On Layout Randomization for Arrays and Functions

Martín Abadi[1,2] and Jérémy Planul[3]

[1] Microsoft Research Silicon Valley
[2] University of California, Santa Cruz
[3] Stanford University

Abstract. Low-level attacks often rely on guessing absolute or relative memory addresses. Layout randomization aims to thwart such attacks. In this paper, we study layout randomization in a setting in which arrays and functions can be stored in memory. Our results relate layout randomization to language-level protection mechanisms, namely to the use of abstract locations (rather than integer addresses). They apply, in particular, when each abstract location can hold an entire array which, concretely, compilation implements with a memory buffer at a random base address.

1 Introduction

Many attacks on software systems rely on predicting the absolute or relative locations of particular pieces of data in memory. For instance, in a system without proper bounds checking, if an attacker has access to one buffer b in the heap and can guess that an immediately contiguous buffer b' contains some sensitive data, then the attacker may try to tamper with the data in b' by overflowing b. The data in b' might for example be an authentication flag that indicates whether the attacker has been properly authenticated, and then the tampering may toggle it from false to true (e.g., [4]). The data in b' might also be a function (or a function pointer), and then the tampering may replace it so that code of the attacker's choice is executed later, when control is transferred to b'.

Layout randomization aims to thwart attacks that guess locations in this manner (e.g., [5, 6, 13]). Basically, layout randomization consists in placing data in memory at random addresses, which may for example be chosen at load time, and which will vary from system to system.

In practice, layout randomization is often an imperfect mitigation (e.g., [16]). In particular, for performance or compatibility reasons, only parts of the memory layout are randomized, typically at a fairly coarse granularity. Moreover, information about the layout sometimes leaks to attackers through various channels. Finally, layout randomization can prove ineffective against attacks that target large regions of memory, such as heap-spraying attacks.

Despite these limitations, layout randomization is widely used in systems, and it has been beneficial. Furthermore, layout randomization resembles other

D. Basin and J.C. Mitchell (Eds.): POST 2013, LNCS 7796, pp. 167–185, 2013.

attractive forms of randomization [8] (such as in-place code randomization [12]) and also cryptographic protection (via the analogy between locations and encryption keys). Therefore, we believe that there is value in trying to understand its power, to characterize it precisely, and to compare it to other protection techniques.

In this spirit, recent research [3, 9, 15] relates layout randomization to the use of language-level protection mechanisms. In the present paper we aim to contribute to this line of work. Specifically, we treat layout randomization in a setting in which arrays and functions can be stored in memory—so overflows on arrays can affect other arrays and modify functions, much as in the example above.

We consider a high-level language with an abstract notion of location and a lower-level language with integer memory addresses. Both languages support functions and arrays. In the high-level language, a location can hold an entire array, while in the lower-level language array elements are stored at consecutive addresses in memory. We also consider a translation from the former language to the latter one that maps locations to randomly chosen base addresses for memory buffers. The choice is random but not necessarily uniform. For instance, all base addresses may be chosen to satisfy alignment constraints. On the other hand, the randomization is not done at the finer granularity of individual array entries, nor is the layout within arrays randomized; although conceptually tractable, such variants could have a disastrous impact on performance.

Our translation also embodies two additional precautions that complement layout randomization:

- Even with a random memory layout, it is possible that two buffers are contiguous in memory. Much as in some practical systems (e.g. [10, 17]), we can eliminate this possibility entirely by imposing the introduction of "guard regions" between buffers. Such guard regions contribute to the security guarantees that we establish.
- In the high-level language, an assignment such as $l := M$ completely overwrites the value in l. Therefore, in the low-level language, it is important that a corresponding assignment do not leave any observable traces of that value, even if M is shorter. We introduce a dynamic check to treat this point; other solutions (e.g., with special padding) may be viable.

We study the correctness and security of the translation. Viewing attackers as contexts, we prove that low-level contexts (with integer addresses) correspond to high-level contexts (with abstract locations), thus showing that the translation does not enable any new attacks [1]. We also prove that the translation preserves security properties that can be expressed as program equivalences. Both of these results are probabilistic, with probabilities that approach 1 for suitable distributions on memory layouts.

In the next section, Section 2, we discuss some small examples, informally. We define the high-level language and the low-level language in Sections 3 and 4, respectively. In Section 5, we consider probability distributions on memory layouts. In Section 6, we define and study the translation. Throughout, our approach is

often analogous to that of Abadi and Plotkin [3]. We discuss this and other related work in Section 7, and then we conclude with brief comments on further work. Because of space constraints, we leave auxiliary results and proofs to an online version of this paper [2].

2 Examples

As a small introductory example, we consider the following program, which (to first approximation) we can express in both the high-level language and the low-level language defined below:

$$\begin{array}{l}\lambda x.\ l_1 := M;\\ \quad\ l_2 := x;\\ \quad\ N\end{array}$$

This program inputs a value x, executes M and stores the result in the location l_1, then stores x in the location l_2, and finally executes N. For instance, M might be an array with two locations, the first location could hold a function f and the second an integer n, and N could apply f to the input x and to n after extracting f and n from l_1 and x from l_2. In that case, the program would be:

$$\begin{array}{l}\lambda x.\ l_1 := [f; n];\\ \quad\ l_2 := x;\\ \quad\ (1\,\mathtt{ith}\,!l_1)(!l_2)(2\,\mathtt{ith}\,!l_1)\end{array}$$

Here, $[f; n]$ forms an array with two elements; $!l_1$ and $!l_2$ return the contents of l_1 and l_2, respectively; and $1\,\mathtt{ith}$ and $2\,\mathtt{ith}$ extract the first and the second elements of an array, respectively.

In the high-level language, we view l_1 and l_2 as two independent, abstract locations, so (under call-by-value semantics) we would expect that this code would behave just like the result of in-lining f, n, and x, namely:

$$\begin{array}{l}\lambda x.\ l_1 := [f; n];\\ \quad\ l_2 := x;\\ \quad\ f(x)(n)\end{array}$$

In the low-level language, on the other hand, the locations are integer addresses in memory, so we need to worry about buffer overflows and similar errors. In particular, when the value of x is too large to fit into the space allocated for l_2, the input may cause an error. If the error is not caught, it is possible that some of the value of x will clobber the contents of memory including l_1, and then $(1\,\mathtt{ith}\,!l_1)(!l_2)(2\,\mathtt{ith}\,!l_1)$ may not behave like $f(x)(n)$. In that case, an attacker that chooses the value of x may well be able to execute a function of their choice instead of f.

Layout randomization can however ensure that an overflow on l_2 will clobber the contents of l_1 only with a small probability, thus countering the attack. Thus, layout randomization can imply that properties that hold when l_1 and l_2

are two independent, abstract locations continue to hold when they are mapped to concrete memory addresses. Our theorems capture this preservation of properties. The properties could easily fail if instead l_1 and l_2 were mapped to fixed, contiguous memory addresses, and if bounds checking was not done properly, as the attack above indicates.

Some of the same themes appear in many other examples. For instance, consider a piece of code where the location l_1 holds a flag that indicates whether some security check has been completed satisfactorily. For instance, l_1 could hold an authentication flag, as mentioned at the start of the Introduction, and much as in the code of an SSH implementation that Chen et al. attacked [4]. The flag is initially false. After some input x is stored in another location l_2, the security checks are performed, and the flag is set to true if the checks are successful. Later, various sensitive operations may be permitted if the flag is true.

$$\begin{array}{l}\lambda x.\ l_1 := \texttt{false};\\ \quad l_2 := x;\\ \quad \texttt{if}\ \textit{some checks}\ \texttt{then}\, l_1 := \textit{true};\\ \quad \ldots\\ \quad \texttt{if}\, !l_1\, \texttt{then}\ \textit{do some sensitive operation}\end{array}$$

The security of this code depends on the integrity of the contents of l_1. Layout randomization can protect this integrity, thwarting direct writes to l_1 by attackers who would try to guess its absolute address in memory and also writes via overflows on l_2 and other locations. Our results account for this application of layout randomization.

In the examples above, we are primarily concerned about an adversary that may provide a dangerous input, but which need not modify locations such as l_1 and l_2 directly. In general, however, we are also interested in adversaries that have direct access to some locations, which we call public locations. We refer to other locations as private locations. At the high level, we restrict adversaries so that they cannot refer to private locations; at the low level, we study the protection of the private locations via layout randomization. For instance, letting l be a private location and l' a public location, we may consider the following program:

$$\lambda f.\, l := 0; l' := 0; f(1)$$

At the high level, if an adversary provides an input function f, this function may read or write l', but it cannot read or write l since l is private. Therefore, when $f(1)$ terminates, l should hold the value 0. So, with respect to such adversaries, the program is equivalent to:

$$\lambda f.\, l' := 0; f(1)$$

At the low level, l will be mapped to an integer memory address. If this mapping is predictable, then the adversary may be able to read or write l, either via an absolute address or via a relative address with an offset from l'. With proper layout randomization, on the other hand, l will be mapped to a random address,

and the offset between l and l' will be random as well, so the adversary will not be able to find and to access l, with high probability. Therefore, the program equivalence will be preserved, at least in a probabilistic sense.

Furthermore, adversaries may do more than provide a single input. They may be contexts that interact with the systems that we wish to protect, for example invoking them multiple times with different inputs, and accessing public locations before and after those invocations. Our approach addresses such interactions.

3 The High-Level Language

In this section, we define our high-level language. In this language, memory locations are symbolic names, and the semantics uses an abstract store to link locations to values.

3.1 Syntax and Informal Semantics

The high-level language is a call-by-value λ-calculus with natural numbers, arrays, and location-labeled dereference and assignment operations. For general background on the λ-calculus, see [11, 14].

The terms of our language are defined by:

$$\begin{array}{rcl} M & ::= & x \mid c \mid [M; \ldots; M] \mid \lambda x.\, M \mid MM \\ c & ::= & * \mid n \mid + \mid =_{\mathtt{nat}} \mid \ldots \\ & & \mid \ \mathtt{ith} \mid \mathtt{length} \\ & & \mid\, !_{\mathtt{loc}} l_{\mathtt{loc}} \mid l_{\mathtt{loc}} :=_{\mathtt{loc}} \end{array}$$

where M and N range over terms, c ranges over a set of constants, and l ranges over a finite set Loc of locations. The terms include variables, constants, arrays, abstractions, and applications. The constants include $*$ (the "unit" value); the usual arithmetic constants, operations, and relations (such as the numerals n, addition $+$, and equality $=_{\mathtt{nat}}$); array access $\mathtt{ith}$ and length measurement $\mathtt{length}$; and constants for accessing locations $!_{\mathtt{loc}} l_{\mathtt{loc}}$ and $l_{\mathtt{loc}} :=_{\mathtt{loc}}$.

We adopt 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. We also adopt standard infix notations, for example sometimes writing $M\,\mathtt{ith}\,N$ instead of $\mathtt{ith}\ M\ N$.

Intuitively, $!_{\mathtt{loc}} l_{\mathtt{loc}}$ outputs the contents of location l and $l_{\mathtt{loc}} :=_{\mathtt{loc}}$ writes its argument in location l. Each location can hold arrays of a given length, and writing produces an error if the argument is not an array of the appropriate length. So, for simplicity, we do not allow storing integers and functions directly into memory, but we do allow storing one-element arrays that contain integers and functions, and in examples we may for instance write $l_{\mathtt{loc}} :=_{\mathtt{loc}} 0$ as an abbreviation for $l_{\mathtt{loc}} :=_{\mathtt{loc}} [0]$. We allow nested arrays (such as $[[M; N]]$) but consider only the top level for calculating lengths, and do not differentiate "short" vs. "long" elements in an array (so for example $[0]$, $[\lambda x.\, x(x7)]$, and $[[M; N]]$ all have length 1). A more elaborate definition could be introduced, and would make sense provided corresponding adjustments are made in the compilation function (see Section 6).

The subscript in $!_{\mathtt{loc}} l_{\mathtt{loc}}$ and $l_{\mathtt{loc}} :=_{\mathtt{loc}}$ is intended to differentiate these constants from the syntax of the low-level language of Section 4. We omit the subscripts sometimes, when they are clear from context or when we wish to discuss both the high-level and the low-level language, as in the examples of Section 2.

Note that $l_{\mathtt{loc}}$ is not itself a term in this language, so locations are not first-class values. This restriction constitutes a simplification (see [3, Section 2]), and contributes to the gap between the high-level language and the low-level language of the next section (in which addresses are first-class values). On the other hand, functions for reading and writing locations can be passed as arguments, returned as results, and stored in memory, so encodings of locations as first-class values are straightforward.

Various other standard abbreviations and encodings are convenient. These include encodings of booleans and other datatypes, and recursive function definitions, as usual in untyped call-by-value lambda calculus. Using these, we can program control constructs, including loops of the form $\mathtt{for}\ i = 1\ \mathtt{to}\ e\ \mathtt{do}\ e'\ \mathtt{done}$. We sometimes write $\mathtt{skip}$ for $*$, and $M; N$ for $\mathtt{let}\, x\ \mathtt{be}\, M\ \mathtt{in}\, N$ (where x is not free in N). We also use $\mathtt{raise\_error}$ as syntactic sugar for an error-raising term, for instance $j\ k$ where j and k are integers.

For simplicity, this language does not support dynamic allocation, which could perhaps be handled as in the work of Jagadeesan et al. [9], at least in the bounded form that they consider. Treating more general memory-management and scoping facilities remains a challenging subject for further research.

3.2 Values

We designate a subset of the expression of the programming language as *values*:

$$\begin{array}{rcl} V & ::= & d \mid e \mid [V; \ldots; V] \mid \lambda x.\, M \\ d & ::= & * \mid n \mid + \mid =_{\mathtt{nat}} \mid \ldots \\ & & \mid\ \mathtt{ith} \mid \mathtt{length} \\ & & \mid\ l_{\mathtt{loc}} :=_{\mathtt{loc}} \\ e & ::= & n + \mid n\ =_{\mathtt{nat}} \mid \ldots \\ & & \mid\ n\ \mathtt{ith} \end{array}$$

Values can be thought of as (syntax for) completed computations. We include all constants in the set of values, except for the constants for reading locations. We also include partially evaluated binary operators. We write $\mathbb{HV}$ for the set of values of the high-level language.

Values of the form $[V_1; \ldots; V_n]$ are *array values*. We define their length by: $|[V_1; \ldots; V_n]| = n$.

3.3 Memory Model

The semantics of the high-level language is based on a simple model of memory.

We assume a fixed mapping $\mathrm{sig} : \mathrm{Loc} \to \mathbb{N}$ that, intuitively, gives the length of each location. We call such a mapping a *signature*.

A *store* is a mapping $s : \mathrm{Loc} \to \mathbb{HV}$ that sends locations to values of the high-level language, such that $s(l)$ is an array value and $|s(l)| = \mathrm{sig}(l)$ for every $l \in \mathrm{Loc}$.

In order to consider security properties, we assume that the set of locations Loc is the disjoint union of two sets, PubLoc and PriLoc, of *public* and *private* locations. As explained in Section 2 and in Section 6.3, below, we model attackers as programs that have direct access to public locations (but not, by default, to private locations).

3.4 Operational Semantics

We define a small-step operational semantics of the high-level language in the style of Felleisen and Friedman [7].

Redexes include terms of the forms:

$$(\lambda x.\, M)V \qquad V\,\mathtt{ith}\,[V;\ldots;V] \qquad \mathtt{length}\,[V;\ldots;V]$$
$$!_{\mathtt{loc}} l_{\mathtt{loc}} \qquad\qquad l_{\mathtt{loc}} :=_{\mathtt{loc}} V$$

and other redexes that involve the various arithmetic constants, operations, and relations, such as $0 =_{\mathtt{nat}} 0$. Redexes also include "ill-typed" constructions, such as the application 00; these redexes will raise errors. (For brevity, we do not list all these ill-typed constructions.) We define a reduction relation $R \to M$ between redexes and terms, and an error property $R \downarrow_{\mathrm{error}}$ on redexes:

$$(\lambda x.\, M)V \to M[V/x]$$
$$i\,\mathtt{ith}\,[V_1;\ldots;V_n] \to V_i \qquad \mathtt{length}\,[V_1;\ldots;V_n] \to n$$
$$i\,\mathtt{ith}\,[V_1;\ldots;V_n] \downarrow_{\mathrm{error}} \qquad \text{for } i = 0 \text{ and } i > n$$
$$\ldots$$

where the ellipses indicate missing arithmetic and error transitions, such as:

$$0 =_{\mathtt{nat}} 0 \to \mathtt{true} \qquad 0 + 1 \to 1 \qquad 00 \downarrow_{\mathrm{error}}$$

We define *evaluation contexts* by:

$$E ::= [-] \mid [V;\ldots;V;E;\ldots;M] \mid EM \mid VE$$

We write $E[M]$ for the term obtained by replacing the "hole" $[-]$ in an evaluation context E with the term M. For every term M, either M is a value, or M can be analyzed uniquely in the form $E[R]$, with R a redex.

A *configuration* is a pair (s, M) with s a store and M a term. The small-step semantics consists of a transition relation and two error properties on configurations:

$$(s, M) \to (s', M') \qquad (s, M) \downarrow_{\mathrm{error}} \qquad (s, M) \downarrow^{l}_{\mathrm{error}}$$

The error property $(s, M) \downarrow^{l}_{\mathrm{error}}$ distinguishes buffer-overflow errors, to which we give a specific treatment below.

For redexes, we set:

$$\begin{array}{ll} (s, !_{\mathtt{loc}} l_{\mathtt{loc}}) \to (s, V) & (\text{if } s(l) = V) \\ (s, l_{\mathtt{loc}} :=_{\mathtt{loc}} V) \to (s[l \mapsto V], \mathtt{skip}) & (\text{if } |V| = \mathrm{sig}(l)) \\ (s, l_{\mathtt{loc}} :=_{\mathtt{loc}} V) \downarrow_{\mathrm{error}} & (\text{if } |V| < \mathrm{sig}(l)) \\ (s, l_{\mathtt{loc}} :=_{\mathtt{loc}} V) \downarrow^{l}_{\mathrm{error}} & (\text{if } |V| > \mathrm{sig}(l)) \end{array}$$

and:

$$\frac{R \to M'}{(s, R) \to (s, M')} \qquad \frac{R \downarrow_{\mathrm{error}}}{(s, R) \downarrow_{\mathrm{error}}}$$

The general case follows via three rules:

$$\frac{(s, R) \to (s', M')}{(s, E[R]) \to (s', E[M'])} \qquad \frac{(s, R) \downarrow_{\mathrm{error}}}{(s, E[R]) \downarrow_{\mathrm{error}}} \qquad \frac{(s, R) \downarrow^{l}_{\mathrm{error}}}{(s, E[R]) \downarrow^{l}_{\mathrm{error}}}$$

We can also define a corresponding big-step semantics by:

$$\begin{array}{lcl} (s, M) \Rightarrow (s', V) & \iff & (s, M) \to^* (s', V) \\ (s, M) \Downarrow_{\mathrm{error}} & \iff & \exists s', M'.\, (s, M) \to^* (s', M') \downarrow_{\mathrm{error}} \\ (s, M) \Downarrow^{l}_{\mathrm{error}} & \iff & \exists s', M'.\, (s, M) \to^* (s', M') \downarrow^{l}_{\mathrm{error}} \\ (s, M) \Uparrow & \iff & \forall n.\, \exists s', M'.\, (s, M) \to^n (s', M') \end{array}$$

These relations and properties are mutually exclusive. The relation $(s, M) \Rightarrow (s', V)$ holds if M evaluates to the value V with final store s' when the initial store is s; the property $(s, M) \Downarrow^{l}_{\mathrm{error}}$ holds if the term M causes a buffer-overflow error on location l when the initial store is s, and $(s, M) \Downarrow_{\mathrm{error}}$ holds if M results in a different error; the property $(s, M) \Uparrow$ holds if M diverges when the initial store is s.

4 The Low-Level Language

Our low-level language, which we define in this section, mainly differs from the high-level language in employing integer addresses.

4.1 Syntax and Informal Semantics

The syntax of the low-level language is a variant of that of the high-level language in which we replace high-level constants for accessing locations by distinguished locations and memory-access constants:

$$\begin{array}{lcl} M & ::= & x \mid c \mid [M; \ldots; M] \mid \lambda x.\, M \mid MM \\ c & ::= & * \mid n \mid + \mid =_{\mathtt{nat}} \mid \ldots \\ & & \mid \mathtt{ith} \mid \mathtt{length} \\ & & \mid l_{\mathtt{nat}} \mid !_{\mathtt{nat}} \mid :=_{\mathtt{nat}} \end{array}$$

where l ranges over the finite set Loc of locations.

Informally, the constant $l_{\mathtt{nat}}$ evaluates to the index of location l; it returns an integer. The constant $!_{\mathtt{nat}}$, when applied to an integer n, reads the contents of memory at address n. The constant $:=_{\mathtt{nat}}$, when applied to a first argument and an integer n, writes the first argument at address n. (So, with infix notation, we write $N := M$ for $:= \ M \ N$.)

Because each constant $l_{\mathtt{nat}}$ is a first-class, legal expression on its own, we can write programs that pass these constants and that store them in memory, such as $l_{\mathtt{nat}} :=_{\mathtt{nat}} l'_{\mathtt{nat}}$. Moreover, $!_{\mathtt{nat}}$ and $:=_{\mathtt{nat}}$ are legal expressions even when they are not applied to location constants (unlike the corresponding notations in the high-level language). Thus, we can write not only $!_{\mathtt{nat}} l_{\mathtt{nat}}$ and $l_{\mathtt{nat}} :=_{\mathtt{nat}} 0$, but also for example $!_{\mathtt{nat}} x$, $!_{\mathtt{nat}}(x+1)$, $(x+1) :=_{\mathtt{nat}} 8$, or $(x + l_{\mathtt{nat}}) :=_{\mathtt{nat}} (x + l'_{\mathtt{nat}})$, where l is a location and x is a variable. As these small examples illustrate, addresses can be the result of integer computations, on variables and constants (including location constants). This flexibility could allow an attacker to try to access memory at a computed offset from a known location, for instance.

We assume that each address in memory is either used to hold a value or unused, and assume that access to an unused address results in an immediate fatal error. These assumptions are as in the "fatal-error model" of Abadi and Plotkin [3]. It should be possible to adapt our work to the alternative "recoverable-error model".

Both the high-level language and the low-level language allow storing a program in memory, retrieving it, then invoking it. In particular, an attacker may be able to inject code into memory via a direct assignment. On the other hand, neither language allows computing on the code of programs, nor confusing natural numbers with programs. In this respect, the low-level language remains fairly high-level. For example, in the low-level language, after the assignment $l_{\mathtt{nat}} :=_{\mathtt{nat}} (\lambda x.\, !_{\mathtt{nat}} l'_{\mathtt{nat}})$, an attacker that can read the contents of $l_{\mathtt{nat}}$ would be able to execute $(\lambda x.\, !_{\mathtt{nat}} l'_{\mathtt{nat}})$, but not extract $l'_{\mathtt{nat}}$ from it, nor the syntax tree of $(\lambda x.\, !_{\mathtt{nat}} l'_{\mathtt{nat}})$. It seems unlikely that one could obtain strong guarantee without some such restrictions.

4.2 Values

As in the high-level language, we designate a subset of the expression of the programming language as *values*. In particular, we take $!_{\mathtt{nat}}$ and $:=_{\mathtt{nat}}$ to be values:

$$
\begin{array}{rcl}
V & ::= & d \mid e \mid [V; \ldots; V] \mid \lambda x.\, M \\
d & ::= & * \mid n \mid + \mid =_{\mathtt{nat}} \mid \ldots \\
 & & \mid \ \mathtt{ith} \mid \mathtt{length} \\
 & & \mid !_{\mathtt{nat}} \mid :=_{\mathtt{nat}} \\
e & ::= & n + \mid n \ =_{\mathtt{nat}} \mid \ldots \\
 & & \mid \ n \ \mathtt{ith} \\
 & & \mid :=_{\mathtt{nat}} V
\end{array}
$$

We write $\mathbb{LV}$ for the set of values of the low-level language.

4.3 Memory Model

Concretely, we let a *memory* be a mapping $m : \mathrm{Mem} \to (\mathbb{LV} + \{\varepsilon\})$, where:

- Mem is the set $0, \ldots, \kappa$ of memory addresses, for a given $\kappa \geq 0$,
- we assume that $|\sum_{l \in \mathrm{Loc}} \mathrm{sig}(l)| \leq \kappa + 1$,
- $\mathbb{LV} + \{\varepsilon\}$ is the disjoint union of $\mathbb{LV}$ and $\{\varepsilon\}$, and
- $m(a) = \varepsilon$ indicates that a is an unused address of m.

Since the low-level language contains location constants, its semantics depends on how these are laid out in memory. A *memory layout* is an injective mapping $w : \mathrm{Loc} \hookrightarrow \mathrm{Mem}$. Such a memory layout connects the abstract and the concrete memory models. A location that stores an array in the abstract model corresponds to a range of memory addresses in the concrete model. For a location l, that range starts at address $w(l)$ and includes $\mathrm{sig}(l)$ consecutive address. We restrict attention to those memory layouts that do not cause range overflows or overlaps, that is, such that there exist no $l_1 \in \mathrm{Loc}$ such that $w(l_1) \leq \kappa < w(l_1) + \mathrm{sig}(l_1) - 1$, and no distinct $l_1, l_2 \in \mathrm{Loc}$ such that $w(l_1) \leq w(l_2) < w(l_1) + \mathrm{sig}(l_1)$. We let $\mathrm{Ran}(w)$ be the set $\{a \in \mathrm{Mem} \mid \exists l.\, w(l) \leq a < w(l) + \mathrm{sig}(l)\}$.

A *public layout* $w_p : \mathrm{PubLoc} \hookrightarrow \mathrm{Mem}$ maps public locations to addresses. We assume that one public layout is fixed throughout, and we consider only those memory layouts that extend it.

We also define stores for the low-level language; we call them low-level stores to distinguish them from those of the high-level language, to which they are directly analogous. Thus, a *low-level store* is a mapping $s : \mathrm{Loc} \to \mathbb{LV}$ that sends locations to values of the low-level language, such that $s(l)$ is an array value and $|s(l)| = \mathrm{sig}(l)$ for every $l \in \mathrm{Loc}$. For every low-level store s and memory layout w, there is a corresponding memory $\mathrm{mem}(s, w)$ defined by:

$$\mathrm{mem}(s, w)(a) = \begin{cases} s(l).(i) & \text{if there exists } i \in 1..\mathrm{sig}(l) \text{ such that } w(l) + i - 1 = a \\ \varepsilon & \text{otherwise } (a \notin \mathrm{Ran}(w)) \end{cases}$$

where $s(l).(i)$ is the ith element of the array value $s(l)$. The mapping $s \mapsto \mathrm{mem}(s, w)$ is 1-1, from low-level stores to memories, but not onto. We say that m *has the form* $\mathrm{mem}(s, w)$ if it equals $\mathrm{mem}(s, w)$ for some w. We abbreviate $\mathrm{mem}(s, w)$ to s_w.

4.4 Operational Semantics

The operational semantics resembles that of the high-level language in its treatment of functions and numbers. In particular, the redexes and the reduction relation between redexes and terms are both much as in the high-level language, but with

$$l_{\mathtt{nat}} \quad (\text{for } l \in \mathrm{Loc}) \qquad !_{\mathtt{nat}} V \qquad V :=_{\mathtt{nat}} V$$

as redexes. Evaluation contexts are as in the high-level language.

A configuration is a pair (m, M) of a memory m and a term M. The semantics consists of a transition relation and two error properties on configurations, all relative to the memory layout chosen:

$$w \models (m, M) \to (m', M')$$
$$w \models (m, M) \downarrow_{\text{error}} \qquad w \models (m, M) \downarrow^a_{\text{error}}$$

The error property $w \models (m, M) \downarrow^a_{\text{error}}$ distinguishes accesses to out-of-range or unused addresses; in particular, if $w \models (m, M) \downarrow^a_{\text{error}}$ and m has the form s_w, then $a \notin \text{Ran}(w_p)$.

For redexes, the transition relation and error properties are given by the rules:

$$\frac{R \to M'}{w \models (m, R) \to (m, M')} \qquad \frac{R \downarrow_{\text{error}}}{w \models (m, R) \downarrow_{\text{error}}}$$

together with:

$$w \models (m, l_{\texttt{nat}}) \to (m, w(l)) \qquad (\text{for } l \in \text{Loc})$$

and:

$$\begin{array}{ll}
w \models (m, !_{\texttt{nat}} a) \to (m, V) & (\text{if } a \in \text{Mem and } m(a) = V) \\
w \models (m, !_{\texttt{nat}} a) \downarrow^a_{\text{error}} & (\text{if } a \notin \text{Mem or } m(a) = \varepsilon) \\
w \models (m, a :=_{\texttt{nat}} V) \to (m[a \mapsto V], \texttt{skip}) & (\text{if } a \in \text{Mem and } m(a) \neq \varepsilon) \\
w \models (m, a :=_{\texttt{nat}} V) \downarrow^a_{\text{error}} & (\text{if } a \notin \text{Mem or } m(a) = \varepsilon)
\end{array}$$

The general case follows by the rules:

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

and:

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

Much as in the high-level language, too, this small-step semantics induces a big-step semantics:

$$\begin{array}{lll}
w \models (m, M) \Rightarrow (m', V) & \iff & w \models (m, M) \to^* (m', V) \\
w \models (m, M) \Downarrow_{\text{error}} & \iff & \exists m', M'. w \models (m, M) \to^* (m', M') \downarrow_{\text{error}} \\
w \models (m, M) \Downarrow^a_{\text{error}} & \iff & \exists m', M'. w \models (m, M) \to^* (m', M') \downarrow^a_{\text{error}} \\
w \models (m, M) \Uparrow & \iff & \forall n. \exists m', M'. w \models (m, M) \to^n (m', M')
\end{array}$$

These relations and properties are mutually exclusive.

5 Layout Distributions

The effectiveness of layout randomization requires the use of unpredictable layouts. In this section, we define distributions on layouts, and introduce several quantities that below we employ in quantitative security bounds.

Let d be a probability distribution over the layouts that extend w_p. When $\varphi(w)$ is a statement, we write $\mathrm{P_d}(\varphi(w))$ for the probability that it holds with respect to the distribution d. For example, $\varphi(w)$ might be the assertion that, with the layout w, an execution that starts from a particular low-level configuration (m, M) will produce an error. In that case, $\mathrm{P_d}(\varphi(w))$ is the probability that such an execution will produce an error for a random layout chosen according to d.

For any $n \in \mathbb{N}$, we write $w\#n$ as an abbreviation for $n \notin (\mathrm{Ran}(w) \backslash \mathrm{Ran}(w_p))$. Informally, when we think of n as an address that an attacker is guessing (not out of the memory bounds, and not at public locations, since the attacker knows those), $w\#n$ means that the attacker does not guess the address of a private location. Then $\mathrm{P_d}(w\#n)$ is the corresponding probability, for w chosen according to d. Furthermore, we define:

$$\delta_{\mathrm{d}} = \min\{\mathrm{P_d}(w\#n) \mid n \in \mathrm{Mem} \setminus \mathrm{Ran}(w_p)\}$$

For example, suppose that l is the only private location and that $\mathrm{sig}(l) = 1$. If layouts chosen according to d always map l to the integer 5, then $\delta_{\mathrm{d}} = 0$, simply because $\mathrm{P_d}(w\#5) = 0$. Obviously, such layouts enable an attacker to access the contents of l, trivially, via the address 5. On the other hand, if layouts chosen according to d map l to each of $\mathrm{Mem} \setminus \mathrm{Ran}(w_p)$ with uniform probability, then $\delta_{\mathrm{d}} = 1 - (1/|\mathrm{Mem} \setminus \mathrm{Ran}(w_p)|)$.

As these examples illustrate, a small value for δ_{d} indicates a lack of security. Therefore, we consider lower bounds on δ_{d} for certain choices of d. These lower bounds approach 1 as the size of memory grows, thus indicating that attacker guesses should succeed with vanishing probability in the limit.

In particular, a system may map all public locations to contiguous addresses starting at address 0, and all private locations to contiguous addresses starting at some random base address in the remaining space. There are $|\mathrm{Mem}| - \sum_{l \in \mathrm{Loc}} \mathrm{sig}(l) + 1$ possible positions for the base address. Moreover, any address is the image of a private location with at most $\sum_{l \in \mathrm{PriLoc}} \mathrm{sig}(l)$ of those possible positions. Hence, for this simple distribution, we can show that:

$$\delta_{\mathrm{d}} \geq 1 - \frac{\sum_{l \in \mathrm{PriLoc}} \mathrm{sig}(l)}{|\mathrm{Mem}| - \sum_{l \in \mathrm{Loc}} \mathrm{sig}(l) + 1}$$

assuming $\sum_{l \in \mathrm{Loc}} \mathrm{sig}(l) \leq |\mathrm{Mem}|$. Even with such a coarse scheme, δ_{d} approaches 1 as $|\mathrm{Mem}|$ grows. For example, if there is no public location, the private locations have total length 2^{32} (which represents a reasonable volume to hold in an actual memory), and the size of memory is 2^{64} (as in a large virtual address space), then this bound is $\delta_{\mathrm{d}} \geq (1 - (2^{32}/(2^{64} - 2^{32} + 1)))$, roughly $(1 - 2^{-32})$.

Much more sophisticated arrangements are possible, in particular ones that map each private location independently. Overall, it is fairly easy to pick distributions on layouts that ensure that δ_{d} approaches 1. Basically, with such

distributions, when an attacker looks for private locations in a large enough memory, getting only one try, the attacker is almost certain to miss provided the memory is large enough.

Similarly, for any $l \in \mathrm{Loc}$, we write $w\#\!\!\#l$ as an abbreviation for $w(l) + \mathrm{sig}(l) \notin \mathrm{Ran}(w)$. Thus, $w\#\!\!\#l$ holds precisely when the end of the array located in l is not contiguous to any other location in use. When this property holds, direct buffer overflows from l will raise an error in the implementation, even without proper bounds checking. We define:

$$\varrho_{\mathrm{d}} = \min\{\mathrm{P}_{\mathrm{d}}(w\#\!\!\#l) \mid l \in \mathrm{Loc}\}$$

As for δ_{d}, we would like ϱ_{d} to be large. Fortunately, assuming that memory is large enough, we can easily focus attention on layouts w such that $w\#\!\!\#l$ for all l, so $\varrho_{\mathrm{d}} = 1$. In such layouts, all arrays are separated by unused locations. These unused locations are analogous to the "guard zones" or "guard regions" that appear in practical systems (e.g. [10, 17]).

Moreover, the wishes for δ_{d} that approaches 1 and for $\varrho_{\mathrm{d}} = 1$ are compatible. For instance, a system may keep all public memory together starting at address 0, and private memory together at some random base address in the remaining space, but separate any two arrays with one unused location.

Our results hold for any probability distribution d. For the rest of the paper, we fix a choice of d, and write $\mathrm{P}(\varphi(w))$, δ, and ϱ as abbreviations for $\mathrm{P}_{\mathrm{d}}(\varphi(w))$, δ_{d}, and ϱ_{d}, respectively.

6 Compilation and Its Properties

In this section we define the translation discussed in the introduction. We then prove its correctness and its security.

6.1 The Translation

We translate terms M of the high-level language to terms $M^{\downarrow}$ of the low-level language. Crucially, this translation maps abstract locations to their low-level counterparts. Since these are interpreted relatively to a memory layout, and since this memory layout is chosen according to a probability distribution, the translation embodies layout randomization.

The translation is trivial for all constructs of the high-level language with the exception of the constants $!_{\mathtt{loc}}l_{\mathtt{loc}}$ and $l_{\mathtt{loc}} :=_{\mathtt{loc}}$, which we compile as follows:

$$\begin{aligned}
(!_{\mathtt{loc}}l_{\mathtt{loc}})^{\downarrow} &= [!_{\mathtt{nat}}l_{\mathtt{nat}}; !_{\mathtt{nat}}l_{\mathtt{nat}} + 1; \ldots; !_{\mathtt{nat}}l_{\mathtt{nat}} + \mathrm{sig}(l) - 1] \\
(l_{\mathtt{loc}} :=_{\mathtt{loc}})^{\downarrow} &= \lambda x.\, \mathtt{for}\ i = 1\ \mathtt{to}\ \mathtt{length}\, x\ \mathtt{do}\ l_{\mathtt{nat}} + i - 1 :=_{\mathtt{nat}} i\ \mathtt{ith}\, x;\ \mathtt{done}; \\
&\qquad \mathtt{if}\ i < \mathrm{sig}(l)\ \mathtt{then}\ \mathtt{raise_error}
\end{aligned}$$

Both $(!_{\mathtt{loc}}l_{\mathtt{loc}})^{\downarrow}$ and $(l_{\mathtt{loc}} :=_{\mathtt{loc}})^{\downarrow}$ employ the signature $\mathrm{sig}(l)$. In the case of $(!_{\mathtt{loc}}l_{\mathtt{loc}})^{\downarrow}$, this signature indicates how much to read from memory. In the case of $(l_{\mathtt{loc}} :=_{\mathtt{loc}})^{\downarrow}$, it serves to ensure that what is being written to memory is

not too short. Alternatively, as suggested in the Introduction, we could add distinguished padding values to fill the space available.

However, $(l_{\mathtt{loc}} :=_{\mathtt{loc}})^{\downarrow}$ does not check whether its argument (the data being written to memory) is too long. This absence of bounds checking leads to the possibility of buffer overflows. Although the absence of bounds checking is deliberate in this definition, it models common mistakes (poor design decisions or implementation blunders). In general, without layout randomization or other mitigations, such mistakes could jeopardize security. Nevertheless, our security results (presented below) apply despite these buffer overflows. (Note that, in addition to the buffer overflows that the translation may introduce, contexts may attempt other problematic operations, such as accessing memory at an offset from a known location, as in $(l_{\mathtt{nat}} + 256) :=_{\mathtt{nat}} 0$; our results still apply.)

Since high-level stores may contain functions, and those may contain occurrences of the constructs $!_{\mathtt{loc}} l_{\mathtt{loc}}$ and $l_{\mathtt{loc}} :=_{\mathtt{loc}}$, we extend the translation so that it maps each high-level store s to a low-level store $s^{\downarrow}$. We define $s^{\downarrow}$ by setting, for every $l \in \mathrm{Loc}$,

$$s^{\downarrow}(l) = s(l)^{\downarrow}$$

Given a layout w, we can then obtain a memory $s^{\downarrow}_{w}$.

6.2 Correctness

The translation is correct in the sense that $M^{\downarrow}$ simulates M, under the corresponding big-step semantics:

Proposition 1. *Suppose that M is a term of the high-level language, and w a layout. Then:*

1. *If $(s, M) \Rightarrow (s', V)$ then, $w \models (s^{\downarrow}_{w}, M^{\downarrow}) \Rightarrow (s'^{\downarrow}_{w}, V^{\downarrow})$.*
2. *If $(s, M) \Downarrow_{\mathrm{error}}$ then $w \models (s^{\downarrow}_{w}, M^{\downarrow}) \Downarrow_{\mathrm{error}}$.*
3. *If $(s, M) \Downarrow^{l}_{\mathrm{error}}$ then, if $w(l) + \mathrm{sig}(l) \notin \mathrm{Ran}(w)$, $w \models (s^{\downarrow}_{w}, M^{\downarrow}) \Downarrow^{w(l)+\mathrm{sig}(l)}_{\mathrm{error}}$.*
4. *If $(s, M) \Uparrow$ then $w \models (s^{\downarrow}_{w}, M^{\downarrow}) \Uparrow$.*

Fixing a distribution on layouts, we can derive a probabilistic statement from Proposition 1:

Proposition 2. *Suppose that M is a term of the high-level language. Then:*

1. *If $(s, M) \Rightarrow (s', V)$, then, for every w, $w \models (s^{\downarrow}_{w}, M^{\downarrow}) \Rightarrow (s'^{\downarrow}_{w}, V^{\downarrow})$.*
2. *If $(s, M) \Downarrow_{\mathrm{error}}$ then, for every w, $w \models (s^{\downarrow}_{w}, M^{\downarrow}) \Downarrow_{\mathrm{error}}$.*
3. *If $(s, M) \Downarrow^{l}_{\mathrm{error}}$ then $\mathrm{P}(w \models (s^{\downarrow}_{w}, M^{\downarrow}) \Downarrow^{w(l)+\mathrm{sig}(l)}_{\mathrm{error}}) \geq \varrho$.*
4. *If $(s, M) \Uparrow$ then, for every w, $w \models (s^{\downarrow}_{w}, M^{\downarrow}) \Uparrow$.*

Further, we can restate the correctness of compilation in terms of a coarse notion of evaluation. For any store s and term M of the high-level language, we define $\mathrm{Eval}(M, s)$ by:

$$\mathrm{Eval}(M, s) = \begin{cases} \checkmark \text{ if } (s, M) \Rightarrow (s', V) \text{ for some } (s', V) \\ \mathtt{E} \text{ if } (s, M) \Downarrow^{u}_{\mathrm{error}} \\ \Omega \text{ if } (s, M) \Uparrow \end{cases}$$

Here, $(s, M) \Downarrow^u_{\text{error}}$ means that $(s, M) \Downarrow_{\text{error}}$ or $(s, M) \Downarrow^l_{\text{error}}$ for some l, and $\checkmark$, $\mathtt{E}$, and Ω are tokens that indicate normal termination, error, and divergence, respectively. Similarly, for any low-level term M, memory m, and layout w, we define $\text{Eval}_w(M, m)$ by:

$$\text{Eval}_w(M, m) = \begin{cases} \checkmark \text{ if } w \models (m, M) \Rightarrow (m', V) \text{ for some } (m', V) \\ \mathtt{E} \text{ if } w \models (m, M) \Downarrow^u_{\text{error}} \\ \Omega \text{ if } w \models (m, M) \Uparrow \end{cases}$$

where $w \models (m, A) \Downarrow^u_{\text{error}}$ means that $w \models (m, A) \Downarrow_{\text{error}}$ or $w \models (m, A) \Downarrow^a_{\text{error}}$ for some a. We obtain:

Proposition 3. *Let M be a high-level term and s a high-level store. Then:*

$$\text{P}(\text{Eval}(M, s) = \text{Eval}_w(M^{\downarrow}, s^{\downarrow}_w)) \geq \varrho$$

This statement is simpler and weaker than those above. It expresses that evaluating M from a store s and $M^{\downarrow}$ from a corresponding memory $s^{\downarrow}_w$ lead to the same outcome with probability at least ϱ. Here, an outcome is, coarsely, normal termination, error, or divergence. Note that the probability does not depend on δ; since (as explained in Section 5) we can often take $\varrho = 1$, we can then have that evaluating M from a store s and $M^{\downarrow}$ from a corresponding memory $s^{\downarrow}_w$ lead to the same outcome with probability 1.

6.3 Security: Mapping Contexts

Although we cannot hope to establish that every program of the high-level language is secure in some absolute sense, we would like to argue that compiling a program from the high-level language to the low-level language does not introduce vulnerabilities. In other words, we regard programs of the high-level language as security specifications, and expect that the corresponding programs of the low-level language conform to those specifications.

Our notion of security relies on the distinction between public and private locations (much as in [3]). As stated in Section 3, we assume that the set of locations Loc is the disjoint union of two sets, PubLoc and PriLoc, of *public* and *private* locations. We then say that a high-level (respectively low-level) term is *public* if all its occurrences of $!_{\mathtt{loc}}l_{\mathtt{loc}}$ and $l_{\mathtt{loc}} :=_{\mathtt{loc}}$ (respectively $l_{\mathtt{nat}}$) are with $l \in \text{PubLoc}$. We model attackers as public contexts, that is, as programs of our languages that interact with the programs that we aim to protect, and that have direct access to public locations.

Contexts (both public contexts and general contexts) have different capabilities in each language. In particular, in the high-level language, contexts can refer to abstract locations, while in the low-level language contexts can use integer addresses, and this might permit exploits that rely on guessing integer addresses, perhaps using offset calculations. So, if the contexts of the low-level language were much more expressive than those of the high-level language, security might

be jeopardized. We aim to show that, in fact, the extra flexibility of the low-level language does not affect security, at least with high probability.

Therefore, we show that a low-level term $M^{\downarrow}$ is as secure as its high-level counterpart M by arguing that the behavior $M^{\downarrow}$ in each public context C of the low-level language corresponds to the behavior of M in some corresponding public context $C^{\uparrow}$ of the high-level language. Since we model attackers as public contexts, this result indicates that, for every low-level attack, there is a corresponding high-level attack with the same effect.

In order to relate behaviors at the two levels, we introduce *pure* stores. A store is *pure* if it contains no location, assignment, or dereference. Therefore, every pure store is both a high-level and a low-level public store. For simplicity, in what follows, we consider only pure initial stores.

We obtain the following theorem:

Theorem 1. *Suppose that M is a high-level term and C is a public low-level term. Then $CM^{\downarrow}$ is a public low-level term, and there exists a public high-level term $C^{\uparrow}$ such that one of the following three mutually exclusive statements holds for any pure store s:*

- *there exist s', s'', V', and V'' such that, for all w, $w \models (s_w, CM^{\downarrow}) \Rightarrow (s'_w, V')$ and $(s, C^{\uparrow}M) \Rightarrow (s'', V'')$,*
- *$\mathrm{P}(w \models (s_w, CM^{\downarrow}) \Downarrow^{u}_{\mathrm{error}}) \geq min(\delta, \varrho)$ and $(s, C^{\uparrow}M) \Downarrow^{u}_{\mathrm{error}}$, or*
- *for all w, $w \models (s_w, CM^{\downarrow}) \Uparrow$ and $(s, C^{\uparrow}M) \Uparrow$.*

The probability bound as a function of δ and ϱ arises, basically, because a non-public, low-level memory access is made independently of the layout.

Using the coarse evaluation function again, we derive a weaker but simpler statement:

Corollary 1. *Suppose that M is a high-level term and C is a public low-level term. Then there exists a public high-level term $C^{\uparrow}$ such that, for any pure store s, we have:*

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

Intuitively, for every attack (represented by C) on $M^{\downarrow}$ there is a corresponding attack (represented by $C^{\uparrow}$) on M that leads to the same outcome (normal termination, error, or divergence).

6.4 Security: Preservation of Equivalences

We introduce a relation $\approx_{h,p}$ of *public (contextual) operational equivalence* for the high-level language, and a relation $\approx_{l,p}$ of *public (contextual) operational partial equivalence* for the low-level language. These two relations refine the corresponding standard relations of operational equivalence. Much as in other settings, they can be used for capturing security properties, such as those discussed informally in Section 2. Therefore, we aim to show that compilation preserves equivalences, at least in a probabilistic sense.

We define $\approx_{h,p}$ by setting, for any two high-level terms M_0 and N_0:

$$M_0 \sim_{h,p} N_0 \iff \forall \text{pure } s.\, \mathrm{Eval}(M_0, s) = \mathrm{Eval}(N_0, s)$$

and then, for any two high-level terms M and N:

$$M \approx_{h,p} N \iff \forall \text{public high-level } C.\, CM \sim_{h,p} CN$$

Thus, $M_0 \sim_{h,p} N_0$ means that M_0 and N_0 yield the same outcome from all pure high-level stores, and $M \approx_{h,p} N$ means the same in an arbitrary public context.

Although Eval produces only a coarse outcome, and although the definition of $\sim_{h,p}$ focuses on pure stores, the quantification over all public contexts leads to fine distinctions between terms. For instance, when n and n' are two different numbers, we have $n \sim_{h,p} n'$, but we do not have $n \approx_{h,p} n'$, simply because a context C may compare its argument to n, terminate if the comparison succeeds, and diverge otherwise. Similarly, we do not have $(l_{\mathtt{loc}} :=_{\mathtt{loc}} n) \approx_{h,p} (l_{\mathtt{loc}} :=_{\mathtt{loc}} n')$ if l is a public location, because a context C may read from $l_{\mathtt{loc}}$, compare the result to n, terminate if the comparison succeeds, and diverge otherwise. On the other hand, the equivalence $(l'_{\mathtt{loc}} :=_{\mathtt{loc}} n) \approx_{h,p} (l'_{\mathtt{loc}} :=_{\mathtt{loc}} n')$ does hold when l' is a private location, and captures a secrecy property.

Note that this equivalence would not hold if we had quantified over arbitrary stores rather than pure stores in the definition of $M_0 \sim_{h,p} N_0$. Let s be a store that maps the public location l to the function $\lambda x.\, !_{\mathtt{loc}} l'_{\mathtt{loc}}$. Let C be a context that consumes an argument, reads from $l_{\mathtt{loc}}$, applies its contents to $*$, compares the result to n, terminates if the comparison succeeds, and diverges otherwise. Then $\mathrm{Eval}(C(l'_{\mathtt{loc}} :=_{\mathtt{loc}} n), s)$ and $\mathrm{Eval}(C(l'_{\mathtt{loc}} :=_{\mathtt{loc}} n'), s)$ yield different outcomes ($\checkmark$ and Ω, respectively).

As this small example illustrates, the quantification over pure stores (rather than arbitrary stores) in the definition of $M_0 \sim_{h,p} N_0$ is crucial because stores may contain functions. In particular, in an arbitrary store, a public location l could contain functions that, when invoked, read or write private locations; a context could then read from l and use those functions to access private locations. Thus, assuming that there is at least one public location l, removing the restriction to pure stores would effectively erase the distinction between public and private locations, and would yield a standard relation of contextual equivalence.

Analogously, for the low-level language, for any M_0 and N_0, we say that $M_0 \sim_{l,p} N_0$ holds if and only if, for every pure store s, at least one of the following three possibilities holds:

- there exist s', s'', V', and V'' such that, for all w, $w \models (s_w, M_0) \Rightarrow (s'_w, V')$ and $w \models (s_w, N_0) \Rightarrow (s''_w, V'')$,
- $\mathrm{P}(w \models (s_w, M_0) \Downarrow^u_{\mathrm{error}}) \geq min(\delta, \varrho)$ and $\mathrm{P}(w \models (s_w, N_0) \Downarrow^u_{\mathrm{error}}) \geq min(\delta, \varrho)$, or
- for all w, $w \models (s_w, M_0) \Uparrow$ and $w \models (s_w, N_0) \Uparrow$.

This relation is a partial equivalence; as in [3], reflexivity may fail (because terms that branch on the concrete addresses of private locations do not behave

identically for all layouts). If $\delta > 0$ and $\varrho > 0$ then the three possibilities are mutually exclusive; also, if the first of them holds, then s', s'', V', and V'' are uniquely determined. Further, for any two low-level terms M and N, we set:

$$M \approx_{l,p} N \iff \forall \text{public low-level } C.\, CM \sim_{l,p} CN$$

The following theorem shows that compilation preserves and reflects equivalences:

Theorem 2. *Let M and N be high-level terms. If $M \approx_{h,p} N$, then $M^{\downarrow} \approx_{l,p} N^{\downarrow}$. The converse holds as well if $\delta > 0$ and $\varrho > 0$.*

7 Related and Further Work

The pioneering work of Pucella and Schneider treats a small C-like language with arrays, and relates obfuscation and type systems [15]. Their theorems focus on integrity properties, and do not explicitly mention probabilities. As explained by Abadi and Plotkin, those theorems basically pertain to protection from a potentially dangerous input, while we consider more general attackers, represented by arbitrary contexts, and also treat program equivalences that can express integrity and secrecy properties.

Our approach is most similar to that of Abadi and Plotkin (specifically, in their "fatal error" model) [3], though with several substantial differences. In particular, we treat a dynamically typed language (rather than a statically typed language), which gives us additional flexibility in the typing of memory. Furthermore, we allow arrays and functions to be stored in memory (rather than just integers). This extension enables the formulation of examples suggested by practical attacks. It also entails a number of complications and opportunities, such as the compilation of array operations, the quantification over pure stores in defining equivalences, and the consideration of guard regions.

The choice of an untyped language and the possibility of storing functions in memory appear also in the work of Jagadeesan et al. [9]. Their programming languages include not only functions but also continuations, a bounded form of local state, and some novel, non-standard constructs. In particular, they do not view addresses as integers even in low-level systems, but pointer arithmetic is available via encodings. On the other hand, the languages do not include arrays, nor the resulting concerns about overflows that appear prominently in this paper.

Despite these distinctions, all these works aim to contribute to the understanding of randomization in the context of programming languages and their implementations. There remain opportunities for research towards this goal. In particular, further work may treat additional constructs and models of computation, such as concurrency. It may also consider combinations of layout randomization with other techniques. Our use of guard regions, in this paper, is a small step in that direction. Other relevant techniques include stack canaries for protecting return addresses and various inline reference monitors that aim to guarantee control-flow integrity. All these techniques may be used in concert in practical systems; a principled study may be able to shed light on their synergies and overlaps.

Acknowledgments. We are grateful to Úlfar Erlingsson and to Gordon Plotkin for discussions on this work. Jérémy Planul's work is supported by DARPA PROCEED.

References

1. Abadi, M.: Protection in Programming-Language Translations. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 868–883. Springer, Heidelberg (1998)
2. Abadi, M., Planul, J.: On layout randomization for arrays and functions (2013), Long version of this paper, at `http://www.msr-inria.inria.fr/~jplanul/libraries-long.pdf`
3. Abadi, M., Plotkin, G.D.: On protection by layout randomization. ACM Transactions on Information and System Security 15(2), 8:1–8:29 (2012)
4. Chen, S., Sezer, E.C., Xu, J., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: Proceedings of the Usenix Security Symposium, pp. 177–192 (2005)
5. Druschel, P., Peterson, L.L.: High-performance cross-domain data transfer. Technical Report TR 92-11, Department of Computer Science, The University of Arizona (March 1992)
6. Erlingsson, Ú.: Low-Level Software Security: Attacks and Defenses. In: Aldini, A., Gorrieri, R. (eds.) FOSAD 2007. LNCS, vol. 4677, pp. 92–134. Springer, Heidelberg (2007)
7. Felleisen, M., Friedman, D.P.: Control operators, the secd-machine, and the lambda-calculus. In: 3rd Working Conference on the Formal Description of Programming Concepts, pp. 193–219 (1986)
8. Forrest, S., Somayaji, A., Ackley, D.H.: Building diverse computer systems. In: 6th Workshop on Hot Topics in Operating Systems, pp. 67–72 (1997)
9. Jagadeesan, R., Pitcher, C., Rathke, J., Riely, J.: Local memory via layout randomization. In: Proceedings of the 24th IEEE Computer Security Foundations Symposium, pp. 161–174 (2011)
10. McCamant, S., Morrisett, G.: Evaluating SFI for a CISC architecture. In: Proceedings of the 15th USENIX Security Symposium, pp. 209–224 (2006)
11. Mitchell, J.: Foundations for Programming Languages. MIT Press (1996)
12. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In: IEEE Symposium on Security and Privacy, pp. 601–615 (2012)
13. PaX Project. The PaX project (2004), `http://pax.grsecurity.net/`
14. Pierce, B.: Types and Programming Languages. MIT Press (2002)
15. Pucella, R., Schneider, F.B.: Independence from obfuscation: A semantic framework for diversity. Journal of Computer Security 18(5), 701–749 (2010)
16. Sotirov, A., Dowd, M.: Bypassing browser memory protections: Setting back browser security by 10 years (2008), `https://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf`
17. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. In: Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, pp. 203–216 (1993)