Independence From Obfuscation: A Semantic Framework for Diversity[*]

Riccardo Pucella
Northeastern University
Boston, MA 02115 USA
riccardo@ccs.neu.edu

Fred B. Schneider
Cornell University
Ithaca, NY 14853 USA
fbs@cs.cornell.edu

January 30, 2006

Abstract

A set of replicas is diverse to the extent that all implement the same functionality but differ in their implementation details. Diverse replicas are less prone to having vulnerabilities in common, because attacks typically depend on memory layout and/or instruction-sequence specifics. Recent work advocates using mechanical means, such as program rewriting, to create such diversity. A correspondence between the specific transformations being employed and the attacks they defend against is often provided, but little has been said about the overall effectiveness of diversity per se in defending against attacks. With this broader goal in mind, we here give a precise characterization of attacks, applicable to viewing diversity as a defense, and also show how mechanically-generated diversity compares to a well-understood defense, strong typing.

1 Introduction

Computers that execute the same program risk being vulnerable to the same attacks. This explains why the Internet, whose machines typically have much software in common, is so susceptible to viruses, worms, and other forms of malware. It is also a reason that replication of servers does not necessarily enhance the availability of a service in the presence of attacks—geographically-separated or not, server replicas, by definition, will all exhibit the same vulnerabilities and thus are unlikely to exhibit the independence required for obtaining enhanced availablity.

A set of replicas is *diverse* if all implement the same functionality (and likely even the same interface) but differ in their implementation details. Diverse replicas are less prone to having vulnerabilities in common, because attacks typically depend on memory layout and/or instruction sequence specifics. But building multiple distinct versions of a program

[*]This work was mainly performed while the first author was at Cornell University. Supported in part by AFOSR grant F9550-06-0019, National Science Foundation Grants 0430161 and CCF-0424422 (TRUST), ONR Grant N00014-01-1-0968, and a grant from Intel Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

is expensive, so researchers have turned to mechanical means for creating diverse sets of replicas.

Various approaches have been proposed, including relocation and/or padding the run-time stack by random amounts [10, 6, 20], re-arranging basic blocks and code within basic blocks [10], randomly changing the names of system calls [7] or instruction opcodes [13, 4, 3], and randomizing the heap memory allocator [5]. Some of these approaches are more effective than others. For example, Sacham *et al.* [17] derive experimental limits on the address space randomization scheme proposed by Xu *et al.* [20], while Sovarel *et al.* [18] discuss the effectiveness of instruction set randomization and outline some attacks against it.

For mechanically-generated diversity to work as a defense, not only must implementations differ (so they have few vulnerabilities in common), but the detailed differences must be kept secret from attackers. For example, buffer-overflow attacks are generally written relative to some specific run-time stack layout. Alter this layout by rearranging the relative locations of variables and the return address on the stack, and an input designed to perpetrate an attack for the original stack layout is unlikely to succeed. But were the new stack layout to become known by the adversary, then crafting an attack again becomes straightforward.

The idea of transforming a program so that its internal logic is difficult to discern is not new; programs to accomplish such transformations have been called *obfuscators* [8]. An obfuscator τ takes two inputs—a program P and a secret key K—and produces a *morph* $\tau(P, K)$, which is a program whose semantics is equivalent to P but whose implementation differs. Secret key K prescribes which exact transformations are applied in producing $\tau(P, K)$. Note that since P and τ is assumed to be public, knowledge of K would enable an attacker to learn implementation details for morph $\tau(P, K)$ and perhaps even automate the generation of attacks for different morphs.

Barak *et al.* [2] and Goldwasser and Kalai [11] give theoretical limitations on the effectiveness of obfuscators as a way to keep secret the details of an algorithm or its embodiment as a program. This work, however, says little about using obfuscators to create diversity. For creating diversity, we are concerned with preventing an attacker from learning details about the output of the obfuscator (since these details are presumed needed for designing an attack), whereas Barak *et al.* and Goldwasser and Kalai are concerned with preventing an attacker from learning the input to the obfuscator.

Different classes of transformations are more or less effective in defending against different classes of attacks. Although knowing this correspondence is important when designing a set of defenses for a given threat model, knowing the specific correspondences is not the same as knowing the overall power of mechanically-generated diversity as a defense. This paper explores that latter, broader, issue, by

- giving a semantics for proving results about the defensive power of obfuscation;
- giving a precise characterization of attacks, applicable to viewing diversity as a defense;
- developing the thesis that mechanically-generated diversity is comparable to type systems, and deriving an admittedly unusual type system equivalent to obfuscation in the presence of finitely many keys;

- exhibiting, for a C-like language and non-trivial obfuscator, an increasingly tighter sequence of type systems for soundly approximating obfuscation under arbitrary finite sets of keys. Surprisingly, the more accurate type systems are based on information flow. We also show that no type system corresponds exactly to the obfuscator under arbitrary finite sets of keys, and therefore approximations are the best we can achieve.

We proceed as follows. In §2, we outline a semantic framework for exploring obfuscation and its limits as a defense. Subsequent sections then illustrate a use of that framework. In particular, we show how the framework can be applied to C-like languages (§3), and use it to obtain insights about a specific obfuscator for a programming notation (§4) and various type systems (§5). Appendices A, B, and C give detailed semantics for the language and type systems we describe in the main text. Appendix D contains a summary of notation.

2 Attacks and Obfuscators: A Semantic Framework

We assume that a program interacts with its environment through inputs and output actions. Inputs include initial arguments supplied to the program when it is invoked, additional data supplied to the program during execution through communication, and so on. Output actions are presumably sensed by the environment, and they can include changes to part of the state. A program's *behavior* defines a sequence of inputs and output actions.

We define a *semantics* for program P and inputs *inps* to be a set of *executions* comprising sequences of states that represent possible behaviors of the program with the given inputs. An *implementation semantics* $[\![\cdot]\!]_I$ describes behaviors of programs at the level of machine execution: for a program P and inputs *inps*, $[\![P]\!]_I(\mathit{inps})$ is the set of executions of program P on inputs *inps*. For high-level languages, an implementation semantics typically will include an account of memory layout and other machine-level details about execution. Given a program P and input *inps*, the executions given by two different implementation semantics could well be different.

We associate an implementation semantics $[\![P]\!]_I^{\tau,K}$ with each morph $\tau(P, K)$. This approach is quite general and allows us to model various kind of obfuscations:

- If the original program is in a high-level language, then we can take obfuscators to be source-to-source translators and take morphs to be programs in the same high-level language;
- If the original program is object code, then we can take obfuscators to be binary rewriters, and take morphs to be object code as well;
- If the original program is in some source language, then we can take obfuscators to be compilers with different compilation choices, and take morphs to be compiled code.

Attacks are conveyed through inputs. An implementation attack is defined relative to some program P, an obfuscator τ, and a finite set of keys $K_1, \ldots, K_n$. The definition of an *implementation attack on program P relative to obfuscator τ and keys $K_1, \ldots, K_n$* is an input that produces a behavior in some morph $\tau(P, K_i)$ and that cannot be produced by some other morph $\tau(P, K_j)$—presumably because implementation details differ from morph to morph.[1] When morphs are deterministic, the definition of an implementation attack simplifies to being an input that produces different behaviors in some pair of morphs $\tau(P, K_i)$ and $\tau(P, K_j)$.

Whether executions from two different morphs reading the same input encode different behaviors is a bit subtle. On the one hand, their output actions must be identical for two executions to encode the same behavior. On the other hand, different morphs might represent state components and sequence state changes in different ways. Therefore, whether two executions encode the same behavior had better not be defined in terms of the states of these executions being equal or even occurring in the same order. For example, different morphs of a routine that returns a tree (where we consider returning a value from the routine an output action) might allocate that tree in different regions of memory. And even though different addresses are returned by each morph, we would want these executions to be considered equivalent if the trees at the different addresses are equivalent.

We formalize execution equivalence for obfuscator τ using relations $\mathcal{B}_n^\tau(\cdot)$, one for every $n \geq 0$. It is tempting to define $\mathcal{B}_n^\tau(\cdot)$ in terms of an equivalence relation on executions, where executions σ_1 and σ_2 are put in the same equivalence class if and only if they encode the same behavior. This, however, does not work, for reasons we detail below. So for a tuple of executions $(\sigma_1, \ldots, \sigma_n)$ where each execution σ_i is produced by morph $\tau(P, K_i)$ run on an input *inps* (i.e., $\sigma_i \in [\![P]\!]_I^{\tau, K_i}(\mathit{inps})$ holds), we define

$$(\sigma_1, \ldots, \sigma_n) \in \mathcal{B}_n^\tau(P, K_1, \ldots, K_n)$$

to hold if and only if executions σ_1, ..., σ_n all encode the same behavior. When morphs are deterministic programs, and thus σ_i is a unique execution of morph $\tau(P, K_i)$ on input *inps*, then by definition *inps* is an implementation attack whenever $(\sigma_1, \ldots, \sigma_n) \notin \mathcal{B}_n^\tau(P, K_1, \ldots, K_n)$. In the general case when morphs are nondeterministic programs, an input *inps* is an implementation attack if there exists an execution $\sigma_j \in [\![P]\!]_I^{\tau, K_j}(\mathit{inps})$ for some $j \in \{1, \ldots, n\}$ such that for all choices of $\sigma_i \in [\![P]\!]_I^{\tau, K_i}(\mathit{inps})$ (for $i \in \{1, \ldots, j-1, j+1, \ldots, n\}$) we have $(\sigma_1, \ldots, \sigma_n) \notin \mathcal{B}_n^\tau(P, K_1, \ldots, K_n)$.

$\mathcal{B}_n^\tau(\cdot)$ cannot be an equivalence relation on executions because our notion of execution equivalence involves supposing an interpretation for states—an implicit existential quantifier—rather than requiring strict equality of all state components (as we do require for output actions). For example, consider executions σ_1, σ_2, and σ_3, each from a different morph with the same input. Let $\sigma[i]$ denote the ith state of σ, $\sigma[i].v$ the value of variable v in state $\sigma[i]$, and suppose that for all i, $\sigma_1[i].x = \sigma_2[i].x = 10$, $\sigma_3[i].x = 22$ and that location 10 in σ_2 has the same value as location 22 in σ_3. Now, by interpreting x as an integer variable, we conclude $\sigma_1[i].x$ and $\sigma_2[i].x$ are equivalent; by interpreting x as a pointer variable, we conclude $\sigma_2[i].x$ and $\sigma_3[i].x$ are equivalent; but it would be wrong to conclude the transitive consequence: $\sigma_1[i].x$ and $\sigma_3[i].x$ are equivalent. Since equivalence relations are necessarily transitive, an equivalence relation is not well suited to our purpose.

[1] An attack that produces equivalent behavior in all morphs might be called an *interface attack* because it exploits the intended (although apparently poorly chosen) semantics of the program's interface [9]. Without some independent specification, interface attacks are indistinguishable from ordinary program inputs; and mechanically-generated diversity is useless against interface attacks.

3 Example: Execution Equivalence for C-like Languages

States. States in C-like languages model snapshots of memory. In the implementation semantics for such a language, a state must not only associate a value with each variable but the state must also capture details of memory layout so that, for example, pointer arithmetic works. We therefore would model a state as a triple (L, V, M), where

- L is the set of memory locations;
- V is a *variable map*, which associates relevant information with every variable. For variables available to programs, V associates the memory locations where the content of the variable is stored; and for variables used to model program execution, V associates information such as sequences of output actions, inputs, or memory locations holding the current stack location or next instruction to execute;
- M is a *memory map*, which gives the contents of every memory location; thus, $dom(M) = L$ holds.

The domain of variable map V includes *program variables* and *hidden variables.* Program variables are manipulated by programmers explicitly, and each program variable is bound to a finite not necessarily contiguous sequence $\langle \ell_1, \ldots, \ell_k \rangle$ of memory locations in L:

- If $k = 1$, then the variable holds a single value, and ℓ_1 is the memory location where the value of the variable is stored;
- If $k > 1$, then the variable holds multiple values (for instance, it may be an array variable, or a C-like struct variable), and $\ell_1, \ldots, \ell_k$ are the memory locations where the values of the variable are stored;
- If $k = 0$, the variable is not bound in that state.

Hidden variables are not directly accessible to the programmer, being artifacts of the language implementation and execution environment. For our purposes, it suffices to assume that the following hidden variables exist:

- pc records the memory location of the next instruction to execute; it is assumed always bound to an element of $L \cup \{\bullet\}$, where $\bullet$ indicates that the program has terminated;
- $\mathsf{actions}$ records the sequence of output actions that the program has performed. Given a set *Action* of output actions that the program is capable of performing, $\mathsf{actions}$ is a finite sequence of elements from *Action*;
- input holds a (possibly infinite) sequence of inputs still available for reading by the program.

Memory map M assigns to every location in L a value representing the content stored there. A memory location can contain either a data value (perhaps representing an instruction or integer) or another memory location (i.e., a pointer). Thus, what is stored in a memory location is ambiguous, being capable of interpretation as a data value or as a memory location. This ambiguity reflects an unfortunate reality of system implementation languages, such as C, that do not distinguish between integers and pointers.

Executions. Let Σ be the set of states. An execution $\sigma \in [\![P]\!]_I(\mathit{inps})$ of program P in a C-like language, when given input *inps*, can be represented as an infinite sequence σ of states from Σ in which each state corresponds to execution of a single instruction in the preceding state, and in which the following general requirements are also satisfied.

(1) L is the same at all states of σ; in other words, the set of memory locations does not change during execution.

(2) If $\sigma[i].\mathsf{pc} = \bullet$ for some i, then $\sigma[j] = \sigma[i]$ for all $j \geq i$; in other words, if the program has terminated in state $\sigma[i]$, then the state remains unchanged in all subsequent states.

(3) There is either an index i with $\sigma[i].\mathsf{pc} = \bullet$ or for every index i there is an index $j > i$ with $\sigma[j] \neq \sigma[i]$; in other words, an execution either terminates with pc set to $\bullet$ or it does not terminate and changes state infinitely many times.[2]

(4) $\sigma[1].\mathsf{actions} = \langle\rangle$ and for all i, $\sigma[i+1].\mathsf{actions}$ is either exactly $\sigma[i].\mathsf{actions}$, or $\sigma[i].\mathsf{actions}$ with a single additional output action appended; in other words, the initial sequence of output actions performed is empty, and it can increase by at most one at every state.

(5) $\sigma[1].\mathsf{inputs} = \mathit{inps}$ and for all i, $\sigma[i+1].\mathsf{inputs}$ is either exactly $\sigma[i].\mathsf{inputs}$, or $\sigma[i].\mathsf{inputs}$ with the first input removed; in other words, input values only get consumed, and at most one input is consumed at every execution step.

Equivalence of Executions. The formal definition of $\mathcal{B}_n^\tau(P, K_1, \ldots, K_n)$ for a C-like language is based on relating the executions of morphs to executions in a suitably chosen *high-level semantics* of the original program. A high-level semantics $[\![\cdot]\!]_H$ also associates a sequence of states with an input but comes closer to capturing the intention of a programmer—it may, for example, be expressed as execution steps of a virtual machine that abstracts away how data is represented in memory, or it may distinguish the intended use of values that have the same internal representation (e.g., integer values and pointer values in C). Intuitively, executions from different morphs of P are deemed equivalent if it is possible to rationalize each of those executions in terms of a single execution in the high-level semantics of P.

To relate executions of morphs to executions in the high-level semantics, we assume a *deobfuscation relation* $\delta(P, K_i)$ between executions σ_i of $\tau(P, K_i)$ and executions $\widehat{\sigma}$ in the high-level semantics $[\![P]\!]_H(\cdot)$ of P, where $(\sigma_i, \widehat{\sigma}) \in \delta(P, K_i)$ means that execution σ_i can be rationalized to execution $\widehat{\sigma}$ in the high-level semantics of P. A necessary condition for morphs to be equivalent is that they perform the same output actions and read the same inputs; therefore, relation $\delta(P, K_i)$ must satisfy

$$\text{For all } (\sigma_i, \widehat{\sigma}) \in \delta(P, K_i): \quad \mathit{Obs}(\sigma_i) = \mathit{Obs}(\widehat{\sigma}),$$

where $\mathit{Obs}(\sigma)$ extracts the sequence of output actions performed and inputs remaining to be consumed by execution σ. So $\mathit{Obs}(\sigma)$ is defined by projecting the bindings of the $\mathsf{actions}$

[2]This rules out direct loops, such as statements of the form ℓ : *goto* ℓ. This restriction does not fundamentally affect our results, but is technically convenient.

```
main(i : int) {
   observable ret
   var ret : int;
       buf : int[3];
       tmp : *int;
   ret := 99;
   tmp := &buf + i;
   *tmp := 42;
}
```

Figure 1: Typical Toy-C program

and the inputs hidden variables

$$(\sigma[1].\mathsf{actions}, \sigma[1].\mathsf{inputs})\ (\sigma[2].\mathsf{actions}, \sigma[2].\mathsf{inputs})\ \ldots$$

and removing repetitions in the resulting sequence. Such a relation for a family of obfuscations and a C-like language is given in §4.

Given a tuple of executions $(\sigma_1, \ldots, \sigma_n)$ for a given input *inps* where each σ_i is produced by morph $\tau(P, K_i)$, these executions are equivalent if they all correspond to the same execution in the high-level semantics $[\![P]\!]_H(\cdot)$ of program P. This is formalized as follows.

$$\begin{array}{l} (\sigma_1, \ldots, \sigma_n) \in \mathcal{B}^{\tau}_n(P, K_1, \ldots, K_n) \text{ if and only if} \\ \quad \text{Exists } \widehat{\sigma} \in [\![P]\!]_H(\mathit{inps}) \\ \qquad \text{For all } i: \quad \sigma_i \in [\![P]\!]^{\tau,K_i}_I(\mathit{inps}) \wedge (\sigma_i, \widehat{\sigma}) \in \delta(P, K_i). \end{array} \tag{1}$$

4 Concrete Example: The Toy-C Language

4.1 The Language

In order to give a concrete example of how to use our framework to reason about diversity and attacks, we introduce a toy C-like language, Toy-C. This language is similar to languages used in the literature to study C features (e.g., aliasing [19]). The syntax and operational semantics of Toy-C programs should be self-explanatory. We only outline the language here, giving complete details in Appendix A.

Figure 1 presents a typical Toy-C program. A program is a list of procedure declarations, where each procedure declaration gives local variable declarations (introduced by var) followed by a sequence of statements. Additionally, every procedure can optionally be annotated with the variables that are observable—that is, variables that can be examined by the environment. An update to an observable variable is an output action.[3] Observable variables are specified on a per-procedure basis. Whether a variable is observable or not does not affect the execution of a program; the annotation is used only for determining equivalence of executions (see §4.4).

[3]Although it is possible to model this behavior directly by defining suitable output actions corresponding to updating these observable variables, the technical development turns out simpler when we track observable variables separately from output actions.

Procedure main is the entry point of the program. Procedure parameters and local variables are declared with types, which are used only to convey representations for values. Types such as $*$int represent pointers to values (in this case, pointers to values of type int). Types such as int[4] represent arrays (in this case, an array with four entries); arrays are 0-indexed and can appear only as the type of local variables.

Toy-C statements include standard statements of imperative programming languages, such as conditionals, loops, and assignment. Additionally, we assume the following statements are available:

- A statement corresponding to every output action in *Action*, such as printing and sending to the network. For simplicity, we identify a statement with the output action that it performs.

- The statement fail simply terminates execution with an error.

As in most imperative languages, we distinguish between expressions that evaluate to values (*value-denoting expressions*, or VD-expressions for short), and expressions that evaluate to memory locations (*address-denoting expressions*, or AD-expressions for short). Expressions appearing on the left-hand side of an assignment statement are AD-expressions. VD-expressions include constants, variables, pointer dereference, and address-of and arithmetic operations, while AD-expressions include variables and pointer dereferences. Array operations can be synthesized from existing expressions using pointer arithmetic, in the usual way.

4.2 Reference Semantics

The execution of Toy-C programs is described by a *reference semantics* $[\![\cdot]\!]_I^{ref}$, which we will use as a basis for other semantics defined in subsequent sections. Full details of the reference semantics appear in Appendix A.2.

Reference semantics $[\![\cdot]\!]_I^{ref}$ captures the stack-based allocation found in standard implementations of C-like languages. Values manipulated by Toy-C programs are integers, which are used as the representation both for integers and pointers—the set of memory locations used by the semantics is just the set of integers. To model stack-based allocation, a hidden variable stores a pointer to the top of the stack; when a procedure is called, the arguments to the procedure are pushed on the stack, the return address is pushed on the stack, and space for storing the local variables is allocated on the stack. Upon return from a procedure, the stack is restored by popping off the allocated space, return address, and arguments of the call. We assume that push increments the stack pointer, and pop decrements it.

4.3 Vulnerabilities

Reference semantics $[\![\cdot]\!]_I^{ref}$ of Toy-C does not mandate safety checks when dereferencing a pointer or when adding integers to pointers. Attackers can take advantage of this possibility to execute programs in a way never intended by the programmer, causing undesirable behavior through techniques such as:

- Stack smashing: overflowing a stack-allocated buffer to overwrite the return address of a procedure with a pointer to attacker-supplied code (generally supplied in the buffer itself);
- Arc injection: using a buffer overflow to change the control flow of the program and jumping to an arbitrary location in memory;
- Pointer subterfuge: modifying a pointer's value (e.g., a function pointer) to point to attacker-supplied code;
- Heap smashing: exploiting the implementation of the dynamic memory allocator, such as overwriting the header information of allocated blocks so that an arbitrary memory location is modified when the block is freed.

Pincus and Baker [16] gives an overview of these techniques. All involve updating a memory location that the programmer thought could not be affected.

Let us consider a threat model in which attackers are allowed to invoke programs and supply inputs to that invocation. These inputs are used as arguments to the main procedure of the program. For example, consider the program of Figure 1. According to reference semantics $[\![\cdot]\!]_I^{ref}$, on input 0,1, or 2, the program terminates in a final state where ret is bound to a memory location containing the integer 99. However, on input -1, the program terminates in a final state where ret is bound to a memory location containing the integer 42; the input -1 makes the variable tmp point to the memory location bound to variable ret, which (according to reference semantics $[\![\cdot]\!]_I^{ref}$) precedes buf on the stack, so that the assignment $*$tmp := 42 stores 42 in the location associated with ret. Presumably, this behavior is undesirable, and input -1 ought to be considered an attack.

4.4 An Obfuscator

An obfuscator that relies on *address obfuscation* to protect against buffer overflows was defined by Bhaktar *et al.* [6]. It attempts to ensure that memory outside an allocated buffer cannot be reliably accessed using statements intended for accessing the buffer.

This obfuscator, which we will call τ_{addr}, relies on the following transformations: varying the starting location of the stack; adding padding around procedure arguments on the stack, blocks of local variables on the stack, and the return location of a procedure call on the stack; permuting the allocation order of variables and the order of procedure arguments on the stack; and supplying a potential different initial memory map.[4]

Keys for τ_{addr} are tuples $(\ell_s, d, \Pi, M_{init})$ describing which transformations to apply: ℓ_s is a starting location for the stack; d is a padding size; $\Pi = (\pi_1, \pi_2, \dots)$ is a sequence of permutations, with π_n (for each $n \geq 1$) a permutation of the set $\{1, \dots, n\}$; and M_{init} represents the initial memory map in which to execute the morph. Morph $\tau_{addr}(P, K)$ is program P compiled under the above transformations.

Implementation semantics $[\![P]\!]_I^{\tau_{addr},K}$ specifying how to execute morph $\tau(P, K)$ is obtained by modifying reference semantics $[\![P]\!]_I^{ref}$ to take into account the transformations

[4]This also lets us model the unpredictability of values stored in memory on different machines running morphs.

prescribed by key K. These modifications affect procedure calls; more precisely, with implementation semantics $[\![P]\!]_I^{\tau_{addr},K}$ for $K = (\ell_s, d, \Pi, M_{init})$, procedure calls now execute as follows:

- d locations of padding are pushed on the stack;
- the arguments to the procedure are pushed on the stack, in the order given by permutation π_n, where n is the number of arguments—thus, if $v_1, \ldots, v_n$ are arguments to the procedure, then they are pushed in order $v_{\pi_n(1)}, \ldots, v_{\pi_n(n)}$;
- d locations of padding are pushed on the stack;
- the return address of the procedure call is pushed on the stack;
- d locations of padding are pushed on the stack;
- memory for the local variables is allocated on the stack, in the order given by permutation π_n, where n is the number of local variables;
- d locations of padding are pushed on the stack;
- the body of the procedure executes.

Full details of implementation semantics $[\![P]\!]_I^{\tau_{addr},K}$ are given in Appendix B.

Notice, which input values cause undesirable behavior (e.g., input -1 causing ret to get value 42 if supplied to the program of Figure 1) depends on which morph is executing—if the morph uses a padding value d of 2 and an identity permutation, for instance, then -3 causes the undesirable behavior in the morph that -1 had caused.

To instantiate $\mathcal{B}_n^{\tau_{addr}}(\cdot)$ for Toy-C and τ_{addr}, we need a description of the intended high-level semantics and deobfuscation relations.

The high-level semantics $[\![\cdot]\!]_H$ that serves our purpose is a variant of reference semantics $[\![\cdot]\!]_I^{ref}$, but where values are only used as the high-level language programmer expects. For example, integers are not used as pointers. Our high-level semantics for Toy-C distinguishes between *direct values* and *pointers*. Roughly speaking, a direct value is a value that is intended to be interpreted literally—for instance, an integer representing some count. In contrast, a pointer is intended to be interpreted as a stand-in for the value stored at the memory location pointed to; the actual memory location given by a pointer is typically irrelevant.[5]

Executions in high-level semantics $[\![\cdot]\!]_H$ are similar to executions described in §3, using states of the form $(\widehat{L}, \widehat{V}, \widehat{M})$, where the set of locations $\widehat{L}$ is $\mathbb{N}$, $\widehat{V}$ is the variable map, and $\widehat{M}$ is the memory map. To account for the intended use of values, the memory map associates with every memory location a tagged value $c(v)$, where tag c indicates whether the value v is meant to be used as a direct value or as a pointer. Specifically, a memory map $\widehat{M}$ associates with every memory location $\widehat{\ell} \in \widehat{L}$ a tagged value

[5]Other high-level semantics are possible, of course, and the framework we propose can accommodate them. For instance, a high-level semantics could additionally model that arrays are never accessed beyond their declared extent. Different high-level semantics generally lead to different notions of equivalence of executions.

- $\mathit{direct}(v)$ with $v \in \mathit{Value}$, indicating that $\widehat{M}(\widehat{\ell})$ contains direct value v; or
- $\mathit{pointer}(\widehat{\ell'})$ with $\widehat{\ell'} \in \widehat{L}$, indicating that $\widehat{M}(\widehat{\ell})$ contains pointer $\widehat{\ell'}$.

Deobfuscation relations $\delta(P, K)$ for τ_{addr} are based on the existence of relations between individual states of executions, where these relations rationalize an implementation state in terms of a high-level state. More precisely, an execution $\sigma \in [\![P]\!]_I^{\tau_{addr},K}(\mathit{inps})$ in the implementation semantics of $\tau_{addr}(P, K)$ and an execution $\widehat{\sigma} \in [\![P]\!]_H(\mathit{inps})$ in the high-level semantics of P are related through $\delta(P, K)$ if there exists a relation $\precsim$ on states (subject to a property that we describe below) such that for some stuttered sequence $\widehat{\sigma}'$ of $\widehat{\sigma}$,[6] we have

$$\text{For all } j: \quad \sigma[j] \precsim \widehat{\sigma}'[j].$$

The properties we require of relation $\precsim$ capture how we are allowed to interpret the states of morph $\tau_{addr}(P, K)$. There is generally a lot of flexibility in this interpretation. For our purposes, it suffices that $\precsim$ allows morphs to allocate variables at different locations in memory, and captures the intended use of values. Generally, relation $\precsim$ might also need to relate states in which values have different representations; there is no need for this with obfuscator τ_{addr}, since τ_{addr} does not change how values are represented.

The required property of relation $\precsim$ is that there exists a map h (indexed by implementation states in σ) that, for any given j, maps memory locations in $\sigma[j]$ to memory locations in $\widehat{\sigma}'[j]$, such that h *determines* $\precsim$. The map is parameterized by implementation states so that it may be different at every state of an execution, since a morph might reuse the same memory location for different variables at different points in time.

A map h determines $\precsim$ when, roughly speaking, $\precsim$ relates implementation states and high-level states that are equal in all components, except that data in memory location ℓ in the implementation state s is found at memory location $h(s, \ell)$ in the high-level state. Formally, h determines $\precsim$ when the relation satisfies the following property: $(L, V, M) \precsim (\widehat{L}, \widehat{V}, \widehat{M})$ holds if and only if

(1) Either $V(\mathsf{pc}) = \bullet$ and $\widehat{V}(\mathsf{pc}) = \bullet$, or $h((L, V, M), V(\mathsf{pc})) = \widehat{V}(\mathsf{pc})$;

(2) $V(\mathsf{actions}) = \widehat{V}(\mathsf{actions})$;

(3) $V(\mathsf{inputs}) = \widehat{V}(\mathsf{inputs})$;

(4) For every observable program variable x, there exists $k \geq 0$ such that $V(x) = \langle \ell_1, \ldots, \ell_k \rangle$, $\widehat{V}(x) = \langle \widehat{\ell_1}, \ldots, \widehat{\ell_k} \rangle$, and for all $i \leq k$ we have $\ell_i \precsim \widehat{\ell_i}$,

where $\ell \precsim \widehat{\ell}$ relates implementation locations $\ell \in L$ and high-level locations $\widehat{\ell} \in \widehat{L}$ and captures when these locations hold similar structures. It is the smallest relation such that $\ell \precsim \widehat{\ell}$ holds if whenever $h(\sigma[j], \ell) = \widehat{\ell}$ holds then so does one of the following conditions:

- $M(\ell) = v$ and $\widehat{M}(\widehat{\ell}) = \mathit{direct}(v)$;
- $M(\ell) = \ell'$, $\widehat{M}(\widehat{\ell}) = \mathit{pointer}(\widehat{\ell'})$ and $\ell' \precsim \widehat{\ell'}$.

Given this definition of deobfuscation relations, it is now immediate to define equivalence of executions for morphs using definition (1).

[6] $\widehat{\sigma}'$ is a *stuttered sequence* of $\widehat{\sigma}$ if $\widehat{\sigma}'$ can be obtained from $\widehat{\sigma}$ by replacing individual states by a finite number of copies of that state.

5 Obfuscation and Type Systems

Obfuscation does not eliminate vulnerabilities—it just makes exploiting them more difficult. Systematic methods for eliminating vulnerabilities not only form an alternative defense but arguably define standards against which obfuscation could be compared. The obvious candidate is type systems, which can prevent attackers from abusing knowledge of low-level implementation details and performing unexpected operations. For example, strong typing as found in Java would prevent overflowing a buffer (in order to alter a return address on the stack) because it is a type violation to store more data into a variable than that variable was declared to accommodate.

Type systems for system programming languages, and strong typing in particular, are generally concerned with ruling out two kinds of behaviors:

(1) Assigning an inappropriate value to some variable.

(2) Accessing memory past the end of a buffer under the pretense of accessing the buffer.

In a language with values of different sizes, we might want to defend against (1), but there is no need to worry about this with Toy-C because every value fits in a single memory location. Thus, our focus here is on (2).

Eliminating vulnerabilities is clearly preferable to having them be difficult to exploit. So why bother with obfuscation? The answer is that there are settings where type systems are not an option—legacy code. The relative success of recent work [12, 14] in adding strong typing to languages like C not withstanding, obfuscation is applicable to any object code, independent of what language was originally used. There are also settings where type systems are not desirable because of cost. For example, most strongly-typed languages involve checking that every access to an array is within bounds. Such checks can be expensive. A careful comparison between obfuscation and type systems then helps understand the trade-offs between the two approaches.

To compare obfuscation with type systems, we can profitably regard obfuscation as a form of *probabilistic type checking*, whereby type-incorrect operations cause the program to halt with some probability p but with probability $1-p$ a type-incorrect operation is allowed to proceed. With a sufficiently good obfuscator, an attempt to overwrite a variable will, with high probability, trigger an illegal operation and cause the program to halt (because the attacker will not have known enough about storage layout), which is exactly the behavior expected from probabilistic type checking.

We start our comparison by discussing how the kind of strong typing being advocated with programming languages, such as Java, compares to what can be achieved with obfuscation, and by obfuscator τ_{addr} in particular. Specifically, we show that strong typing does eliminate all vulnerabilities targeted by τ_{addr}, but strong typing also signals type errors for programs and inputs that are not considered attacks relative to τ_{addr}. This discrepancy prompts us to investigate how to weaken strong typing to capture more accurately what τ_{addr} accomplishes.

All of the type systems we study are *dynamic* type systems—extra information is associated with values, and this information is checked during execution. When the check detects a type error, execution is halted. Admittedly, this is a very general notion of type

```
main() {                          main() {
  observable x                      observable pa
  var a : int[5];                   var a : int[5];
      x : int;                          pa : *int;
  x := *(&a + 10);                  pa := &a + 10;
}                                   *pa := 0;
                                  }

         (a)                               (b)
```

Figure 2: Accessing memory outside a buffer

system. It encompasses all type systems in the literature, but also includes approaches that are not typically viewed as type systems.

5.1 Strong Typing for Toy-C

Obfuscator τ_{addr} is intended to defend against attacks that involve accessing memory outside the extent of a buffer. Thus, to eliminate the vulnerabilities targeted by τ_{addr}, a type system only has to check that a memory read or write through a pointer into a buffer allocated to a variable does not access memory outside that buffer.

In Toy-C, there are only two ways in which this memory access can happen. First, the program can read a value using a pointer that has been moved past the extent of a buffer, as in Figure 2(a).[7] Second, the program can write through a pointer that has been moved past either end of a buffer, as in Figure 2(b). Our type system must abort executions of these programs.

To put strong typing into Toy-C, we associate information with values manipulated by programs. More precisely, values will be represented as pairs $\langle i, \mathbf{int}\rangle$—an *integer value* i—and $\langle i, \mathbf{ptr}(\mathit{start}, \mathit{end})\rangle$—a *pointer value* i pointing to a buffer starting at address *start* and ending at address *end* [12, 14]. Our type system T^{strg} enforces the following invariant: whenever a pointer value $\langle i, \mathbf{ptr}(\mathit{start}, \mathit{end})\rangle$ is dereferenced, it must satisfy $\mathit{start} \leq i \leq \mathit{end}$.[8] Information associated with values is tracked and checked during expression evaluation, as follows.

S1. The representation of a constant i is $\langle i, \mathbf{int}\rangle$.

[7] While reading a value is not by itself generally considered an attack, allowing an attacker to read an arbitrary memory location can be used to mount attacks.

[8] An alternate form of strong typing is to enforce the following, stronger, invariant: every pointer value $\langle i, \mathbf{ptr}(\mathit{start}, \mathit{end})\rangle$ satisfies $\mathit{start} \leq i \leq \mathit{end}$. This alternate form has the advantage of being enforceable whenever a pointer value is constructed, rather than when a pointer value is used. Compare the two programs in Figure 3. According to this alternate form of strong typing, both programs signal a type error when evaluating expression $\&\mathsf{a} + 10$: it evaluates to a pointer value outside its allowed range (viz., extent of a). But although signalling a type error seems reasonable for program (a) in Figure 3, it seems inappropriate with program (b) because this problematic pointer value is never actually used. Note that morphs created by τ_{addr} will not differ in behavior when executing program (b).

```
main() {
  observable x
  var a : int[5];
      pa : *int;
      x : int;
  pa := &a + 10;
  x := *pa;
}
```

(a)

```
main() {
  observable x
  var a : int[5];
      pa : *int;
      x : int;
  pa := &a + 10;
  x := 10;
}
```

(b)

Figure 3: Signalling type errors at pointer value construction versus use

S2. Dereferencing an integer value results in a type error. The result of dereferencing a pointer value $\langle i, \mathbf{ptr}(\mathit{start}, \mathit{end})\rangle$ returns the content of memory location i; however, if i is not in the range delimited by *start* and *end*, then a type error is signalled.

S3. Taking the address of an AD-expression *lv* denoting an address i returns a pointer value $\langle i, \mathbf{ptr}(\mathit{start}, \mathit{end})\rangle$, where *start* and *end* are the start and end of the buffer in which address i is located.

S4. An addition operation signals a type error if both summands are pointer values; if both summands are integer values, the result is an integer value; if one of the summands is a pointer value $\langle i, \mathbf{ptr}(\mathit{start}, \mathit{end})\rangle$ and the other an integer value $\langle i', \mathbf{int}\rangle$, the operation returns $\langle i + i', \mathbf{ptr}(\mathit{start}, \mathit{end})\rangle$.

S5. An equality test signals a type error if the operands are not both integer values or both pointer values.

To illustrate type system T^{strg}, consider the program of Figure 2(a). Assume that variable a is allocated at memory location ℓ_{a}. To execute x := $*(\&\mathsf{a} + 10)$, the expression $*(\&\mathsf{a} + 10)$ is evaluated. The expression &a evaluates to

$$\langle \ell_{\mathsf{a}}, \mathbf{ptr}(\ell_{\mathsf{a}}, \ell_{\mathsf{a}} + 4)\rangle,$$

by S3. The constant 10 evaluates to

$$\langle 10, \mathbf{int}\rangle,$$

by S1. The sum &a + 10 is type correct (see S4), because no more than one summand is a pointer, and yields

$$\langle \ell_{\mathsf{a}} + 10, \mathbf{ptr}(\ell_{\mathsf{a}}, \ell_{\mathsf{a}} + 4)\rangle.$$

However, the subsequent dereference $*\langle \ell_{\mathsf{a}} + 10, \mathbf{ptr}(\ell_{\mathsf{a}}, \ell_{\mathsf{a}} + 4)\rangle$ signals a type error because location $\ell_{\mathsf{a}} + 10$ is out the range delimited by ℓ_{a} and $\ell_{\mathsf{a}} + 4$.

The same thing happens in the program of Figure 2(b). Assume that variable a is allocated at memory location ℓ_{a}, and that variable pa is allocated at memory location ℓ_{pa}. The first statement executes by evaluating &a + 10 to a value

$$\langle \ell_{\mathsf{a}} + 10, \mathbf{ptr}(\ell_{\mathsf{a}}, \ell_{\mathsf{a}} + 4)\rangle,$$

as before. The left-hand side of the assignment statement, pa is evaluated to the value representing the memory location allocated to variable pa, namely

$$\langle \ell_{\mathsf{pa}}, \mathbf{ptr}(\ell_{\mathsf{pa}}, \ell_{\mathsf{pa}}) \rangle;$$

the assignment stores value $\langle \ell_{\mathsf{a}} + 10, \mathbf{ptr}(\ell_{\mathsf{a}}, \ell_{\mathsf{a}} + 4) \rangle$ in location ℓ_{pa}. Note that values are not checked when they are stored. The next assignment statement, however, signals a type error. The right-hand side evaluates to the value

$$\langle 0, \mathbf{int} \rangle,$$

but the left-hand side attempts a dereference of the value stored in pa, that is, attempts the dereference $*\langle \ell_{\mathsf{a}} + 10, \mathbf{ptr}(\ell_{\mathsf{a}}, \ell_{\mathsf{a}} + 4) \rangle$, which signals a type error because location $\ell_{\mathsf{a}} + 10$ is not in the range delimited by ℓ_{a} and $\ell_{\mathsf{a}} + 4$.

To formalize how Toy-C programs execute under type system T^{strg}, we extend reference semantics $[\![\cdot]\!]_I^{ref}$ to track the types of values. The resulting implementation semantics $[\![\cdot]\!]_I^{strg}$ appears in Appendix C.1. One modification to $[\![\cdot]\!]_I^{ref}$ is that $[\![\cdot]\!]_I^{strg}$ uses values of the form $\langle i, t \rangle$, where i is an integer and t is a type, as described above.

Attacks disrupted by obfuscator τ_{addr} lead to type errors in Toy-C equipped with T^{strg}. The following theorem makes this precise.

Theorem 5.1. *Let $K_1, \ldots, K_n$ be arbitrary keys for τ_{addr}. For any program P and inputs inps, if inps is an attack on P relative to τ_{addr} and $K_1, \ldots, K_n$, then $\sigma \in [\![P]\!]_I^{strg}(inps)$ signals a type error. Equivalently, if $\sigma \in [\![P]\!]_I^{strg}(inps)$ does not signal a type error, then inps is not an attack on P relative to τ_{addr} and $K_1, \ldots, K_n$*

Proof. See Appendix C.1. ∎

Thus, T^{strg} is a sound approximation of τ_{addr}, in the sense that it signals type errors for all inputs that are attacks relative to τ_{addr} and a finite set of keys $K_1, \ldots, K_n$. This confirms our thesis that there is a connection between type systems and obfuscation. Moreover, any type system that is more restrictive than T^{strg} and therefore causes more executions to signal a type error will also have the property given in Theorem 5.1.

Notice that τ_{addr} and T^{strg} do not impose equivalent restrictions. Not every input for which T^{strg} signals a type error corresponds to what we consider to be an attack. When executing the program of Figure 4(a), for instance, T^{strg} signals a type error because $\&\mathsf{a} + 10$ yields a pointer that cannot be dereferenced. But there is no an attack relative to τ_{addr} because morphs created by τ_{addr} do not differ in their behavior, since output action $\mathsf{print}(0)$ is always going to be executed.

Figure 4(a) is a program for which strong typing is stronger than necessary—at least if one accepts our definition of an attack as input that leads to differences in observable behavior. So in the remainder of this section, we examine weakenings of T^{strg} with the intent of more tightly characterizing the attacks τ_{addr} targets.

5.2 A Tighter Type System for Obfuscator τ_{addr}

One way to understand the difference between τ_{addr} and T^{strg} is to think about integrity of values. Intuitively, if a program accesses a memory location through a corrupted pointer,

```
main() {
  var a : int[5];
      x : int;
  x := *(&a + 10);
  print(0);
}
```

(a)

```
main() {
  var a : int[5];
      x : int;
  x := *(&a + 10);
  if (x = 0) then {
    print(1);
  } else {
    print(2);
  }
}
```

(b)

```
main() {
  var a : int[5];
      x : int;
  x := *(&a + 10);
  if (x = x) then {
    print(1);
  } else {
    print(2);
  }
}
```

(c)

Figure 4: Sample programs

then the value computed from that memory access has low integrity. This is enforced with τ_{addr} by having different morphs compute different values. We thus distinguish between values having *low integrity*, which are obtained by somehow abusing pointers, and values having *high integrity*, which are not. This suggests equating integrity with variability under τ_{addr}; a value has low integrity if and only if it differs across morphs.

If we require that output actions cannot depend on values with low integrity, then execution should be permitted to continue after reading a value with low integrity. This is the key insight for a defense, and it will be exploited for the type system in this section.

Tracking whether high-integrity values depend on low-integrity values can be accomplished using information flow analyses, and type systems have been developed for this, both statically [1] and dynamically [15].

We adapt T^{strg} and design a new type system T^{info} that takes integrity into account. Roughly speaking, a new type **low** is associated with any value having low integrity. Rather than signalling a type error when dereferencing a pointer to a memory location that lies outside its range, the type of the value extracted from the memory location is set to **low**. The resulting implementation semantics $[\![P]\!]_I^{info}$ appears in Appendix C.2.

T^{info} will signal a type error whenever an output action is attempted and that output action depends on a value with type **low**. In other words, if control reaches an output action due to a value with type **low**, then a type error is signalled. So, for example, if a conditional statement branches based on a guard that depends on values with type **low**, and one of the branches performs an output action, then a type error is signalled. To implement T^{info}, we track when control flow depends on values with type **low**. This is achieved by associating a type not only with values stored in program variables, but with the content of the program counter itself, in such a way that the program counter has type **low** if and only if control flow somehow depended on values with type **low**.

Consider Figure 4(a). When executing $\mathsf{x} := *(\&\mathsf{a} + 10)$ in that program, the expression $\&\mathsf{a} + 10$ evaluates to $\langle \ell_{\mathsf{a}} + 10, \mathbf{ptr}(\ell_{\mathsf{a}}, \ell_{\mathsf{a}} + 4)\rangle$ (using a similar reasoning as for T^{strg}), and therefore, because location $\ell_{\mathsf{a}} + 10$ is outside its range, $*(\&\mathsf{a} + 10)$ evaluates to $\langle i, \mathbf{low}\rangle$ for some integer i—the actual integer is unimportant, since it having type **low** will prevent the integer from having an observable effect. The value $\langle i, \mathbf{low}\rangle$ is never used in the rest of the program, so execution proceeds without signalling a type error (in contrast to T^{strg}, which does signal a type error).

By way of contrast, consider Figure 4(b). When executing the if statement in that program, $*(\&\mathsf{a} + 10)$ evaluates to $\langle i, \mathbf{low}\rangle$ (for some integer i), and 0 evaluates to $\langle 0, \mathbf{int}\rangle$. Comparing these two values yields a value with type **low**, since one of the values in the guard had type **low**. (Computing using a value of low integrity yields a result of low integrity.) Because the guard's value affects the control flow of the program, the program counter receives type **low** as well. When execution reaches output action $\mathsf{print}(1)$, a type error is signalled because the program counter has type **low**.

Theorem 5.2. *Let $K_1, \ldots, K_n$ be arbitrary keys for τ_{addr}. For any program P and inputs inps, if inps is an attack on P relative to τ_{addr} and $K_1, \ldots, K_n$, then $\sigma \in [\![P]\!]_I^{info}(inps)$ signals a type error.*

Proof. See Appendix C.2. ∎

Thus, just like T^{strg}, T^{info} is a sound approximation of τ_{addr}. And as illustrated by the programs of Figure 4, T^{info} corresponds more closely to τ_{addr} than does T^{strg}. Somewhat surprisingly, information flow therefore captures our definition of attack relative to τ_{addr} more closely than strong typing. But, as we shall see below, T^{info} still aborts executions on inputs that are not attacks relative to τ_{addr}, so T^{info} is still stronger than τ_{addr}.

5.3 Exact Type Systems for Obfuscator τ_{addr}

Consider the program of Figure 4(c). Here, the value read from location $\&\mathsf{a} + 10$ has type **low**, and it is being used in a conditional test that can potentially select between different output actions. However, because equality is reflexive, the fact that we are comparing to a value with type **low** is completely irrelevant, as the guard always yields true. We believe that it would be quite difficult to develop a type system that can identify guards that are validities, because doing so requires a way to decide when two expressions have the same value in all executions. Yet, if we had a more precise way to establish the integrity of the program counter (for instance, by being able to establish that two expressions affecting control flow have the same value in all executions), then we would have a type system that more closely correspond to τ_{addr}.

A type system, admittedly unusual, that signals a type error for exactly those executions corresponding to inputs that are attacks relative to τ_{addr} and some fixed and finite set $K_1, \ldots, K_n$ of keys is the trivial type system, $T^{mrph}_{K_1,\ldots,K_n}$, instantiated by an implementation semantics $[\![P]\!]_I^{mrph,K_1,\ldots,K_n}(inps)$ that repeatedly applies the following procedure. This type system essentially runs all morphs in parallel, taking unanimous consensus before performing an observable action.

> Execute program P up to the next output action (including updates to observable program variables), and also execute morphs $\tau_{addr}(P, K_1), \ldots, \tau_{addr}(P, K_n)$ up to their next output action:

- If the same output action is next about to be performed by all morphs, then the type system allows P to perform its output action, and repeats the procedure;
- If not, then the type system signals a type error and aborts execution.

Theorem 5.3. *Let $K_1, \ldots, K_n$ be arbitrary keys for τ_{addr}. For any program P and inputs inps, inps is an attack on P relative to τ_{addr} and $K_1, \ldots, K_n$ if and only if $\sigma \in [\![P]\!]_I^{mrph,K_1,\ldots,K_n}(inps)$ signals a type error.*

Proof. Immediate from the description of the procedure and the definition of an attack relative to τ_{addr} and $K_1, \ldots, K_n$. ∎

This theorem then establishes that type systems are in fact equivalent to obfuscation under a fixed finite set of keys. As we shall see below, this correspondence can be used to construct a probabilistic type system.

As defined in §4.4, obfuscator τ_{addr} admits infinitely many keys. Although for many applications we care only about a finite set of keys at any given time (e.g., when using morphs to implement server replicas, of which there are only finitely many), the exact set of keys might not be known in advance or may change during the lifetime of the application. Therefore, it is sensible to try to identify inputs that are attacks relative to τ_{addr} and any finite subset of the possible keys, or equivalently, to recognize inputs that are not attacks relative to τ_{addr} and every finite subset of the possible keys.

If we are interested in a type system that signals a type error for exactly executions corresponding to inputs that are attacks relative to τ_{addr} and any finite set of keys, then a type system such as $T^{mrph}_{K_1,\ldots,K_n}$ is no longer feasible. This is because there are infinitely many possible finite sets of keys available for τ_{addr}. Therefore, a type system like $T^{mrph}_{K_1,\ldots,K_n}$ would need to execute infinitely many morphs.

Type systems like $T^{mrph}_{K_1,\ldots,K_n}$ can be viewed as approximating a type system that aborts exactly those executions corresponding to inputs that are attacks relative to τ_{addr} and some finite set of keys. Adding more keys, that is, considering type system $T^{mrph}_{K_1,\ldots,K_n,K'}$, improves the approximation because there are fewer programs and inputs for which $T^{mrph}_{K_1,\ldots,K_n,K'}$ will fail to signal a type error even though the inputs are attacks. This is because every attack relative to τ_{addr} and $K_1, \ldots, K_n$ is an attack relative to τ_{addr} and $K_1, \ldots, K_n, K'$, but not vice versa.

The approximation embodied by type system $T^{mrph}_{K_1,\ldots,K_n}$ can become a probabilistic approximation of the type system that aborts exactly those executions corresponding to inputs that are attacks relative to τ_{addr} and some finite set of keys. Consider a type system T^{rand} that works as follows: before executing a program, keys $K_1, \ldots, K_n$ are chosen at random, and then the type system acts as $T^{mrph}_{K_1,\ldots,K_n}$. For any fixed finite set $K_1, \ldots, K_n$ of keys, $T^{mrph}_{K_1,\ldots,K_n}$ will identify inputs that are attacks relative to τ_{addr} and $K_1, \ldots, K_n$, but may miss inputs that are attacks relative to τ_{addr} and some other finite set of keys. By choosing the set of keys at random, type system T^{rand} has some probability of identifying any input that is an attack relative to some finite set of keys.

Note that $T^{mrph}_{K_1,\ldots,K_n}$ (similarly, T^{rand}) is a fundamentally different kind of approximation than we have with type systems T^{strg} and T^{info}, which are sound approximations: if T^{strg} and T^{info} do not signal a type error for a program P and input *inps*, then *inps* is not an attack relative to τ_{addr} and every finite set of keys. In fact, as we now show, it is impossible to design a type system that aborts executions for exactly those inputs for which there exists a finite set $K_1, \ldots, K_n$ of keys and the input is an attack relative to τ_{addr} and $K_1, \ldots, K_n$.

To simplify the exposition, we focus on type systems that restrict $[\![\cdot]\!]_I^{ref}$: if an execution of program P does not signal a type error, then that execution can be viewed as an execution of $[\![P]\!]_I^{ref}$. Assume a function *val* on the extended values of implementation semantics $[\![\cdot]\!]_I^{ref}$ that extracts the integer being represented by the extended value, stripped of all typing information. For example, the *val* function for $[\![\cdot]\!]_I^{strg}$ is defined by $val(\langle i, t\rangle) = i$. Given an execution σ in an implementation semantics $[\![\cdot]\!]_I$, define the execution $\ulcorner\sigma\urcorner$ to be the execution obtained by replacing every value v in every state of σ by $val(i)$. An implementation semantics $[\![\cdot]\!]_I$ is a *restriction* of $[\![\cdot]\!]_I^{ref}$ if for every program P and input *inps*, whenever $\sigma \in [\![P]\!]_I(inps)$ does not signal a type error, then $\widetilde{\sigma} \in [\![P]\!]_I^{ref}(inps)$ satisfies, for all $i \geq 0$:

(i) $\ulcorner\sigma\urcorner[i].\mathsf{actions} = \widetilde{\sigma}[i].\mathsf{actions}$;

(ii) $\ulcorner\sigma\urcorner[i].\mathsf{inputs} = \widetilde{\sigma}[i].\mathsf{inputs}$;

(iii) For every observable program variable x, $\ulcorner\sigma\urcorner[i].x = \widetilde{\sigma}[i].x$.

It is straightforward to check that the implementation semantics corresponding to type systems T^{strg} and T^{info} are restrictions of $[\![\cdot]\!]_I^{ref}$.

Lemma 5.4. *$[\![\cdot]\!]_I^{strg}$ and $[\![\cdot]\!]_I^{info}$ are restrictions of $[\![\cdot]\!]_I^{ref}$.*

Proof. See Appendix C.2. ∎

Theorem 5.5. *Let $[\![\cdot]\!]_I$ be an implementation semantics for Toy-C such that:*

(i) For every P and inps, $[\![P]\!]_I(inps)$ is computable;

(ii) $[\![\cdot]\!]_I$ is a restriction of $[\![\cdot]\!]_I^{ref}$;

(iii) $\sigma \in [\![P]\!]_I(inps)$ signals a type error whenever inps is an attack relative to τ_{addr} and some finite set of keys.

Then, there exists a program P and input inps such that $\sigma \in [\![P]\!]_I(inps)$ signals a type error, but for all finite sets of keys $K_1, \ldots, K_n$, inps is not an attack relative to τ_{addr} and $K_1, \ldots, K_n$.

Proof. See Appendix C.2. ∎

This shows that it is impossible, in general, to devise a type system that signals a type error exactly when an input is an attack relative to τ_{addr} and an arbitrary finite set of keys. Any type system must therefore approximate this.

This result relies on obfuscator τ_{addr} admitting infinitely many keys. In reality, machines have a bounded amount of memory, and memory locations can only store a bounded number of bits, so program size is bounded. Therefore, it is likely that only finitely many keys are needed to describe all morphs that can be executed on a given machine. This means that there is a possibility of devising a type system that exactly corresponds to τ_{addr} on a finite machine. One possibility might be $T^{mrph}_{K_1,\ldots,K_n}$, although that type system requires a factor of n additional memory to execute programs. We leave the question of devising such exact type systems for finite machines open.

6 Concluding Remarks

This paper gives a reduction from defenses created by mechanically-generated diversity to probabilistic dynamic type checking. We have ignored the probabilities, but for practical application, these values really do matter, because if the dynamic type checking is performed with low probability, then checks are frequently skipped and attacks are likely to succeed. The probabilities, then, are the measure of interest when trying to decide in practice whether mechanically-generated diversity actually adds value. And unfortunately, obtaining these probabilities appears to be a difficult problem. They depend on how much diversity is introduced and how robust attacks are to the resulting diverse semantics. Our framework is thus best seen as a first step in trying to characterize the effectiveness of program obfuscation and other means of mechanically introducing diversity.

A reduction to non-probabilistic type checking—although clearly a stronger result—would not help in characterizing the effectiveness of mechanically-generated diversity, either. This is because there is (to our knowledge) no non-trivial and complete characterization of the attacks that strong typing blunts. Various specific attacks (such as buffer overflows) are known to be blocked. But enumerating which attacks are blocked and which are not constitutes an unsatisfying basis for defining the effectiveness of a defense in a world where new attacks are constantly being perpetrated. We should strive for characterizations that are more abstract—a threat model based on the resources or information available to the attacker, for example. And in the absence of suitable abstract threat models, reductions from one defense to another, like what is being introduced in this paper, might well be the only way to get insight into the relative powers of defenses. Moreover, such reductions remain valuable even after suitable threat models have been developed.

We treat in this paper a specific language, a single obfuscator, and a few simple type systems. Our primary goal, however, was not to analyze these particular artifacts, although the analysis does shed light on how the obfuscators and type systems defend against attacks (and some of the results for these artifacts are surprising). Rather, our goal has been to create a framework that allows such an analysis to be performed for any language, obfuscator, or type system. The hard part, then, was finding a suitable, albeit unconventional, definition of attack and appreciating that probabilistic variants of type systems constitute a useful vocabulary for describing the power of mechanically-generated diversity.

Acknowledgments

Thanks to Michael Clarkson, Matthew Fluet, Greg Morrisett, Andrew Myers, and Tom Roeder for their comments on a draft of this paper.

A A Semantics for Toy-C

A.1 Syntax

Let *Var* be a set of variables, *Proc* be a set of procedure names, and *Action* be a set of output actions.

Syntax of Toy-C:

$P ::=$	program
$\quad pd_1 \ \ldots \ pd_k$	
$pd ::=$	procedure declaration
$\quad m(x_1 : py_1, \ldots, x_m : py_m) \ \{ \ ld; sts \ \}$	$\quad m \in \mathit{Proc}, x_1, \ldots, x_m \in \mathit{Var}$
$ty ::=$	type
$\quad py$	pointer type
$\quad py[i]$	array
$py ::=$	pointer type
$\quad \mathsf{int}$	integer
$\quad *ty$	pointer
$ld ::=$	local declaration list
$\quad \mathsf{var} \ x_1 : ty_1; \ldots ; x_n : ty_n$	$\quad x_1, \ldots, x_n \in \mathit{Var}$
$st ::=$	statement
$\quad lv := ex$	assignment
$\quad m(ex_1, \ldots, ex_n)$	procedure call, $m \in \mathit{Proc}$
$\quad \mathsf{if} \ ex \ \mathsf{then} \ \{ \ sts_1 \ \} \ \mathsf{else} \ \{ \ sts_2 \ \}$	conditional
$\quad \mathsf{while} \ ex \ \mathsf{do} \ \{ \ sts \ \}$	loop
$\quad ac$	output action, $ac \in \mathit{Action}$
$\quad \mathsf{fail}$	exception
$sts ::=$	statement sequence
$\quad \epsilon$	empty sequence
$\quad st; sts$	sequencing
$ct ::=$	constant
$\quad i$	integer
$\quad \mathsf{null}$	null pointer
$ex ::=$	value-denoting expression
$\quad ct$	constant
$\quad *ex$	pointer dereference
$\quad x$	variable, $x \in \mathit{Var}$
$\quad \& lv$	address
$\quad ex_1 + ex_2$	arithmetic
$\quad ex_1 = ex_2$	comparison
$lv ::=$	address-denoting expression
$\quad x$	variable, $x \in \mathit{Var}$
$\quad *lv$	pointer dereference

A program is a set of procedure declarations, where each procedure declaration consists of some local variable declarations followed by a sequence of statements. Every well-formed program defines a procedure main, which is the entry point of the program; the inputs to the programs are just inputs to the procedure main.

Procedure parameters and local variable are declared with types, which are used only to indicate the representation of values. We distinguish between general types *ty* (local variables can be of general types) and pointer types *py* (procedure parameters must be of pointer type). Pointer types represent constants and memory locations (i.e., pointers); we assume that pointer types fit in a single location in memory—that is, constants and pointers fit in a single memory location. In contrast, general types include array types, which can span multiple memory locations. The following function, which computes the size of values of a type, formalizes this.

Size of Types: $size(ty)$

$size(py) \triangleq 1$
$size(py[i]) \triangleq i$

Statements include standard statements of imperative programming languages: assignment, procedure call, conditional, and iteration. Output actions are identified with statements that perform those output actions; thus, we have a statement *ac* for every output action in the set *Action*. Statement fail terminates an execution with an error. Execution of a sequence of statements is terminated with an empty sequence statement ϵ, which we generally omit.

As pointed out in §4.1, we distinguish VD-expressions, which evaluate to values, from AD-expressions, which denote memory locations. For simplicity, we take only integers and the null pointer as constants. Notation $*ex$ (and $*lv$) is used to dereference a pointer, while $\&lv$ is used to return the address of a variable (or, more generally, of an AD-expression). Toy-C does not contain an array dereferencing operator, since it can be synthesized from other operations. For instance, array dereference x[5] in a VD-expression can be written:

$$*(\&\mathsf{x} + 5)$$

(recall that all values fit in a single memory location), and similarly, assignment x[5] := 10 can be written using a temporary variable:

```
var y : *int;
y := (&x + 5);
*x := 10.
```

(To reduce the number of rules we need to consider in the semantics, the more direct $*(\&\mathsf{x} + 5) := 10$ is not allowed by our syntax.) The only operations we include in our semantics are + and =. It is, of course, completely straightforward to add new operations.

A.2 Reference Semantics

In order to define the reference semantics for Toy-C, we need to describe the states; in particular, we need to describe what we take as location structures, variable maps (including all hidden variables), and memory maps.

For simplicity, we take $\mathbb{N}$ as set of memory locations, where the stack and variables live. We assume that first locations of $\mathbb{N}$ are locations that store program code, and that those locations are not readable or writable. Let ℓ_{RW} be the first readable/writable memory location. We assume a map *loc* that returns the location $\mathit{loc}(\mathit{sts})$ in the first K locations of $\mathbb{N}$ where the statement sequence *sts* is stored. Conversely, we assume a map *code* that extracts code from memory: $\mathit{code}(\ell)$ returns a sequence of statements *sts*. Clearly, the functions *loc* and *code* are inverse to each other: $\mathit{code}(\mathit{loc}(\mathit{sts})) = \mathit{sts}$.

Variable maps are standard. We assume the hidden variables required by our semantic framework. The hidden variables pc, $\mathsf{actions}$, and inputs are as before. We also have additional hidden variables: sp holds the current stack pointer, the current memory location at the top of the stack (initially ℓ_s), return holds the location on the stack where the return address of the current procedure invocation lives, and status records whether an evaluation terminated successfully (value **Success**) or failed (value **Fail**). We shall need the empty variable map V_0, which maps every variable to the empty sequence $\langle\rangle$ of locations.

Memory maps are also standard. We shall need the empty memory map M_0, which maps every location to the value 0: $M_0(\ell) = 0$ for all $\ell \in \mathbb{N}$. Our semantics will ensure that programs cannot access memory locations below ℓ_{RW}; any read/write to a memory location below ℓ_{RW} will be interpreted as a read/write to memory location ℓ_{RW}. (An alternative would be to signal a failure when a read/write to a memory location below ℓ_{RW} is attempted.)

Before presenting the semantics, we give functions to evaluate AD-expressions in the language to locations, and VD-expressions to values. The evaluation function $A[\![lv]\!](V, M)$ is used to evaluate an AD-expression *lv* in variable map V and memory map M, and $E[\![ex]\!](V, M)$ is used to evaluate an VD-expression *ex* in variable map V and memory map M.

Evaluation Functions: $A[\![lv]\!](V, M)$**,** $E[\![ex]\!](V, M)$

$$A[\![x]\!](V, M) \triangleq \ell_1 \quad \text{when } V(x) = \langle \ell_1, \ldots, \ell_k \rangle$$

$$A[\![{*lv}]\!](V, M) \triangleq \begin{cases} M(\ell_{RW}) & \text{if } A[\![lv]\!](V, M) < \ell_{RW} \\ M(A[\![lv]\!](V, M)) & \text{otherwise} \end{cases}$$

$$E[\![{*ex}]\!](V, M) \triangleq \begin{cases} M(\ell_{RW}) & \text{if } E[\![ex]\!](V, M) < \ell_{RW} \\ M(E[\![ex]\!](V, M)) & \text{otherwise} \end{cases}$$

$$E[\![i]\!](V, M) \triangleq i$$

$$E[\![\mathsf{null}]\!](V, M) \triangleq 0$$

$$E[\![x]\!](V, M) \triangleq M(\ell_1) \quad \text{when } V(x) = \langle \ell_1, \ldots, \ell_k \rangle$$

$$E[\![\&lv]\!](V, M) \triangleq A[\![lv]\!](V, M)$$

$$E[\![ex_1 + ex_2]\!](V, M) \triangleq E[\![ex_1]\!](V, M) + E[\![ex_2]\!](V, M)$$

$$E[\![ex_1 = ex_2]\!](V, M) \triangleq \begin{cases} 1 & \text{if } E[\![ex_1]\!](V, M) = E[\![ex_2]\!](V, M) \\ 0 & \text{otherwise} \end{cases}$$

Executions making up semantics $[\![P]\!]_I^{ref}(\cdot)$ are constructed using standard reduction rules between *operational states* of the form $(sts, V, M, V\!s)$, where V is a variable map, $V\!s = \langle V_1, \dots, V_k \rangle$ is a stack of variable maps, M is a memory map, and sts is a sequence of statements to execute. A reduction rule of the form $(sts, V, M, V\!s) \longrightarrow (sts', V, M', V\!s')$, describes one step of the execution of sts. To simplify the description of the semantics, we extend the set of statement sequences with special token $\bullet$ to represent termination, in analogy with the values of pc. In the rules below, we use the notation $V\{x \mapsto \langle \ell_1, \dots, \ell_k \rangle\}$ to represent the map V updated to map variable x to $\langle \ell_1, \dots, \ell_k \rangle$, and similarly for memory map M.

Reduction Rules:

$(\bullet, V, M, V\!s) \longrightarrow (\bullet, V, M, V\!s)$ (R1)

$(\epsilon, V, M, V\!s) \longrightarrow (\bullet, V\{\mathsf{pc} \mapsto \bullet\}, M, V\!s)$ (R2)
  if $V(\mathsf{return}) = \bullet$

$(\epsilon, V, M, \langle V_1, \dots, V_k \rangle) \longrightarrow (sts, V_1\{\mathsf{pc} \mapsto \ell\}, M, \langle V_2, \dots V_k \rangle)$ (R3)
  if $V(\mathsf{return}) = \ell'$, $M(\ell') = \ell$, and $code(\ell) = sts$

$(lv := ex; sts, V, M, V\!s) \longrightarrow (sts, V\{\mathsf{pc} \mapsto loc(sts)\}, M\{\ell \mapsto v\}, V\!s)$ (R4)
  if $E[\![ex]\!](V, M) = v$ and $A[\![lv]\!](V, M) = \ell$

$(\mathsf{if}\ ex\ \mathsf{then}\ \{\ sts_1\ \}\ \mathsf{else}\ \{\ sts_2\ \}; sts, V, M, \langle V_1, \dots, V_k \rangle) \longrightarrow$ (R5)
  $(sts_1, V', M\{V(\mathsf{sp}) \mapsto loc(sts)\}, \langle V, V_1, \dots, V_k \rangle)$
  if $E[\![ex]\!](V, M) \neq 0$
  where $V' \triangleq V\{\mathsf{pc} \mapsto loc(sts_1),$
    $\mathsf{sp} \mapsto V(\mathsf{sp}) + 1,$
    $\mathsf{return} \mapsto V(\mathsf{sp})\}$

$(\mathsf{if}\ ex\ \mathsf{then}\ \{\ sts_1\ \}\ \mathsf{else}\ \{\ sts_2\ \}; sts, V, M, \langle V_1, \dots, V_k \rangle) \longrightarrow$ (R6)
  $(sts_2, V', M\{V(\mathsf{sp}) \mapsto loc(sts)\}, \langle V', V_1, \dots, V_k \rangle)$
  if $E[\![ex]\!](V, M) = 0$
  where $V' \triangleq V\{\mathsf{pc} \mapsto loc(sts_2),$
    $\mathsf{sp} \mapsto V(\mathsf{sp}) + 1,$
    $\mathsf{return} \mapsto V(\mathsf{sp})\}$

$(\mathsf{while}\ ex\ \mathsf{do}\ \{\ sts_1\ \}; sts, V, M, \langle V_1, \dots, V_k \rangle) \longrightarrow$ (R7)
  $(sts_1, V', M\{V(\mathsf{sp}) \mapsto loc(\mathsf{while}\ ex\ \mathsf{do}\ \{\ sts_1\ \}; sts)\}, \langle V, V_1, \dots, V_k \rangle)$
  if $E[\![ex]\!](V, M) \neq 0$
  where $V' \triangleq V\{\mathsf{pc} \mapsto loc(sts_1),$
    $\mathsf{sp} \mapsto V(\mathsf{sp}) + 1,$
    $\mathsf{return} \mapsto V(\mathsf{sp})\}$

$(\mathsf{while}\ ex\ \mathsf{do}\ \{\ sts_1\ \}; sts, V, M, V\!s) \longrightarrow (sts, V\{\mathsf{pc} \mapsto loc(sts)\}, M, V\!s)$ (R8)
  if $E[\![ex]\!](V, M) = 0$

$$(\mathsf{fail}; sts, V, M, Vs) \longrightarrow (\bullet, V\{\mathsf{pc} \mapsto \bullet, \mathsf{status} \mapsto \mathbf{Fail}\}, M, Vs) \tag{R9}$$

$$\begin{aligned}&(ac; sts, V, M, Vs) \longrightarrow (sts, V', M, Vs) \\ &\quad \text{where } V' \triangleq V\{\mathsf{pc} \mapsto loc(sts), \\ &\qquad\qquad\qquad \mathsf{actions} \mapsto V(\mathsf{actions}) +\!\!+ \langle ac\rangle\}\end{aligned} \tag{R10}$$

$$\begin{aligned}&(m(ex_1, \ldots, ex_n); sts, V, M, \langle V_1, \ldots, V_k\rangle) \longrightarrow (sts_m, V', M', \langle V, V_1, \ldots, V_k\rangle) \\ &\quad \text{if } E[\![ex_i]\!](V, M) = ct_i \text{ for all } i \\ &\quad \text{where } V' \triangleq V\{\mathsf{pc} \mapsto loc(sts_m), \\ &\qquad\qquad \mathsf{sp} \mapsto V(\mathsf{sp}) + n + \textstyle\sum_{i=1}^{k} size(ty_i) + 1, \\ &\qquad\qquad \mathsf{return} \mapsto V(\mathsf{sp}) + n, \\ &\qquad\qquad x_1 \mapsto \langle V(\mathsf{sp})\rangle, \\ &\qquad\qquad \vdots \\ &\qquad\qquad x_n \mapsto \langle V(\mathsf{sp}) + n - 1\rangle, \\ &\qquad\qquad y_1 \mapsto \langle V(\mathsf{sp}) + n + 1, \ldots, V(\mathsf{sp}) + n + 1 + size(ty_1) - 1\rangle \\ &\qquad\qquad \vdots \\ &\qquad\qquad y_k \mapsto \langle V(\mathsf{sp}) + n + 1 + \textstyle\sum_{i=1}^{k-1} size(ty_i), \\ &\qquad\qquad\qquad \ldots, \\ &\qquad\qquad\qquad V(\mathsf{sp}) + n + 1 + \textstyle\sum_{i=1}^{k-1} size(ty_i) + size(ty_k) - 1\rangle\} \\ &\qquad M' \triangleq M\{V(\mathsf{sp}) \mapsto ct_1, \\ &\qquad\qquad \vdots \\ &\qquad\qquad V(\mathsf{sp}) + n - 1 \mapsto ct_n, \\ &\qquad\qquad V(\mathsf{sp}) + n \mapsto loc(sts) \\ &\qquad\qquad V(\mathsf{sp}) + n + 1 \mapsto 0, \\ &\qquad\qquad \vdots \\ &\qquad\qquad V(\mathsf{sp}) + n + 1 + \textstyle\sum_{i=1}^{k-1} size(ty_i) \mapsto 0\} \\ &\quad \text{for } m(x_1 \!:\! py_1, \ldots, x_n \!:\! py_n)\ \{\ \mathsf{var}\ y_1 \!:\! ty_1; \ldots; y_k \!:\! ty_k; sts_m\ \}\text{ a procedure}\end{aligned} \tag{R11}$$

The reference semantics $[\![P]\!]_I^{ref}(\cdot)$ proper is defined by extracting a sequence of states from a sequence of reductions starting from an initial operational state that corresponds to invoking the procedure main with the arguments supplied to the semantic function.

Reference Semantics $[\![\cdot]\!]_I^{ref}$:

$$\begin{aligned}[\![P]\!]_I^{ref}(\langle ct_1, \ldots, ct_k\rangle) \triangleq \{\langle s_1, s_2, s_3, \ldots\rangle \mid\ & (sts_1, V_1, M_1, Vs_1) \longrightarrow (sts_2, V_2, M_2, Vs_2) \longrightarrow \ldots, \\ & s_i = (\mathbb{N}, V_i, M_i) \text{ for all } i \geq 1\}\end{aligned}$$

$$\begin{aligned}
\text{where } & \mathit{Vs}_1 \triangleq \langle\rangle \\
& V_1 \triangleq V_0\{\mathsf{sp} \mapsto \ell_{RW}, \\
&\qquad \mathsf{return} \mapsto \bullet, \\
&\qquad \mathsf{pc} \mapsto \mathit{loc}(\mathsf{main}(ct_1, \ldots, ct_k);), \\
&\qquad \mathsf{inputs} \mapsto \langle\rangle, \\
&\qquad \mathsf{status} \mapsto \textbf{Success}, \\
&\qquad \mathsf{actions} \mapsto \langle\rangle\} \\
& M_1 \triangleq M_0 \\
& \mathit{sts}_1 \triangleq \mathsf{main}(ct_1, \ldots, ct_k);
\end{aligned}$$

Some remarks on the above definition: the stack pointer sp is initially set to the first readable/writable location in memory, the program counter pc is set to a code location corresponding to invoking the procedure main with the supplied inputs before returning immediately; all the inputs are consumed in the first step of execution, and therefore the inputs variable remains empty for the whole execution.

To simplify the presentation of the reduction rules, some information in operational states is somewhat redundant—the statement being executed in an operational state not only appears in the operational state, but is also given by the code at the location in the program counter. As the following lemma shows, this redundancy is consistent.

Lemma A.1. *Let $(\mathit{sts}, V, M, \mathit{Vs})$ be an operational state reachable from $(\mathit{sts}_1, V_1, M_1, \mathit{Vs}_1)$. The following properties hold:*

(1) $V(\mathsf{pc}) = \bullet$ *if and only if* sts *is* $\bullet$*;*

(2) *If* sts *is not* $\bullet$*, then* $\mathit{code}(V(\mathsf{pc})) = \mathit{sts}$*.*

Proof. This is a straightforward induction on the length of executions. Every reduction of the form $(\mathit{sts}, V, M, \mathit{Vs}) \longrightarrow (\mathit{sts}', V', M', \mathit{Vs}')$ satisfies $V'(\mathsf{pc}) = \mathit{loc}(\mathit{sts}')$. ∎

Finally, we verify that the resulting traces are in fact executions as we defined them in §3.

Lemma A.2. *The infinite trace $[\![P]\!]_I^{\mathit{ref}}(\langle ct_1, \ldots, ct_k\rangle)$ is an execution.*

Proof. let σ be the infinite trace in $[\![P]\!]_I^{\mathit{ref}}(\langle ct_1, \ldots, ct_k\rangle)$, generated by the sequence of reductions $(\mathit{sts}_1, V_1, M_1, \mathit{Vs}_1) \longrightarrow (\mathit{sts}_2, V_2, M_2, \mathit{Vs}_2) \longrightarrow \ldots$. We check the five conditions defining an execution:

(1) By definition, every state in σ uses $\mathbb{N}$ as its set of locations.

(2) By Lemma A.1, if $\sigma[i].\mathsf{pc} = \bullet$, then $\mathit{sts}_i = \bullet$. Only rule (R1) can apply from operational state i on, and thus $\mathit{sts}_j = \bullet$ for all $j \geq i$, that is, $\sigma[j].\mathsf{pc} = \bullet$ for all $j \geq i$.

(3) If there exists an $i \geq 1$ with $\sigma[i].\mathsf{pc} = \bullet$, we are done. If there is no $i \geq 1$ with $\sigma[i].\mathsf{pc} = \bullet$, then by Lemma A.1, there is no $i \geq 1$ such that $\mathit{sts}_i = \bullet$. Therefore, for every $i \geq 1$, the reduction rule applied at step i must be one of (R3), (R4), (R5), (R6), (R7), (R8), (R10), or (R11). Each of these rules changes the state (if only because they change pc). Thus, we cannot have $\sigma[i] = \sigma[i + 1]$.

(4) By definition, $\sigma[1].\mathsf{actions} = \langle\rangle$. The only rule that adds output actions to $\mathsf{actions}$ is (R10), which adds a single output action.

(5) By definition, $\sigma[1].\mathsf{inputs} = \langle\rangle$, and examination of the rules shows that inputs is unchanged throughout execution. ∎

B Detailed Account of Obfuscator τ_{addr}

The implementation semantics $[\![P]\!]_I^{\tau,K}$ is obtained by modifying that $[\![P]\!]_I^{ref}$ so that it is parameterized by:

(1) A memory location $\ell_s \geq \ell_{RW}$ representing the start of the stack;

(2) A positive integer d, which is a padding size for data on the stack;

(3) A sequence of permutations $\Pi = \{\pi_n \mid n \geq 0\}$, where π_n is a permutation of $\{1, \ldots, n\}$, which is used to permute the allocation of parameters and local variables on the stack;

(4) An initial memory map M_{init}.

The implementation semantics $[\![P]\!]_I^{\tau_{addr},K}$ corresponding to the morph $\tau_{addr}(P, K)$ is obtained by modifying the reference semantics $[\![P]\!]_I^{ref}(\cdot)$ in a rather simple way. To account for (1), we initially set sp to ℓ_s instead of ℓ_{RW}. To account for (4), we initially set M to M_{init}. To account for (2) and (3), we replace the reduction rule (R11) for procedure calls by the following rule (R11*).

Implementation Semantics $[\![P]\!]_I^{\tau_{addr},K}$ with $K = (\ell_s, d, \Pi, M_{init})$:

$$
\begin{array}{l}
(m(ex_1, \ldots, ex_n); sts, V, M, \langle V_1, \ldots, V_k\rangle) \longrightarrow (sts_m, V', M', \langle V, V_1, \ldots, V_k\rangle) \qquad \text{(R11*)} \\
\quad \text{if } E[\![ex_i]\!](V, M) = ct_i \text{ for all } i \\
\quad \text{where } V' \triangleq V\{\mathsf{pc} \mapsto loc(sts_m), \\
\qquad\qquad \mathsf{sp} \mapsto V(\mathsf{sp}) + n + \sum_{i=1}^{k} size(ty_i) + 1 + 4d, \\
\qquad\qquad x_{\pi_n(1)} \mapsto \langle V(\mathsf{sp}) + d\rangle, \\
\qquad\qquad \vdots \\
\qquad\qquad x_{\pi_n(n)} \mapsto \langle V(\mathsf{sp}) + d + n - 1\rangle, \\
\qquad\qquad y_{\pi_k(1)} \mapsto \langle V(\mathsf{sp}) + 3d + n + 1, \ldots, V(\mathsf{sp}) + 3d + n + 1 + size(ty_{\pi_k(1)}) - 1\rangle \\
\qquad\qquad \vdots \\
\qquad\qquad y_{\pi_k(k)} \mapsto \langle V(\mathsf{sp}) + 3d + n + 1 + \sum_{i=1}^{k-1} size(ty_{\pi_k(i)}), \\
\qquad\qquad\qquad \ldots, \\
\qquad\qquad\qquad V(\mathsf{sp}) + 3d + n + 1 + \sum_{i=1}^{k-1} size(ty_{\pi_k(i)}) + size(ty_{\pi_k(k)}) - 1\rangle\} \\
\qquad M' \triangleq M\{V(\mathsf{sp}) + d \mapsto ct_1, \\
\qquad\qquad \vdots \\
\qquad\qquad V(\mathsf{sp}) + d + n - 1 \mapsto ct_n, \\
\qquad\qquad V(\mathsf{sp}) + 2d + n \mapsto loc(sts) \\
\qquad\qquad V(\mathsf{sp}) + 3d + n + 1 \mapsto 0, \\
\qquad\qquad \vdots \\
\qquad\qquad V(\mathsf{sp}) + 3d + n + 1 + \sum_{i=1}^{k-1} size(ty_{\pi_k(i)}) \mapsto 0\} \\
\quad \text{for } m(x_1 : py_1, \ldots, x_n : py_n)\ \{\ \mathsf{var}\ y_1 : ty_1; \ldots; y_k : ty_k; sts_m\ \} \text{ a procedure}
\end{array}
$$

C Type Systems for τ_{addr}

The type systems for τ_{addr} described in §5 are folded into implementation semantics obtained by modifying reference semantics $[\![P]\!]^{ref}_I$.

C.1 Implementation Semantics $[\![\cdot]\!]^{strg}_I$

As described in §5.1, values in the semantics are of the form $\langle i, t\rangle$, where i is an integer, and t is the type of the value, either integer or pointer.

Type checking for T^{strg} occurs when operations $*$, $+$, and $=$ are evaluated (rules S2, S4, and S5 in §5.1). Recall, Toy-C supports two kinds of expressions: VD- and AD-expressions, which denote values (integer or pointer) and memory locations, respectively. VD-expressions ex include constants (for simplicity, only integers i, as well as the null pointer null), variables, pointer dereference ($*ex$), operations such as $ex_1{+}ex_2$ and $ex_1 = ex_2$, and taking the address of an AD-expression ($\&lv$). AD-expressions lv include variables and pointer dereferences ($*lv$). Reference semantics $[\![\cdot]\!]^{ref}_I$ (see Appendix A.2) uses evaluation function $A[\![lv]\!](V, M)$ to evaluate an AD-expression lv in variable map V and memory map M to a location, and evaluation function $E[\![ex]\!](V, M)$ to evaluate a VD-expression ex in variable map V and memory map M to a value. To implement type checking for T^{strg}, it suffices to replace these evaluation functions by the *type-checking evaluation functions* AT and ET given below, which check suitable conditions on extended values.

Type-Checking Evaluation Functions for $[\![\cdot]\!]^{strg}_I$:

$$AT[\![x]\!](V, M) \triangleq \langle \ell_1, \mathbf{ptr}(\ell_1, \ell_k)\rangle \quad \text{when } V(x) = \langle \ell_1, \ldots, \ell_k\rangle$$

$$AT[\![*lv]\!](V, M) \triangleq \begin{cases} M(i) & \text{if } AT[\![lv]\!](V, M) = \langle i, \mathbf{ptr}(start, end)\rangle,\ i \geq \ell_{RW}, \\ & \text{and } start \leq i \leq end \\ \mathbf{TypeErr} & \text{otherwise} \end{cases}$$

$$ET[\![*ex]\!](V, M) \triangleq \begin{cases} M(i) & \text{if } ET[\![ex]\!](V, M) = \langle i, \mathbf{ptr}(start, end)\rangle,\ i \geq \ell_{RW}, \\ & \text{and } start \leq i \leq end \\ \mathbf{TypeErr} & \text{otherwise} \end{cases}$$

$$ET[\![i]\!](V, M) \triangleq \langle i, \mathbf{int}\rangle$$

$$ET[\![\mathsf{null}]\!](V, M) \triangleq \langle 0, \mathbf{ptr}(0, 0)\rangle$$

$$ET[\![x]\!](V, M) \triangleq M(\ell_1) \quad \text{when } V(x) = \langle \ell_1, \ldots, \ell_k\rangle$$

$$ET[\![\&lv]\!](V, M) \triangleq AT[\![lv]\!](V, M)$$

$$ET[\![ex_1 + ex_2]\!](V, M) \triangleq \begin{cases} \langle i_1 + i_2, \mathbf{int}\rangle & \text{if } t_1 = t_2 = \mathbf{int} \\ \langle i_1 + i_2, \mathbf{ptr}(start, end)\rangle & \text{if } t_1 = \mathbf{ptr}(start, end) \text{ and } t_2 = \mathbf{int}, \\ & \text{or } t_1 = \mathbf{int} \text{ and } t_2 = \mathbf{ptr}(start, end) \\ \mathbf{TypeErr} & \text{otherwise} \end{cases}$$

where $ET[\![ex_1]\!](V,M) = \langle i_1, t_1\rangle$
$ET[\![ex_2]\!](V,M) = \langle i_2, t_2\rangle$

$$ET[\![ex_1 = ex_2]\!](V,M) \triangleq \begin{cases} \langle 1, \mathbf{int}\rangle & \text{if } i_1 = i_2 \text{ and } t_1 = t_2 = \mathbf{int} \\ \langle 1, \mathbf{int}\rangle & \text{if } i_1 = i_2,\ t_1 = \mathbf{ptr}(-,-) \text{ and } t_2 = \mathbf{ptr}(-,-) \\ \langle 0, \mathbf{int}\rangle & \text{if } i_1 \neq i_2 \text{ and } t_1 = t_2 = \mathbf{int} \\ \langle 0, \mathbf{int}\rangle & \text{if } i_1 \neq i_2,\ t_1 = \mathbf{ptr}(-,-), \text{ and } t_2 = \mathbf{ptr}(-,-) \\ \mathbf{TypeErr} & \text{otherwise} \end{cases}$$

where $ET[\![ex_1]\!](V,M) = \langle i_1, t_1\rangle$
$ET[\![ex_2]\!](V,M) = \langle i_2, t_2\rangle$

Functions AT and ET return either a value v or **TypeErr** (indicating a type error).

The semantics proper is again obtained from reduction rules. The rules (T4'), (T6'), (T8'), and (T11') are concerned specifically with reporting type errors.

Type-Checking Reduction Rules:

$(\bullet, V, M, Vs) \longrightarrow (\bullet, V, M, Vs)$ (T1)

$(\epsilon, V, M, Vs) \longrightarrow (\bullet, V\{\mathsf{pc} \mapsto \bullet\}, M, Vs)$ (T2)
if $V(\mathsf{return}) = \bullet$

$(\epsilon, V, M, \langle V_1, \ldots, V_k\rangle) \longrightarrow (sts, V_1\{\mathsf{pc} \mapsto \ell\}, M, \langle V_2, \ldots, V_k\rangle)$ (T3)
if $V(\mathsf{return}) = \ell'$, $M(\ell') = \ell$, and $code(\ell) = sts$

$(lv := ex; sts, V, M, Vs) \longrightarrow (sts, V\{\mathsf{pc} \mapsto loc(sts)\}, M\{\ell \mapsto v\}, Vs)$ (T4)
if $ET[\![ex]\!](V,M) = v$ and $AT[\![lv]\!](V,M) = \langle \ell, -\rangle$

$(lv := ex; sts, V, M, Vs) \longrightarrow (\bullet, V\{\mathsf{pc} \mapsto \bullet, \mathsf{status} \mapsto \mathbf{TypeErr}\}, M, Vs)$ (T4')
if $ET[\![ex]\!](V,M) = \mathbf{TypeErr}$ or $AT[\![lv]\!](V,M) = \mathbf{TypeErr}$

$(\mathsf{if}\ ex\ \mathsf{then}\ \{\ sts_1\ \}\ \mathsf{else}\ \{\ sts_2\ \}; sts, V, M, \langle V_1, \ldots, V_k\rangle) \longrightarrow$ (T5)
$(sts_1, V', M\{V(\mathsf{sp}) \mapsto loc(sts)\}, \langle V, V_1, \ldots, V_k\rangle)$
if $ET[\![ex]\!](V,M) \neq \langle 0, \mathbf{int}\rangle$
where $V' \triangleq V\{\mathsf{pc} \mapsto loc(sts_1),$
$\mathsf{sp} \mapsto V(\mathsf{sp}) + 1,$
$\mathsf{return} \mapsto V(\mathsf{sp})\}$

$(\mathsf{if}\ ex\ \mathsf{then}\ \{\ sts_1\ \}\ \mathsf{else}\ \{\ sts_2\ \}; sts, V, M, \langle V_1, \ldots, V_k\rangle) \longrightarrow$ (T6)
$(sts_2, V', M\{V(\mathsf{sp}) \mapsto loc(sts)\}, \langle V', V_1, \ldots, V_k\rangle)$
if $ET[\![ex]\!](V,M) = \langle 0, \mathbf{int}\rangle$
where $V' \triangleq V\{\mathsf{pc} \mapsto loc(sts_2),$
$\mathsf{sp} \mapsto V(\mathsf{sp}) + 1,$
$\mathsf{return} \mapsto V(\mathsf{sp})\}$

$(\mathsf{if}\ ex\ \mathsf{then}\ \{\ sts_1\ \}\ \mathsf{else}\ \{\ sts_2\ \}; sts, V, M, Vs) \longrightarrow$ (T6')
$(\bullet, V\{\mathsf{pc} \mapsto \bullet, \mathsf{status} \mapsto \mathbf{TypeErr}\}, M, Vs)$

if $ET[\![ex]\!](V, M) = \mathbf{TypeErr}$

$$(\textsf{while}\ ex\ \textsf{do}\ \{\ sts_1\ \}; sts, V, M, \langle V_1, \ldots, V_k\rangle) \longrightarrow \qquad \text{(T7)}$$
$$(sts_1, V', M\{V(\textsf{sp}) \mapsto loc(\textsf{while}\ ex\ \textsf{do}\ \{\ sts_1\ \}; sts)\}, \langle V, V_1, \ldots, V_k\rangle)$$

if $ET[\![ex]\!](V, M) \neq \langle 0, -\rangle$

where $V' \triangleq V\{\textsf{pc} \mapsto loc(sts_1),$
$\textsf{sp} \mapsto V(\textsf{sp}) + 1,$
$\textsf{return} \mapsto V(\textsf{sp})\}$

$$(\textsf{while}\ ex\ \textsf{do}\ \{\ sts_1\ \}; sts, V, M, Vs) \longrightarrow (sts, V\{\textsf{pc} \mapsto loc(sts)\}, M, Vs) \qquad \text{(T8)}$$

if $ET[\![ex]\!](V, M) = \langle 0, -\rangle$

$$(\textsf{while}\ ex\ \textsf{do}\ \{\ sts_1\ \}; sts, V, M, Vs) \longrightarrow (\bullet, V\{\textsf{pc} \mapsto \bullet, \textsf{status} \mapsto \mathbf{TypeErr}\}, M, Vs) \qquad \text{(T8')}$$

if $ET[\![ex]\!](V, M) = \mathbf{TypeErr}$

$$(\textsf{fail}; sts, V, M, Vs) \longrightarrow (\bullet, V\{\textsf{pc} \mapsto \bullet, \textsf{status} \mapsto \mathbf{Fail}\}, M, Vs) \qquad \text{(T9)}$$

$$(ac; sts, V, M, Vs) \longrightarrow (sts, V', M, Vs) \qquad \text{(T10)}$$

where $V' \triangleq V\{\textsf{pc} \mapsto loc(sts),$
$\textsf{actions} \mapsto V(\textsf{actions}) +\!\!+ \langle ac\rangle\}$

$$(m(ex_1, \ldots, ex_n); sts, V, M, \langle V_1, \ldots, V_k\rangle) \longrightarrow (sts_m, V', M', \langle V, V_1, \ldots, V_k\rangle) \qquad \text{(T11)}$$

if $ET[\![ex_i]\!](V, M) = v_i$ for all i

where $V' \triangleq V\{\textsf{pc} \mapsto loc(sts_m),$
$\textsf{sp} \mapsto V(\textsf{sp}) + n + \sum_{i=1}^{k} size(ty_i) + 1,$
$\textsf{return} \mapsto V(\textsf{sp}) + n,$
$x_1 \mapsto \langle V(\textsf{sp})\rangle,$
$\vdots$
$x_n \mapsto \langle V(\textsf{sp}) + n - 1\rangle,$
$y_1 \mapsto \langle V(\textsf{sp}) + n + 1, \ldots, V(\textsf{sp}) + n + 1 + size(ty_1) - 1\rangle$
$\vdots$
$y_k \mapsto \langle V(\textsf{sp}) + n + 1 + \sum_{i=1}^{k-1} size(ty_i),$
$\ldots,$
$V(\textsf{sp}) + n + 1 + \sum_{i=1}^{k-1} size(ty_i) + size(ty_k) - 1\rangle\}$

$M' \triangleq M\{V(\textsf{sp}) \mapsto v_1,$
$\vdots$
$V(\textsf{sp}) + n - 1 \mapsto v_n,$
$V(\textsf{sp}) + n \mapsto loc(sts)$
$V(\textsf{sp}) + n + 1 \mapsto 0,$
$\vdots$
$V(\textsf{sp}) + n + 1 + \sum_{i=1}^{k-1} size(ty_i) \mapsto 0\}$

for $m(x_1 : py_1, \ldots, x_n : py_n)\ \{\ \textsf{var}\ y_1 : ty_1; \ldots; y_k : ty_k; sts_m\ \}$ a procedure

$$(m(ex_1, \ldots, ex_n); sts, V, M, Vs) \longrightarrow (\bullet, V\{\textsf{pc} \mapsto \bullet, \textsf{status} \mapsto \mathbf{TypeErr}\}, M, Vs) \qquad \text{(T11')}$$

if $ET[\![ex_i]\!](V, M) = \mathbf{TypeErr}$ for some i

Note that the hidden variable status is set to **Success** if termination is successful, **Fail** if termination is due to a failure, and **TypeErr** if termination is due to a type error. We say an execution trace *signals a type error* if it terminates with status equal to **TypeErr**.

Type-Checking Implementation Semantics $[\![\cdot]\!]_I^{strg}$:

$$[\![P]\!]_I^{strg}(\langle ct_1,\dots,ct_k\rangle)T \triangleq \{\langle s_1,s_2,s_3,\dots\rangle \mid (sts_1,V_1,M_1,Vs_1) \longrightarrow (sts_2,V_2,M_2,Vs_2) \longrightarrow \dots,\ s_i=(\mathbb{N},V_i,M_i) \text{ for all } i\geq 1\}$$

$$\begin{aligned}
\text{where } & Vs_1 \triangleq \langle\rangle\\
& V_1 \triangleq V_0\{\mathsf{sp}\mapsto \ell_{RW},\\
&\qquad \mathsf{return}\mapsto \bullet,\\
&\qquad \mathsf{pc}\mapsto loc(\mathsf{main}(ct_1,\dots,ct_k);),\\
&\qquad \mathsf{inputs}\mapsto \langle\rangle,\\
&\qquad \mathsf{status}\mapsto \textbf{Success},\\
&\qquad \mathsf{actions}\mapsto \langle\rangle\}\\
& M_1 \triangleq M_0\\
& sts_1 \triangleq \mathsf{main}(ct_1,\dots,ct_k);
\end{aligned}$$

Lemma C.1. *For all V, M, lv, and ex:*

(1) *If $AT[\![lv]\!](V,M)=\langle i,-\rangle$, then $A[\![lv]\!](V,\ulcorner M\urcorner)=i$;*

(2) *If $ET[\![ex]\!](V,M)=\langle i,-\rangle$, then $E[\![ex]\!](V,\ulcorner M\urcorner)=i$;*

where $\ulcorner M\urcorner$ is defined by $\ulcorner M\urcorner(\ell)=i$ when $M(\ell)=\langle i,-\rangle$.

Proof. A straightforward proof by induction on the structure of lv and ex. ∎

Lemma C.2. *$[\![\cdot]\!]_I^{strg}$ is a restriction of $[\![\cdot]\!]_I^{ref}$.*

Proof. Let P be a program, and $inps$ an input. Let $\sigma\in[\![P]\!]_I^{strg}(inps)$ be generated by the sequence of reductions $(sts_1,V_1,M_1,Vs_1)\longrightarrow(sts_2,V_2,M_2,Vs_2)\longrightarrow\dots$. Assume that σ does not signal a type error. Thus, the reduction sequence never uses rules (T4'), (T6'), (T8'), or (T11'). Let $\widetilde{\sigma}\in[\![P]\!]_I^{ref}(inps)$ be generated by the sequence of reductions $(\widetilde{sts_1},\widetilde{V_1},\widetilde{M_1},\widetilde{Vs_1})\longrightarrow(\widetilde{sts_2},\widetilde{V_2},\widetilde{M_2},\widetilde{Vs_2})\longrightarrow\dots$. We show by induction on the length of the reduction sequences that for all $i\geq 0$, $\ulcorner\sigma\urcorner[i].\mathsf{actions}=\widetilde{\sigma}[i].\mathsf{actions}$, $\ulcorner\sigma\urcorner[i].\mathsf{inputs}=\widetilde{\sigma}[i].\mathsf{inputs}$, and for every observable program variable x, $\ulcorner\sigma\urcorner[i].x=\widetilde{\sigma}[i].x$.

Without loss of generality, we assume that programs live in the same locations in both semantics. (Otherwise, we need to ensure that we map program locations in one semantics to the appropriate program locations in the other semantics.)

The result holds trivially for the base case, since the initial operational states are the same. The inductive step is based on the following observation. Since no type error is signalled, by Lemma C.1, every evaluation of AT and ET returns the same result as the corresponding evaluation of A and E. It follows by examination of the rules that the result of applying rule (T1)–(T11) is the same as the result of applying the corresponding rule (R1)–(R11). ∎

Theorem 5.1. *Let $K_1, \ldots, K_n$ be arbitrary keys for τ_{addr}. For any program P and inputs inps, if inps is an attack on P relative to τ_{addr} and $K_1, \ldots, K_n$, then $\sigma \in [\![P]\!]_I^{strg}(inps)$ signals a type error. Equivalently, if $\sigma \in [\![P]\!]_I^{strg}(inps)$ does not signal a type error, then inps is not an attack on P relative to τ_{addr} and $K_1, \ldots, K_n$*

Proof. Assume that $\sigma \in [\![P]\!]_I^{strg}(inps)T$ does not signal a type error. We want to show that *inps* is not an attack relative to P and $K_1, \ldots, K_n$. In other words, if σ_i is the execution in $[\![P]\!]_I^{\tau_{addr},K_i}(inps)$, we want $(\sigma_1, \ldots, \sigma_n) \in \mathcal{B}_n^{\tau_{addr}}(P, K_1, \ldots, K_n)$. This requires finding an appropriate $\widehat{\sigma} \in [\![P]\!]_H(inps)$.

We derive the required $\widehat{\sigma}$ from σ as follows. When $\sigma[j] = (\mathbb{N}, V, M)$, we take $\widehat{\sigma}[j]$ to be $(\mathbb{N}, V, \widehat{M})$, with $\widehat{M}$ given by:

$$\widehat{M}(\ell) = \begin{cases} direct(i) & \text{if } M(\ell) = \langle i, \mathbf{int} \rangle \\ pointer(i) & \text{if } M(\ell) = \langle i, \mathbf{ptr}(-,-) \rangle \end{cases}$$

It remains to show that for every i we have $(\sigma_i, \widehat{\sigma}) \in \delta(P, K_i)$. Fix i. We exhibit a relation $\precsim$ determined by a map h such that $\sigma_i[j] \precsim \widehat{\sigma}[j]$ for all j. The map h maps memory locations in the states of execution σ_i to corresponding memory locations in the states of the high-level execution $\widehat{\sigma}$. Roughly speaking, for any $\ell \in L$, if ℓ in the range of any observable program variable x in V, then we put $h(s, \ell)$ in the corresponding range of that same x in $\widehat{V}$.

The relevant portions of the map h are defined inductively over the sequence of reductions $(\overline{sts_1}, \overline{V_1}, \overline{M_1}, \overline{Vs_1}) \longrightarrow (\overline{sts_2}, \overline{V_2}, \overline{M_2}, \overline{Vs_2}) \longrightarrow \ldots$ generating σ_i. Let $(sts_1, V_1, M_1, Vs_1) \longrightarrow (sts_2, V_2, M_2, Vs_2) \longrightarrow \ldots$ be the sequence of reductions generating σ. As in the proof of Lemma C.2, we assume without loss of generality that programs live in the same locations in both semantics; thus, we take $h(s, \ell) = \ell$ for every state s and every location $\ell < \ell_{RW}$. We specify $h(s, \ell)$ only for states s of the form $(\mathbb{N}, \overline{V_j}, \overline{M_j})$, for every $j \geq 1$. For $j = 1$, we can take $h((\mathbb{N}, \overline{V_1}, \overline{M_1}), \ell)$ to be arbitrary, since there are no observable program variables in the initial state of the execution. Inductively, assume that we have $h((\mathbb{N}, \overline{V_j}, \overline{M_j}), \cdot)$. If the reduction rule applied at step j is any reduction but (R11*), then take $h((\mathbb{N}, \overline{V_{j+1}}, \overline{M_{j+1}}), \cdot)$ to be the same as $h((\mathbb{N}, \overline{V_j}, \overline{M_j}), \cdot)$. If the reduction rule applied at step j is (R11*), then take $h((\mathbb{N}, \overline{V_{j+1}}, \overline{M_{j+1}}), \cdot)$ to be $h((\mathbb{N}, \overline{V_j}, \overline{M_j}), \cdot)$, updated to map

$$\overline{V_{j+1}}(\mathsf{sp}) + 3d + n + 1 \text{ to}$$

$$V_{j+1} + n + 1 + \sum_{i=1}^{\pi_k^{-1}(1)-1} size(ty_i)$$

$$\vdots$$

$$\overline{V_{j+1}}(\mathsf{sp}) + 3d + n + 1 + size(ty_{\pi_k(1)}) \text{ to}$$

$$V_{j+1} + n + 1 + \sum_{i=1}^{\pi_k^{-1}(1)-1} size(ty_i) + size(ty_{\pi_k^{-1}(1)})$$

$$\vdots$$

$$\overline{V_{j+1}}(\mathsf{sp}) + 3d + n + 1 + \sum_{i=1}^{k-1} size(ty_{\pi_k(i)}) \text{ to}$$

$$V_{j+1} + n + 1 + \sum_{i=1}^{\pi_k^{-1}(k)-1} size(ty_i)$$

$$\vdots$$

$$\overline{V_{j+1}}(\mathsf{sp}) + 3d + n + 1 + \sum_{i=1}^{k-1} size(ty_{\pi_k(i)}) + size(ty_{\pi_k(k)}) \text{ to}$$

$$V_{j+1} + n + 1 + \sum_{i=1}^{\pi_k^{-1}(k)-1} size(ty_i) + size(ty_{\pi^{-1}(k)})$$

Let $\precsim$ be a relation determined by h satisfying the above. We use induction to show that for all $j \geq 1$:

(i) $\overline{sts_j} = sts_j$;

(ii) for all ex,

- if $ET[\![ex]\!](V_j, M_J) = \langle i', \mathbf{int}\rangle$, then $E[\![ex]\!](\overline{V_j}, \overline{M_j}) = i'$;
- if $ET[\![ex]\!](V_j, M_j) = \langle i', \mathbf{ptr}(-,-)\rangle$, then $E[\![ex]\!](\overline{V_j}, \overline{M_j}) = i''$ and $h((\mathbb{N}, \overline{V_j}, \overline{M_j}), i'') = i'$;

and similarly for all lv;

(iii) If reduction (Rn), for some $n < 11$, applies at step j to produce $\sigma_i[j+1]$, then reduction (Tn) applies at step j to produce $\sigma[j+1]$; if reduction (R11*) applies at step j to produce $\sigma_i[j+1]$, then reduction (T11) applies at step j to produce $\sigma[j+1]$;

(iv) $\sigma_i[j] \precsim \widehat{\sigma}[j]$;

The base case, $j = 1$, is immediate, since the initial states $\sigma[1]$ and $\sigma_i[1]$ are the same, up to the types associated with the values in memory, meaning that the states $\widehat{\sigma}[1]$ and $\sigma_i[1]$ are also the same, up to the tagging of the values required by the high-level semantics $[\![\cdot]\!]_H$.

For the inductive case, assume that we have the result for j; we show it for $j+1$. Establishing (i), (ii), and (iii) is straightforward. To establish (iv), we need to show: (1) either $\overline{V_{j+1}}(\mathsf{pc}) = \bullet$ and $V_{j+1}(\mathsf{pc}) = \bullet$, or $h(\sigma_i[j+1], \overline{V_{j+1}}(\mathsf{pc})) = V_{j+1}(\mathsf{pc})$; (2) $\overline{V_{j+1}}(\mathsf{actions}) = V_{j+1}(\mathsf{actions})$; (3) $\overline{V_{j+1}}(\mathsf{inputs}) = V_{j+1}(\mathsf{inputs})$; (4) for every observable program variable x, there exists $k \geq 0$ such that $\overline{V_{j+1}}(x) = \langle \ell_1, \ldots, \ell_k\rangle$, $V_{j+1}(x) = \langle \widehat{\ell_1}, \ldots, \widehat{\ell_k}\rangle$, and for all $j \leq k$ we have $\ell_j \precsim \widehat{\ell_j}$.

The proof proceeds by case analysis on the reduction rule that applies at step j. (By (iii), we know that corresponding reduction rules apply to produce $\sigma_i[j+1]$ and $\sigma[j+1]$.) Most of the cases are trivial using (i)–(iii). The only case of interest is when the rules that apply at step j are (R11*) and (T11). Thus, $sts_j = \overline{sts_j} = m(ex_1, \ldots, ex_n); sts$, and

$ET[\![ex_k]\!](V_j, M_j) = \langle i_k, t_k\rangle$ for all k. By (ii), $E[\![ex_k]\!](\overline{V_j}, \overline{M_j}) = i'_k$ for all k, where $i_k = i'_k$ if $t_k = \mathbf{int}$, and $h(\sigma_i[j], i'_k) = i_k$ if $t_k = \mathbf{ptr}(-,-)$. (1), (2), and (3) follow immediately. For (4), note that if $\overline{V_{j+1}}(x) = \langle \ell_1, \ldots, \ell_k\rangle$ for an observable program variable x, then either x was not newly allocated with the current reduction rule, in which case we already have (4) by the induction hypothesis, or x is newly allocated, in which case by observation of the rules and by the construction of h, (4) holds. Thus, we have $\sigma_i[j+1] \precsim \widehat{\sigma}[j+1]$. ∎

C.2 Implementation Semantics $[\![\cdot]\!]_I^{\mathit{info}}$

As described in §5.2, this implementation semantics extends $[\![\cdot]\!]_I^{\mathit{strg}}$ by adding a new type, **low**.

To implement type checking for T^{info}, we again replace $[\![\cdot]\!]_I^{\mathit{ref}}$ evaluation functions A and E by the type-checking evaluation functions AT and ET given below, which check suitable conditions on extended values.

Type-Checking Evaluation Functions for $[\![\cdot]\!]_I^{\mathit{info}}$:

$$AT[\![x]\!](V, M) \triangleq \langle \ell_1, \mathbf{ptr}(\ell_1, \ell_k)\rangle \quad \text{when } V(x) = \langle \ell_1, \ldots, \ell_k\rangle$$

$$AT[\![*lv]\!](V, M) \triangleq \begin{cases} M(i) & \text{if } AT[\![lv]\!](V,M) = \langle i, \mathbf{ptr}(\mathit{start}, \mathit{end})\rangle,\ i \geq \ell_{RW}, \\ & \text{and } \mathit{start} \leq i \leq \mathit{end} \\ \langle i', \mathbf{low}\rangle & \text{otherwise, where } AT[\![lv]\!](V,M) = \langle i, -\rangle \text{ and } M(i) = \langle i', -\rangle \end{cases}$$

$$ET[\![*ex]\!](V, M) \triangleq \begin{cases} M(i) & \text{if } ET[\![ex]\!](V,M) = \langle i, \mathbf{ptr}(\mathit{start}, \mathit{end})\rangle,\ i \geq \ell_{RW}, \\ & \text{and } \mathit{start} \leq i \leq \mathit{end} \\ \langle i', \mathbf{low}\rangle & \text{otherwise, where } ET[\![ex]\!](V,M) = \langle i, -\rangle \text{ and } M(i) = \langle i', -\rangle \end{cases}$$

$$ET[\![i]\!](V, M) \triangleq \langle i, \mathbf{int}\rangle$$

$$ET[\![\mathsf{null}]\!](V, M) \triangleq \langle 0, \mathbf{ptr}(0,0)\rangle$$

$$ET[\![x]\!](V, M) \triangleq M(\ell_1) \quad \text{when } V(x) = \langle \ell_1, \ldots, \ell_k\rangle$$

$$ET[\![\&lv]\!](V, M) \triangleq AT[\![lv]\!](V, M)$$

$$ET[\![ex_1 + ex_2]\!](V, M) \triangleq \begin{cases} \langle i_1 + i_2, \mathbf{int}\rangle & \text{if } t_1 = t_2 = \mathbf{int} \\ \langle i_1 + i_2, \mathbf{ptr}(\mathit{start}, \mathit{end})\rangle & \text{if } t_1 = \mathbf{ptr}(\mathit{start}, \mathit{end}) \text{ and } t_2 = \mathbf{int}, \\ & \text{or } t_1 = \mathbf{int} \text{ and } t_2 = \mathbf{ptr}(\mathit{start}, \mathit{end}) \\ \langle i_1 + i_2, \mathbf{low}\rangle & \text{otherwise} \end{cases}$$
$$\text{where } ET[\![ex_1]\!](V,M) = \langle i_1, t_1\rangle$$
$$ET[\![ex_2]\!](V,M) = \langle i_2, t_2\rangle$$

$$ET[\![ex_1 = ex_2]\!](V, M) \triangleq \begin{cases} \langle 1, \mathbf{int}\rangle & \text{if } i_1 = i_2 \text{ and } t_1 = t_2 = \mathbf{int} \\ \langle 1, \mathbf{int}\rangle & \text{if } i_1 = i_2,\ t_1 = \mathbf{ptr}(-,-) \text{ and } t_2 = \mathbf{ptr}(-,-) \\ \langle 0, \mathbf{int}\rangle & \text{if } i_1 \neq i_2 \text{ and } t_1 = t_2 = \mathbf{int} \\ \langle 0, \mathbf{int}\rangle & \text{if } i_1 \neq i_2,\ t_1 = \mathbf{ptr}(-,-), \text{ and } t_2 = \mathbf{ptr}(-,-) \\ \langle 1, \mathbf{low}\rangle & \text{otherwise, where } i_1 = i_2 \\ \langle 0, \mathbf{low}\rangle & \text{otherwise, where } i_1 \neq i_2 \end{cases}$$
$$\text{where } ET[\![ex_1]\!](V, M) = \langle i_1, t_1\rangle$$
$$ET[\![ex_2]\!](V, M) = \langle i_2, t_2\rangle$$

Functions AT and ET return either an extended value v or **TypeErr** (indicating a type error).

The main difference from the reduction rules in §C.1 is that type checking is only performed in rules (T4') and (T10'), when the integrity of a value can influence a variable or control flow. (The values associated pc and the return hidden variables are colored, to track the fact that a value can influence the control flow of programs.) To simplify the presentation of the semantics, we define the following operation on types:

$$t_1 \downarrow t_2 \triangleq \begin{cases} t_1 & \text{if } t_2 \neq \mathbf{low} \\ \mathbf{low} & \text{if } t_2 = \mathbf{low}. \end{cases}$$

Thus, $t_1 \downarrow t_2$ is **low** if and only if one of t_1 or t_2 is **low**.

Type-Checking Reduction Rules:

$(\bullet, V, M, Vs) \longrightarrow (\bullet, V, M, Vs)$ (T1)

$(\epsilon, V, M, Vs) \longrightarrow (\bullet, V\{\mathsf{pc} \mapsto \bullet\}, M, Vs)$ (T2)
if $V(\mathsf{return}) = \bullet$

$(\epsilon, V, M, \langle V_1, \ldots, V_k\rangle) \longrightarrow (sts, V_1\{\mathsf{pc} \mapsto \langle \ell, t \downarrow t'\rangle\}, M, \langle V_2, \ldots, V_k\rangle)$ (T3)
if $V(\mathsf{return}) = \ell'$, $V(\mathsf{pc}) = \langle -, t'\rangle$, $M(\ell') = \langle \ell, t\rangle$, and $code(\ell) = sts$

$(lv := ex; sts, V, M, Vs) \longrightarrow (sts, V\{\mathsf{pc} \mapsto \langle loc(sts), t'\rangle\}, M\{\ell \mapsto v\}, Vs)$ (T4)
if $ET[\![ex]\!](V, M) = v$, $AT[\![lv]\!](V, M) = \langle \ell, t\rangle$, $V(\mathsf{pc}) = \langle \ell', t'\rangle$,
and $t, t' \neq \mathbf{low}$

$(lv := ex; sts, V, M, Vs) \longrightarrow (\bullet, V\{\mathsf{pc} \mapsto \bullet, \mathsf{status} \mapsto \mathbf{TypeErr}\}, M, Vs)$ (T4')
if $V(\mathsf{pc}) = \langle \ell, \mathbf{low}\rangle$ or $AT[\![lv]\!](V, M) = \langle i, \mathbf{low}\rangle$

$(\mathsf{if}\ ex\ \mathsf{then}\ \{\ sts_1\ \}\ \mathsf{else}\ \{\ sts_2\ \}; sts, V, M, \langle V_1, \ldots, V_k\rangle) \longrightarrow$ (T5)
$(sts_1, V', M\{V(\mathsf{sp}) \mapsto \langle loc(sts), t'\rangle\}, \langle V, V_1, \ldots, V_k\rangle)$
if $ET[\![ex]\!](V, M) = \langle i, t\rangle$ and $i \neq 0$
where $V(\mathsf{pc}) = \langle -, t'\rangle$
$V' \triangleq V\{\mathsf{pc} \mapsto \langle loc(sts_1), t \downarrow t'\rangle,$
$\mathsf{sp} \mapsto V(\mathsf{sp}) + 1,$
$\mathsf{return} \mapsto V(\mathsf{sp})\}$

$$
\begin{aligned}
&(\textsf{if } ex \textsf{ then } \{\ sts_1\ \} \textsf{ else } \{\ sts_2\ \}; sts, V, M, \langle V_1, \ldots, V_k\rangle) \longrightarrow && \text{(T6)}\\
&\qquad (sts_2, V', M\{V(\textsf{sp}) \mapsto \langle loc(sts), t'\rangle\}, \langle V', V_1, \ldots, V_k\rangle)\\
&\quad \text{if } ET[\![ex]\!](V, M) = \langle 0, t\rangle\\
&\quad \text{where } V(\textsf{pc}) = \langle -, t'\rangle\\
&\qquad\quad V' \triangleq V\{\textsf{pc} \mapsto \langle loc(sts_2), t \downarrow t'\rangle,\\
&\qquad\qquad\quad \textsf{sp} \mapsto V(\textsf{sp}) + 1,\\
&\qquad\qquad\quad \textsf{return} \mapsto V(\textsf{sp})\}
\end{aligned}
$$

$$
\begin{aligned}
&(\textsf{while } ex \textsf{ do } \{\ sts_1\ \}; sts, V, M, \langle V_1, \ldots, V_k\rangle) \longrightarrow && \text{(T7)}\\
&\qquad (sts_1, V', M\{V(\textsf{sp}) \mapsto \langle loc(\textsf{while } ex \textsf{ do } \{\ sts_1\ \}; sts), t'\rangle\}, \langle V, V_1, \ldots, V_k\rangle)\\
&\quad \text{if } ET[\![ex]\!](V, M) = \langle i, t\rangle \text{ and } i \neq 0\\
&\quad \text{where } V(\textsf{pc}) = \langle -, t'\rangle\\
&\qquad\quad V' \triangleq V\{\textsf{pc} \mapsto \langle loc(sts_1), t \downarrow t'\rangle,\\
&\qquad\qquad\quad \textsf{sp} \mapsto V(\textsf{sp}) + 1,\\
&\qquad\qquad\quad \textsf{return} \mapsto V(\textsf{sp})\}
\end{aligned}
$$

$$
\begin{aligned}
&(\textsf{while } ex \textsf{ do } \{\ sts_1\ \}; sts, V, M, Vs) \longrightarrow (sts, V\{\textsf{pc} \mapsto \langle loc(sts), t \downarrow t'\rangle\}, M, Vs) && \text{(T8)}\\
&\quad \text{if } ET[\![ex]\!](V, M) = \langle 0, t\rangle \text{ and } V(\textsf{pc}) = \langle -, t'\rangle
\end{aligned}
$$

$$
(\textsf{fail}; sts, V, M, Vs) \longrightarrow (\bullet, V\{\textsf{pc} \mapsto \bullet, \textsf{status} \mapsto \mathbf{Fail}\}, M, Vs) \qquad \text{(T9)}
$$

$$
\begin{aligned}
&(ac; sts, V, M, Vs) \longrightarrow (sts, V', M, Vs) && \text{(T10)}\\
&\quad \text{if } V(\textsf{pc}) = \langle \ell, t\rangle \text{ and } t \neq \mathbf{low}\\
&\quad \text{where } V' \triangleq V\{\textsf{pc} \mapsto \langle loc(sts), t\rangle,\\
&\qquad\qquad\quad \textsf{actions} \mapsto V(\textsf{actions}) +\!\!+ \langle ac\rangle\}
\end{aligned}
$$

$$
\begin{aligned}
&(ac; sts, V, M, Vs) \longrightarrow (\bullet, V\{\textsf{pc} \mapsto \bullet, \textsf{status} \mapsto \mathbf{TypeErr}\}, M, Vs) && \text{(T10')}\\
&\quad \text{if } V(\textsf{pc}) = \langle \ell, \mathbf{low}\rangle
\end{aligned}
$$

$$
\begin{aligned}
&(m(ex_1, \ldots, ex_n); sts, V, M, \langle V_1, \ldots, V_k\rangle) \longrightarrow (sts_m, V', M', \langle V, V_1, \ldots, V_k\rangle) && \text{(T11)}\\
&\quad \text{if } ET[\![ex_i]\!](V, M) = v_i \text{ for all } i\\
&\quad \text{where } V(\textsf{pc}) = \langle -, t\rangle\\
&\qquad\quad V' \triangleq V\{\textsf{pc} \mapsto \langle loc(sts_m), t\rangle,\\
&\qquad\qquad\quad \textsf{sp} \mapsto V(\textsf{sp}) + n + \textstyle\sum_{i=1}^{k} size(ty_i) + 1,\\
&\qquad\qquad\quad \textsf{return} \mapsto V(\textsf{sp}) + n,\\
&\qquad\qquad\quad x_1 \mapsto \langle V(\textsf{sp})\rangle,\\
&\qquad\qquad\quad \vdots\\
&\qquad\qquad\quad x_n \mapsto \langle V(\textsf{sp}) + n1\rangle,\\
&\qquad\qquad\quad y_1 \mapsto \langle V(\textsf{sp}) + n + 1, \ldots, V(\textsf{sp}) + n + 1 + size(ty_1) - 1\rangle\\
&\qquad\qquad\quad \vdots\\
&\qquad\qquad\quad y_k \mapsto \langle V(\textsf{sp}) + n + 1 + \textstyle\sum_{i=1}^{k-1} size(ty_i),\\
&\qquad\qquad\qquad\quad \ldots,\\
&\qquad\qquad\qquad\quad V(\textsf{sp}) + n + 1 + \textstyle\sum_{i=1}^{k-1} size(ty_i) + size(ty_k) - 1\rangle\}
\end{aligned}
$$

$$\begin{aligned} M' \triangleq M\{&V(\mathsf{sp}) \mapsto v_1,\\ &\vdots\\ &V(\mathsf{sp}) + n - 1 \mapsto v_n,\\ &V(\mathsf{sp}) + n \mapsto \langle loc(sts), t\rangle,\\ &V(\mathsf{sp}) + n + 1 \mapsto 0,\\ &\vdots\\ &V(\mathsf{sp}) + n + 1 + \textstyle\sum_{i=1}^{k-1} size(ty_i) \mapsto 0\} \end{aligned}$$

for $m(x_1 : py_1, \ldots, x_n : py_n)$ { $\mathsf{var}\ y_1 : ty_1; \ldots ; y_k : ty_k; sts_m$ } a procedure

Note that the hidden variable status is set to **Success** if termination is successful, **Fail** if termination is due to a failure, and **TypeErr** if termination is due to a type error. We say an execution trace *signals a type error* if it terminates with status equal to **TypeErr**.

Type-Checking Implementation Semantics $[\![\cdot]\!]_I^{info}$:

$$[\![P]\!]_I^{info}(\langle ct_1, \ldots, ct_k\rangle)T \triangleq \{\langle s_1, s_2, s_3, \ldots\rangle \mid (sts_1, V_1, M_1, Vs_1) \longrightarrow (sts_2, V_2, M_2, Vs_2) \longrightarrow \ldots,\ s_i = (\mathbb{N}, V_i, M_i) \text{ for all } i \geq 1\}$$

$$\begin{aligned} \text{where } Vs_1 &\triangleq \langle\rangle\\ V_1 &\triangleq V_0\{\mathsf{sp} \mapsto \ell_{RW},\\ &\quad \mathsf{return} \mapsto \bullet,\\ &\quad \mathsf{pc} \mapsto \langle loc(\mathsf{main}(ct_1, \ldots, ct_k);), \mathbf{int}\rangle,\\ &\quad \mathsf{inputs} \mapsto \langle\rangle,\\ &\quad \mathsf{status} \mapsto \mathbf{Success},\\ &\quad \mathsf{actions} \mapsto \langle\rangle\}\\ M_1 &\triangleq M_0\\ sts_1 &\triangleq \mathsf{main}(ct_1, \ldots, ct_k); \end{aligned}$$

Lemma C.3. *For all V, M, lv, and ex:*

(1) *If $AT[\![lv]\!](V, M) = \langle i, -\rangle$, then $A[\![lv]\!](V, \ulcorner M\urcorner) = i$;*

(2) *If $ET[\![ex]\!](V, M) = \langle i, -\rangle$, then $E[\![ex]\!](V, \ulcorner M\urcorner) = i$;*

where $\ulcorner M\urcorner$ is defined by $\ulcorner M\urcorner(\ell) = i$ when $M(\ell) = \langle i, -\rangle$.

Proof. A straightforward proof by induction on the structure of lv and ex. ∎

Lemma C.4. *$[\![\cdot]\!]_I^{info}$ is a restriction of $[\![\cdot]\!]_I^{ref}$.*

Proof. Let P be a program, and $inps$ an input. Let $\sigma \in [\![P]\!]_I^{strg}(inps)$ be generated by the sequence of reductions $(sts_1, V_1, M_1, Vs_1) \longrightarrow (sts_2, V_2, M_2, Vs_2) \longrightarrow \ldots$. Assume that σ does not signal a type error. Thus, the reduction sequence never uses rules (T4') and (T10'). Let $\widetilde{\sigma} \in [\![P]\!]_I^{ref}(inps)$ be generated by the sequence of reductions $(\widetilde{sts_1}, \widetilde{V_1}, \widetilde{M_1}, \widetilde{Vs_1}) \longrightarrow (\widetilde{sts_2}, \widetilde{V_2}, \widetilde{M_2}, \widetilde{Vs_2}) \longrightarrow \ldots$. We show by induction on the length of the reduction sequences that for all $i \geq 0$, $\ulcorner\sigma\urcorner[i].\mathsf{actions} = \widetilde{\sigma}[i].\mathsf{actions}$, $\ulcorner\sigma\urcorner[i].\mathsf{inputs} = \widetilde{\sigma}[i].\mathsf{inputs}$, and for every observable program variable x, $\ulcorner\sigma\urcorner[i].x = \widetilde{\sigma}[i].x$.

We assume that programs live in the same locations in both semantics. (Otherwise, we need to ensure that we map locations to locations, and that just complicates everything.)

The inductive step is based on the following observation. By Lemma C.3, every evaluation of AT and ET returns the same result as the corresponding evaluation of A and E, except tagged with a type. By examination of the rules, the result of applying rule (T1)–(T11) is the same as the result of applying the corresponding rule (R1)–(R11). ∎

Lemma C.5. *Let* $(sts1, V_1, M_1, Vs_1) \longrightarrow (sts_2, V_2, M_2, Vs_2) \longrightarrow \dots$ *be a reduction sequence. If* $V_i(\mathsf{pc}) = \langle -, \mathbf{low}\rangle$ *for some* i*, then for all* $j > i$*,* $V_j(\mathsf{pc}) = \langle -, \mathbf{low}\rangle$*.*

Proof. A straightforward induction on the length of the reduction sequence. ∎

Theorem 5.2. *Let* $K_1, \dots, K_n$ *be arbitrary keys for* τ_{addr}*. For any program* P *and inputs* $inps$*, if* $inps$ *is an attack on* P *relative to* τ_{addr} *and* $K_1, \dots, K_n$*, then* $\sigma \in [\![P]\!]_I^{info}(inps)$ *signals a type error.*

Proof. This proof is essentially the same as that of Theorem 5.1.

Assume that $\sigma \in [\![P]\!]_I^{strg}(inps)T$ does not signal a type error. We want to show that $inps$ is not an attack relative to P and $K_1, \dots, K_n$. In other words, if σ_i is the execution in $[\![P]\!]_I^{\tau_{addr}, K_i}(inps)$, we want $(\sigma_1, \dots, \sigma_n) \in \mathcal{B}_n^{\tau_{addr}}(P, K_1, \dots, K_n)$. This requires finding an appropriate $\widehat{\sigma} \in [\![P]\!]_H(inps)$.

We derive the required $\widehat{\sigma}$ from σ as follows. When $\sigma[j] = (\mathbb{N}, V, M)$, we take $\widehat{\sigma}[j]$ to be $(\mathbb{N}, V, \widehat{M})$, with $\widehat{M}$ given by:

$$\widehat{M}(\ell) = \begin{cases} direct(i) & \text{if } M(\ell) = \langle i, \mathbf{int}\rangle \\ pointer(i) & \text{if } M(\ell) = \langle i, \mathbf{ptr}(-,-)\rangle \end{cases}$$

It remains to show that for every i we have $(\sigma_i, \widehat{\sigma}) \in \delta(P, K_i)$. Fix i. We exhibit a relation $\precsim$ determined by a map h such that $\sigma_i[j] \precsim \widehat{\sigma}[j]$ for all j. The map h maps memory locations in the states of execution σ_i to corresponding memory locations in the states of the high-level execution $\widehat{\sigma}$. Roughly speaking, for any $\ell \in L$, if ℓ in the range of any observable program variable x in V, then we put $h(s, \ell)$ in the corresponding range of that same x in $\widehat{V}$.

The relevant portions of the map h are defined inductively over the sequence of reductions $(\overline{sts_1}, \overline{V_1}, \overline{M_1}, \overline{Vs_1}) \longrightarrow (\overline{sts_2}, \overline{V_2}, \overline{M_2}, \overline{Vs_2}) \longrightarrow \dots$ generating σ_i. Let $(sts_1, V_1, M_1, Vs_1) \longrightarrow (sts_2, V_2, M_2, Vs_2) \longrightarrow \dots$ be the sequence of reductions generating σ. As in the proof of Lemma C.2, we assume without loss of generality that programs live in the same locations in both semantics; thus, we take $h(s, \ell) = \ell$ for every state s and every location $\ell < \ell_{RW}$. We specify $h(s, \ell)$ only for states s of the form $(\mathbb{N}, \overline{V_j}, \overline{M_j})$, for every $j \geq 1$. For $j = 1$, we can take $h((\mathbb{N}, \overline{V_1}, \overline{M_1}), \ell)$ to be arbitrary, since there are no observable program variables in the initial state of the execution. Inductively, assume that we have $h((\mathbb{N}, \overline{V_j}, \overline{M_j}), \cdot)$. If the reduction rule applied at step j is any reduction but (R11*), then take $h((\mathbb{N}, \overline{V_{j+1}}, \overline{M_{j+1}}), \cdot)$ to be the same as $h((\mathbb{N}, \overline{V_j}, \overline{M_j}), \cdot)$. If the reduction rule applied at step j is (R11*), then take $h((\mathbb{N}, \overline{V_{j+1}}, \overline{M_{j+1}}), \cdot)$ to be $h((\mathbb{N}, \overline{V_j}, \overline{M_j}), \cdot)$, updated to map

$$\overline{V_{j+1}}(\mathsf{sp}) + 3d + n + 1 \text{ to}$$
$$V_{j+1} + n + 1 + \sum_{i=1}^{\pi_k^{-1}(1)-1} \mathit{size}(ty_i)$$
$$\vdots$$
$$\overline{V_{j+1}}(\mathsf{sp}) + 3d + n + 1 + \mathit{size}(ty_{\pi_k(1)}) \text{ to}$$
$$V_{j+1} + n + 1 + \sum_{i=1}^{\pi_k^{-1}(1)-1} \mathit{size}(ty_i) + \mathit{size}(ty_{\pi_k^{-1}(1)})$$
$$\vdots$$
$$\overline{V_{j+1}}(\mathsf{sp}) + 3d + n + 1 + \sum_{i=1}^{k-1} \mathit{size}(ty_{\pi_k(i)}) \text{ to}$$
$$V_{j+1} + n + 1 + \sum_{i=1}^{\pi_k^{-1}(k)-1} \mathit{size}(ty_i)$$
$$\vdots$$
$$\overline{V_{j+1}}(\mathsf{sp}) + 3d + n + 1 + \sum_{i=1}^{k-1} \mathit{size}(ty_{\pi_k(i)}) + \mathit{size}(ty_{\pi_k(k)}) \text{ to}$$
$$V_{j+1} + n + 1 + \sum_{i=1}^{\pi_k^{-1}(k)-1} \mathit{size}(ty_i) + \mathit{size}(ty_{\pi^{-1}(k)})$$

Let $\precsim$ be a relation determined by h satisfying the above. We use induction to show that for all $j \geq 1$ such that $V_j(\mathsf{pc}) = \langle -, t \rangle$ and $t \neq \mathbf{low}$:

(i) $\overline{sts_j} = sts_j$;

(ii) for all ex,

- if $ET[\![ex]\!](V_j, M_J) = \langle i', \mathbf{int} \rangle$, then $E[\![ex]\!](\overline{V_j}, \overline{M_j}) = i'$;
- if $ET[\![ex]\!](V_j, M_j) = \langle i', \mathbf{ptr}(-, -) \rangle$, then $E[\![ex]\!](\overline{V_j}, \overline{M_j}) = i''$ and $h((\mathbb{N}, \overline{V_j}, \overline{M_j}), i'') = i'$;

and similarly for all lv;

(iii) If reduction (Rn), for some $n < 11$, applies at step j to produce $\sigma_i[j+1]$, then reduction (Tn) applies at step j to produce $\sigma[j+1]$; if reduction (R11*) applies at step j to produce $\sigma_i[j+1]$, then reduction (T11) applies at step j to produce $\sigma[j+1]$;

and for all $j \geq 1$:

(iv) $\sigma_i[j] \precsim \widehat{\sigma}[j]$;

The base case, $j = 1$, is immediate, since the initial states $\sigma[1]$ and $\sigma_i[1]$ are the same, up to the types associated with the values in memory, meaning that the states $\widehat{\sigma}[1]$ and $\sigma_i[1]$ are also the same, up to the tagging of the values required by the high-level semantics $\llbracket\cdot\rrbracket_H$.

For the inductive case, assume that we have the result for j; we show it for $j+1$. Establishing (i), (ii), and (iii) is straightforward. (This is only needed as long as $V_j(\mathsf{pc}) = \langle -, t\rangle$ with $t \neq \mathbf{low}$, by Lemma C.5.) To establish (iv), we consider two cases. If $V_j(\mathsf{pc}) = \langle -, \mathbf{low}\rangle$, then because execution does not signal a type error, $V_{j+1}(\mathsf{actions}) = V_j(\mathsf{actions})$, and $V_{j+1}(x) = V_j(x)$ for every observable program variable x. The result then follows easily by choice of h, and stuttering. If $V_j(\mathsf{pc}) = \langle -, t\rangle$ with $t \neq \mathbf{low}$, then we need to show: (1) either $\overline{V_{j+1}}(\mathsf{pc}) = \bullet$ and $V_{j+1}(\mathsf{pc}) = \bullet$, or $h(\sigma_i[j+1], \overline{V_{j+1}}(\mathsf{pc})) = V_{j+1}(\mathsf{pc})$; (2) $\overline{V_{j+1}}(\mathsf{actions}) = V_{j+1}(\mathsf{actions})$; (3) $\overline{V_{j+1}}(\mathsf{inputs}) = V_{j+1}(\mathsf{inputs})$; (4) for every observable program variable x, there exists $k \geq 0$ such that $\overline{V_{j+1}}(x) = \langle \ell_1, \ldots, \ell_k\rangle$, $V_{j+1}(x) = \langle \widehat{\ell_1}, \ldots, \widehat{\ell_k}\rangle$, and for all $j \leq k$ we have $\ell_j \precsim \widehat{\ell_j}$.

The proof proceeds by case analysis on the reduction rule that applies at step j. (By (iii), we know that corresponding reduction rules apply to produce $\sigma_i[j+1]$ and $\sigma[j+1]$.) Most of the cases are trivial using (i)–(iii). The only case of interest is when the rules that apply at step j are (R11*) and (T11). Thus, $sts_j = \overline{sts_j} = m(ex_1, \ldots, ex_n); sts$, and $ET\llbracket ex_k\rrbracket(V_j, M_j) = \langle i_k, t_k\rangle$ for all k. By (ii), $E\llbracket ex_k\rrbracket(\overline{V_j}, \overline{M_j}) = i'_k$ for all k, where $i_k = i'_k$ if $t_k = \mathbf{int}$, and $h(\sigma_i[j], i'_k) = i_k$ if $t_k = \mathbf{ptr}(-,-)$. (1), (2), and (3) follow immediately. For (4), note that if $\overline{V_{j+1}}(x) = \langle \ell_1, \ldots, \ell_k\rangle$ for an observable program variable x, then either x was not newly allocated with the current reduction rule, in which case we already have (4) by the induction hypothesis, or x is newly allocated, in which case by observation of the rules and by the construction of h, (4) holds. Thus, we have $\sigma_i[j+1] \precsim \widehat{\sigma}[j+1]$. ∎

Lemma 5.4. *$\llbracket\cdot\rrbracket_I^{strg}$ and $\llbracket\cdot\rrbracket_I^{info}$ are restrictions of $\llbracket\cdot\rrbracket_I^{ref}$.*

Proof. See Lemmas C.2 and C.4. ∎

Theorem 5.5. *Let $\llbracket\cdot\rrbracket_I$ be an implementation semantics for Toy-C such that:*

(i) *For every P and inps, $\llbracket P\rrbracket_I(inps)$ is computable;*

(ii) *$\llbracket\cdot\rrbracket_I$ is a restriction of $\llbracket\cdot\rrbracket_I^{ref}$;*

(iii) *$\sigma \in \llbracket P\rrbracket_I(inps)$ signals a type error whenever inps is an attack relative to τ_{addr} and some finite set of keys.*

Then, there exists a program P and input inps such that $\sigma \in \llbracket P\rrbracket_I(inps)$ signals a type error, but for all finite sets of keys $K_1, \ldots, K_n$, inps is not an attack relative to τ_{addr} and $K_1, \ldots, K_n$.

Proof. Assume by way of contradiction that implementation semantics $\llbracket\cdot\rrbracket_I$ signals a type error on program P and input *inps* if and only if there exists a finite set $K_1, \ldots, K_n$ of keys such that *inps* is an attack relative to τ_{addr} and $K_1, \ldots, K_n$. We derive a contradiction by showing that such an implementation semantics, which is computable by assumption (i), gives us an algorithm for an undecidable problem.

Let ac_0 be an arbitrary but fixed output action. Let PR be the class of Toy-C procedures with a single integer parameter that do not create or dereference pointers and that either do not perform any output action, or perform a single output action ac_0. The problem of deciding whether a procedure in PR performs output action ac_0 for all positive inputs is an undecidable problem, since Toy-C has arithmetic and iteration primitives.

We use implementation semantics $[\![\cdot]\!]_I$ to decide whether a procedure proc in PR performs output action ac_0 for all positive inputs. Consider the following program P_{proc},

```
main(i : int) {
   var a : int[5];
         x : int;
   x := *(&a − 2);
   if (x = 0) {
      $ac_0$;
   } else {
      proc(x);
   }
},
```

parameterized by procedure proc in PR. The intuition is that a morph of P_{proc} will invoke procedure proc with some arbitrary value (viz., the result of dereferencing &a − 2). Observe that for every possible integer value v, there is a morph of P_{proc} that invokes procedure proc with argument v. Consider the infinite set of keys $\mathcal{K} = \{K_1, K_2, \dots\}$, where $K_i = (\ell_{RW} + 1, 0, \Pi_0, M_i)$ with Π_0 the sequence of identity permutations and M_i a memory map that assigns value $\langle i-1, \mathbf{int}\rangle$ to the location ℓ_{RW}. According to semantics $[\![\cdot]\!]_I^{\tau_{addr},K}$ (Appendix B), when passed any input value, any morph of the program with respect to a key $K_i \in \mathcal{K}$ will execute procedure proc, passing it the content of memory location ℓ_{RW} (viz., $i-1$).) Furthermore, morph $\tau_{addr}(P, K_1)$ performs output action ac_0 (since x gets value $\langle 0, \mathbf{int}\rangle$ stored in location ℓ_{RW}). Based on these observations, it is easy to see that procedure proc performs output action ac_0 for all positive inputs if and only if any input to P_{proc} is not an attack relative to τ_{addr} and any finite subset of keys.

This gives an algorithm for deciding whether a procedure proc in PR performs output action ac_0 for all positive arguments to proc: execute P_{proc} under implementation semantics $[\![\cdot]\!]_I$, with an arbitrary input; if a type error is signalled, then return "no", otherwise, return "yes". We claim that this algorithm is correct, that is, it returns "yes" if and only if procedure proc performs output action ac_0 for all positive arguments to proc. Specifically, we show that for any input i, $\sigma \in [\![P_{\mathsf{proc}}]\!]_I(i)$ does not signal a type error if and only if procedure proc performs output action ac_0 for all positive arguments to proc.

- Assume that proc performs output action ac_0 for all positive arguments. It is easy to see that the only difference between the morphs of the program is the value of the argument passed to proc. Thus, every morph of P_{proc}, under every possible key, will perform output action ac_0, because proc performs output action ac_0 when passed any argument. In other words, for every finite set of keys, an input i is not an attack relative to τ_{addr} and $K_1, \dots, K_n$. And therefore, by assumption, execution $\sigma \in [\![P_{\mathsf{proc}}]\!]_I(i)$ does not signal a type error.

- Assume execution $\sigma \in [\![P_{\mathsf{proc}}]\!]_I(i)$, for an arbitrary input i, does not signal a type error. Then, by assumption, input i is not an attack relative to τ_{addr} and any finite set of keys. Thus, every morph of the program will perform output action ac_0, and therefore proc will perform output action ac_0 under every possible argument, since for every possible integer value v there is a morph that invokes proc with argument v.

Thus, we have an algorithm for deciding an undecidable problem, a contradiction. ∎

D Summary of Notation

P	Program
inps	Inputs
$\tau(P, K)$	Morph of program P with key K under obfuscator τ
$[\![P]\!]_I(\mathit{inps})$	Implementation semantics of P with input *inps* (generic)
$[\![P]\!]_I^{\tau,K}(\mathit{inps})$	Implementation semantics of morph $\tau(P, K)$ with input *inps*
$\mathcal{B}_n^\tau(P, K_1, \ldots, K_n)$	Equivalence of execution for morphs $\tau(P, K_1), \ldots, \tau(P, K_n)$
L	Memory location
V	Variable map
M	Memory map
Σ	Set of states
σ	sequence of states in Σ
$\sigma[i]$	ith state of sequence σ
$\sigma[i].v$	value of variable v in ith state of σ
$\delta(P, K)$	Deobfuscation relation for program P and key K
$[\![P]\!]_H(\mathit{inps})$	High-level semantics of program P on input *inps*
$[\![P]\!]_I^{\mathit{ref}}(\mathit{inps})$	Reference semantics of program P on input *inps*
T^{strg}	Type system for strong typing
$[\![P]\!]_I^{\mathit{strg}}(\mathit{inps})$	Implementation semantics for T^{strg}
T^{info}	Type system for integrity-based typing
$[\![P]\!]_I^{\mathit{info}}(\mathit{inps})$	Implementation semantics for T^{info}
$T^{\mathit{mrph}}_{K_1,\ldots,K_n}$	Exact type system corresponding to $K_1, \ldots, K_n$
$[\![P]\!]_I^{\mathit{mrph},K_1,\ldots,K_n}(\mathit{inps})$	Implementation semantics for $T^{\mathit{mrph}}_{K_1,\ldots,K_n}$
T^{rand}	Randomized exact type system

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proc. 26th Annual ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 147–160. ACM Press, 1999.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Proc. 21th Annual International Cryptology Conference (CRYPTO'01)*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2001.

[3] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanovic. Randomized instruction set emulation. *ACM Transactions on Information and System Security*, pages 3–40, 2005.

[4] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. 10th ACM Conference on Computer and Communications Security (CCS'03)*, pages 281–289. ACM Press, 2003.

[5] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. Department of Computer Science Technical Report 05-65, University of Massachusetts Amherst, 2005.

[6] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proc. 12th USENIX Security Symposium*, pages 105–120, 2003.

[7] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, School of Computer Science, Carnegie Mellon University, 2002.

[8] C. S. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Proc. 1998 International Conference on Computer Languages*, pages 28–38. IEEE Computer Society Press, 1998.

[9] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, 2000.

[10] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proc. 6th Workshop on Hot Topics in Operating Systems*, pages 67–72. IEEE Computer Society Press, 1997.

[11] S. Goldwasser and Y. T. Kalai. On the impossibility of obfuscation with auxiliary inputs. In *Proc. 46th IEEE Symposium on the Foundations of Computer Science (FOCS'05)*, 2005.

[12] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proc. USENIX Annual Technical Conference*, pages 275–288, 2002.

[13] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. 10th ACM Conference on Computer and Communications Security (CCS'03)*, pages 272–280. ACM Press, 2003.

[14] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proc. 29th Annual ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 128–139. ACM Press, 2002.

[15] P. Ørbæk and J. Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, 1997.

[16] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.

[17] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proc. 11th ACM Conference on Computer and Communications Security (CCS'04)*, pages 298–307. ACM Press, 2004.

[18] A. N. Sovarel, D. Evans, and N. Paul. Where's the FEEB?: The effectiveness of instruction set randomization. In *Proc. 14th USENIX Security Symposium*, 2005.

[19] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 32–41. ACM Press, 1996.

[20] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proc. 22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 260–269. IEEE Computer Society Press, 2003.