Local Memory via Layout Randomization

Radha Jagadeesan
DePaul University

Corin Pitcher
DePaul University

Julian Rathke
University of Southampton

James Riely
DePaul University

***Abstract*—Randomization is used in computer security as a tool to introduce unpredictability into the software infrastructure. In this paper, we study the use of randomization to achieve the secrecy and integrity guarantees for local memory.**

We follow the approach set out by Abadi and Plotkin (2010). We consider the execution of an idealized language in two environments. In the *strict* environment, opponents cannot access local variables of the user program. In the *lax* environment, opponents may attempt to guess allocated memory locations and thus, with small probability, gain access the local memory of the user program. We model these environments using two novel calculi: $\lambda\mu$hashref and $\lambda\mu$proberef.

Our contribution to the Abadi-Plotkin program is to enrich the programming language with dynamic memory allocation, first class and higher order references and call/cc-style control. On the one hand, these enhancements allow us to directly model a larger class of system hardening principles. On the other hand, the class of opponents is also enhanced since our enriched language permits natural and direct encoding of attacks that alter the control flow of programs.

Our main technical result is a fully abstract translation (upto probability) of $\lambda\mu$hashref into $\lambda\mu$proberef. Thus, in the presence of randomized layouts, the opponent gains no new power from being able to guess local references of the user program. Our numerical bounds are similar to those of Abadi and Plotkin; thus, the extra programming language features do not cause a concomitant increase in the resources required for protection via randomization.

I. Introduction

Randomization is used in computer security as a tool to introduce unpredictability. The aim is to harden the system against opponents who rely on precise deterministic assignment of resources. Examples include the following.

- Random canary values, as used in StackGuard (Cowan et al. 1999), add random values to the stack just before the return address. It is difficult for buffer overflow attacks to overwrite the return address without changing the canary.
- Instruction code randomization, as in (Portokalidis and Keromytis 2010; Kc et al. 2003; Barrantes et al. 2005), is intended to make it difficult for an attacker to carry out code injection attacks.
- Address space layout randomization (ASLR) randomizes the locations of the base of the executable, stack, libraries, etc. For example, the randomized location of libraries is intended to make it difficult for the opponent to carry out "return-to-libc" attacks. ASLR is implemented in systems such as PAX and Vista. (Shacham et al. 2004) studies the effectiveness of ASLR and also traces the history of discovery of the technique.

In this paper, we focus on the use of randomization to achieve secrecy and integrity guarantees for local memory in the presence of an active opponent. Strackx et al. (2009) argue that memory secrecy is essential even for traditional probabilistic countermeasures, such as the ones above, to provide the expected protection. For example, buffer overflow vulnerabilities that permit the opponent to read the private memory of a process can be exploited to weaken the protection provided by ASLR and stack canaries.

Our work is inspired by Abadi and Plotkin (2010), who studied the secrecy and integrity guarantees afforded by layout randomization. Abadi and Plotkin describe an idealized source language, which has the properties expected of local variables, and the translation of this source into a target language where addresses are represented by integers. They model opponents as programming language contexts in both source and target languages; thus, the opponent's power to distinguish user programs is modelled via contextual equivalence. Opponents in the target language can attempt to guess allocated memory locations and thus gain access to local memory of the user program. In this environment, equivalences expected by the programmer do not necessarily hold with deterministic layout.

Consider for the following programs.

$$\mathsf{new}\, x; x := 0; \mathsf{unit} \qquad \mathsf{new}\, x; x := 1; \mathsf{unit} \qquad (*)$$

The programs $(*)$ are equivalent in the source language, because an opponent context in the source language is unable to see the different values assigned to the local variable x. With a deterministic layout strategy, the programs are distinguished in the target language, because an opponent context in the target language can guess the location of the local variable x and read its contents.

The secrecy and integrity of local references can be recovered in target-language contexts using randomized layout. The idea is to use a larger address space than necessary (say twice the maximum required by the user program) and to allocate memory using a stochastic process. In such a scenario, an opponent is unable to find the locations that store the private user program data with high probability. For a suitably large memory, Abadi and Plotkin show that their translation preserves contextual equivalences with high probability; that is, the opponent can only access local user variables with low probability.

In this paper, we revisit the Abadi-Plotkin program in the context of a programming language enriched with dynamic memory allocation, first-class and higher-order references, and control operators.

These enhancements allow us to model a larger class of system hardening principles. For example, first class references permit us to simply encode a model of instruction set

randomization.

The class of opponents is also enhanced. The control operators allow a natural and direct encoding of gotos and therefore of attacks that alter the control flow of programs.

In contrast with Abadi and Plotkin, we permit dynamic growth in the opponent's knowledge of user memory via first-class references. However, we still do not address deallocation nor contiguous memory allocation, which is required to model arrays and structures.

Our view of memory allocation is rich enough to support the reasoning involved in several of the traditional Meyer-Sieber examples (1988), which illustrate the properties of local state in higher order programs.

Concretely, the starting point for our source language is the $\lambda\mu$-calculus of Parigot (1992), which adds the control operator μ to the λ-calculus. The $\lambda\mu\rho$-calculus of Støvring and Lassen (2007) adds second class local state to $\lambda\mu$. Let $\lambda\mu$ref denote the obvious variation of $\lambda\mu\rho$ using ML-style references. In $\lambda\mu$ref, references are first class; the language includes operations to allocate new references, to read and write references and to distinguish references from other values (since the language is untyped).

$\lambda\mu$ref is not suitable for modeling implementation techniques using pointers. We introduce two new calculi which address these shortcomings. The $\lambda\mu$hashref-calculus extends $\lambda\mu$ref with null and hashref. The $\lambda\mu$proberef-calculus extends $\lambda\mu$hashref with proberef.

- null represents a "bad location" or "unallocated reference".
- hashref allows testing the hash of a reference against any possible reference hash.
- proberef allows access to any reference, reversing its hash.

The hashref operator models the ability of the attacker to compare references. The proberef operator models the ability to access memory locations that are local to the user programs (and therefore should be inaccessible).

In our execution model for $\lambda\mu$proberef, the attacker does not know the layout of memory. For a memory of size N, there are $N!$ possible layouts. A nondeterministic interpretation of proberef provides too much power to the attacker. For example, the attacker is able to distinguish the programs $(*)$ given above, since there always exist layouts where the attacker's guess is successful. Instead, we exploit randomized layout to provide a probabilistic interpretation of proberef. Roughly, a proberef succeeds with probability of $\frac{1}{N}$ and fails with the probability of $1-\frac{1}{N}$. Failures are not fatal; an attacker continues to execute after a failed proberef.

Our main technical result is a fully abstract translation (upto probability) of $\lambda\mu$hashref into $\lambda\mu$proberef, thus recovering the results of Abadi and Plotkin for a richer language. In the presence of randomized layouts, the opponent gains no new power from being able to guess local references of the user program. Our results prescribe a relationship between the probability that an execution path succeeds, the number of failed attacker guesses on that path, and the size of memory. These bounds are essentially the same as those of Abadi and Plotkin, so the extra programming language features do not come at a great cost.

In the next section, we introduce the source language, $\lambda\mu$hashref, by example. We follow with formalities in Section III. In Section IV, we introduce the target language, $\lambda\mu$proberef. We introduce an alternative characterization of contextual equivalence in Section V and prove full abstraction in Section VI.

II. $\lambda\mu$hashref-CALCULUS: EXAMPLES

The allocation model for $\lambda\mu$hashref differs from idealized models of allocation in two ways: the number of allocations is bounded and the allocation is deterministic. These differences impact the derived theory of contextual equivalence. In this section we provide an informal account of the allocation model and its application to examples that benefit from layout randomization.

We use several example programs due to (Meyer and Sieber 1988); their examples were originally used to demonstrate the lack of full abstraction of different semantic models of languages with idealized allocation. Perhaps surprisingly, many of Meyer and Sieber's examples can be modified to account for a bounded and deterministic allocation model (and control features arising from $\lambda\mu$). This suggests that the bounded and deterministic allocation model yields a useful theory of local state, albeit distinct from that of the usual unbounded and non-deterministic allocation model.

Our full abstraction result in the sequel establishes that the bounded and deterministic allocation model is sound and complete for reasoning about an allocation model with opponents that can probe arbitrary memory locations but are hindered by layout randomization.

A. Bounded allocation

The bounded allocation model of $\lambda\mu$hashref invalidates contextual equivalences that hold for unbounded allocation models, because the number of allocations performed by a program can be observed indirectly.

Example 1 (Meyer-Sieber example 1). The first Meyer-Sieber example states that $(\mathsf{new}\, x; t)$ and t are equivalent for any program t where x is not free in t; that is, allocation has no observable effect if the reference is unused in the program. However, such equivalences do not hold in $\lambda\mu$hashref because allocation is bounded. For example, $(\mathsf{new}\, x; \mathsf{unit}) \not\equiv \mathsf{unit}$, where unit is the unit value. If the size of the store is N, these programs are distinguished by the context

$$D = \mathsf{new}\, y_1; \ldots; \mathsf{new}\, y_N; [-]$$

which performs N allocations. Then $D[\mathsf{new}\, x; \mathsf{unit}]$ ends in an error at the $(N+1)^{\text{th}}$ allocation, but $D[\mathsf{unit}]$ returns unit. ■

B. Control

The security of a program depends upon the entry points for an opponent (compare with *attack surface*, see (Manadhata and Wing 2004)). The entry points for an opponent in the λ-calculus include invoking the entire program, invoking

functional arguments during a callback, and returning from a callback (where callback refers to a call from the program to the opponent). The last of these demonstrates an asymmetry between calls and returns: an opponent cannot return from a callback more than once. There is a mismatch between such an asymmetric constraint on an opponent's capabilities and the use of jumps seen in attacks. For this reason, we adopt the control primitive from Parigot's $\lambda\mu$ (1992) and thus allow modelling of jumps, which in turn strengthens opponents.

Example 2 (Meyer-Sieber example 2). Consider the following program, where Ω is a divergent term.

$$\lambda f\,.\,\mathsf{new}\,x;\, x{:=}0;\, f\,\mathsf{unit};\, \mathsf{if}\ !x \neq 0\ \mathsf{then}\ \Omega\ \mathsf{else}\ \mathsf{unit}$$

In an unbounded allocation model without control, this program diverges when applied to any argument, because the function f cannot modify the contents of the local reference x. This does not hold in the presence of $\lambda\mu$'s control primitive, however, since f may transfer control to a μ name and never return to the conditional; that is, we must compare against $(f\,\mathsf{unit};\Omega)$ rather than Ω. Moreover, as in Example 1 the number of allocations must be equivalent whenever the opponent f receives control; that is, we must compare against $(\mathsf{new}\,x;\, f\,\mathsf{unit};\,\Omega)$. Thus we have the following equivalence as an adaptation of Meyer and Sieber's example.

$$\begin{array}{l}\mathsf{new}\,x;\, x{:=}0;\, f\,\mathsf{unit};\, \mathsf{if}\ !x \neq 0\ \mathsf{then}\ \Omega\ \mathsf{else}\ \mathsf{unit}\\ \simeq\\ \mathsf{new}\,x;\, f\,\mathsf{unit};\,\Omega\end{array}$$

■

Example 3 illustrates a more subtle attack against local state using control.

Example 3 (Multiple returns via control). The following programs can be distinguished via $\lambda\mu$'s control.

$$\begin{array}{l}\lambda f\,.\,\mathsf{new}\,x;\, x{:=}0;\, f\,\mathsf{unit};\, \mathsf{if}\ !x \neq 0\ \mathsf{then}\ \Omega\ \mathsf{else}\ (x{:=}1;\,\mathsf{unit})\\ \not\simeq\\ \lambda f\,.\,\mathsf{new}\,x;\, x{:=}0;\, f\,\mathsf{unit};\, \mathsf{if}\ !x \neq 0\ \mathsf{then}\ \Omega\ \mathsf{else}\ \mathsf{unit}\end{array}$$

In the first program, the continuation of the call to f performs an update to x. In $\lambda\mu\mathsf{hashref}$, the programs can be distinguished by an opponent that invokes the continuation twice. Using the first program, the second invocation of the continuation diverges because of the update to x. ■

C. Deterministic allocation and reference hashes

The choice of allocation model affects the ability of an opponent to make inferences about references. In particular, non-deterministic allocation may prevent attacks that are to be addressed by layout randomization. To avoid obscuring the behavior of layout randomization, we adopt a deterministic allocation model: the deterministic allocation operator takes the first reference from a list of unallocated references. This list of unallocated references is established before the program is executed.

We assume that opponents have knowledge of the list of unallocated references. This is formalized using a contextual equivalence that quantifies over all lists of unallocated references (as well all contexts, representing opponent behavior). Two programs are compared by execution in identical contexts with identical lists of unallocated references.

However this is not sufficient for an opponent to take advantage of the deterministic allocation model. We first observe that contexts (for example, in the ν-calculus (Pitts and Stark 1993)) are usually not permitted to contain unallocated references because information hiding via local state would be impossible. The problem then is that although an attacking context may depend upon the list of unallocated references, an attacking context cannot take advantage of such a dependency. For example, it would not be possible for an attacking context to test whether a reference received at runtime is equal to the first reference from the initial list of unallocated references.

We resolve this problem by creating a distinction between references and reference hashes that can be used in equality tests. We assume a bijection between references and reference hashes. Hashes of references from the initial list of unallocated references are permitted to occur in an attacking context. Reference hashes cannot be used to read or write to the corresponding reference cell—only a reference can be used to access memory. Reference hashes can only appear in equality tests using the operator $\mathsf{hashref}$. Given a reference hash h and a reference r, the expression $\mathsf{hashref}_h\, r$ is true iff h is the reference hash for r. Example 4 demonstrates the consequence of $\mathsf{hashref}$ with deterministic allocation: leaked references are not opaque to an opponent, and thus the order of allocation is observable.

Example 4 (Deterministic allocation). The following two programs are not equivalent.

$$\mathsf{new}\,x;\, \mathsf{new}\,y;\, x \ \not\simeq\ \mathsf{new}\,x;\, \mathsf{new}\,y;\, y$$

To see why this equivalence does not hold, consider the list of unallocated references: $r_1, r_2, \ldots, r_N$. Then a distinguishing context for the programs above is

$$D \triangleq \mathsf{let}\,z{=}[\text{–}]\,\mathsf{in}\,(\mathsf{hashref}_{r_1}\, z).$$

The program $D[\mathsf{new}\,x;\, \mathsf{new}\,y;\, x]$ always evaluates to the boolean *true* using the allocation order $r_1, r_2, \ldots, r_N$, since $\mathsf{new}\,x$ will always allocate r_1. In contrast, $D[\mathsf{new}\,x;\, \mathsf{new}\,y;\, y]$ will always evaluate to the boolean *false*.

Note that the syntax of $\lambda\mu\mathsf{hashref}$ does not distinguish references and reference hashes. Instead the subscripted argument to $\mathsf{hashref}$ is not constrained by the usual scoping rules for names, and the r_1 in $\mathsf{hashref}$ should be considered a hash. ■

A similar distinction between references and integers (representing reference hashes) can be found in Java. Java's type system prevents conversions from numeric types to references, but the hashCode method of Java's Object class is "typically implemented by converting the internal address of the object into an integer" (Java 6 API). With such a hashCode implementation it is possible to illustrate the essence of Example 4. Consider the following Java class.

```
public class A {
public static void main (String[] args) {
 Object x=new Object(); int i=x.hashCode (); // (1)
 Object y=new Object(); int j=y.hashCode (); // (2)
 System.out.println(
   Integer.toHexString(i) + "," + Integer.toHexString(j));
}}
```

The Java class B is defined with the same code except that lines (1) and (2) are interchanged. Running class A with Java 1.6.0_23 on an Ubuntu 10.04 x64 system yielded 0x4830c221, 0x7919298d on 99 out of 100 runs. Running class B yielded 0x7919298d, 0x4830c221 on 98 out of 100 runs. This particular implementation has an allocation model that is more deterministic than might be expected, and this determinism can be observed using hashCode.

Finally, we turn to the third Meyer-Sieber example. This example fails in $\lambda\mu$hashref because an opponent can observe the allocation order of references that it receives.

Example 5 (Meyer-Sieber example 3). The following two programs are not equivalent.

$$\begin{array}{l}\lambda f.\,\mathsf{new}\,x;\,\mathsf{new}\,y;\,x:=0;\,y:=0;\,f\,x\,y\\ \not\simeq\\ \lambda f.\,\mathsf{new}\,x;\,\mathsf{new}\,y;\,x:=0;\,y:=0;\,f\,y\,x\end{array}$$

A distinguishing context applies the context hole to a function using hashref, similarly to Example 4. ■

D. Local state

In spite of deterministic allocation order, allocation still yields local state that is not directly accessible by a context if its reference is kept secret. Thus a $\lambda\mu$hashref context may test references using hashref but cannot manufacture references. This distinguishes $\lambda\mu$hashref from $\lambda\mu$proberef.

Example 6 (Local state). Allocation order is unimportant when local state is accessed by an opponent via reader/writer functions instead of references as in the following example, where $(\_,\_)$ denotes a pair and $\lambda.\,t$ abbreviates $\lambda z.\,t$ where z does not occur free in t.

$$\begin{array}{l}\mathsf{new}\,x;\,\mathsf{new}\,y;\,\big((\lambda.\,!x),(\lambda z.\,x:=z)\big)\\ \simeq\\ \mathsf{new}\,x;\,\mathsf{new}\,y;\,\big((\lambda.\,!y),(\lambda z.\,y:=z)\big)\end{array}$$ ■

Importantly, the form of local state in $\lambda\mu$hashref *does* support invariants.

Example 7 (Meyer-Sieber example 5). The fifth Meyer-Sieber equivalence holds in $\lambda\mu$hashref.

$$\begin{array}{l}\lambda f.\,(\mathsf{new}\,x;\,\mathsf{let}\,\mathit{addtwo}=(\lambda.\,x:=2+\,!x)\,\mathsf{in}\,x:=0;\,f\,\mathit{addtwo};\\ \qquad \mathsf{if}\,(\,!x \bmod 2)=0\,\mathsf{then}\,\Omega\,\mathsf{else}\,\mathsf{unit})\\ \simeq\\ \lambda f.\,(\mathsf{new}\,x;\,\mathsf{let}\,\mathit{addtwo}=(\lambda.\,x:=2+\,!x)\,\mathsf{in}\,x:=0;\,f\,\mathit{addtwo};\\ \qquad \Omega)\end{array}$$

The invariant that x is even is maintained because an opponent f can only modify x by calling *addtwo*. ■

Full abstraction tells us that Example 6 and Example 7 hold for the probabilistic contextual equivalence of $\lambda\mu$proberef where attackers may attempt to guess and modify x and y inside f.

E. Safety of a stored return address

Many attacks upon programs involve overwriting an area of memory containing the address of code to jump to; for example, by overwriting a stored return address on the call stack via a buffer overflow (Levy 1996). These attacks are sensitive to (a) the opponent's ability to inject data into memory; and (b) the memory layout used by the program (partly determined by the implementation of the programming language).

Example 8 formalizes a simple safety property for systems that store return addresses in memory, where return addresses are modelled as functions that transfer control via $\lambda\mu$. In this example, we (a) permit opponents to execute arbitrary code to inject data into memory, as opposed to modelling buffer overflows; and (b) make the relevant memory of the program (the stored return address) *explicit* in the program as local state.

Example 8 (Safety of stored return address). The following equivalent programs both allocate a reference *return*, call a function f, and jump with the result to a free μ-name *previous*. In the first program, the jump is via a function stored at *return*. In the second program, the jump is direct and *return* is unused. In both programs, the call to f could fail to return due to divergence or a jump to a μ-name.

$$\begin{array}{l}\lambda f.\,\lambda x.\,\mathsf{new}\,\mathit{return};\\ \qquad \mathit{return}:=(\lambda y.\,\mu a.\,[\![\mathit{previous}]\!]\,y);\\ \qquad \mathsf{let}\,\mathit{result}=(f\,x)\,\mathsf{in}\\ \qquad !\,\mathit{return}\,\mathit{result}\\ \simeq\\ \lambda f.\,\lambda x.\,\mathsf{new}\,\mathit{return};\\ \qquad \mathsf{let}\,\mathit{result}=(f\,x)\,\mathsf{in}\\ \qquad \mu a.\,[\![\mathit{previous}]\!]\,\mathit{result}\end{array}$$

In these programs the function f may be an opponent. Taken in conjunction with our full abstraction result, the equivalence above establishes that a "return address" can be safely stored in memory with randomized allocation layout. This holds even in the presence of opponent code — injected via f — that can probe and modify arbitrary memory locations. ■

More generally, Example 8 suggests a strategy for formalizing safety against attacks on the memory locations used for a chosen compilation strategy and runtime system. For example, one might also investigate explicit representations of call stacks or closures using local state, by analogy with the explicit representation of a return location presented above. We speculate that the congruence of equivalence would facilitate the verification of safety for compositional translations that transform a program into its explicit representation, e.g., introducing closures.

F. Instruction set randomization

We now consider formalization of a basic property of instruction set randomization using program equivalence.

Example 9 (Instruction set randomization). We model an instruction set as a collection of locally-allocated references and a function that interprets an instruction. References representing instructions are kept secret from attackers. The interpretation function uses reference-equality tests to determine the code to execute for an instruction. If the instruction received does not match any of the known instructions, control is thrown to a μ name *invalid*, modelling an exception.

In the first function below, the ith instruction $inst_i$ is executed using the closed term t_i. This is equivalent to the second function, which always throws control to *invalid*.

$$\begin{array}{l}\mathsf{new}\, inst_1; \ldots; \mathsf{new}\, inst_n;\\ \lambda x.\, \mathsf{if}\ \mathsf{hashref}_{inst_1}\, x\ \mathsf{then}\ t_1\ \mathsf{else}\\ \quad \ldots\\ \quad \mathsf{if}\ \mathsf{hashref}_{inst_n}\, x\ \mathsf{then}\ t_n\ \mathsf{else}\\ \quad \mu a.\, [\![invalid]\!]\,\mathsf{unit}\\ \simeq\\ \mathsf{new}\, inst_1; \ldots; \mathsf{new}\, inst_n;\\ \lambda x.\, \mu a.\, [\![invalid]\!]\,\mathsf{unit}\end{array}$$

This equivalence demonstrates that an opponent cannot manufacture instructions for the interpretation function in $\lambda\mu\mathsf{hashref}$. Using full abstraction, we conclude that the interpretation function is resistant to $\lambda\mu\mathsf{proberef}$ attackers that guess instructions (encoded as references).

Here we are using random allocation only to generate random numbers. These happen to be locations, but they are used here like names in the ν-calculus (Pitts and Stark 1993). More generally, programs that store instructions in memory would also rely upon the fact that we have *layout* randomization in particular, as opposed to any other type of randomization, in order to be resistant to $\lambda\mu\mathsf{proberef}$ attackers that probe memory to find stored instructions. ■

III. The $\lambda\mu$hashref-calculus: formalities

Here we provide formal underpinnings for the examples given in the previous section. We give the syntax and evaluation semantics for the $\lambda\mu\mathsf{hashref}$-calculus and define the contextual equivalence ($\simeq$) used in Section II.

A. Syntax

We assume disjoint sets of *variables* (x, y, z), *names* (a, b) and *locations* (ι, κ). Names include the reserved name error. The syntax of *named terms* is built up using definitions for *references*, *values* and *terms*. To avoid syntactic bloat, we identify references and reference hashes. We define hashref (and proberef) as reference-indexed families of operations, with the index being the "hash" of a reference which otherwise may not be accessible.

The basic syntax is from the untyped $\lambda\mu$-calculus, as reported by Støvring and Lassen (2007); we refer the reader there for motivations and examples. To this basis we add one new form for values and five new forms for terms.

$$\begin{array}{rcll} r, q & ::= & \iota \mid \mathsf{null} & \text{(References)}\\ v, w & ::= & r \mid x \mid \lambda x.\, t & \text{(Value)}\\ t, u & ::= & v \mid v_1\, v_2 \mid \mathsf{let}\, x{=}t_1\, \mathsf{in}\, t_2 \mid \mu a.\, nt & \text{(Term)}\\ & & \mid \mathsf{alloc} \mid\, !\, v \mid v_1 := v_2 \mid \mathsf{isref}\, v & \\ & & \mid \mathsf{hashref}_r\, v & \\ nt & ::= & [\![a]\!]t & \text{(Named term)}\\ E & ::= & [-] \mid \mathsf{let}\, x{=}E\, \mathsf{in}\, t & \text{(Evaluation context)}\\ NE & ::= & [\![a]\!]E & \text{(Named evaluation context)}\end{array}$$

From ML, we add first class locations with dynamic allocation. If memory is not full, the alloc operator allocates and returns a new location, initialized to null; otherwise alloc blocks (see Example 12, below). The ! and := operators read and write locations, as usual. The isref operator returns true if its argument is a reference and false otherwise, using Church's encoding of the Booleans.

$\lambda\mu\mathsf{hashref}$ extends this basic setup with a value to represent nonallocated memory, null, and a family of operators $\mathsf{hashref}_r$ for comparing the "hash" of r to the "hash" of a program value. References include null; we use the term *location* for non-null references.

There are no binders for references. References may occur in terms as values or as indices in hashref. We define functions *vrefs* and *hrefs* which determine the sets of references occurring in these two roles. For example, $vrefs(\mathsf{let}\, x{=}\,!\, r\, \mathsf{in}\, x := q) = \{r, q\}$, $hrefs(\mathsf{let}\, x{=}\,!\, r\, \mathsf{in}\, x := q) = \emptyset$, $vrefs(\mathsf{hashref}_r\, q) = \{q\}$, and $hrefs(\mathsf{hashref}_r\, q) = \{r\}$.

The variable x is bound in value $\lambda x.t$ with scope t. The variable x is bound in term $\mathsf{let}\, x{=}t_1\, \mathsf{in}\, t_2$ with scope t_2. The name a is bound in term $\mu a.nt$ with scope nt. The reserved name error may not be bound. We identify syntax up to renaming of bound variables and names. We adopt the standard definitions of *free variables* (notation *fv*) and *free names* (notation *fn*). A term t is *closed* if $fv(t) = fn(t) = \emptyset$.

We write $t\{\!\{{}^{v}/_{x}\}\!\}$ for the capture-avoiding substitution of v for x in t, and $nt\{\!\{{}^{NE}/_{a}\}\!\}$ for the capture-avoiding substitution of NE for a in nt. The definition of substitution extends to other syntactic categories in the obvious way. In an evaluation context, $[-]$ represents the hole. We write $NE[t]$ for the term which results by replacing the hole with t.

In examples, we adopt the standard precedence rules and other conventions of the lambda calculus. We use conventional abbreviations. For example $(t_1; t_2)$ stands for $(\mathsf{let}\, x{=}t_1\, \mathsf{in}\, t_2)$ where $x \notin fv(t_2)$. Similarly, $(\lambda .\, t)$ stands for $(\lambda x.t)$ where $x \notin fv(t)$. $(\mathsf{new}\, x; t)$ stands for $(\mathsf{let}\, x{=}\mathsf{alloc}\, \mathsf{in}\, t)$. We use the following abbreviations.

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{tru} & \triangleq & \lambda x.\, \lambda y.\, x & \mathsf{err} & \triangleq & \mu a.\, [\![\mathsf{error}]\!]\,\mathsf{null}\\ \mathsf{fls} & \triangleq & \lambda x.\, \lambda y.\, y & \mathsf{unit} & \triangleq & \lambda .\, \mathsf{err}\\ \multicolumn{3}{r@{\qquad}}{\mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ u \triangleq v\, (\lambda .\, t)\, (\lambda .\, u)\,\mathsf{unit}} & \Omega & \triangleq & (\lambda x.\, xx)\, (\lambda x.\, xx)\end{array}$$

We use other notations in examples, such as integers and pairs, which can be encoded in the λ-calculus. We often elide explicit sequencing, using terms in the place of values; these should be interpreted left to right, for example $(t_1\, t_2)$

$$
\begin{array}{lll}
\langle s, NE[\mathsf{let}\, x = v\, \mathsf{in}\, t]\rangle \longrightarrow. & \langle s, NE[t\{\!\{^{v}/_{x}\}\!\}]\rangle & \\
\langle s, NE[(\lambda x.t)\, v]\rangle \longrightarrow. & \langle s, NE[t\{\!\{^{v}/_{x}\}\!\}]\rangle & \\
\langle s, NE[\mu a.nt]\rangle \longrightarrow. & \langle s, nt\{\!\{^{NE}/_{a}\}\!\}\rangle & \\
\langle s, NE[\mathsf{alloc}]\rangle \longrightarrow_{\mathbf{A}_\iota} & \langle s \uplus \{\iota := \mathsf{null}\}, NE[\iota]\rangle & \text{if } \eta(s) = \iota \\
\langle s, NE[\iota := v]\rangle \longrightarrow_{\mathbf{W}_{\iota,v}} & \langle s[\iota := v], NE[v]\rangle & \\
\langle s, NE[\mathsf{null} := v]\rangle \longrightarrow_{\mathbf{W}_{\mathsf{null},v}} & \langle s, NE[\mathsf{err}]\rangle & \\
\langle s, NE[!\,\iota]\rangle \longrightarrow_{\mathbf{R}_\iota} & \langle s, NE[v]\rangle & \text{if } s(\iota) = v \\
\langle s, NE[!\,\mathsf{null}]\rangle \longrightarrow_{\mathbf{R}_{\mathsf{null}}} & \langle s, NE[\mathsf{err}]\rangle & \\
\langle s, NE[\mathsf{isref}\, v]\rangle \longrightarrow. & \langle s, NE[\mathsf{tru}]\rangle & \text{if } \mathit{isref}(v) \\
\langle s, NE[\mathsf{isref}\, v]\rangle \longrightarrow. & \langle s, NE[\mathsf{fls}]\rangle & \text{if } \neg \mathit{isref}(v) \\
\langle s, NE[\mathsf{hashref}_r\, v]\rangle \longrightarrow. & \langle s, NE[\mathsf{tru}]\rangle & \text{if } r = v \\
\langle s, NE[\mathsf{hashref}_r\, v]\rangle \longrightarrow. & \langle s, NE[\mathsf{fls}]\rangle & \text{if } r \neq v
\end{array}
$$

Fig. 1. Evaluation in $\lambda\mu\mathsf{hashref}$ $(\eta \vdash \mathscr{C} \to_\alpha \mathscr{C}')$

stands for $(\mathsf{let} x_1 = t_1\, \mathsf{in}\, (\mathsf{let} x_2 = t_2\, \mathsf{in}\, x_1\, x_2))$, where $x_1 \notin fv(t_2)$. A usual names are not first class, but can be encoded: $\hat{a} \triangleq (\lambda x.\, \mu b.\, [\![a]\!]x)$.

B. Evaluation and contextual equivalence

Evaluation is defined using stores, allocation orders, configurations and annotations. Locations can store both references and functions, not simply integers.

$$
\begin{array}{rll}
s & ::= \{\iota_1 := v_1, \ldots, \iota_n := v_n\} & \text{(Store)} \\
\eta & ::= \overline{\iota_1, \ldots, \iota_N} & \text{(Allocation order)} \\
\mathscr{C} & ::= \langle s, nt\rangle & \text{(Configuration)} \\
\alpha & ::= \cdot \mid \mathbf{A}_r \mid \mathbf{W}_{r,v} \mid \mathbf{R}_r & \text{(Annotation)}
\end{array}
$$

A *store* is a partial map from locations to values. We adopt standard notation for partial maps. If $s = \{\iota_1 := v_1, \ldots, \iota_n := v_n\}$, then define $dom(s) \triangleq \{\iota_1, \ldots, \iota_n\}$. If s and s' have disjoint domains, then $s \uplus s'$ denotes their disjoint union. If $s = s' \uplus \{\iota := v\}$ then define $s(\iota) \triangleq v$ and $s[\iota := v'] \triangleq s' \uplus \{\iota := v'\}$.

Our semantics is parameterized with respect to an *allocation order* η, which is a sequence of locations. Suppose $\eta = \overline{\iota_1, \ldots, \iota_N}$. If $dom(s) = \{\iota_1, \ldots, \iota_k\}$ and $k < N$ then define $\eta(s) \triangleq \iota_{k+1}$, otherwise $\eta(s) \triangleq \mathsf{null}$.

Define $\mathit{isref}(v)$ to be true whenever v is a reference and false otherwise.

A *configuration* $\mathscr{C}$ is a pair of a store and a named term.

An *annotation* α marks information about how the term has reduced. Annotations are used to define probabilities and to state composition/decomposition lemmas. They can be ignored on first reading. Most reductions are annotated with the silent annotation "$\cdot$", which often elide, writing "$\longrightarrow.$" as "$\longrightarrow$". The annotation $\mathbf{A}_r$ indicates that r has been allocated, $\mathbf{W}_{r,v}$ indicates that r has been written with value v, and $\mathbf{R}_r$ indicates that r has been read.

Evaluation is defined in Figure 1 as an annotated relation on configurations, parameterized by an allocation order $(\eta \vdash \mathscr{C} \longrightarrow_\alpha \mathscr{C}')$. Since η is constant throughout, we have elided it from the rules. Let $\Longrightarrow$ be the reflexive transitive closure of $\longrightarrow_\alpha$, ignoring annotations.

Define $\eta \vdash nt \downarrow a$ to mean that there exists s such that

$$\eta \vdash \langle \emptyset, nt\rangle \Longrightarrow \langle s, [\![a]\!]\mathsf{null}\rangle.$$

$$
\begin{array}{ll}
\langle s, NE[\mathsf{proberef}_r]\rangle \longrightarrow_{\mathbf{Y}_r} \langle s, NE[r]\rangle & \text{if } r \in dom(s) \\
\langle s, NE[\mathsf{proberef}_r]\rangle \longrightarrow_{\mathbf{N}_r} \langle s, NE[\mathsf{null}]\rangle & \text{if } r \notin dom(s)
\end{array}
$$

Fig. 2. Additional evaluation rules for $\lambda\mu\mathsf{proberef}$

Definition 10 (Contextual equivalence for $\lambda\mu\mathsf{hashref}$). Closed values v and w are *contextually equivalent with memory size N* (notation $v \simeq^N w$) if for any allocation order η such that $|\eta| \geq N$ and for any named evaluation context NE such that $\mathit{vrefs}(NE) = \emptyset$ and $\mathit{hrefs}(NE) \subseteq \eta$, and for any $a \in fn(NE)$,

$$\eta \vdash NE[v] \downarrow a \text{ implies } \eta \vdash NE[w] \downarrow a$$

and symmetrically for w. We extend this notion to closed terms by defining $t \simeq^N u$ whenever $\lambda.t \simeq^N \lambda.u$. ■

We drop the script on $\simeq^N$ when it is universally quantified (for $N > 1$) or can be inferred from context.

We have defined contextual equivalence on values using evaluation contexts. This is equivalent to the definition using more general contexts D, since $\langle s, [\![a]\!]\mathsf{let} x = v\, \mathsf{in}\, D[x]\rangle \longrightarrow \langle s, [\![a]\!]D[v]\rangle$ This observation can be elaborated to prove the following.

Proposition 11. $\simeq$ is a congruence on terms. ■

We conclude this section with some short examples to explain the status of errors in the language.

Example 12. Unlike Abadi and Plotkin, we allow null references. Consequently we have the following.

$$\lambda x.\, \Omega \;\not\simeq\; \lambda x.\, !x; \Omega$$

The distinguishing context applies the function to null.

Once all memory is allocated, alloc blocks.

$$\mathsf{alloc}; \mathsf{unit} \;\not\simeq\; \mathsf{unit} \qquad\qquad \mathsf{alloc}; \Omega \;\simeq\; \Omega$$

As noted in Example 1, given memory η, the distinguishing context allocates $|\eta|$ references, then runs the term.

Other behaviors for alloc on a full memory are possible, with subtle effects on the equivalence. If alloc returns null, then both pairs of terms above are equated. If alloc evaluates to err, then both pairs are distinguished. Our results are easily adapted to these alternative semantics. ■

IV. The $\lambda\mu$proberef-calculus

In this section, we define $\lambda\mu$proberef and its probabilistic semantics. We first present the minor syntactic changes. We then provide a series of examples to show that the standard contextual equivalence is very fine in $\lambda\mu$proberef with deterministic layout. This motivates the study of randomized layouts. To calculate the relevant probabilities, we must record the set of references known to the environment. We describe the encoding of knowledge in Section IV-C and the calculation of probabilities in Section IV-D. Finally we define probabilistic contextual equivalence in Section IV-E.

A. Syntax

$\lambda\mu$proberef includes a new reference-indexed family of operators. The evaluation rules for the new construct are given in Figure 2 using two new annotations.

$$\begin{array}{rll} t, u & ::= \cdots \mid \mathsf{proberef}_r & \text{(Unnamed term)} \\ \alpha & ::= \cdots \mid \mathbf{Y}_r \mid \mathbf{N}_r & \text{(Annotation)} \end{array}$$

Following hashref, we have $\mathit{vrefs}(\mathsf{proberef}_r) = \emptyset$, and $\mathit{hrefs}(\mathsf{proberef}_r) = \{r\}$. The annotation $\mathbf{Y}_r$ that r has been probed with a yes response, and $\mathbf{N}_r$ indicates that r has been probed with a no response.

B. Examples

The next two examples demonstrate that references with hashref/proberef and deterministic allocation are no more abstract than pointers, which represent memory locations as integers. We distinguish references from other values in order to compute the knowledge of the context, where pointers must be distinguished from other integers. This can also be accomplished by tagging integers with provenance information to indicate whether it is a pointer or not. The approach we have chosen is equivalent, but easier to work with.

Example 13. Given allocation order $\overline{\iota_1, \ldots, \iota_N}$, one can encode pointer arithmetic. For example, $x+1$ can be written

$$\begin{array}{l} \mathsf{if}\ \mathsf{hashref}_{\iota_1}\, x\ \mathsf{then}\ \iota_2\ \mathsf{else} \\ \mathsf{if}\ \mathsf{hashref}_{\iota_2}\, x\ \mathsf{then}\ \iota_3\ \mathsf{else} \\ \cdots \\ \mathsf{if}\ \mathsf{hashref}_{\iota_N}\, x\ \mathsf{then}\ \iota_1\ \mathsf{else}\ \mathsf{err}. \end{array}$$

■

Example 14. Consider the *exposed* terms $(\mathsf{new}\, x; x := \mathsf{null}; x)$ and $(\mathsf{new}\, x; x := \mathsf{unit}; x)$, which publish their storage, and the *concealed* terms $(\mathsf{new}\, x; x := \mathsf{null})$ and $(\mathsf{new}\, x; x := \mathsf{unit})$, which do not.

The exposed terms can be distinguished by the $\lambda\mu$hashref context $(\mathsf{let}\, x = [-]\, \mathsf{in}\, !x)$. The concealed terms cannot be distinguished by any $\lambda\mu$hashref context. However, given allocation order $\overline{\iota_1, \ldots, \iota_N}$, the concealed terms can be distinguished by the $\lambda\mu$proberef context

$$([-]; \mathsf{let}\, x = \mathsf{proberef}_{\iota_1}\, \mathsf{in}\, !x).$$

We refer this as the *lucky* context.

The situation changes with randomization. We assume a uniform distribution of allocation orders over $\{\iota_1, \ldots, \iota_N\}$.

The exposed terms are distinguished by the given $\lambda\mu$hashref context with probability 1.

The concealed terms are distinguished by the lucky context with probability $\frac{(N-1)!}{N!} = \frac{1}{N}$. This context fails with probability $1 - \frac{1}{N} = \frac{N-1}{N}$, evaluating to err on both concealed terms. (It must be lucky to succeed!)

Consider the following *hard-working* context (where $\neg$ is implemented by swapping the branches of the conditional).

$$\begin{array}{l} [-]; \mathsf{let}\, x = \mathsf{proberef}_{\iota_1}\, \mathsf{in}\, \mathsf{if}\ \neg\mathsf{hashref}_{\mathsf{null}}\, x\ \mathsf{then}\ !x\ \mathsf{else} \\ \quad \mathsf{let}\, x = \mathsf{proberef}_{\iota_2}\, \mathsf{in}\, \mathsf{if}\ \neg\mathsf{hashref}_{\mathsf{null}}\, x\ \mathsf{then}\ !x\ \mathsf{else} \\ \quad \cdots \\ \quad \mathsf{let}\, x = \mathsf{proberef}_{\iota_n}\, \mathsf{in}\, \mathsf{if}\ \neg\mathsf{hashref}_{\mathsf{null}}\, x\ \mathsf{then}\ !x \end{array}$$

A single branch of the hard-working context is no more likely to succeed than the lucky context, distinguishing the concealed terms with probability $\frac{1}{N}$. Taken as a whole, however, the hard-working context succeeds in distinguishing the concealed terms with probability 1. ■

The different results for exposed and concealed terms in Example 14 indicate that we must record the *knowledge* of the context, i.e., the set of references known to the context.

Example 15. We narrate the execution of a particular *user* executing against an *opponent* encoded as a $\lambda\mu$proberef context. Assume a uniform distribution of allocation orders over a memory of size N where $N > 3$. The allocation order is not known to the opponent and the store is initially empty.

(1) In the first two steps of evaluation, suppose the user allocates two new locations and therefore store has size 2.

(2) Suppose the opponent now executes $\mathsf{proberef}_r$, guessing that one of the two allocated references is $r \neq \mathsf{null}$. What is the probability that one of the two allocated references is r? There are $N!$ possible permutations of the allocation order, with $(N-1)!$ permutations that have r in any given position. Thus there are $2 \times (N-1)!$ permutations with r in first or second position and the chance of success is $\frac{2\times(N-1)!}{N!} = \frac{2}{N}$.

Let us assume that the probe fails. The above probability calculation yields that the possible sample space is now reduced to $N! \times \frac{N-2}{N} = (N-1)! \times (N-2)$.

(3) Consider a further evolution where the user gives the opponent a reference, say ι. The opponent knows that ι is one of the two allocated references and that r is not an allocated reference as yet. The possible sample space is $(N-2)! \times 2 \times (N-2)$. That is $N-2$ for r, 2 for ι and $N-2!$ for the rest.

(4) Consider another probe of the opponent, say $\mathsf{proberef}_q$, where $q \notin \{\mathsf{null}, r\}$. The opponent succeeds if the two allocated references are ι and q. There are $2 \times (N-2)!$ permutations that start with ι and q. Thus the chance that the second probe succeeds is $\frac{1}{N-2}$.

Let us again assume that the opponent probe fails. The opponent then knows that ι is one of the first two allocated references and that r and q are not in the first two allocated references. The above probability calculation yields the size of the potential sample space to be $(N-2)! \times 2 \times (N-2) \times \frac{N-3}{N-2} = (N-2)! \times 2 \times (N-3)$.

(5) Suppose that the user allocates another reference.

(6) Consider a third opponent probe, repeating the quest for reference r. Under the above circumstances, opponent succeeds if and only if the third allocated reference is r.

The number of permutations with ι in one of first two positions, r in third position, and q not in first two positions is $2 \times (N-3) \times 1 \times (N-3)!$. That is 2 for ι, $N-3$ for q, 1 for r and $N-3!$ for the rest. The probability of success is $\frac{1}{N-3}$. ■

C. Encoding knowledge

As demonstrated by Example 15, the probability that a opponent probe succeeds depends upon the set of references known to the opponent. Determining this set is complicated by the fact that opponents are defined as contexts. The separation

between context and term is not preserved by evaluation, and thus we must somehow instrument the evaluation relation. One approach is to define an LTS, as we do in the following section. Another approach is extend the syntax with *tags* to record the provenance of terms.

Here we define a transformation $\mathscr{T}$ to record accumulated knowledge in an isolated *kernel* memory. We apply the transformation to opponent contexts, but not user terms. Define *kmem* to be a function on allocation orders such that $|kmem(\eta)| = |\eta|$ and $(kmem(\eta) \cap \eta) = \emptyset$. Suppose $\eta = \overline{\iota_1, \ldots, \iota_N}$ and $kmem(\eta) = \overline{\kappa_1, \ldots, \kappa_N}$.

- Define $korder(\eta) \triangleq \overline{\iota_1, \ldots, \iota_N, \kappa_1, \ldots, \kappa_N}$.
- Define $kstore(\eta) \triangleq \{\kappa_1 := \mathsf{null}, \ldots, \kappa_n := \mathsf{null}\}$.
- Define $K_s \triangleq \{\iota_i \mid s(\kappa_i) \neq \mathsf{null}\}$.

For a store s with kernel memory, K_s is the set of user references known to the context.

An alternative characterization is given in Section V. The remainder of this subsection can be skipped on first reading.

The side-effecting function $setk^\eta$ is a map over $\eta \cup \{\mathsf{null}\}$, defined as follows. $setk^\eta(v)$ immediately returns null if $v \notin \eta$, otherwise if $\eta = \overline{\iota_1, \ldots, \iota_N}$ and $kmem(\eta) = \overline{\kappa_1, \ldots, \kappa_N}$ then $setk^\eta(\iota_i)$ writes $\kappa_i := \mathsf{unit}$ and returns ι_i. For any η, $setk^\eta$ can be internalized as a value in our language (as an abstraction using $\mathsf{hashref}_{\iota_i}$ for $1 \leq i \leq N$).

We define the transformation function $\mathscr{T}$ as follows, where $setk^\eta(t)$ is shorthand for $\mathsf{let}\, y = t \,\mathsf{in}\, setk^\eta(y)$ and similarly for $setk^\eta(E)$.

$$\mathscr{T}(v) \triangleq \begin{cases} \lambda x.\, \mathscr{T}(t) & \text{if } v = \lambda x.t \\ v & \text{otherwise} \end{cases}$$

$$\mathscr{T}(t) \triangleq \begin{cases} \mathsf{let}\, x = \mathscr{T}(t_1) \,\mathsf{in}\, \mathscr{T}(t_2) & \text{if } t = \mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2 \\ setk^\eta(\mathsf{alloc}) & \text{if } t = \mathsf{alloc} \\ t & \text{otherwise} \end{cases}$$

$$\mathscr{T}(E) \triangleq \begin{cases} setk^\eta([\text{–}]) & \text{if } E = [\text{–}] \\ \mathsf{let}\, x = \mathscr{T}(E) \,\mathsf{in}\, setk^\eta(\mathscr{T}(t)) & \text{if } E = \mathsf{let}\, x = E \,\mathsf{in}\, t \end{cases}$$

Define $\mathscr{T}([\![a]\!]t) \triangleq [\![a]\!]\mathscr{T}(t)$ and $\mathscr{T}([\![a]\!]E) \triangleq [\![a]\!]\mathscr{T}(E)$.

D. Computing probabilities

Following Abadi and Plotkin, we give an explicit formula to characterize of the probabilities associated with evaluation sequences. We follow up by showing that the calculations in the style of Example 15 agree with our formulas to calculate probabilities. We elide the precise statement of the theorem equating these two approaches. (The proof of the unstated theorem follows Example 15.)

Let ϖ range over *evaluation paths* of the form

$$korder(\eta) \vdash \mathscr{C}_{i-1} \rightarrow_{\alpha_i} \mathscr{C}_i \text{ for } 1 \leq i \leq n.$$

Given such an evaluation path, we define several sets of interest, where $\mathscr{C}_i = \langle s_i, \ldots \rangle$.

- Define $N_\varpi \triangleq |\eta|$ be the size of user memory.
- Define $s_\varpi \triangleq dom(s_n) \setminus kmem(\eta)$ be the final set of user references allocated.
- Define $K_\varpi \triangleq K_{s_n}$ be the final set of user references known to the context.
- Define $\mathbf{N}_\varpi \triangleq \{r \mid \exists i.\, \mathscr{C}_{i-1} \rightarrow_{\mathbf{N}_r} \mathscr{C}_i\}$ be the set of user references for which proberef has failed.

The following definitions split a path at a failed proberef.

- Define $upto(\varpi, r)$ to be the largest prefix of ϖ whose final transition is annotated $\mathbf{N}_r$, if one exists, and to be the empty path ε otherwise.
- Define $after(\varpi, r)$ to be the suffix of ϖ after $upto(\varpi, r)$.

Let ε represent an empty path and let $\varpi \rightarrow_\alpha$ represent a path whose final transition is $\rightarrow_\alpha$.

We can now define the probability of a path ϖ in the evaluation relation as follows:

$$\delta_\varpi(r) \triangleq \frac{|s_\varpi \setminus (s_{upto(\varpi,r)} \cup K_\varpi)|}{N_\varpi - |s_{upto(\varpi,r)} \cup K_\varpi| - |\mathbf{N}_{after(\varpi,r)}|}$$

$$\phi(\varepsilon) \triangleq 1$$

$$\phi(\varpi \rightarrow_\alpha) \triangleq \begin{cases} \phi(\varpi) \times \delta_\varpi(r) & \text{if } \alpha = \mathbf{Y}_r, r \notin K_\varpi \\ \phi(\varpi) \times (1 - \delta_\varpi(r)) & \text{if } \alpha = \mathbf{N}_r, r \neq \mathsf{null} \\ \phi(\varpi) & \text{otherwise} \end{cases}$$

Observe that in the special case where no previous failed checks on reference r have occurred then $\phi(\varpi)$ becomes identical to the corresponding formula of Abadi and Plotkin. This is because K_ϖ represents the public references PubLoc, $s_\varpi \setminus K_\varpi$ represents the private references PriLoc, $s_{upto(\varpi,r)}$ is empty, and $|\mathbf{N}_{after(\pi,r)}|$ counts the number of distinct probes that do not hit any of the private references. In our setting, it is not sufficient to only consider the set of distinct probes over a trace. This is due to the presence of dynamic allocation in our model: a failed probe may on a subsequent occurrence become a successful probe. Therefore our model takes in to account the *list* of failed probes over a trace. Indeed, if we were to allow deallocation of references also then we would need to consider the list of *all* probes, including successful ones over a trace. We can however render our calculation as an iterated version of that of Abadi and Plotkin: consider a path ϖ and any r such that ϖ has no $\mathbf{Y}_r$ but k_r many $\mathbf{N}_r$ annotations. Let $\varpi_{k_r} = \varpi$ and $\varpi_{i-1} = upto(\varpi_i, r)$ so that $\varpi_i = \varpi_{i-1} after(\varpi_i, r)$. We can write $\phi(\varpi)$ as

$$\prod_{r \in \varpi} \left(\prod_{i \leq k_r} (1 - \delta_{\varpi_i}(r)) \right).$$

Now, for any given i we write

$$\begin{array}{lcl} N_i & \triangleq & N_\varpi - |s_{\varpi_{i-1}}| \\ \mathsf{PubLoc}_i & \triangleq & K_{\varpi_i} \setminus K_{\varpi_{i-1}} \\ \mathsf{Store}_i & \triangleq & s_{\varpi_i} \setminus s_{\varpi_{i-1}} \\ \mathsf{PriLoc}_i & \triangleq & \mathsf{Store}_i \setminus \mathsf{PubLoc}_i \\ \mathbf{N}_i & \triangleq & \mathbf{N}_{\varpi_i \setminus \varpi_{i-1}} \end{array}$$

We see then that $s_{\varpi_i} \setminus (s_{upto(\varpi_i,r)} \cup K_{\varpi_i})$ is $\mathsf{Store}_i \setminus \mathsf{PubLoc}_i$, which is just PriLoc_i. Also note that, $N_{\varpi_i} - |s_{upto(\varpi_i,r)} \cup K_{\varpi_i}| - |\mathbf{N}_{after(\varpi_i,r)}|$ is equal to $N_i - |\mathsf{PubLoc}_i| - |\mathbf{N}_i|$. Therefore we can write

$$\delta_{\varpi_i}(r) = \frac{|\mathsf{PriLoc}_i|}{N_i - |\mathsf{PubLoc}_i| - |\mathbf{N}_i|}$$

as it appears in Abadi and Plotkin.

Example 16. We revisit Example 15 in the light of the calculations above, describing a particular evaluation path. Initially, the user store and opponent knowledge are empty. We use ϖ to stand for the trace *so far*, at each point of the evaluation. Let N be the constant N_ϖ

(1) The user allocates twice. $s_\varpi = \{\iota, \kappa_1\}$ and $K_\varpi = \emptyset$.

(2) The opponent executes $\mathsf{proberef}_r$. $upto(\varpi, r) = \varepsilon$ and $after(\varpi, r) = \varpi$. The calculation above gives the probability of success as $\frac{2}{N}$.

Assume that the proberef failed.

(3) The user reveals ι. $s_\varpi = \{\iota, \kappa_1\}$ and $K_\varpi = \{\iota\}$.

(4) The opponent executes $\mathsf{proberef}_q$. $upto(\varpi, q) = \varepsilon$ and $after(\varpi, q) = \varpi$. The calculation above gives $s_{upto(\varpi,q)} = \emptyset$ and $\mathbf{N}_{after(\varpi,q)} = \{r\}$. The calculation above gives the probability of success as $\frac{2-1}{N-1-1} = \frac{1}{N-2}$.

Assume that the proberef failed.

(5) The user allocates once. $s_\varpi = \{\iota, \kappa_1, \kappa_2\}$ and $K_\varpi = \{\iota\}$.

(6) The opponent executes $\mathsf{proberef}_r$. $upto(\varpi, r)$ gives the path up to step (2) and $after(\varpi, r)$ gives the path afterwards. The calculation above gives $s_{upto(\varpi,r)} = \{\iota, \kappa_1\}$ and $\mathbf{N}_{after(\varpi,r)} = \{q\}$. We have $s_\varpi \setminus (s_{upto(\varpi,r)} \cup K_\varpi) = \{\kappa_2\}$ and $s_{upto(\varpi,r)} \cup K_\varpi = \{\iota, \kappa_1\}$. The chance of success is therefore $\frac{1}{N-2-1} = \frac{1}{N-3}$.

These calculations agree with the derivation of Example 15. ∎

E. Contextual equivalence

The lucky context of Example 14 shows that contextual equivalence must ignore low-probability execution paths. The subsequent hard-working context indicates that it must also ignore executions in a single proberef conveys too much information. When a probe fails, subsequent probes convey more information, and therefore (Abadi and Plotkin 2010) limit the number of failed probes. We generalize their results to include dynamic allocation. The set of references which can be unfruitfully probed are those which are unallocated and unprobed. Therefore we bound the sum of the number of references allocated and the number of failed probes.

We define paths with parameters for the probability (p) and the sum of the references allocated and the number of failed probes (n).

Define $\eta \vdash \mathscr{C} \Rightarrow_{p,n} \mathscr{C}'$ to hold whenever there exists a path ϖ with the given source configuration and target configuration such that

$$p = \phi(\varpi) \text{ and } n = |s_\varpi| + |\mathbf{N}_\varpi|.$$

Define $\eta \vdash nt \not\downarrow_{p,n} a$ to mean that there exists s such that

$$korder(\eta) \vdash \langle kstore(\eta), nt\rangle \Rightarrow_{p,n} \langle s, [\![a]\!]\mathsf{null}\rangle.$$

We define contextual equivalence with bounds for the minimum probability (ε) and maximum number of failed probes (B).

Definition 17 (Contextual equivalence for $\lambda\mu$proberef). Closed values v and w are *contextually equivalent up to ε and B with memory size N* (notation $v \simeq^N_{\varepsilon,B} w$) if for any allocation order η such that $|\eta| \geq N$ and for any named evaluation context NE such that $vrefs(NE) = \emptyset$ and $hrefs(NE) \subseteq \eta$, and for any $a \in fn(NE)$,

$$\eta \vdash \mathscr{T}(NE)[v] \not\downarrow_{p,n} a \text{ and } p > \varepsilon \text{ and } n \leq B \text{ implies } \eta \vdash \mathscr{T}(NE)[w] \not\downarrow_{p,n} a$$

and symmetrically for w. We extend this notion to closed terms by defining $t \simeq^N u$ whenever $\lambda\,.\,t \simeq^N_{\varepsilon,B} \lambda\,.\,u$. ∎

We can now state the main result of the paper.

Theorem 18 (Full abstraction). For closed $\lambda\mu$hashref terms t and u and for values of N, ε and B such that $B \leq \varepsilon N$:

$$t \simeq^N u \text{ iff } t \simeq^N_{\varepsilon,B} u$$

PROOF. The $\Leftarrow$ direction is immediate, since every $\lambda\mu$hashref context is a $\lambda\mu$proberef context. The $\Rightarrow$ direction is Corollary 29 in Section VI. ∎

V. LABELLED TRANSITION SYSTEM

We develop a labelled transition system model for $\lambda\mu$proberef (and $\lambda\mu$hashref) and define a suitable notion of trace equivalence. We relate these to contextual equivalence in Section VI.

A. The LTS

Each node of the LTS includes a reference set and valuation, recording the knowledge of the opponent context. We distinguish two types of nodes, indicating whether the term or context has control.

$$\begin{array}{rcll} K & ::= & \{r_1, \ldots, r_n\} & \text{(Reference set)} \\ V & ::= & \{x_1 := v_1, \ldots, x_n := v_n, & \text{(Valuation)} \\ & & a_1 := NE_1, \ldots, a_m := NE_m\} & \\ \mathscr{N} & ::= & \langle s, nt, V, K\rangle & \text{(Term node)} \\ & | & \langle s, \_, V, K\rangle & \text{(Context node)} \\ \ell & ::= & \boldsymbol{\tau} \mid \mathbf{tvf}_a y \mid \mathbf{tvr}_a r \mid \mathbf{taf}_a xy \mid \mathbf{tar}_a xr & \text{(Label)} \\ & & \mid \mathbf{cvf}_a x \mid \mathbf{cvr}_a r \mid \mathbf{caf}_a yx \mid \mathbf{car}_a yr & \\ & & \mid \mathbf{crf}_a rx \mid \mathbf{crr}_a rq \mid \mathbf{cra}_a \mid \mathbf{crg}_a r \mid \mathbf{crp}_a \iota & \end{array}$$

Define $\langle v\rangle \triangleq \langle \emptyset, \_, \{x := v\}, \{\mathsf{null}\}\rangle$, where $x \notin fv(v)$, to be the *initial node* of v.

The LTS for $\lambda\mu$proberef is defined in Figure 3 as a labeled and annotated relation on nodes, parameterized by an allocation order ($\eta \vdash \mathscr{N} \xrightarrow{\ell}_\alpha \mathscr{N}$). A in Figure 1, η is constant throughout and we have elided it from the rules.

We obtain an LTS for $\lambda\mu$hashref by omitting the transition rule for probes (Context Ref Probe) and the evaluation rules of Figure 2. We say that the inaccessibility of local references is *strict* in $\lambda\mu$hashref and *lax* in $\lambda\mu$proberef. We therefore refer to the LTS for $\lambda\mu$hashref as the *strict* LTS, and that for $\lambda\mu$proberef as the *lax* LTS.

Note that the LTS is bipartite. Term evaluation is deterministic and therefore only one transition is available to each term node. Context nodes have many transitions, each of which give control to a term node.

Context transitions labeled **cr?** are always followed immediately by two term transitions, labeled $\boldsymbol{\tau}$ and **tv?**. Context transitions labeled **cv?** and **ca?** may cause any number of

(Term Evaluation)	$\langle s, nt, V, K\rangle$	$\xrightarrow{\boldsymbol{\tau}}_\alpha$	$\langle s', nt', V, K\rangle$	if $\langle s, nt\rangle \longrightarrow_\alpha \langle s', nt'\rangle$
(Term Value Fun)	$\langle s, [\![b]\!]v, V, K\rangle$	$\xrightarrow{\mathbf{tvf}_b\, x}$.	$\langle s, \_, V \uplus \{x := v\}, K\rangle$	if $\neg isref(v)$ and x fresh
(Term Value Ref)	$\langle s, [\![b]\!]v, V, K\rangle$	$\xrightarrow{\mathbf{tvr}_b\, v}$.	$\langle s, \_, V, K \cup \{v\}\rangle$	if $isref(v)$
(Term Apply Fun)	$\langle s, NE[y\, v], V, K\rangle$	$\xrightarrow{\mathbf{taf}_a\, y\, x}$.	$\langle s, \_, V \uplus \{a := NE\} \uplus \{x := v\}, K\rangle$	if a fresh and $\neg isref(v)$ and x fresh
(Term Apply Ref)	$\langle s, NE[y\, v], V, K\rangle$	$\xrightarrow{\mathbf{tar}_a\, y\, v}$.	$\langle s, \_, V \uplus \{a := NE\}, K \cup \{v\}\rangle$	if a fresh and $isref(v)$
(Context Value Fun)	$\langle s, \_, V, K\rangle$	$\xrightarrow{\mathbf{cvf}_b\, x}$.	$\langle s, NE[x], V, K\rangle$	if $V(b) = NE$ and x fresh
(Context Value Ref)	$\langle s, \_, V, K\rangle$	$\xrightarrow{\mathbf{cvr}_b\, q}$.	$\langle s, NE[q], V, K\rangle$	if $V(b) = NE$ and $q \in K$
(Context Apply Fun)	$\langle s, \_, V, K\rangle$	$\xrightarrow{\mathbf{caf}_a\, y\, x}$.	$\langle s, [\![a]\!](v\, x), V, K\rangle$	if a fresh and $V(y) = v$ and x fresh
(Context Apply Ref)	$\langle s, \_, V, K\rangle$	$\xrightarrow{\mathbf{car}_a\, y\, q}$.	$\langle s, [\![a]\!](v\, q), V, K\rangle$	if a fresh and $V(y) = v$ and $q \in K$
(Context Ref Fun)	$\langle s, \_, V, K\rangle$	$\xrightarrow{\mathbf{crf}_a\, r\, x}$.	$\langle s, [\![a]\!](r := x), V, K\rangle$	if a fresh and $r \in K$ and x fresh
(Context Ref Ref)	$\langle s, \_, V, K\rangle$	$\xrightarrow{\mathbf{crr}_a\, r\, q}$.	$\langle s, [\![a]\!](r := q), V, K\rangle$	if a fresh and $r \in K$ and $q \in K$
(Context Ref Get)	$\langle s, \_, V, K\rangle$	$\xrightarrow{\mathbf{crg}_a\, r}$.	$\langle s, [\![a]\!](!\, r), V, K\rangle$	if a fresh and $r \in K$
(Context Ref Alloc)	$\langle s, \_, V, K\rangle$	$\xrightarrow{\mathbf{cra}_a}$.	$\langle s, [\![a]\!]\mathsf{alloc}, V, K\rangle$	if a fresh
(Context Ref Probe)	$\langle s, \_, V, K\rangle$	$\xrightarrow{\mathbf{crp}_a\, r}$.	$\langle s, [\![a]\!]\mathsf{proberef}_r, V, K\rangle$	if a fresh

Fig. 3. Labeled transition system $(\eta \vdash \mathcal{N} \xrightarrow{\ell}_\alpha \mathcal{N}')$

term $\boldsymbol{\tau}$ transitions, ending in a term transition labeled **tv?** or **ta?**.

The knowledge of the context increases on term transitions labeled **t?r**.

Transitions labeled **ca?** model applicative tests (Gordon 1995), performed symbolically (Sangiorgi 1996; Lassen 2005, 2006; Jagadeesan, Pitcher, and Riely 2009). Transitions labeled **cv?** model the context control tests. Both **ca?** and **cv?** transitions use stored information, and thus the tests may be performed repeatedly. Transitions labeled **cr?** model the context's ability to manipulate references.

Example 19. Consider an execution path for LTS generated by the left-or operator $(\lambda x.\, \lambda y.\, x\ \mathsf{tru}\ y)$. Let

$$\begin{aligned} V_1 &= \{z_1 := (\lambda x.\, \lambda y.\, x\ \mathsf{tru}\ y)\},\\ V_2 &= V_1 \uplus \{z_2 := (\lambda y.\, x'\ \mathsf{tru}\ y)\},\ \text{and}\\ V_3 &= V_2 \uplus \{a := [\![a]\!]([-]\, y'), z_3 := \mathsf{tru}\}. \end{aligned}$$

Then the LTS includes the following.

$$\begin{aligned} &\langle \emptyset, \_, V_1, \emptyset\rangle\\ \xrightarrow{\mathbf{caf}_a\, z_1\, x'}\ &\langle \emptyset, [\![a]\!](\lambda x.\, \lambda y.\, x\ \mathsf{tru}\ y)\, x', V_1, \emptyset\rangle\\ \xrightarrow{\boldsymbol{\tau}}\ &\langle \emptyset, [\![a]\!](\lambda y.\, x'\ \mathsf{tru}\ y), V_1, \emptyset\rangle\\ \xrightarrow{\mathbf{tvf}_a\, z_2}\ &\langle \emptyset, \_, V_2, \emptyset\rangle\\ \xrightarrow{\mathbf{caf}_a\, z_2\, y'}\ &\langle \emptyset, [\![a]\!](\lambda y.\, x'\ \mathsf{tru}\ y)\, y', V_2, \emptyset\rangle\\ \xrightarrow{\boldsymbol{\tau}}\ &\langle \emptyset, [\![a]\!](x'\ \mathsf{tru}\ y'), V_2, \emptyset\rangle\\ \xrightarrow{\mathbf{taf}_a\, x'\, z_3}\ &\langle \emptyset, \_, V_3, \emptyset\rangle \end{aligned}$$

This prefix of the path is sufficient to see the difference with the right-or operator $(\lambda x.\, \lambda y.\, y\ \mathsf{tru}\ x)$. Let

$$\begin{aligned} V_1' &= \{z_1 := (\lambda x.\, \lambda y.\, y\ \mathsf{tru}\ x)\},\\ V_2' &= V_1 \uplus \{z_2 := (\lambda y.\, y\ \mathsf{tru}\ x')\},\ \text{and}\\ V_3' &= V_2 \uplus \{a := [\![a]\!]([-]\, x'), z_3 := \mathsf{tru}\}. \end{aligned}$$

Starting with $\langle \emptyset, \_, V_1', \emptyset\rangle$, execution proceeds with the same labels as before, except that the final transition is labeled $\mathbf{taf}_a\, y'\, z_3$. The context can observe the difference between the final labels, thus distinguishing the left-or and right-or operators. ∎

Example 20. Consider an execution path for LTS generated by the first process of Example 4, $\mathsf{new}\, x;\ \mathsf{new}\, y;\ x$. Executing with allocation order $\overline{\iota_1, \iota_2}$, and eliding $\boldsymbol{\tau}$-annotations, we observe the following path.

$$\begin{aligned} &\langle \emptyset, [\![a]\!]\mathsf{new}\, x;\ \mathsf{new}\, y;\ x, \emptyset, \emptyset\rangle\\ \xrightarrow{\boldsymbol{\tau}}\xrightarrow{\boldsymbol{\tau}}\ &\langle \{\iota_1 := \mathsf{null}\}, [\![a]\!]\mathsf{new}\, y;\ \iota_1, \emptyset, \emptyset\rangle\\ \xrightarrow{\boldsymbol{\tau}}\xrightarrow{\boldsymbol{\tau}}\ &\langle \{\iota_1 := \mathsf{null}, \iota_2 := \mathsf{null}\}, [\![a]\!]\iota_1, \emptyset, \emptyset\rangle\\ \xrightarrow{\mathbf{tvr}_a\, \iota_1}\ &\langle \{\iota_1 := \mathsf{null}, \iota_2 := \mathsf{null}\}, \_, \emptyset, \{\iota_1\}\rangle \end{aligned}$$

This prefix of the path is sufficient to see the difference with the second process of Example 4, $\mathsf{new}\, x;\ \mathsf{new}\, y;\ y$, whose final transition is labelled $\mathbf{tvr}_a\, \iota_2$. ∎

Example 21. Consider a variation of the first process given in Example 3. Let

$$\begin{aligned} V_1'' &= \{z_1 := (\lambda f.\, \mathsf{new}\, x;\ x := \mathsf{fls};\ f\, \mathsf{unit};\\ &\qquad\qquad !\, x\, (\lambda .\, \Omega)\, (\lambda .\, x := \mathsf{tru};\ \mathsf{unit})\, \mathsf{unit})\},\\ V_2'' &= V_1'' \uplus \{b := [\![a]\!]([-];\ !\, \iota\, (\lambda .\, \Omega)\, (\lambda .\, \iota := \mathsf{tru};\ \mathsf{unit})\, \mathsf{unit}),\\ &\qquad z_2 := \mathsf{unit}\},\ \text{and}\\ V_3'' &= V_2'' \uplus \{z_3 := \mathsf{unit}\}. \end{aligned}$$

Executing with allocation order $\overline{\iota}$, and again eliding $\boldsymbol{\tau}$-annotations, we observe the following path.

$$\begin{aligned} &\langle \emptyset, \_, V_1'', \emptyset\rangle\\ \xrightarrow{\mathbf{caf}_a\, z_1\, f'}\xrightarrow{\boldsymbol{\tau}}\ &\langle \emptyset, [\![a]\!]\mathsf{new}\, x;\ x := \mathsf{fls};\ f'\, \mathsf{unit};\\ &\qquad\qquad !\, x\, (\lambda .\, \Omega)\, (\lambda .\, x := \mathsf{tru};\ \mathsf{unit})\, \mathsf{unit}, V_1'', \emptyset\rangle\\ \xrightarrow{\boldsymbol{\tau}}{}^{+}\ &\langle \{\iota := \mathsf{fls}\}, [\![a]\!]f'\, \mathsf{unit};\\ &\qquad\qquad !\, \iota\, (\lambda .\, \Omega)\, (\lambda .\, \iota := \mathsf{tru};\ \mathsf{unit})\, \mathsf{unit}, V_1'', \emptyset\rangle\\ \xrightarrow{\mathbf{taf}_b\, f'\, z_2}\ &\langle \{\iota := \mathsf{fls}\}, \_, V_2'', \emptyset\rangle\\ \xrightarrow{\mathbf{cvf}_b\, x'}\ &\langle \{\iota := \mathsf{fls}\}, [\![a]\!]x';\\ &\qquad\qquad !\, \iota\, (\lambda .\, \Omega)\, (\lambda .\, \iota := \mathsf{tru};\ \mathsf{unit})\, \mathsf{unit}, V_2'', \emptyset\rangle\\ \xrightarrow{\boldsymbol{\tau}}{}^{+}\ &\langle \{\iota := \mathsf{fls}\}, [\![a]\!]\iota := \mathsf{tru};\ \mathsf{unit}, V_2'', \emptyset\rangle\\ \xrightarrow{\boldsymbol{\tau}}{}^{+}\ &\langle \{\iota := \mathsf{tru}\}, [\![a]\!]\mathsf{unit}, V_2'', \emptyset\rangle\\ \xrightarrow{\mathbf{tvf}_a\, z_3}\ &\langle \{\iota := \mathsf{tru}\}, \_, V_3'', \emptyset\rangle\\ \xrightarrow{\mathbf{cvf}_b\, x''}\ &\langle \{\iota := \mathsf{tru}\}, [\![a]\!]x'';\\ &\qquad\qquad !\, \iota\, (\lambda .\, \Omega)\, (\lambda .\, \iota := \mathsf{tru};\ \mathsf{unit})\, \mathsf{unit}, V_2'', \emptyset\rangle\\ \xrightarrow{\boldsymbol{\tau}}{}^{+}\ &\langle \{\iota := \mathsf{tru}\}, [\![a]\!]\Omega, V_2'', \emptyset\rangle\\ \xrightarrow{\boldsymbol{\tau}}{}^{+}\ &\langle \{\iota := \mathsf{tru}\}, [\![a]\!]\Omega, V_2'', \emptyset\rangle \end{aligned}$$

This path is not possible for the second process given in Example 3, since ι remains fls.

The path illustrates how the LTS formalism uses (Term Apply Fun) transitions provide the programmatic power available in $\lambda\mu$ to capture continuations, and (Context Value Fun) to use and reuse them. In contrast, our earlier work on LTSs for aspects (Jagadeesan, Pitcher, and Riely 2009) maintained a stack for the current evaluation context, enforcing a linear regime that prohibits the reuse of evaluation contexts. ■

B. Trace Equivalence

Let π range over LTS *paths* of the form

$$\eta \vdash \mathscr{N}_{i-1} \xrightarrow{\ell_i}_{\alpha_i} \mathscr{N}_i \text{ for } 1 \le i \le n.$$

A path is *strict* if it contains no transitions labeled **crp** or annotated $\mathbf{Y}$ or $\mathbf{N}$. A path is *balanced* if the first and last nodes are context nodes.

LTS paths contain all of the information required to compute probabilities. Define N_π, $\mathbf{N}_\pi$, $upto(\pi, r)$ and $after(\pi, r)$ as in Section IV-D. When $\mathscr{N}_i = \langle s_i, \ldots, \ldots, K_i \rangle$, define $s_\pi = dom(s_n)$ and $K_\pi = K_n$. Given these notations, the definition of $\phi(\varpi)$ in Section IV-D lifts directly to define $\phi(\pi)$ on LTS paths, simply replacing ϖ with π.

As before, define $(\eta \vdash \mathscr{N} \stackrel{\bar{\ell}}{\Longrightarrow}_{p,n} \mathscr{N}')$ to be a path π with the given source node, target node and label sequence such that $p = \phi(\pi)$ and $n = |s_\pi| + |\mathbf{N}_\pi|$.

Definition 22 (Trace equivalence). Closed values v and w are *trace equivalent up to ε and B with memory size N* (notation $v \approx^N_{\varepsilon,B} w$) if for any allocation order η such that $|\eta| \ge N$ and any context node $\mathscr{N}$,

$$\eta \vdash \langle v \rangle \stackrel{\bar{\ell}}{\Longrightarrow}_{p,n} \mathscr{N} \text{ and } p > \varepsilon \text{ and } n \le B \text{ implies } \eta \vdash \langle w \rangle \stackrel{\bar{\ell}}{\Longrightarrow}_{p,n} \mathscr{N}'$$

and symmetrically for w. We extend this notion to closed terms by defining $t \approx^N u$ whenever $\lambda\,.\,t \approx^N_{\varepsilon,B} \lambda\,.\,u$. ■

The same definition is suitable for the strict LTS. All strict paths have probability 1, eliminating the need to count allocations and failed probes. This means we may safely omit the annotations on strict paths, writing simply $(\eta \vdash \mathscr{N} \stackrel{\bar{\ell}}{\Longrightarrow} \mathscr{N}')$. We write $(t \approx^N u)$ to denote trace equivalence over the strict LTS and note that this is shorthand for $(t \approx^N_{0,\infty} u)$.

VI. Full abstraction

We now turn our attention to the main results of the paper. We will show that contextual equivalences in $\lambda\mu$hashref are preserved probabilistically as contextual equivalences in $\lambda\mu$proberef. We do this via the labelled transition system model by showing that contextual equivalence in $\lambda\mu$hashref implies trace equivalence in the LTS model with no probes. We then describe how trace equivalence is preserved probabilistically when probes are included as possible trace actions. Furthermore, we prove that trace equivalence implies contextual equivalence in $\lambda\mu$proberef. We begin by arguing that balanced traces with sufficiently high probability are in fact strict traces.

Given a balanced path π as $\eta \vdash \langle t \rangle \stackrel{\bar{\ell}}{\Longrightarrow}_{p,n} \mathscr{N}'$ in the LTS we write $\uparrow \pi$ to be the path π with all occurrences of the sequence of three edges $\xrightarrow{\mathbf{crp}_a\, r} \xrightarrow{\tau}_{\mathbf{N}_r} \xrightarrow{\mathbf{tvr}_a\, \mathsf{null}}$ removed. We also remove all occurrences of π $\xrightarrow{\mathbf{crp}_a\, r} \xrightarrow{\tau}_{\mathbf{Y}_r} \xrightarrow{\mathbf{tvr}_a\, r}$ where $r \in K_\pi$. If π contains no other transitions annotated with $\mathbf{Y}_r$ it is not too hard to see that $\uparrow \pi$ is actually a balanced path of the strict LTS. This is formalized in the following lemma.

For closed, strict terms t and u and for values of N, ε and B such that $B \le \varepsilon N$:

Lemma 23. If $\langle t \rangle \stackrel{\bar{\ell}}{\Longrightarrow}_{p,n} \mathscr{N}'$ is a balanced path π then $p > \varepsilon$ and $n \le B$ implies $\uparrow \pi$ is a strict path.

PROOF. We proceed by induction over the length of the balanced path. We need to establish that the path π contains no reduction edges annotated with $\mathbf{Y}_r$, from which it follows that $\uparrow \pi$ is a strict path. For the inductive case, we also need to ensure that the final node of $\uparrow \pi$ is the same as the final node of π.

The base case of the induction, in which the path is empty, is trivial. Suppose then that the path π is $\pi' \stackrel{\ell_1\ell_2}{\Longrightarrow}_{p,n} \mathscr{N}'$ and suppose that $p > \varepsilon$. If ℓ_1 is anything other than a $\mathbf{crp}_a\, r$ action then by the inductive hypothesis we know that there is a balanced strict path $\uparrow \pi'$. As the final node of π' and $\uparrow \pi'$ are the same, this path clearly extends to the strict path $\uparrow \pi' \stackrel{\ell_1\ell_2}{\Longrightarrow} \mathscr{N}'$.

In the case that π is $\pi' \xrightarrow{\mathbf{crp}_a\, r} \xrightarrow{\tau}_{\mathbf{N}_r} \xrightarrow{\mathbf{tvr}_a\, \mathsf{null}}$, or $\pi' \xrightarrow{\mathbf{crp}_a\, r} \xrightarrow{\tau}_{\mathbf{Y}_r} \xrightarrow{\mathbf{tvr}_a\, r}$ with $r \in K_{\pi'}$, we have, by the inductive hypothesis, that $\uparrow \pi'$ is a strict path. Notice also that, in this case, $\uparrow \pi = \uparrow \pi'$ and moreover, the final node of $\uparrow \pi'$ is the same as the final node of π. Therefore $\uparrow \pi$ is a strict path as required.

Finally, the case π is $\pi' \xrightarrow{\mathbf{crp}_a\, r} \xrightarrow{\tau}_{\mathbf{Y}_r} \xrightarrow{\mathbf{tvr}_a\, r}$ where $r \notin K_{\pi'}$ cannot arise. If π were of this form we would have:

$$\begin{aligned} p &= \phi(\pi') \times \delta_\pi(r) \\ &\le \delta_\pi(r) \\ &\le |s_\pi|/(N_\pi - |\mathbf{N}_\pi|) \\ &\le (|s_\pi| + |\mathbf{N}_\pi|)/N_\pi \\ &= n/N_\pi \le B/N_\pi \le \varepsilon \end{aligned}$$

This contradicts the assumption that $p > \varepsilon$. ■

Theorem 24 (Preservation). $v \approx^N w$ implies $v \approx^N_{\varepsilon,B} w$ whenever $B \le \varepsilon N$

PROOF. Take any η and any balanced path π as $\eta \vdash \langle v \rangle \stackrel{\bar{\ell}}{\Longrightarrow}_{p,n} \mathscr{N}'$ of the LTS such that $p > \varepsilon$ and $n \le B$. By the previous Lemma we have that $\uparrow \pi$ is a strict path originating at $\langle v \rangle$. Therefore, by the hypothesis, there is a path π' of the form $\eta \vdash \langle w \rangle \stackrel{\bar{\ell}'}{\Longrightarrow}$ such that $\bar{\ell}'$ and $\bar{\ell}$ differ only in that $\bar{\ell}$ may contain failed probes or probes on known references. We insert all such missing probes in to the path π' in line with π to obtain a path of the LTS proper $\eta \vdash \langle w \rangle \stackrel{\bar{\ell}}{\Longrightarrow}_{p,n}$ as required. It is easy to check that this path has probability p and bound n as required.

Theorem 25 (Completeness of $\lambda\mu$hashref). $v \simeq^N w$ implies $v \approx^N w$

PROOF. Take any η and a balanced path π of the form $\eta \vdash \langle v \rangle \stackrel{\bar{\ell}}{\Longrightarrow} \mathscr{N}$. We must show that there is a matching path from

$\langle w \rangle$. To do this we build a named evaluation context $D^{\eta}_{\overline{\ell}}$ with the property that $\eta \vdash D^{\eta}_{\overline{\ell}}[v_0] \not\downarrow a$ iff $\eta \vdash \langle v_0 \rangle \stackrel{\overline{\ell}}{\Longrightarrow} \mathscr{N}_0$ for some $\mathscr{N}_0$. We omit the details of this construction here. Once we have this however, it is clear to see that

$$\eta \vdash D^{\eta}_{\overline{\ell}}[v] \not\downarrow a$$

which implies by hypothesis that

$$\eta \vdash D^{\eta}_{\overline{\ell}}[w] \not\downarrow a$$

and hence by construction of $D^{\eta}_{\overline{\ell}}$ we have $\eta \vdash \langle w \rangle \stackrel{\overline{\ell}}{\Longrightarrow} \mathscr{N}'$ as required. ∎

At this point, as a technical convenience, we introduce an intermediate representation for terms that allows a transition system closer in spirit to the LTS semantics than the evaluation relation. We will call this the *dual* LTS. The nodes of this model, ranged over by $\mathscr{M}$, are of the form

$$\langle s, ct, W, nt, V \rangle$$

where s is a (symbolic) store, ct is _ or a named term that represents computation provided by the evaluation context, W is a binding of "opponent" variables to lambda terms originating from the context, nt is _ or a named term under investigation, and V is a binding of "player" variables to lambda terms originating from the named term. An invariant that is maintained on these nodes is that the only free variables in nt and V are "opponent" variables and similarly, the only free variables in ct and W are "player" variables. The store s does not hold any lambda abstractions, but rather variables which are bound in V or W. Another useful invariant is that if ct is not _ then nt is and if nt is not _ then ct is. We will use the notation $(ct|nt)$ to mean ct if nt is _, nt if ct is _ and _ otherwise. We will also write $s[WV]^*$ and $nt[WV]^*$ to represent the store and term, respectively, after iterating the substitutions in W and V on them to a fixed point.

The initial dual configuration for a given allocation order η, named evaluation context NE and value v is:

$$\langle\!\langle NE, v \rangle\!\rangle^{\eta} = \langle kstore(\eta), \_, \{a_0 := NE\}, [\![a_0]\!]x_0, \{x_0 := v\} \rangle$$

The transitions of the dual LTS are derived from those of the LTS model but fewer of the actions are used. After the Term Evaluation rule, we only make use of the Term Value, Term Apply, Context Value and Context Apply rules of Figure 3. The two main rules are

$$\frac{\langle s[V], nt, V, K_s \rangle \xrightarrow{\ell}_{\alpha} \langle s[V], nt', V', K' \rangle}{\langle s, ct, W, nt, V \rangle \xrightarrow{\cdot:\ell}_{\alpha} \langle s[K'], ct, W, nt', V' \rangle}$$

$$\frac{\langle s[W], ct, W, K_s \rangle \xrightarrow{\ell}_{\alpha} \langle s[W], ct', W', K' \rangle}{\langle s, ct, W, nt, V \rangle \xrightarrow{\ell:\cdot}_{\alpha} \langle s, ct', W', nt, V \rangle}$$

where $s[K']$ is the store s but with an updated system memory to record K'. As well as these rules we need special cases for actions that interact with the store:

$$\frac{\langle s[V], nt, V, K_s \rangle \xrightarrow{\ell}_{\mathbf{A}_r} \langle s[V] \uplus \{r := \mathsf{null}\}, nt', V, K_s \rangle}{\langle s, ct, W, nt, V \rangle \xrightarrow{\cdot:\ell}_{\mathbf{A}_r} \langle s \uplus \{r := \mathsf{null}\}, ct, W, nt', V \rangle}$$

$$\frac{\langle s[V], nt, V, K_s \rangle \xrightarrow{\ell}_{\mathbf{R}_r} \langle s[V], nt', V, K_s \rangle}{\langle s, ct, W, nt, V \rangle \xrightarrow{\cdot:\ell}_{\mathbf{R}_r} \langle s, ct, W, nt'[V], V \rangle}$$

$$\frac{\langle s[V], nt, V, K_s \rangle \xrightarrow{\ell}_{\mathbf{W}_{r,v}} \langle s[V] \uplus \{r := v\}, nt', V, K_s \rangle}{\langle s, ct, W, nt, V \rangle \xrightarrow{\cdot:\ell}_{\mathbf{W}_{r,v}} \langle s \uplus \{r := x\}, ct, W, nt', V \uplus \{x := v\} \rangle}$$

There are dual versions of these for transitions originating in the ct term also. As for the LTS we use the notation $\stackrel{\overline{j}:\overline{\ell}}{\Longrightarrow}{}^{p,n}_{\overline{\alpha}}$ to denote a path in the dual LTS. The calculations of probabilities on these paths lifts in the obvious way by making use of the annotations $\overline{\alpha}$. We are now ready to state the Lemma that relates evaluation and the dual LTS.

Lemma 26 (Decomposition/Composition). If

$$korder(\eta) \vdash \langle kstore(\eta), NE[v] \rangle \Longrightarrow_{p,n} \langle s, nt \rangle$$

then there exists a dual transition sequence:

$$korder(\eta) \vdash \langle\!\langle NE, v \rangle\!\rangle^{\eta} \stackrel{\overline{j}:\overline{\ell}}{\Longrightarrow}{}^{p,n}_{\overline{\alpha}} \langle s', ct, W, nt', V \rangle$$

such that $s = s'[WV]^*$ and $nt = (ct|nt')[WV]^*$.
Moreover, for any other w such that

$$korder(\eta) \vdash \langle\!\langle NE, w \rangle\!\rangle^{\eta} \stackrel{\overline{j}:\overline{\ell}}{\Longrightarrow}{}^{p,n}_{\overline{\alpha}} \langle s'', ct', W', nt', V' \rangle$$

we have $ct = ct'$, $W = W'$ and

$$korder(\eta) \vdash \langle kstore(\eta), NE[w] \rangle \Longrightarrow_{p,n} \langle s''[W'V']^*, (ct'|nt')[W'V']^* \rangle$$

PROOF. We outline how the proof of this Lemma is constructed but for the sake of brevity omit the full details. We use an induction over the length of the evaluation sequence and a case analysis on the possible single step evaluations from $\langle s[WV]^*, nt[WV]^* \rangle$. Assume (wlog by symmetry) that we have reached a node $\langle s, \_, W, nt, V \rangle$ in the decomposed dual trace. Now, if $\langle s, nt \rangle$ performs the next single evaluation step independently then we embed that directly as a transition of the dual system. Otherwise, nt is a stuck term such that the substitution $nt[WV]^*$ enables the next evaluation step. There are two possibilities here: nt is of the form $NE[xv]$ and W binds x to a lambda abstraction $\lambda z.t$, say, or nt is of the form $[\![a_0]\!]v$ and W binds a_0 to a named evaluation context. We illustrate the proof by showing the former case and by assuming that v is a lambda abstraction. Note that nt would generate a **taf** action in the LTS. In this case we decompose as follows:

$$\begin{aligned}\langle s, \_, W, NE[xv], V \rangle &\xrightarrow{\cdot:\mathbf{taf}_{a'xy}} . \langle s, \_, W, \_, V' \rangle \\ &\xrightarrow{\mathbf{caf}_{a'xy}:\cdot} . \langle s, [\![a']\!](\lambda z.t)y, W, \_, V' \rangle \\ &\xrightarrow{\boldsymbol{\tau}:\cdot} . \quad \langle s, [\![a']\!]t\{\!|^{y}/_{z}|\!\}, W, \_, V' \rangle\end{aligned}$$

where V' is $V \uplus \{a' := NE\} \uplus \{y := v\}$. Note, that, after the substitution $[WV']^*$, the named term here exactly what is required as the target of the evaluation step. ∎

The dual transition system provides a useful stepping-stone towards extracting the contribution of a given term in its interaction with an enclosing context. We now complete this step by describing how to project a path in the dual LTS down to a path in the LTS proper. For the most part this is straightforward: a node $\langle s, ct, W, nt, V \rangle$ maps to a node $\langle s[V],$

$nt, V', K_s\rangle$ where V' differs from V only on variables created in the write annotated dual transitions to maintain a symbolic store. Any transition $\xrightarrow{:\ell}_\alpha$ maps directly to the same transition in the LTS in the obvious way. Otherwise, transitions are of the form $\xrightarrow{\bar{j}:}_\alpha$. In this case we inspect the annotation α — if it is empty then we simply drop this transition from the projection. In all other cases, three transitions are generated in the projected trace: for example, for annotation $\mathbf{A}_r$ we have transitions

$$\xrightarrow{\mathbf{cra}_a}.\xrightarrow{\tau}_{\mathbf{A}_r}\xrightarrow{\mathbf{tvr}_a r}.$$

similar projections use $\mathbf{crg}_a r$ actions for read annotations, $\mathbf{crf}_a r x$ actions for write annotations and $\mathbf{crp}_a r$ actions for probe annotations.

Lemma 27 (Projection). If

$$korder(\eta) \vdash \langle\!\langle NE, v\rangle\!\rangle^\eta \overset{\bar{j}:\bar{\ell}}{\Longrightarrow}{}^{p,n}_{\alpha}\ \mathscr{M}$$

then there exists a projected trace $\eta \vdash \langle v\rangle \overset{\bar{\ell}^+}{\Longrightarrow}_{p,n} \mathscr{N}$ *such that for any other* $\eta \vdash \langle w\rangle \overset{\bar{\ell}^+}{\Longrightarrow}_{p,n} \mathscr{N}'$ *then*

$$korder(\eta) \vdash \langle\!\langle NE, w\rangle\!\rangle^\eta \overset{\bar{j}:\bar{\ell}}{\Longrightarrow}{}^{p,n}_{\alpha}\ \mathscr{M}'$$

for some $\mathscr{M}'$. ∎

Theorem 28 (Soundness of $\lambda\mu$proberef*).* $v \approx^N_{\varepsilon,B} w$ *implies* $v \simeq^N_{\varepsilon,B} w$

PROOF. Take any η and any context NE such that $vrefs(NE) = \emptyset$ and $hrefs(NE) \subseteq \eta$, and suppose that $\eta \vdash \mathscr{T}(NE)[v] \downarrow_{p,n} a$ with $p > \varepsilon$ and $n \leq B$. We know by the Decomposition Lemma above that there is a dual transition path

$$korder(\eta) \vdash \langle\!\langle NE, v\rangle\!\rangle^\eta \overset{\bar{j}:\bar{\ell}}{\Longrightarrow}{}^{p,n}_{\alpha}\ \mathscr{M}$$

for some $\bar{\ell}, \bar{j}, \overline{\alpha}, \mathscr{M}$. Note that $\mathscr{M}$ must contain $[\![a]\!]$null in the context-term or term position (we write $\mathscr{M} \downarrow a$ for this). Further, by projection we know that this yields a trace $\eta \vdash \langle v\rangle \overset{\bar{\ell}^+}{\Longrightarrow}_{p,n} \mathscr{N}$ for some $\bar{\ell}^+$ and $\mathscr{N}$. It is easy to check that either this trace is balanced, or may be deterministically extended to a balanced one. This follows because $\mathscr{M} \downarrow a$ and thus either the trace is balanced and node contains the _ marker or $\mathscr{N} \downarrow a$. In the latter case, we simply extend the trace with a $\mathbf{tvr}_a$null transition. By hypothesis we therefore also have $\eta \vdash \langle w\rangle \overset{\bar{\ell}^+}{\Longrightarrow}_{p,n} \mathscr{N}'$ and hence, by Lemma 27, we know

$$korder(\eta) \vdash \langle\!\langle \mathscr{T}(NE), w\rangle\!\rangle^\eta \overset{\bar{j}:\bar{\ell}}{\Longrightarrow}{}^{p,n}_{\alpha}\ \mathscr{M}'$$

Also note that $\mathscr{M}' \downarrow a$ holds because either the context-term has converged to null as above, or $\mathscr{N}' \downarrow a$ is guaranteed by trace equivalence. Now, Lemma 26 tells us that $\eta \vdash \mathscr{T}(NE)[w] \downarrow_{p,n} a$ as required. ∎

Corollary 29. For closed $\lambda\mu$hashref *terms,* $t \simeq^N u$ *implies* $t \simeq^N_{\varepsilon,B} u$ *whenever* $B \leq \varepsilon N$

PROOF. Suppose $t \simeq^N u$. By definition this tells us that $\lambda\,.\,t \simeq^N \lambda\,.\,u$ and so by Theorem 25, Theorem 24, and Theorem 28 we have $\lambda\,.\,t \simeq^N_{\varepsilon,B} \lambda\,.\,u$. This is sufficient to establish the result. ∎

VII. CONCLUSION

Abadi and Plotkin (2010) initiated the language-based study of the security properties afforded by randomized layout. Their results demonstrate that the expected locality properties of local variables in a high-level user language are preserved with high probability in a low-level implementation language where memory locations are integers and the opponent has the ability to construct attacks to expose user memory by guessing these locations.

Abadi and Plotkin leave open the question of whether these results continue to hold in the presence of more powerful attackers (in terms of control and probability) and for more realistic languages (with memory management and first class references).

In this paper, we take a further step along the way towards establishing a fully general foundational infrastructure to study the general subject of layout randomization. We consider more powerful programming language constructs such as dynamic memory allocation, first class references and continuations. Even with these enhancements, our bounds on the extra size of the memory required to achieve effective randomization are similar to that of Abadi and Plotkin.

While we do not develop the details in this paper, our framework is easily adapted to model probabilistic contexts, which allow opponents to use randomization, thus restoring the symmetry between the user program and opponent.

In our study, we have not formalized the interaction of layout randomization with data structures that are usually laid out contiguously in memory, such as stacks and arrays. We propose to investigate this in future work.

This research was supported by NSF CCF-0915704.

References

M. Abadi and G. Plotkin. On protection by layout randomization. *Computer Security Foundations Symposium*, 0:337–351, 2010.

E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur.*, 8:3–40, February 2005. ISSN 1094-9224.

C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle, and E. Walthinsen. Protecting systems from stack smashing attacks with stackguard. In *In Linux Expo*, 1999.

A. D. Gordon. Bisimilarity as a theory of functional programming. *Electr. Notes Theor. Comput. Sci.*, 1, 1995.

R. Jagadeesan, C. Pitcher, and J. Riely. Open bisimulation for aspects. *T. Aspect-Oriented Software Development*, 5: 72–132, 2009.

Java 6 API. http://download.oracle.com/javase/6/docs/api/, 2011.

G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM. ISBN 1-58113-738-9.

S. B. Lassen. Eager normal form bisimulation. In *LICS*, pages 345–354. IEEE Computer Society, 2005. ISBN 0-7695-2266-1.

S. B. Lassen. Head normal form bisimulation for pairs and the $\lambda\mu$-calculus. In *LICS*, pages 297–306. IEEE Computer Society, 2006.

E. Levy. Smashing the stack for fun and profit. *Phrack*, 49, 1996.

P. Manadhata and J. M. Wing. Measuring a system's attack surface. Technical report, IN, 2004.

A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 191–203, New York, NY, USA, 1988. ACM.

M. Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *LPAR*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 1992. ISBN 3-540-55727-X.

A. Pitts and I. Stark. On the observable properties of higher order functions that dynamically create local names. In *In Mathematical Foundations of Computer Science, Proc. 18th Int. Symp*, pages 122–141. Springer-Verlag, 1993.

G. Portokalidis and A. D. Keromytis. Fast and practical instruction-set randomization for commodity systems. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 41–48, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0133-6.

D. Sangiorgi. A theory of bisimulation for the pi-calculus. *Acta Inf.*, 33(1):69–97, 1996.

H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM. ISBN 1-58113-961-6.

K. Støvring and S. B. Lassen. A complete, co-inductive syntactic theory of sequential control and state. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 161–172. ACM, 2007. ISBN 1-59593-575-4.

R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pages 1–8, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-472-0.