{array}{l} |\mathrm{Acc}_w(CN^{\downarrow}, s_w)| \leq b \quad \wedge \\ \mathrm{Eval}(C^{\uparrow}N, s)_w = \mathrm{Eval}_w(CN^{\downarrow}, s_w) \end{array} \right) \geq \delta_b$$

   By assumption we have

$$\mathrm{Eval}(C^{\uparrow}M, s) =_p \mathrm{Eval}(C^{\uparrow}N, s)$$

   so there are two subcases.

   (a) In the first, there are s', s'', and V such that $s' \lceil \mathrm{PubLoc} = s'' \lceil \mathrm{PubLoc}$, $\mathrm{Eval}(C^{\uparrow}M, s) = (s', V)$, and $\mathrm{Eval}(C^{\uparrow}N, s) = (s'', V)$. But then we have:

$$\mathrm{P}(\mathrm{Eval}^{b}_{w}(CM^{\downarrow}, s_w) = (s'_w, V)) \geq \delta_b$$

   and

$$\mathrm{P}(\mathrm{Eval}^{b}_{w}(CN^{\downarrow}, s_w) = (s''_w, V)) \geq \delta_b$$

   concluding this subcase.

   (b) In the second,

$$\mathrm{Eval}(C^{\uparrow}M, s) = \mathrm{Eval}(C^{\uparrow}N, s) = \Omega$$

   and we obtain:

$$P(\mathrm{Eval}^{b}_{w}(CM^{\downarrow}, s_w) = \Omega) \geq \delta_b$$

   and

$$P(\mathrm{Eval}^{b}_{w}(CN^{\downarrow}, s_w) = \Omega) \geq \delta_b$$

   concluding this subcase.

3. In the third case we have:

$$P(|\mathrm{Acc}_w(CM^{\downarrow}, s_w)| > b) \geq \delta_{b+1}$$

   and

$$P\left(\begin{array}{c} |\mathrm{Acc}_w(CN^{\downarrow}, s_w)| \leq b \quad \wedge \\ \mathrm{Eval}(C^{\uparrow}N, s)_w = \mathrm{Eval}_w(CN^{\downarrow}, s_w) \end{array}\right) \geq \delta_b$$

   There are again two subcases.

   (a) In the first, $\mathrm{Eval}(C^{\uparrow}N, s)$ has the form (s', V) for some s' and V. So $(s, CN) \overset{A'}{\Longrightarrow} (s', V)$ for some A' with $A' \cap \{0, \ldots, c\} = \mathrm{Acc}(CN, s)$, as otherwise (i.e., if $(s, CN) \Uparrow^{A''}$, for some A''), by Proposition 5.7 we would have a contradiction with the form of $\mathrm{Eval}(C^{\uparrow}N, s)$.

   By the remark after Theorem 5.16, and since the alternatives there are mutually exclusive, we have

$$|\mathrm{Acc}(CM, s)| > b \geq |\mathrm{Acc}(CN, s)|$$

   So $\mathrm{Acc}(CM, s) \backslash \mathrm{Acc}(CN, s)$ is non-empty, and we choose an element a of it; note that $a \notin \mathrm{Ran}(w_p)$.

   Then, on the one hand, $(s, CN) \overset{A'}{\Longrightarrow} (s', V)$, so, as $(CN)^{\uparrow}_{a} = C^{\uparrow}_{a}N$, by Proposition 5.8.2 we have $(s, C^{\uparrow}_{a}N) \Rightarrow (s', V)$. On the other hand, by parts 3 and 4 of Proposition 5.8 we have $(s, C^{\uparrow}_{a}M) \Uparrow$. So we obtain a contradiction with the assumption that $M \approx_{h,p} N$.

   (b) In the second, $\mathrm{Eval}(C^{\uparrow}N, s) = \Omega$. But then we have

$$P(\mathrm{Eval}^{b}_{w}(CM^{\downarrow}, s_w) = \Omega) \geq \delta_{b+1}$$

   and

$$P(\mathrm{Eval}^{b}_{w}(CN^{\downarrow}, s_w) = \Omega) \geq \delta_b \geq \delta_{b+1}$$

   concluding this subcase.

4. The fourth case is similar to the third.

In the other direction, assume that $M^{\downarrow} \approx^b_{l,p} N^{\downarrow}$ and then consider a public high-level term $C : \sigma \rightarrow \texttt{bool}$ in order to show, for a given store s, that $\mathrm{Eval}(CM, s) =_p \mathrm{Eval}(CN, s)$. We know that $C^{\downarrow}M^{\downarrow} \sim^b_{l,p} C^{\downarrow}N^{\downarrow}$. For any w we also know by Proposition 5.14 that $\mathrm{Acc}_w(C^{\downarrow}M^{\downarrow}, s_w) = \emptyset$, so, by Corollary 5.15, that

$$\begin{array}{rcl}\mathrm{Eval}(CM, s)_w & = & \mathrm{Eval}_w(C^{\downarrow}M^{\downarrow}, s_w)\\ & = & \mathrm{Eval}^b_w(C^{\downarrow}M^{\downarrow}, s_w)\end{array}$$

The same holds for N.

The definition of $\sim^b_{l,p}$ then yields two cases.

1. In the first case we have:

$$\mathrm{P}(\mathrm{Eval}_w(C^{\downarrow}M^{\downarrow}, s_w) = (s'_w, V)) \geq \delta_b$$

   and

$$\mathrm{P}(\mathrm{Eval}_w(C^{\downarrow}N^{\downarrow}, s_w) = (s''_w, V)) \geq \delta_b$$

   for some $s' \restriction \mathrm{PubLoc} = s'' \restriction \mathrm{PubLoc}$ and V. As $\delta_b > 0$, we have that $\mathrm{Eval}_{w'}(C^{\downarrow}M^{\downarrow}, s_{w'}) = (s'_{w'}, V)$ and $\mathrm{Eval}_{w''}(C^{\downarrow}N^{\downarrow}, s_{w''}) = (s''_{w''}, V)$, for some w' and w''. So,

$$\mathrm{Eval}(CM, s)_{w'} = \mathrm{Eval}_{w'}(C^{\downarrow}M^{\downarrow}, s_{w'}) = (s'_{w'}, V)$$

   and

$$\mathrm{Eval}(CN, s)_{w''} = \mathrm{Eval}_{w''}(C^{\downarrow}N^{\downarrow}, s_{w''}) = (s''_{w''}, V)$$

   As the map $s \mapsto s_w$ is injective, it follows that $\mathrm{Eval}(CM, s) = (s', V)$ and $\mathrm{Eval}(CN, s) = (s'', V)$. Therefore, $\mathrm{Eval}(CM, s) =_p \mathrm{Eval}(CN, s)$, concluding this case.

2. In the second case we have:

$$\mathrm{P}(\mathrm{Eval}_w(C^{\downarrow}M^{\downarrow}, s_w) = \Omega) \geq \delta_{b+1}$$

   and

$$\mathrm{P}(\mathrm{Eval}_w(C^{\downarrow}N^{\downarrow}, s_w) = \Omega) \geq \delta_{b+1}$$

   As $\delta_{b+1} > 0$ there are w' and w'' such that

$$\mathrm{Eval}_{w'}(C^{\downarrow}M^{\downarrow}, s_{w'}) = \mathrm{Eval}_{w''}(C^{\downarrow}N^{\downarrow}, s_{w''}) = \Omega$$

   So $\mathrm{Eval}(CM, s) = \mathrm{Eval}(CN, s) = \Omega$ concluding the proof.

■

It would be interesting to look for stronger computational-soundness results. For example, one might consider changing $\sim^b_{l,p}$ so as not to conflate nontermination with too many erroneous accesses.

6 Conclusion

Given the abundance of disparate techniques for protection, it is useful to compare those techniques. Our results relate layout randomization to the static guarantees of the syntax of a high-level language in which the programs that represent attackers can neither mention private locations directly nor access them via natural-number addresses. Our work follows that of Pucella and Schneider [32], which related obfuscation and type systems. However, their theorems do not explicitly mention resource bounds or probabilities, and focus on integrity properties. These theorems basically pertain to the protection—by obfuscation or typing—of a program from a potentially dangerous input. We consider more general attackers, represented by arbitrary contexts, and also treat program equivalences, capturing not only integrity but also secrecy properties. Despite these substantial differences, we share their goal of understanding randomization in the context of programming languages and their implementations.

Going further, one could study layout randomization for richer languages. Those languages may include richer type systems, concurrency, and dynamic allocation, in particular. For instance, they may allow the passing of locations (see Section 2), much like security protocols pass communication channels and cryptographic keys. Thus, despite the differences mentioned in the introduction, methods currently being developed in the study of security protocols could also be helpful in the study of layout randomization. In another direction, one could explore variants and extensions—for instance, with replication (e.g., [9])— as well as other forms of randomization. For instance, our methods seem to apply to techniques that rename opcodes randomly.

In such further advances, it may be tempting to develop and analyze sophisticated implementations that yield the strongest possible guarantees. Again, the analogy with security protocols may prove helpful. Nevertheless, those implementations would be of only limited interest unless they correspond to methods that could plausibly be used in actual systems. For instance, in models where the attacker may corrupt all shared memory (not common in security protocols), it may be futile to consider protection approaches that rely on frequent, extensive memory checks. Such difficulties should however encourage the development of programming models and constructs for which security guarantees can be realistically obtained. A promising step in this direction is the identification of the memory locations that are critical to security and require protection [30].

Acknowledgments

We are grateful to Adriana Compagnoni, Peter Druschel, Úlfar Erlingsson, Cédric Fournet, Sergio Maffeis, Vaughan Pratt, Fred Schneider, and Ben Zorn for their questions, comments, and encouragement.

References

[1] Martín Abadi. Protection in programming-language translations. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 868–883. Springer, 1998.

[2] Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: principles, implementations, and applications. *ACM Transactions on Information and System Security*, 13(1):1–40, 2009.

[4] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (The computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[5] Anonymous. Bypassing PaX ASLR protection. *Phrack*, 11(59), 2002.

[6] Michael Backes, Dennis Hofheinz, and Dominique Unruh. Cosp: a general framework for computational soundness proofs. In *16th ACM Conference on Computer and Communications Security*, pages 66–78, 2009.

[7] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanović. Randomized instruction set emulation. *ACM Transactions on Information and System Security*, 8(1):3–40, 2005.

[8] Gilles Barthe, Tamara Rezk, and Amitabh Basu. Security types preserving compilation. *Computer Languages, Systems and Structures*, 33(2):35–59, 2007.

[9] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 158–168, 2006.

[10] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium*, 2003.

[11] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *14th USENIX Security Symposium*, 2005.

[12] Hubert Comon-Lundh and Véronique Cortier. Computational soundness of observational equivalence. In *15th ACM Conference on Computer and Communications Security*, pages 109–118, 2008.

[13] Peter Druschel and Larry L. Peterson. High-performance cross-domain data transfer. Technical Report TR 92-11, Department of Computer Science, The University of Arizona, March 1992.

[14] Úlfar Erlingsson. Low-level software security: Attacks and defenses. In Alessandro Aldini and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design IV, FOSAD 2006/2007 Tutorial Lectures*, volume 4677 of *Lecture Notes in Computer Science*, pages 92–134. Springer, 2007.

[15] Matthias Felleisen and Daniel P. Friedman. Control operators, the secd-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, pages 193–219, 1986.

[16] Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building diverse computer systems. In *6th Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[17] Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms. In *16th ACM Conference on Computer and Communications Security*, pages 432–441, 2009.

[18] Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 323–335, 2008.

[19] Masahito Hasegawa and Yoshihiko Kakutani. Axioms for recursion in call-by-value. *Higher-Order and Symbolic Computation*, 15(2-3):235–264, 2002.

[20] Michael Howard and Matt Thomlinson. Windows Vista ISV security, April 2007. `http://msdn2.microsoft.com/en-us/library/bb430720.aspx`.

[21] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *10th ACM Conference on Computer and Communications security*, pages 272–280, 2003.

[22] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, pages 191–206, 2002.

[23] John Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[24] Eugenio Moggi. Computational lambda-calculus and monads. In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, 1989.

[25] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[26] James H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.

[27] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[28] Gene Novark and Emery D. Berger. Dieharder: securing the heap. In *17th ACM Conference on Computer and Communications security*, pages 573–584, 2010.

[29] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *Communications of the ACM*, 51(12):87–95, 2008.

[30] Karthik Pattabiraman, Vinod Grover, and Benjamin G. Zorn. Samurai: protecting critical data in unsafe languages. In *EuroSys*, pages 219–232, 2008.

[31] PaX Project. The PaX project, 2004. `http://pax.grsecurity.net/`.

[32] Riccardo Pucella and Fred B. Schneider. Independence from obfuscation: A semantic framework for diversity. In *19th IEEE Computer Security Foundations Workshop*, pages 230–241, 2006.

[33] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *11th ACM Conference on Computer and Communications Security*, pages 298–307, 2004.

[34] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections: Setting back browser security by 10 years. `http://taossa.com/archive/bh08sotirovdowd.pdf`, 2008.

[35] Anna Nora Sovarel, David Evans, and Nathanael Paul. Where's the FEEB? the effectiveness of instruction set randomization. In *14th USENIX Security Symposium*, pages 145–160, 2005.

[36] Curtis Yarvin, Richard Bukowski, and Tom Anderson. Anonymous RPC: Low-latency protection in a 64-bit address space. In *USENIX Summer Technical Conference*, pages 175–186, 1993.