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

May 11, 2009

Abstract

A set of replicas is diverse to the extent that they implement the same functionality but differ in their implementation details. Diverse replicas are less likely to succumb to the same attacks, when attacks depend on memory layout and/or other implementation details. 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, this paper gives a precise characterization of attacks, applicable to viewing diversity as a defense, and also shows how mechanically-generated diversity compares to a well-understood defense: type checking.

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 malware. It is also a reason that replication of servers does not necessarily enhance the availability of a service in the face 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 enhanced availablity.

A set of replicas is *diverse* if all implement the same functionality but differ in their implementation details. Diverse replicas are less prone to having vulnerabilities in common.

*This work was mainly performed while the first author was at Cornell University. A preliminary version of this paper appears in the *Proc. 19th IEEE Computer Security Foundations Workshop*, pp. 230–241, 2006. 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 grants from Microsoft and Intel. 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.

Thus, they exhibit a form of independence with respect to attacks. Building multiple distinct versions of a program is expensive, however, so researchers have advocated 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 [13, 6, 29], rearranging basic blocks and code within basic blocks [13], randomly changing the names of system calls [9] or instruction opcodes [16, 4, 3], and randomizing the heap memory allocator [5]. Some of these approaches are more effective than others. For example, Shacham *et al.* [24] derive experimental limits on the address space randomization scheme proposed by Xu *et al.* [29], while Sovarel *et al.* [25] and Weiss and Barrantes [28] 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. 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 intended to accomplish such transformations have been called *obfuscators* [10]. 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, where secret key K prescribes which transformations are applied in producing $\tau(P, K)$.[1] Since P and τ are 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 [14] give theoretical limits on the effectiveness of obfuscators for keeping secret the details of an algorithm or its embodiment as a program. This work, however, says nothing 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 this prior work is 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 the correspondence is important when designing a set of defenses for a given threat model, this is not the same as knowing the overall power of mechanically-generated diversity as a defense. We explore that latter, broader, issue here, by

- giving definitions suited 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

[1] Thus, an obfuscator is a way to mechanically generate diversity, where the output of an obfuscator is called a morph.

systems. We show that there can be no computable type system corresponding to a rich obfuscator for an expressive programming language. We also show that under suitable restrictions, it is possible to give an admittedly unusual type system equivalent to obfuscation;

- exhibiting, for a C-like language and an obfuscator that rearranges memory, an increasingly tighter sequence of type systems for soundly approximating the defenses provided by that obfuscator. The most accurate of these type systems is based on information flow.

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. Note that type systems are believed to be useful for improving reliability—by catching violations of safety (for various notions of safety). Thus an implication of our results is that mechanically-generated diversity also can help improve reliability by converting a fault into a crash rather than leaving an insidious error that is hard to track down.

2 Attacks and Obfuscators: A Framework

We assume, without loss of generality, that a program interacts with its environment through inputs and outputs. Inputs include initial arguments supplied to the program when it is invoked, additional data supplied during execution through communication, and so on. Outputs are presumably sensed by the environment, and they can include state changes (e.g., updates to observable program variables or memory locations). A program's *behavior* is defined as a sequence of inputs and outputs.

An *implementation semantics* $[\![\cdot]\!]_I$ describes *executions* of programs, generally at the level of machine instructions: for a program P and inputs *inps*, $[\![P]\!]_I(inps)$ is the set of executions of program P comprising sequences of states that engender possible behaviors of the program with 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*, executions given by two different implementation semantics could well be different.

Associate an implementation semantics $[\![P]\!]_I^{\tau,K}$ with each morph $\tau(P, K)$. This allows us to model various kinds of obfuscations:

- An obfuscator can be seen as a source-to-source translator, and we can take morphs to be programs in the same language.

- An obfuscator can be seen as a compiler with different compilation choices, and we can take morphs to be compiled code.

- In the case of object code, an obfuscator can be seen as a binary rewriter, and we can take morphs to be object code.

Notice that an obfuscator is not precluded from adding runtime checks. So, our characterization of an obfuscator admits program rewriters that add checks to protect against bad programming practices.

Attacks are conveyed through inputs and are defined relative to some program P, an obfuscator τ, and a finite set of keys $K_1, \ldots, K_n$. A *resistable attack on program P relative to obfuscator τ and keys $K_1, \ldots, K_n$* is defined to be an input that produces a behavior in some morph $\tau(P, K_i)$ where that behavior cannot be produced by some other morph $\tau(P, K_j)$—presumably because implementation details differ from morph to morph.[2] When morphs are deterministic, the definition of a resistable attack simplifies to being an input that produces different behaviors in some pair of morphs $\tau(P, K_i)$ and $\tau(P, K_j)$.

Obfuscators may admit 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 a finite number of server replicas), 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 resistable attacks relative to an obfuscator τ and some finite subset of the possible keys. Accordingly, a *resistable attack on program P relative to obfuscator τ* is defined to be an input *inps* for which there exists a finite set of keys $K_1, \ldots, K_n$ such that *inps* is a resistable attack on P relative to τ and $K_1, \ldots, K_n$.

Whether executions from two different morphs reading the same input constitute different behaviors is subtle. Different morphs might represent state components and sequence state changes in different ways (e.g., by reordering instructions). Therefore, whether two executions engender the same behavior is best not 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) might create that tree in different regions of memory. Even though different addresses are returned by each morph, we would want these executions to be considered equivalent if the trees at those different addresses are equivalent.

We formalize execution equivalence of n executions from the morphs produced by obfuscator τ using relations $\mathcal{B}_n^\tau(\cdot)$, one for every $n \geq 0$. These relations are parameters in our framework, and they must be chosen appropriately to capture the desired notion of execution equivalence. 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 engender the same behavior. This, however, may be too restrictive, 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., $\sigma_i \in [\![P]\!]_I^{\tau, K_i}(inps)$ holds), we define

$$(\sigma_1, \ldots, \sigma_n) \in \mathcal{B}_n^\tau(P, K_1, \ldots, K_n) \tag{1}$$

to hold if and only if executions σ_1, …, σ_n all engender equivalent behavior. Equation (1) has to be instantiated for each given language and obfuscator. A common way to define the $\mathcal{B}_n^\tau(\cdot)$ relation, but by no means the only way, is to relate implementation semantics to an idealized execution, perhaps given by a more abstract semantics. We do this in §4 for a particular language, Toy-C.

[2] An attack that produces equivalent behavior in all morphs might indicate an obfuscator that does not introduce sufficient diversity in the morphs, or it might indicate poorly chosen semantics of the program's interface [11]. We refer to the latter case as an *interface attack*. Without some independent specification, interface attacks are indistinguishable from other program inputs; mechanically-generated diversity is useless against interface attacks.

When morphs are deterministic programs, and thus each σ_i is a unique execution of morph $\tau(P, K_i)$ on input *inps*, then by definition *inps* is a resistable 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 a resistable attack if there exists an execution $\sigma_j \in [\![P]\!]_I^{\tau, K_j}(inps)$ for some $j \in \{1, \ldots, n\}$ such that for all choices of $\sigma_i \in [\![P]\!]_I^{\tau, K_i}(inps)$ (for $i \in \{1, \ldots, j-1, j+1, \ldots, n\}$) we have $(\sigma_1, \ldots, \sigma_n) \notin \mathcal{B}_n^\tau(P, K_1, \ldots, K_n)$.[3]

$\mathcal{B}_n^\tau(\cdot)$ need not in general be an equivalence relation on executions because execution equivalence for some languages might involve supposing an interpretation for states—an implicit existential quantifier—rather than requiring strict equality of all state components (as we do require for outputs). 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.

3 Obfuscation and Type Systems

Even when obfuscation does not eliminate vulnerabilities, it can make 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 is declared to accommodate. More generally, the property commonly enforced by type systems is *code safety*: every variable is associated with a set of values it can store, every operation has a signature indicating what values it can take as arguments, and execution is aborted when an attempt is made to store a unacceptable value in a variable or invoke an operation with an unacceptable value as argument.

Eliminating vulnerabilities is clearly preferable to having them be difficult to exploit. So why bother with obfuscation? The answer is that strong type systems are not always an option with legacy code. The relative success of recent work [15, 18] in adding strong typing to languages like C not withstanding, obfuscation is applicable to any object code, independent of what high-level language it derives from. 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.

[3] This definition implies that in programs that use randomization or are subject to race conditions, inputs will generally be identified as attacks. This is a limitation of our framework. However, it is not clear how to do things differently—the difficulty seems inherent to defining attacks in terms of observable behaviors rather than looking at or analyzing program code.

We distinguish two forms of type systems:

- *Compile-time type systems* report type errors before execution by analyzing the program code. For an expressive enough programming language, analyzing program code does not suffice to determine everything about its run-time behaviors. Therefore, compile-time type systems may rule out programs whose behaviors are acceptable.

- *Run-time type systems* add checks to the program, and if one of those checks fails at run-time then the program halts. Given an execution, even if it does not halt due to a type error, there might be another execution that does. Therefore, run-time type systems may accept programs capable of exhibiting unacceptable behaviors. Run-time type systems are, in effect, reference monitors. Hence, run-time type systems are restricted to enforcing safety properties [23].

A type system is *computable* if the checks it implements terminate. Thus, a compile-time type system is computable if it always terminates in response to the question "does this program type check?" A run-time type system is computable if all the run-time checks it implements terminate.

To compare obfuscation with type systems, we view obfuscation as a form of *probabilistic type checking*. A compile-time or run-time type system is probabilistic if there is some probability of error associated with the result. Thus, for a probabilistic run-time type system, 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 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 run-time type checking.

3.1 Exact Type Systems

We start our comparison by first showing that for any expressive programming language and a rich obfuscator, there is no computable type system that signals a type error on an input exactly when that input is a resistable attack. By restricting the programming language, however, we can (and do) exhibit an equivalence between obfuscators and type systems. In §4.6, we will build on these results and discuss, for a concrete programming language, how the kind of strong typing being advocated for programming languages compares to what can be achieved with obfuscation.

It is well known that type checking is undecidable for an expressive programming language and type system. What about type systems that attempt to signal a type error for resistable attacks relative to an obfuscator? As we now show, these type systems are also undecidable for an expressive programming language and obfuscator. More precisely, we show there can exist no compile-time or run-time computable type system that signals a type error when executing program P on input *inps* exactly when *inps* is a resistable attack relative to τ if P is written in a language whose programs can detect differences among morphs.

That differences between morphs can be detected by a program is captured by the existence of a program we call a *key classifier* for obfuscator τ with associated relation

$\mathcal{B}^n_\tau(\cdot)$. Intuitively, a key classifier for τ is a program KC_τ that reads no input and classifies each key into an equivalence class, where every key in an equivalence class produces a morph of KC_τ that returns the same element of some universe U. The intent is to use a key classifier as a subroutine, where the value it returns is not observable by the environment (for instance, by being stored in a variable that is not observable by the environment).

Formally, KC_τ is a key classifier for τ if there exits a countable set $A \subseteq U$ with at least two elements satisfying:

(1) KC_τ is a deterministic program that reads no inputs, halts, and returns an encoding of an element in U (but see below);

(2) for all keys K, $\tau(KC_\tau, K)$ returns an encoding of an element in A;

(3) $\mathcal{B}^n_\tau(\cdot)$ is such that for all finite sets of keys $K_1, \ldots, K_n$, if σ_i is the execution of $\tau(KC_\tau, K_i)$ for $i = 1, \ldots, n$, then $(\sigma_1, \ldots, \sigma_n) \in \mathcal{B}^n_\tau(KC_\tau, K_1, \ldots, K_n)$; and

(4) for every element $a \in A$, there is a key K such that morph $\tau(KC_\tau, K)$ returns an encoding of a.

Roughly speaking, a key classifier distinguishes among τ-morphs to some extent.

We require that Turing-completeness of the programming language not be destroyed by obfuscator τ. More precisely, not only must there be a way to write a Turing machine simulator in the programming language, but the simulator must exhibit the same behavior under obfuscation with any key.

Theorem 3.1. *Let L be a programming language and let τ be an obfuscator for programs in L. If*

(1) there exists a key classifier KC_τ for τ in L;

(2) there exists a Turing machine simulator $sim(\cdot,\cdot)$ written in L such that $sim(M, x)$ simulates Turing machine M on input x and has the same behavior as $\tau(sim(M, x), K)$ for all keys K;

(3) for all keys K and programs P, morph $\tau(P; output(0), K)$ has the same behavior as $\tau(P, K); output(0)$;

then there is no computable type system for L that signals a type error on input inps to program P if and only if inps is a resistable attack on program P relative to τ.

Proof. Assume by way of contradiction that there is a computable type system T that signals a type error on program P and input *inps* if and only if *inps* is a resistable attack relative to τ. In other words, there is a computable type system T that 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 a resistable attack relative to τ and $K_1, \ldots, K_n$. We derive a contradiction by showing that such a computable type system can be used to construct an algorithm for deciding the following undecidable problem: given a nonempty set I, determine whether a Turing machine halts on all inputs in I.[4]

[4]This problem is easily shown undecidable by reduction from the problem of determining whether a Turing machine accepts all inputs in some given nonempty set I of inputs, which can be shown undecidable using Rice's Theorem [21].

Let A be the set of elements into which KC_τ classifies keys. (For simplicity of exposition, we equate elements and their encoding.) Let a be an arbitrary element of A, and let $I = A - \{a\}$. Here is an algorithm to determine whether a Turing machine M halts on all inputs in I:

Algorithm $\textsc{Halt}_I$:

1. Construct Turing machine M_a that behaves just like Turing machine M, except that it halts immediately on input a.
2. Construct program *Test*:

$$let\ x = KC_\tau\ in\ sim(M_a, x); output(0).$$

3. If type system T is a run-time type system, run *Test* (on a void input).
4. If a type error is signaled by T, return "No, M does not halt on all inputs in I".
5. If no type error is signaled by T, return "Yes, M halts on all inputs in I".

We claim that this algorithm is correct: it returns "Yes, M halts on all inputs in I" if and only if Turing machine M halts on all inputs in I. Two facts are important for the proof of correctness of algorithm $\textsc{Halt}_I$.

Fact 1: M halts on all inputs in I if and only if M_a halts on all inputs in I, because $a \notin I$, by construction of I.

Fact 2: Morph $\tau(Test, K)$ has the same behavior as

$$let\ x = \tau(KC_\tau, K)\ in\ sim(M_a, x); output(0),$$

using hypotheses (2) and (3).

Here now is the correctness proof for $\textsc{Halt}_I$.

- Assume that M halts on all inputs in I. We need to prove that algorithm $\textsc{Halt}_I$ returns "Yes, M halts on all inputs in I". According to step 5 of $\textsc{Halt}_I$, it suffices to show that *Test* does not signal a type error. By assumption, *Test* does not signal a type error on its void input if and only if that input is not a resistable attack relative to τ. Thus, if we want *Test* not to signal a type error, then it suffices to show that given any finite set of keys $K_1, \ldots, K_n$, morphs $\tau(Test, K_1), \ldots, \tau(Test, K_n)$ have the same behavior.

  Let $K_1, \ldots, K_n$ be any finite set of keys. Let $a_1, \ldots, a_n$ the values into which key classifier KC_τ classifies keys $K_1, \ldots, K_n$, respectively. (That is, $\tau(KC_\tau, K_i)$ returns value a_i.) By definition, every a_i is in A. By assumption, M halts on all inputs in I; thus, by Fact 1 above, M_a halts on all inputs in I as well. Moreover, M_a also halts on input a by construction. Thus, M_a halts on all inputs in $I \cup \{a\} = A$. This means that

$$let\ x = \tau(KC_\tau, K_i)\ in\ sim(M_a, x); output\ (0)$$

halts and outputs 0 for $i = 1, \ldots, n$. By Fact 2, this means that $\tau(\mathit{Test}, K_i)$ halts and outputs 0 for $i = 1, \ldots, n$, and thus morphs $\tau(\mathit{Test}, K_1), \ldots, \tau(\mathit{Test}, K_n)$ have the same behavior, and therefore T does not signal a type error; so $\textsc{Halt}_I$ returns "Yes".

- Assume that M does not halt on all inputs in I. We need to prove that algorithm $\textsc{Halt}_I$ returns "No, M does not halt on all inputs in I". According to step 4 of $\textsc{Halt}_I$, it suffices to show that *Test* signals a type error. By assumption, *Test* signals a type error on its void input if and only if that input is a resistable attack relative to τ and some finite set of keys $K_1, \ldots, K_n$; that is, if $\tau(\mathit{Test}, K_1), \ldots, \tau(\mathit{Test}, K_n)$ do not all have the same behavior. Thus, it suffices to show that there exists a finite set of keys $K_1, \ldots, K_n$ such that $\tau(\mathit{Test}, K_1), \ldots, \tau(\mathit{Test}, K_n)$ do not all have the same behavior.

  Let a' be an input in I on which M does not halt. By construction, M_a also does not halt on a'. Let $K_{a'}$ be a key that key classifier KC_τ classifies as returning value a'. Such an a' exists by definition of KC_τ. By construction, M_a halts on input a. Let K_a be a key that key classifier KC_τ classifies as returning value a. Again, such an a exists by definition of KC_τ. We claim that keys $K_a, K_{a'}$ are such that $\tau(\mathit{Test}, K_a), \tau(\mathit{Test}, K_{a'})$ do not have the same behavior, since $\tau(\mathit{Test}, K_a)$ halts but $\tau(\mathit{Test}, K_{a'})$ does not. By Fact 2, $\tau(\mathit{Test}, K_a)$ has the same behavior as

  $$\mathit{let}\ x = \tau(KC_\tau, K_a)\ \mathit{in}\ \mathit{sim}(M_a, x); \mathit{output}\ (0)$$

  and $\tau(\mathit{Test}, K_{a'})$ has the same behavior as

  $$\mathit{let}\ x = \tau(KC_\tau, K_{a'})\ \mathit{in}\ \mathit{sim}(M_a, x); \mathit{output}\ (0).$$

  But the first of these programs outputs 0, because M_a halts on input a, while the second of these programs does not output 0, because M_a does not halt on input a'. Thus, these two programs do not have the same behavior, and $\tau(\mathit{Test}, K_a), \tau(\mathit{Test}, K_{a'})$ do not have the same behavior, and therefore T signals a type error; so $\textsc{Halt}_I$ returns "No".

Correctness of $\textsc{Halt}_I$ implies that algorithm $\textsc{Halt}_I$ decides an undecidable problem, which is the sought contradiction. ∎

Theorem 3.1 establishes that it is impossible, for a general enough programming language and a rich enough obfuscator, to devise a computable type system that signals a type error exactly when an input is a resistable attack relative to τ and an arbitrary finite set of keys. Any computable type system must therefore approximate this.

It is possible to obtain a computable type system by limiting either the programming language or the property checked by the type system. This suggests that if we restrict the programming language, then it could be possible to devise a type system that exactly signals a type error when an input is a resistable attack relative to an obfuscator. For example, if we are willing to restrict the behavior of morphs and require that only finitely many execution steps separate successive outputs, then the following type system signals a type error for exactly those executions corresponding to inputs that are resistable attacks relative to any τ and some fixed and finite set $K_1, \ldots, K_n$ of keys. (For simplicity, here, we

assume execution equivalence is defined as having the same sequence of observable outputs.) This type system, admittedly unusual, will be called the trivial type system, $T^{mrph}_{K_1,\ldots,K_n}$, instantiated by an implementation semantics $[\![P]\!]^{mrph,K_1,\ldots,K_n}_I(inps)$ that repeatedly runs all morphs in parallel alongside P with the same input, checking for unanimous consensus on each value to be output before performing each output action by P:

> Execute morphs $\tau(P,K_1),\ldots,\tau(P,K_n)$ up to their next output statement:
>
> - If the same output is next about to be produced by all morphs, then the type system allows that output to be produced, and repeats the procedure;
> - If not, then the type system signals a type error and aborts execution.

This run-time type system implements a form of N-version programming [8].

Theorem 3.2. *Let $K_1,\ldots,K_n$ be arbitrary keys for τ. For any program P and inputs inps satisfying*

> *For every K_i, $\tau(P,K_i)$ takes finitely many execution steps between subsequent outputs;*

then inps is a resistable attack on P relative to τ and $K_1,\ldots,K_n$ if and only if $\sigma \in [\![P]\!]^{mrph,K_1,\ldots,K_n}_I(inps)$ signals a type error.

Proof. Immediate from the description of the procedure and the definition of a resistable attack relative to τ and $K_1,\ldots,K_n$. ∎

This theorem establishes that when programs are restricted enough, run-time type systems are equivalent to obfuscation under a fixed finite set of keys for defending against attacks. This correspondence is used below.

$T^{mrph}_{K_1,\ldots,K_n}$ can be viewed as approximating a type system that aborts exactly those executions corresponding to inputs that are resistable attacks relative to τ and any finite set of keys. Adding 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 resistable attacks. This is because every resistable attack relative to τ and $K_1,\ldots,K_n$ is a resistable attack relative to τ and $K_1,\ldots,K_n,K'$, but not vice versa.

The approximation embodied by type system $T^{mrph}_{K_1,\ldots,K_n}$ serves as a basis for a probabilistic approximation of the type system that aborts exactly those executions corresponding to inputs that are resistable attacks relative to τ 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 resistable attacks relative to τ and $K_1,\ldots,K_n$ but may miss inputs that are resistable attacks relative to τ 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 a resistable attack relative to some finite set of keys.

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

Figure 1: Example Toy-C program

4 A 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 simplified C-like language, Toy-C. Our interest in C-like languages is motivated by an interest in understanding obfuscation for legacy code written in C. Despite its simplicity, Toy-C supports pointer-addressable memory and allocatable buffers, so it remains subject to many of the same attacks as the full C language. 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 an example 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. Every procedure can optionally be annotated to indicate which variables are observable—that is, variables that can be examined by the environment. Whether a variable is observable does not affect execution of a program; the annotation is used only for determining equivalence of executions (see §4.4).

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. We assume the following statements also are available:

- An output statement corresponding to every output, such as printing and sending to the network. For simplicity, we identify an output statement with the output that it produces.
- A statement fail that simply terminates execution with an error.

As in most imperative languages, we distinguish between two kinds of expressions, whose meaning depends on where they appear in a program. Expressions may evaluate to values

(*value-denoting expressions*, or VD-expressions for short), and expressions may evaluate to memory locations (*address-denoting expressions*, or AD-expressions for short). AD-expressions are assignable expressions, and they appear on the left-hand side of assignment statements. All other expressions in a program are VD-expressions. VD-expressions include constants, variables, pointer dereference, and address-of and arithmetic operations, while AD-expressions are restricted to variables and pointer dereferences. However, array operations can still be synthesized from existing expressions using pointer arithmetic, in the usual way, as illustrated in Figure 1 which in effect implements buf[i] := 42.

4.2 Toy-C Semantics

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 (i.e., addresses);
- 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 other facets of program execution, V associates information such as sequences of outputs, 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 explicitly manipulated by programs, and each program variable is bound to a finite, though 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; memory location ℓ_1 stores that value;
- If $k > 1$, then the variable holds multiple values (for instance, it may be an array variable, or a C-like struct variable); $\ell_1, \ldots, \ell_k$ stores its values;
- If $k = 0$, the variable is not bound in that state.

Hidden variables are artifacts of the language implementation and execution environment. For our purposes, it suffices to assume the following hidden variables exist:

- pc records the memory location of the next instruction to execute; it is always bound to an element of $L \cup \{\bullet\}$, where $\bullet$ indicates the program terminated;
- outputs records the finite sequence of outputs the program has produced;
- inputs 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, interpretable 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 $\sigma \in [\![P]\!]_I(inps)$ 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; 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 $j \geq i$; if the program has terminated in state $\sigma[i]$, then that state is stuttered for the remainder of the execution.[5]

(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 $\sigma[j] \neq \sigma[i]$; an execution either terminates with pc set to $\bullet$ or it does not terminate and changes state infinitely many times.[6]

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

(5) $\sigma[1].\mathsf{inputs} = inps$ 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; input values only get consumed, and at most one input is consumed at every execution step.

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

Base semantics $[\![\cdot]\!]_I^{base}$ 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; push increments the stack pointer and pop decrements it.

[5] This simplifies the technical development by allowing infinite sequences of states to represent all executions.

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

4.3 Vulnerabilities

Base semantics $[\![\cdot]\!]_I^{base}$ of Toy-C does not mandate safety checks when dereferencing a pointer or when adding integers to pointers. Attackers can take advantage of this freedom to execute Toy-C programs in a way never intended by the programmer, causing undesirable behavior through techniques such as [20]:

- Stack smashing: overflowing a stack-allocated buffer (array) 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;

These techniques involve updating a memory location that the programmer thought could not be affected by that operation.[7]

Consider a threat model in which attackers are allowed to invoke programs and supply inputs. Inputs are used as arguments to the main procedure of the program. For example, consider the program of Figure 1. According to base semantics $[\![\cdot]\!]_I^{base}$, 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 base semantics $[\![\cdot]\!]_I^{base}$) precedes buf on the stack, so that the assignment $*\mathsf{tmp} := 42$ 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 implements *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 accessed reliably 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 different initial memory maps.[8]

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.

[7] Pincus and Baker [20] describe two further attack techniques, namely pointer subterfuge (modifying a function pointer's value to point to attacker-supplied code) and 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). It is straightforward to extend Toy-C to model these attacks, by adding function pointers and dynamic memory allocation, respectively.

[8] Different initial memory maps model the unpredictability of values stored in memory on different machines running morphs.

An implementation semantics $[\![P]\!]_I^{\tau_{addr},K}$ specifying how to execute morph $\tau(P,K)$ is obtained by modifying base semantics $[\![P]\!]_I^{base}$ to take into account the transformations 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 that whether an input causes 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 input -3 causes the undesirable behavior in the morph that -1 had caused.

4.5 Execution Equivalence for τ_{addr}

To instantiate $\mathcal{B}_n^{\tau_{addr}}(\cdot)$ for Toy-C and τ_{addr}, we first need to develop a general theory of execution equivalence for C-like languages.

The formal definition of $\mathcal{B}_n^{\tau}(P, K_1, \ldots, K_n)$ for a C-like language can be based on relating executions of morphs to executions in a suitably chosen *high-level semantics* of the original program, which serves as an idealized specification for the language. A high-level semantics $[\![\cdot]\!]_H$ 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). Executions from different morphs of P are deemed equivalent if it is possible to rationalize each execution 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 produce equivalent outputs and read the same inputs; therefore, relation $\delta(P, K_i)$ must satisfy

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

where $Obs(\sigma)$ extracts the sequence of outputs produced and inputs remaining to be consumed by execution σ. $Obs(\sigma)$ is defined by projecting the bindings of the outputs and inputs hidden variables and removing repetitions in the resulting sequence.[9]

Given a tuple of executions $(\sigma_1, \ldots, \sigma_n)$ for a given input *inps* where each σ_i is produced by morph $\tau(P, K_i)$, we define these executions to be equivalent if and only if they all correspond to the same execution in the high-level semantics $[\![P]\!]_H(\cdot)$ of program P. This is formalized by instantiating Equation (1) as follows.

$$\begin{array}{l}(\sigma_1, \ldots, \sigma_n) \in \mathcal{B}_n^{\tau}(P, K_1, \ldots, K_n) \text{ if and only if} \\ \quad \text{Exists } \widehat{\sigma} \in [\![P]\!]_H(inps): \\ \quad\quad \text{For all } i: \quad \sigma_i \in [\![P]\!]_I^{\tau, K_i}(inps) \wedge (\sigma_i, \widehat{\sigma}) \in \delta(P, K_i).\end{array} \tag{2}$$

Accordingly, 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.

A high-level semantics $[\![\cdot]\!]_H$ that serves our purpose can be defined similarly to the base semantics $[\![\cdot]\!]_I^{base}$, except that values are used only 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 interpreted literally—for instance, an integer representing some count. In contrast, a pointer is 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.[10]

Executions in high-level semantics $[\![\cdot]\!]_H$ are similar to executions described in §4.2, using states of the form $(\widehat{L}, \widehat{V}, \widehat{M})$, where 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 value v is meant to be used as a direct value or as a pointer. Specifically, memory map $\widehat{M}$ associates with every memory location $\widehat{\ell} \in \widehat{L}$ a tagged value

- *direct*(v) with $v \in$ *Value*, indicating that $\widehat{M}(\widehat{\ell})$ contains direct value v; or
- *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 give a correspondence between an implementation state and a high-level state. Roughly speaking, we can view deobfuscation relations as refinements of high-level states into implementation states. More precisely, an execution $\sigma \in [\![P]\!]_I^{\tau_{addr}, K}(inps)$ in the implementation semantics of $\tau_{addr}(P, K)$ and an

[9] Removing repetitions is necessary so that the sequence has one element per output produced or input read.

[10] Other high-level semantics are possible, of course, and our framework 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.

execution $\widehat{\sigma} \in [\![P]\!]_H(inps)$ 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[11] $\widehat{\sigma}'$ of $\widehat{\sigma}$, 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 analyzing τ_{addr}, 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.

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

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

(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 $\mathcal{B}_n^{\tau_{addr}}(\cdot)$ of executions for morphs of τ_{addr} using definition (2).

4.6 Type Systems for Toy-C

We can check that Toy-C and τ_{addr} satisfy the premises of Theorem 3.1, so there can be no computable type system for Toy-C that exactly identifies inputs that are resistable attacks

[11] $\widehat{\sigma}'$ 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.

relative to τ_{addr}. It is straightforward to implement a Turing machine simulator in Toy-C, and here is a key classifier for τ_{addr}:

```
classify(ret : *int) {
  var a : int[5];
  *ret := *(&a - 2);
}
```

where ret holds a memory location in which to store the return value. The intuition is that a morph of classify will return an arbitrary value (viz., the result of dereferencing &a − 2). Observe that for every possible integer value v, there is a morph of classify that returns v. (For example, take the infinite set of keys $\mathcal{K} = \{K_1, K_2, \dots\}$, where $K_i = (s, 0, \Pi_0, M_i)$ with s an initial stack position, 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 return the content of memory location ℓ_{RW} (viz., $i - 1$).) It is easy to verify that classify is a key classifier for τ_{addr}. The remaining hypotheses, (2) and (3), of Theorem 3.1 are similarly easy to discharge.

We also saw in §3 that we can approximate an arbitrary obfuscator by considering only finite sets of keys. However, if we consider a particular programming language and a particular obfuscator, we may devise more interesting approximations. In particular, for Toy-C, we examine how strong typing approximates τ_{addr}.

4.6.1 Strong Typing for Toy-C

Obfuscator τ_{addr} is intended to defend against attacks that involve overflowing a buffer to corrupt memory. Thus, to eliminate the vulnerabilities targeted by τ_{addr}, a type system only has to check that a memory read[12] or write through a pointer into a buffer does not access memory outside that buffer.

In Toy-C, there are only two ways in which such a memory access can happen. First, the program can read a value using a pointer that addresses a location outside the extent of a buffer, as in Figure 2(a). (A buffer is simply the area of memory allocated for storing the value of a variable—a single location for an integer or pointer variable, and a sequence of contiguous locations for an array variable.) 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 and perform bounds checks before every dereference. More precisely, values will be represented as pairs $\langle i, \mathbf{int}\rangle$—an *integer value* i—and $\langle i, \mathbf{ptr}(start, end)\rangle$—a *pointer value* i pointing to a buffer starting at address *start* and ending at address *end* [15, 18]. Our type system T^{strg} enforces the following invariant: whenever a pointer value $\langle i, \mathbf{ptr}(start, end)\rangle$ is dereferenced, it must satisfy $start \leq i \leq end$. Information associated with values is tracked and checked during expression evaluation, as follows.

[12]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.

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

(a)

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

(b)

Figure 2: Accessing memory outside a buffer

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

S2. Dereferencing an integer value results in a type error. The result of dereferencing a pointer value $\langle i, \mathbf{ptr}(start, end) \rangle$ 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 $\langle i, \mathbf{ptr}(start, end) \rangle$, 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 $\langle i, \mathbf{ptr}(start, end) \rangle$ and the other an integer value $\langle i', \mathbf{int} \rangle$, the operation returns $\langle i + i', \mathbf{ptr}(start, end) \rangle$.

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

An alternate form of strong typing is to enforce the following, stronger, invariant: every pointer value $\langle i, \mathbf{ptr}(start, end) \rangle$ always satisfies $start \leq i \leq end$. 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. Moreover, morphs created by τ_{addr} will not differ in behavior when executing program (b).

To illustrate type system T^{strg}, consider the program of Figure 2(a) again. Assume that variable a is allocated at memory location ℓ_{a}. To execute $\mathsf{x} := *(\&\mathsf{a} + 10)$, expression $*(\&\mathsf{a}+10)$ is evaluated. 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 $\&\mathsf{a}+10$ 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 $\ell_{\mathsf{a}} + 4$.

```
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

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 according to S1–S5, 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 base semantics $[\![\cdot]\!]_I^{base}$ to track the types of values. Details of the resulting implementation semantics $[\![\cdot]\!]_I^{strg}$ appear in Appendix C.1. One modification to $[\![\cdot]\!]_I^{base}$ 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 4.1. *Let $K_1, \ldots, K_n$ be arbitrary keys for τ_{addr}. For any program P and inputs inps, if inps is a resistable 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 a resistable 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 resistable attacks relative to τ_{addr} and a finite set of keys $K_1, \ldots, K_n$. This supports our thesis about 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 4.1.

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 a resistable attack. When executing the program of Figure 4(a), for instance, T^{strg} signals a type error because &a + 10 yields a pointer that cannot be dereferenced. But there is no resistable attack relative to τ_{addr}

```
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

because morphs created by τ_{addr} do not differ in their behavior according to $\mathcal{B}_n^{\tau_{addr}}(\cdot)$ (see §4.4), since output statement 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 a resistable attack as being an 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} defends against.

4.6.2 A Tighter Type System for 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, then the value computed from that memory access has low integrity. This is enforced with τ_{addr} when 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 only that outputs 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 [19]. That information flow arises when trying to develop more precise type systems corresponding to obfuscator is, in retrospect, not surprising. As Tse and Zdancewic [26] have shown, there is a tight relationship between information-flow type systems and representation independence, where type systems guarantee type safety even though underlying data representations may vary; the kind of obufscation performed by τ_{addr} can be seen as varying data representation.

We adapt T^{strg} and design a new type system T^{info} that takes integrity into account. 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 statement is attempted and that output statement depends on a value with type **low**. In other words, if control reaches an output statement 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 produces an output, 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 that program, 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 having type **low** will prevent the integer from having an observable effect. Value $\langle i, \mathbf{low}\rangle$ is never actually 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 the print statement—either in the then or the else—a type error is signalled because the program counter has type **low**.

Theorem 4.2. *Let $K_1, \ldots, K_n$ be arbitrary keys for τ_{addr}. For any program P and inputs inps, if inps is a resistable 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}, type system T^{info} is a sound approximation of τ_{addr}. Moreover, as illustrated by the programs of Figure 4, T^{info} corresponds more closely to τ_{addr} than does T^{strg}. Information flow therefore captures our definition of resistable attack relative to τ_{addr} more closely than strong typing. But, as we see below, T^{info} still aborts executions on inputs that are not resistable attacks relative to τ_{addr}, so T^{info} is still stronger than τ_{addr}.

One problem is the familiar "label creep" that affects dynamic enforcement mechanisms for information-flow properties. Label creep here refers to the propensity for integrity labels to be coerced downward while upward relabeling is rare and difficult to do correctly. Label creep occurs in our type system when the program counter gets a **low** type because of a conditional branch that depends on a low-integrity value, even though the two branches of the conditional may have exactly the same behavior in every morph.

Another problem with T^{info} is illustrated by 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 statements. However, because equality is

reflexive, the fact that we are comparing to a value with type **low** is completely irrelevant—the guard always yields true. We believe it would be quite difficult to develop a type system[13] 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 could obtain a type system that more closely correspond to τ_{addr}. Theorem 3.1, however, indicates that obtaining a type system equivalent to τ_{addr} would be impossible.

5 Concluding Remarks

There is a relationship between diversity for resisting attacks and diversity for guaranteeing availability [7]. Eckhardt and Lee [12] and Littlewood and Miller [17], for instance, formalize some of the concepts involved in assessing independence of failures, or dually, what they term *coincident failures*, which are failures that affect more than one implementation. That work assumes that the probability that a program fails on a given input is proportional to how much processing is required for that input.

There has been work on probabilistic verification of properties at runtime, a generalization of probabilistic dynamic type systems, such as Zheng *et al*'s [30] work on statistical debugging, or Sammapun *et al.*'s [22] work in the runtime verification community. None of these projects attempt to bridge the gap between probabilistic properties and type systems.

Our work has, however, ignored the probabilities that this other work investigates. For practical applications, these probabilities actually do matter, because if the dynamic type checking works only with low probability, then attacks are likely to succeed. The probabilities, then, are the interesting metric when trying to decide in practice whether mechanically-generated diversity actually is useful. 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. Indeed, it is not simply a matter of defining a measure of how much diversity is introduced—the kind of diversity introduced also affects probabilities, since different attacks will respond to different obfuscating transformations; for instance, renaming opcodes against buffer-overflow attacks. Our framework is thus best seen as only a first step in characterizing the effectiveness of program obfuscation and other mechanically-generated diversity.

Note that a reduction from obfuscation to non-probabilistic type checking—although clearly stronger than the results we give—would not help in characterizing the effectiveness of mechanically-generated diversity. This is because there is (to our knowledge) no non-trivial and complete characterization of the attacks that strong typing repels. Simply enumerating which specific known attacks are blocked and which are not does not give a satisfying basis for characterizing a defense in a world where new kinds of attacks are constantly being devised. We strive for characterizations that are more abstract—a threat model based on the resources or information available to the attacker, for example. In the absence of suitable abstract threat models, reductions from one defense to another, like

[13]There are static analyses, such as constant propagation with conditional branches [27], that achieve some, but not all, of what is needed.

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 focus in this paper on 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 was finding a suitable, albeit unconventional, definition of resistable 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. Thanks as well to Martín Abadi for pointing out the relationship between obfuscation and representation independence.

A A Semantics for Toy-C

A.1 Syntax

Let *Var* be a set of variables, *Proc* be a set of procedure names, and *Outputs* be a set of possible outputs. The syntax of Toy-C is given by the following grammar.

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

A program is a sequence of procedure declarations, where each procedure declaration consists of local variable declarations followed by a sequence of statements. Every program P must define a procedure main, which is the entry point of P; inputs to P are inputs to procedure main.

Procedure parameters and local variable are declared with types, which describe representation of values. General types ty include base types and array types. Base types py

characterize variables whose value fit in a single memory location (viz., integer constants and pointers); variables of array types can refer to values in multiple memory locations. While local variables may have general types, procedure parameters, as in C, must have base types. The following function computes the size of values of a given type:

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

Statements include standard statements of imperative programming languages: assignment, procedure call, conditional, and iteration. We identify outputs with statements that produce those outputs; thus, there is a statement *out* for every output in *Outputs*. Statement fail terminates an execution with an error. Every sequence of statements is terminated by an empty sequence statement ϵ, which we generally omit.

We distinguish VD-expressions ex, which evaluate to values, from AD-expressions lv, which denote memory locations. For simplicity, integers and the null pointer are the only constants. Notation $*ex$ (and $*lv$) dereferences a pointer, while $\&lv$ returns 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);
*y := 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 completely straightforward to add new operations.

A.2 Base Semantics

In order to define base semantics $[\![\cdot]\!]_I^{base}$ for Toy-C, we need to describe the states; in particular, we need to describe locations, 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. Assume the first locations of $\mathbb{N}$ are locations that store program code; 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 $loc(sts)$ in the first ℓ_{RW} locations of $\mathbb{N}$ where the statement sequence *sts* is stored. Conversely, we assume a map *code* that extracts code from memory: $code(\ell)$ returns a sequence of statements *sts*. Functions *loc* and *code* are inverses: $code(loc(sts)) = sts$.

Variable maps are standard. Hidden variables pc, outputs, and inputs are as before. We also have additional hidden variables: sp holds the current stack pointer—that is, the

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 $\surd$) or failed (value $\times$). 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 value 0. Our semantics ensures programs cannot access memory locations below ℓ_{RW}; any read/write to a memory location below ℓ_{RW} is interpreted as a read/write to memory location ℓ_{RW}. (An acceptable 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 to locations, and VD-expressions to values. Evaluation function $A[\![lv]\!](V, M)$ is used to evaluate an AD-expression lv in variable map V and memory map M:

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

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

Evaluation function $E[\![ex]\!](V, M)$ is used to evaluate a VD-expression ex in variable map V and memory map M:

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

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

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

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

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

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

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

Executions comprising semantics $[\![P]\!]_I^{base}(\cdot)$ are constructed using standard reduction rules between *operational states* of the form (sts, V, M, Vs), where V is a variable map, $Vs = \langle V_1, \ldots, V_k \rangle$ is a stack of variable maps, M is a memory map, and sts is a sequence of statements to execute. (Stacks of variable maps record the layers of environment introduced by the block structure of the language.) The reduction rules are given in Figures 5 and 6. A reduction rule of the form $(sts, V, M, Vs) \longrightarrow (sts', V, M', Vs')$, describes one step of the execution of sts. To simplify the description of the semantics, we extend statement sequences with token $\bullet$ to represent termination. We use notation $V\{x \mapsto \langle \ell_1, \ldots, \ell_k \rangle\}$ to represent map V updated so that variable x is mapped to $\langle \ell_1, \ldots, \ell_k \rangle$; similarly for memory map M. We use $+\!\!+$ to denote sequence concatenation in rule (R10).

Base semantics $[\![P]\!]_I^{base}(\cdot)$ proper is defined by extracting a sequence of states from a sequence of reductions starting from an initial operational state that corresponds to invoking procedure main:

$$(\bullet, V, M, \mathit{Vs}) \longrightarrow (\bullet, V, M, \mathit{Vs}) \tag{R1}$$

$$(\epsilon, V, M, \mathit{Vs}) \longrightarrow (\bullet, V\{\mathsf{pc} \mapsto \bullet\}, M, \mathit{Vs}) \tag{R2}$$
if $V(\mathsf{return}) = \bullet$

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

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

$$(\mathsf{if}\ \mathit{ex}\ \mathsf{then}\ \{\ \mathit{sts}_1\ \}\ \mathsf{else}\ \{\ \mathit{sts}_2\ \}; \mathit{sts}, V, M, \mathit{Vs}) \longrightarrow (\mathit{sts}_1; \mathit{sts}, V\{\mathsf{pc} \mapsto \mathit{loc}(\mathit{sts}_1)\}, M, \mathit{Vs}) \tag{R5}$$
if $E[\![\mathit{ex}]\!](V, M) \neq 0$

$$(\mathsf{if}\ \mathit{ex}\ \mathsf{then}\ \{\ \mathit{sts}_1\ \}\ \mathsf{else}\ \{\ \mathit{sts}_2\ \}; \mathit{sts}, V, M, \mathit{Vs}) \longrightarrow (\mathit{sts}_2; \mathit{sts}, V\{\mathsf{pc} \mapsto \mathit{loc}(\mathit{sts}_2)\}, M, \mathit{Vs}) \tag{R6}$$
if $E[\![\mathit{ex}]\!](V, M) = 0$

$$(\mathsf{while}\ \mathit{ex}\ \mathsf{do}\ \{\ \mathit{sts}_1\ \}; \mathit{sts}, V, M, \mathit{Vs}) \longrightarrow (\mathit{sts}_1; \mathsf{while}\ \mathit{ex}\ \mathsf{do}\ \{\ \mathit{sts}_1\ \}; \mathit{sts}, V\{\mathsf{pc} \mapsto \mathit{loc}(\mathit{sts}_1)\}, M, \mathit{Vs}) \tag{R7}$$
if $E[\![\mathit{ex}]\!](V, M) \neq 0$

$$(\mathsf{while}\ \mathit{ex}\ \mathsf{do}\ \{\ \mathit{sts}_1\ \}; \mathit{sts}, V, M, \mathit{Vs}) \longrightarrow (\mathit{sts}, V\{\mathsf{pc} \mapsto \mathit{loc}(\mathit{sts})\}, M, \mathit{Vs}) \tag{R8}$$
if $E[\![\mathit{ex}]\!](V, M) = 0$

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

$$(\mathit{out}; \mathit{sts}, V, M, \mathit{Vs}) \longrightarrow (\mathit{sts}, V', M, \mathit{Vs}) \tag{R10}$$
where $V' \triangleq V\{\mathsf{pc} \mapsto \mathit{loc}(\mathit{sts}),$
$\mathsf{outputs} \mapsto V(\mathsf{outputs}) +\!\!+ \langle \mathit{out}\rangle\}$

Figure 5: Reduction rules for $[\![\cdot]\!]_I^{base}$

$$(m(ex_1,\ldots,ex_n);sts,V,M,\langle V_1,\ldots,V_k\rangle) \longrightarrow (sts_m,V',M',\langle V,V_1,\ldots,V_k\rangle) \qquad \text{(R11)}$$

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

where $V' \triangleq V\{\mathsf{pc} \mapsto loc(sts_m),$

$$\begin{array}{l}
\mathsf{sp} \mapsto V(\mathsf{sp}) + n + \sum_{i=1}^{k} size(ty_i) + 1,\\
\mathsf{return} \mapsto V(\mathsf{sp}) + n,\\
x_1 \mapsto \langle V(\mathsf{sp})\rangle,\\
\vdots\\
x_n \mapsto \langle V(\mathsf{sp}) + n - 1\rangle,\\
y_1 \mapsto \langle V(\mathsf{sp}) + n + 1,\ldots,V(\mathsf{sp}) + n + 1 + size(ty_1) - 1\rangle\\
\vdots\\
y_k \mapsto \langle V(\mathsf{sp}) + n + 1 + \sum_{i=1}^{k-1} size(ty_i),\\
\qquad \ldots,\\
\qquad V(\mathsf{sp}) + n + 1 + \sum_{i=1}^{k-1} size(ty_i) + size(ty_k) - 1\rangle\}
\end{array}$$

$M' \triangleq M\{V(\mathsf{sp}) \mapsto ct_1,$

$$\begin{array}{l}
\vdots\\
V(\mathsf{sp}) + n - 1 \mapsto ct_n,\\
V(\mathsf{sp}) + n \mapsto loc(sts)\\
V(\mathsf{sp}) + n + 1 \mapsto 0,\\
\vdots\\
V(\mathsf{sp}) + n + 1 + \sum_{i=1}^{k-1} size(ty_i) \mapsto 0\}
\end{array}$$

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

Figure 6: Reduction rules for $[\![\cdot]\!]_I^{base}$, continued

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

where

$$\begin{aligned}
Vs_1 &\triangleq \langle\rangle \\
V_1 &\triangleq V_0\{\mathsf{sp} \mapsto \ell_{RW}, \\
&\qquad \mathsf{return} \mapsto \bullet, \\
&\qquad \mathsf{pc} \mapsto loc(\mathsf{main}(ct_1, \ldots, ct_k);), \\
&\qquad \mathsf{inputs} \mapsto \langle\rangle, \\
&\qquad \mathsf{status} \mapsto \surd, \\
&\qquad \mathsf{outputs} \mapsto \langle\rangle\} \\
M_1 &\triangleq M_0 \\
sts_1 &\triangleq \mathsf{main}(ct_1, \ldots, ct_k).
\end{aligned}$$

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

To simplify the presentation of the reduction rules, operational states hold redundant information; for instance, 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 (sts, V, M, Vs) be an operational state reachable from (sts_1, V_1, M_1, 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 $V(\mathsf{pc}) = loc(sts)$.*

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

We can verify that the resulting traces are executions as defined in §4.2.

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

Proof. Let σ be the single infinite trace in $[\![P]\!]_I^{base}(\langle ct_1, \ldots, ct_k \rangle)$, generated by the sequence of reductions $(sts_1, V_1, M_1, Vs_1) \longrightarrow (sts_2, V_2, M_2, Vs_2) \longrightarrow \ldots$. We check the five conditions defining an execution:

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

(2) By Lemma A.1, if $\sigma[i].\mathsf{pc} = \bullet$, then $sts_i = \bullet$. Only rule (R1) can apply from operational state i on, and thus $sts_j = \bullet$ 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 $sts_i = \bullet$. 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, $\sigma[1].\mathsf{outputs} = \langle\rangle$. The only rule that adds outputs to hidden variable $\mathsf{outputs}$ is (R10), which adds a single output at every application of the rule.

(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}

Implementation semantics $[\![P]\!]_I^{\tau,K}$ is obtained by modifying $[\![P]\!]_I^{base}$ using:

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

(2) An initial memory map M_{init};

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

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

Implementation semantics $[\![P]\!]_I^{\tau_{addr},K}$ corresponding to morph $\tau_{addr}(P,K)$ is obtained by modifying base semantics $[\![P]\!]_I^{base}(\cdot)$ in a simple way. To account for (1), we initially set sp to ℓ_s instead of ℓ_{RW}. To account for (2), we initially set M to M_{init}. To account for (3) and (4), we replace the procedure call reduction rule (R11) in Figure 6 by rule (R11*) in Figure 7.

C Type Systems for τ_{addr}

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

Values in implementation semantics $[\![\cdot]\!]_I^{strg}$ have the form $\langle i, t\rangle$, where i is an integer, and t is a type (integer or pointer).

Type checking for T^{strg} occurs when operations $*$, $+$, and $=$ are evaluated (rules S2, S4, and S5 in §4.6.1). Recall, Toy-C supports two kinds of expressions: VD- and AD-expressions, denoting values (integer or pointer) and memory locations, respectively. Base semantics $[\![\cdot]\!]_I^{base}$ (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 following *type-checking evaluation functions* A^s and E^s, which check suitable conditions on values:

$$(m(ex_1,\ldots,ex_n);sts,V,M,\langle V_1,\ldots,V_k\rangle) \longrightarrow (sts_m,V',M',\langle V,V_1,\ldots,V_k\rangle) \qquad \text{(R11*)}$$

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

where $V' \triangleq V\{\mathsf{pc} \mapsto loc(sts_m),$

$$\begin{aligned}
&\mathsf{sp} \mapsto V(\mathsf{sp}) + n + \sum_{i=1}^{k} size(ty_i) + 1 + 4d,\\
&\mathsf{return} \mapsto V(\mathsf{sp}) + n + 2d,\\
&x_{\pi_n(1)} \mapsto \langle V(\mathsf{sp}) + d\rangle,\\
&\vdots\\
&x_{\pi_n(n)} \mapsto \langle V(\mathsf{sp}) + d + n - 1\rangle,\\
&y_{\pi_k(1)} \mapsto \langle V(\mathsf{sp}) + 3d + n + 1,\ldots,V(\mathsf{sp}) + 3d + n + 1 + size(ty_{\pi_k(1)}) - 1\rangle\\
&\vdots\\
&y_{\pi_k(k)} \mapsto \langle V(\mathsf{sp}) + 3d + n + 1 + \sum_{i=1}^{k-1} size(ty_{\pi_k(i)}),\\
&\qquad\qquad \ldots,\\
&\qquad\qquad V(\mathsf{sp}) + 3d + n + 1 + \sum_{i=1}^{k-1} size(ty_{\pi_k(i)}) + size(ty_{\pi_k(k)}) - 1\rangle\}
\end{aligned}$$

$M' \triangleq M\{V(\mathsf{sp}) + d \mapsto ct_{\pi_n(1)},$

$$\begin{aligned}
&\vdots\\
&V(\mathsf{sp}) + d + n - 1 \mapsto ct_{\pi_n(n)},\\
&V(\mathsf{sp}) + 2d + n \mapsto loc(sts)\\
&V(\mathsf{sp}) + 3d + n + 1 \mapsto 0,\\
&\vdots\\
&V(\mathsf{sp}) + 3d + n + 1 + \sum_{i=1}^{k-1} size(ty_{\pi_k(i)}) \mapsto 0\}
\end{aligned}$$

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

Figure 7: Replacement reduction rule for $[\![\cdot]\!]_I^{\tau_{addr},(\ell_s,d,\Pi,M_{init})}$

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

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

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

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

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

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

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

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

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

These functions return either a value or **TE** (indicating a type error).

The semantics proper is obtained from the reduction rules given in Figures 8 and 9. Rules (T4'), (T6'), (T8'), and (T11') are concerned specifically with reporting type errors. Hidden variable status is set to $\surd$ if termination is successful, $\times$ if termination is due to a failure, and **TE** if termination is due to a type error. We say an execution trace *signals a type error* if it terminates with status equal to **TE**.

The semantics $[\![P]\!]_I^{strg}$ is defined as follows:

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

$$(\bullet, V, M, Vs) \longrightarrow (\bullet, V, M, Vs) \tag{T1}$$

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

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

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

$$(lv := ex; sts, V, M, Vs) \longrightarrow (\bullet, V\{\mathsf{pc} \mapsto \bullet, \mathsf{status} \mapsto \mathbf{TE}\}, M, Vs) \tag{T4'}$$
if $E^s[\![ex]\!](V, M) = \mathbf{TE}$ or $A^s[\![lv]\!](V, M) = \mathbf{TE}$

$$(\mathsf{if}\ ex\ \mathsf{then}\ \{\ sts_1\ \}\ \mathsf{else}\ \{\ sts_2\ \}; sts, V, M, Vs) \longrightarrow (sts_1; sts, V\{\mathsf{pc} \mapsto loc(sts_1)\}, M, Vs) \tag{T5}$$
if $E^s[\![ex]\!](V, M) \neq \langle 0, \mathbf{int}\rangle$

$$(\mathsf{if}\ ex\ \mathsf{then}\ \{\ sts_1\ \}\ \mathsf{else}\ \{\ sts_2\ \}; sts, V, M, Vs) \longrightarrow (sts_2; sts, V\{\mathsf{pc} \mapsto loc(sts_2)\}, M, Vs) \tag{T6}$$
if $E^s[\![ex]\!](V, M) = \langle 0, \mathbf{int}\rangle$

$$(\mathsf{if}\ ex\ \mathsf{then}\ \{\ sts_1\ \}\ \mathsf{else}\ \{\ sts_2\ \}; sts, V, M, Vs) \longrightarrow (\bullet, V\{\mathsf{pc} \mapsto \bullet, \mathsf{status} \mapsto \mathbf{TE}\}, M, Vs) \tag{T6'}$$
if $E^s[\![ex]\!](V, M) = \mathbf{TE}$

$$(\mathsf{while}\ ex\ \mathsf{do}\ \{\ sts_1\ \}; sts, V, M, Vs) \longrightarrow (sts_1; \mathsf{while}\ ex\ \mathsf{do}\ \{\ sts_1\ \}; sts, V\{\mathsf{pc} \mapsto loc(sts_1)\}, M, Vs) \tag{T7}$$
if $E^s[\![ex]\!](V, M) \neq \langle 0, - \rangle$

$$(\mathsf{while}\ ex\ \mathsf{do}\ \{\ sts_1\ \}; sts, V, M, Vs) \longrightarrow (sts, V\{\mathsf{pc} \mapsto loc(sts)\}, M, Vs) \tag{T8}$$
if $E^s[\![ex]\!](V, M) = \langle 0, - \rangle$

$$(\mathsf{while}\ ex\ \mathsf{do}\ \{\ sts_1\ \}; sts, V, M, Vs) \longrightarrow (\bullet, V\{\mathsf{pc} \mapsto \bullet, \mathsf{status} \mapsto \mathbf{TE}\}, M, Vs) \tag{T8'}$$
if $E^s[\![ex]\!](V, M) = \mathbf{TE}$

Figure 8: Reduction rules for $[\![\cdot]\!]_I^{strg}$

$$(\mathsf{fail}; sts, V, M, Vs) \longrightarrow (\bullet, V\{\mathsf{pc} \mapsto \bullet, \mathsf{status} \mapsto \times\}, M, Vs) \tag{T9}$$

$$(out; sts, V, M, Vs) \longrightarrow (sts, V', M, Vs) \tag{T10}$$

where $V' \triangleq V\{\mathsf{pc} \mapsto loc(sts),$
$\mathsf{outputs} \mapsto V(\mathsf{outputs}) +\!\!+ \langle out \rangle\}$

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

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

where $V' \triangleq V\{\mathsf{pc} \mapsto loc(sts_m),$

$$\begin{array}{l}
\mathsf{sp} \mapsto V(\mathsf{sp}) + n + \sum_{i=1}^{k} size(ty_i) + 1, \\
\mathsf{return} \mapsto V(\mathsf{sp}) + n, \\
x_1 \mapsto \langle V(\mathsf{sp}) \rangle, \\
\vdots \\
x_n \mapsto \langle V(\mathsf{sp}) + n - 1 \rangle, \\
y_1 \mapsto \langle V(\mathsf{sp}) + n + 1, \ldots, V(\mathsf{sp}) + n + 1 + size(ty_1) - 1 \rangle \\
\vdots \\
y_k \mapsto \langle V(\mathsf{sp}) + n + 1 + \sum_{i=1}^{k-1} size(ty_i), \\
\qquad \ldots, \\
\qquad V(\mathsf{sp}) + n + 1 + \sum_{i=1}^{k-1} size(ty_i) + size(ty_k) - 1 \rangle\}
\end{array}$$

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

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

$$(m(ex_1, \ldots, ex_n); sts, V, M, Vs) \longrightarrow (\bullet, V\{\mathsf{pc} \mapsto \bullet, \mathsf{status} \mapsto \mathbf{TE}\}, M, Vs) \tag{T11'}$$

if $E^s[\![ex_i]\!](V, M) = \mathbf{TE}$ for some i

Figure 9: Reduction rules for $[\![\cdot]\!]_I^{strg}$, continued

where

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

Theorem 4.1. *Let $K_1, \ldots, K_n$ be arbitrary keys for τ_{addr}. For any program P and inputs inps, if inps is a resistable 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 a resistable attack on P relative to τ_{addr} and $K_1, \ldots, K_n$*

Proof. Assume that $\sigma \in [\![P]\!]_I^{strg}(inps)$ does not signal a type error. We need to prove that *inps* is not a resistable attack relative to P and $K_1, \ldots, K_n$. In other words, if σ_i is the single execution in $[\![P]\!]_I^{\tau_{addr}, K_i}(inps)$, we need to prove that $(\sigma_1, \ldots, \sigma_n) \in \mathcal{B}_n^{\tau_{addr}}(P, K_1, \ldots, K_n)$. By definition of $\mathcal{B}_n^{\tau_{addr}}(\cdot)$, 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 $(\mathbb{N}, V, \widehat{M})$, where $\widehat{M}$ is defined by:

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

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 $\ell \in \mathbb{N}$, if ℓ is in the range of some 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 $(\overline{\mathit{sts}_1}, \overline{V_1}, \overline{M_1}, \overline{\mathit{Vs}_1}) \longrightarrow (\overline{\mathit{sts}_2}, \overline{V_2}, \overline{M_2}, \overline{\mathit{Vs}_2}) \longrightarrow \ldots$ generating σ_i for morph $\tau_{addr}(P, K_i)$. Let $(\mathit{sts}_1, V_1, M_1, \mathit{Vs}_1) \longrightarrow (\mathit{sts}_2, V_2, M_2, \mathit{Vs}_2) \longrightarrow \ldots$ be the sequence of reductions generating σ for P. Without loss of generality, we can assume that programs occupy the same locations in both semantics. (If not, we simply map program locations in one semantics to the appropriate program locations in the other 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 function $h((\mathbb{N}, \overline{V_j}, \overline{M_j}), \cdot)$. If the reduction rule applied at step j is any reduction but (R11*) of Figure 7, then take function $h((\mathbb{N}, \overline{V_{j+1}}, \overline{M_{j+1}}), \cdot)$ 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 function $h((\mathbb{N}, \overline{V_{j+1}}, \overline{M_{j+1}}), \cdot)$ 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. (It is easy to see that such a relation can be found.) We use induction to show that for all $j \geq 1$:

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

(ii) for all ex,

- if $E^s[\![ex]\!](V_j, M_J) = \langle i', \mathbf{int}\rangle$, then $E[\![ex]\!](\overline{V_j}, \overline{M_j}) = i'$;
- if $E^s[\![ex]\!](V_j, M_j) = \langle i', \mathbf{ptr}(-,-)\rangle$, 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 initial states $\sigma[1]$ and $\sigma_i[1]$ are the same, up to the types associated with the values in memory, meaning that states $\widehat{\sigma}[1]$ and $\sigma_i[1]$ are also the same, up to the tagging of the values required by high-level semantics $[\![\cdot]\!]_H$.

For the inductive case, assume the result holds for j; we show it for $j+1$. Establishing (i), (ii), and (iii) is straightforward. To establish (iv), we need to establish:

(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) $\overline{V_{j+1}}(\mathsf{outputs}) = V_{j+1}(\mathsf{outputs})$;

(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 $E^s[\![ex_k]\!](V_j, M_j) = \langle i_k, t_k \rangle$ 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 $[\![\cdot]\!]_I^{info}$

Implementation semantics $[\![\cdot]\!]_I^{info}$ extends $[\![\cdot]\!]_I^{strg}$ by adding a new type, **low**. To implement type checking for T^{info}, we replace evaluation functions A and E in semantics $[\![\cdot]\!]_I^{base}$ by the following type-checking evaluation functions A^i and E^i, which check suitable conditions on values:

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

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

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

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

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

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

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

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

$$\text{where } E^i[\![ex_1]\!](V,M) = \langle i_1, t_1\rangle \quad E^i[\![ex_2]\!](V,M) = \langle i_2, t_2\rangle$$

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

$$\text{where } E^i[\![ex_1]\!](V,M) = \langle i_1, t_1\rangle \quad E^i[\![ex_2]\!](V,M) = \langle i_2, t_2\rangle.$$

These functions return either a value or **TE** (indicating a type error).

Reduction rules for T^{info} are given in Figures 10 and 11. The main difference from the reduction rules in §C.1 is that type checking is only performed in rules (T4') and (T10'), when integrity of a value can influence a variable or control flow. (Hidden variables pc and return also carry a type, to account for a value influencing control flow.) To simplify the presentation of the semantics, we define an operation $t_1 \downarrow t_2$ 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 at least one of t_1 or t_2 is **low**.

Hidden variable status is set to $\surd$ if termination is successful, $\times$ if termination is due to a failure, and **TE** if termination is due to a type error. As in §C.1, we say an execution trace *signals a type error* if it terminates with status equal to **TE**.

The semantics $[\![P]\!]_I^{info}$ is defined as follows:

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

where

$$\begin{aligned} Vs_1 &\triangleq \langle\rangle \\ V_1 &\triangleq V_0\{\mathsf{sp} \mapsto \ell_{RW}, \\ &\qquad \mathsf{return} \mapsto \bullet, \\ &\qquad \mathsf{pc} \mapsto \langle loc(\mathsf{main}(ct_1, \ldots, ct_k);), \mathbf{int}\rangle, \\ &\qquad \mathsf{inputs} \mapsto \langle\rangle, \\ &\qquad \mathsf{status} \mapsto \surd, \\ &\qquad \mathsf{outputs} \mapsto \langle\rangle\} \end{aligned}$$

$$(\bullet, V, M, Vs) \longrightarrow (\bullet, V, M, Vs) \tag{T1}$$

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

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

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

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

$$(\mathsf{if}\ ex\ \mathsf{then}\ \{\ sts_1\ \}\ \mathsf{else}\ \{\ sts_2\ \}; sts, V, M, Vs) \longrightarrow (sts_1; sts, V\{\mathsf{pc} \mapsto \langle loc(sts_1), t \downarrow t'\rangle\}, M, Vs) \tag{T5}$$
if $E^i[\![ex]\!](V, M) = \langle i, t\rangle$, $i \neq 0$, and $V(\mathsf{pc}) = \langle -, t'\rangle$

$$(\mathsf{if}\ ex\ \mathsf{then}\ \{\ sts_1\ \}\ \mathsf{else}\ \{\ sts_2\ \}; sts, V, M, Vs) \longrightarrow (sts_2; sts, V\{\mathsf{pc} \mapsto \langle loc(sts_2), t \downarrow t'\rangle\}, M, Vs) \tag{T6}$$
if $E^i[\![ex]\!](V, M) = \langle 0, t\rangle$ and $V(\mathsf{pc}) = \langle -, t'\rangle$

$$(\mathsf{while}\ ex\ \mathsf{do}\ \{\ sts_1\ \}; sts, V, M, Vs) \longrightarrow (sts_1; \mathsf{while}\ ex\ \mathsf{do}\ \{\ sts_1\ \}; sts, V\{\mathsf{pc} \mapsto \langle loc(sts_1), t \downarrow t'\rangle\}, M, Vs) \tag{T7}$$
if $E^i[\![ex]\!](V, M) = \langle i, t\rangle$, $i \neq 0$, and $V(\mathsf{pc}) = \langle -, t'\rangle$

$$(\mathsf{while}\ ex\ \mathsf{do}\ \{\ sts_1\ \}; sts, V, M, Vs) \longrightarrow (sts, V\{\mathsf{pc} \mapsto \langle loc(sts), t \downarrow t'\rangle\}, M, Vs) \tag{T8}$$
if $E^i[\![ex]\!](V, M) = \langle 0, t\rangle$ and $V(\mathsf{pc}) = \langle -, t'\rangle$

Figure 10: Reduction rules for $[\![\cdot]\!]_I^{info}$

$$(\mathsf{fail}; sts, V, M, Vs) \longrightarrow (\bullet, V\{\mathsf{pc} \mapsto \bullet, \mathsf{status} \mapsto \times\}, M, Vs) \tag{T9}$$

$$(out; sts, V, M, Vs) \longrightarrow (sts, V', M, Vs) \tag{T10}$$
if $V(\mathsf{pc}) = \langle \ell, t \rangle$ and $t \neq \mathbf{low}$
where $V' \triangleq V\{\mathsf{pc} \mapsto \langle loc(sts), t \rangle,$
$\mathsf{outputs} \mapsto V(\mathsf{outputs}) +\!\!+ \langle out \rangle\}$

$$(out; sts, V, M, Vs) \longrightarrow (\bullet, V\{\mathsf{pc} \mapsto \bullet, \mathsf{status} \mapsto \mathbf{TE}\}, M, Vs) \tag{T10'}$$
if $V(\mathsf{pc}) = \langle \ell, \mathbf{low} \rangle$

$$(m(ex_1, \ldots, ex_n); sts, V, M, \langle V_1, \ldots, V_k \rangle) \longrightarrow (sts_m, V', M', \langle V, V_1, \ldots, V_k \rangle) \tag{T11}$$
if $E^i[\![ex_i]\!](V, M) = v_i$ for all i
where $V(\mathsf{pc}) = \langle -, t \rangle$

$$\begin{aligned}
V' \triangleq V\{&\mathsf{pc} \mapsto \langle loc(sts_m), t \rangle,\\
&\mathsf{sp} \mapsto V(\mathsf{sp}) + n + \sum_{i=1}^{k} size(ty_i) + 1,\\
&\mathsf{return} \mapsto V(\mathsf{sp}) + n,\\
&x_1 \mapsto \langle V(\mathsf{sp}) \rangle,\\
&\vdots\\
&x_n \mapsto \langle V(\mathsf{sp}) + n1 \rangle,\\
&y_1 \mapsto \langle V(\mathsf{sp}) + n + 1, \ldots, V(\mathsf{sp}) + n + 1 + size(ty_1) - 1 \rangle\\
&\vdots\\
&y_k \mapsto \langle V(\mathsf{sp}) + n + 1 + \sum_{i=1}^{k-1} size(ty_i),\\
&\qquad \ldots,\\
&\qquad V(\mathsf{sp}) + n + 1 + \sum_{i=1}^{k-1} size(ty_i) + size(ty_k) - 1 \rangle\}
\end{aligned}$$

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

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

Figure 11: Reduction rules for $[\![\cdot]\!]_I^{info}$, continued

$$M_1 \triangleq M_0$$
$$sts_1 \triangleq \mathsf{main}(ct_1, \ldots, ct_k).$$

Lemma C.1. *Let* $(sts1, V_1, M_1, Vs_1) \longrightarrow (sts_2, V_2, M_2, Vs_2) \longrightarrow \ldots$ *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. This is a straightforward induction on the length of the reduction sequence. ∎

Theorem 4.2. *Let* $K_1, \ldots, K_n$ *be arbitrary keys for* τ_{addr}*. For any program* P *and inputs* $inps$*, if* $inps$ *is a resistable 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. This proof has the same structure as that of Theorem 4.1.

Assume that $\sigma \in [\![P]\!]_I^{strg}(inps)$ does not signal a type error. We need to prove that $inps$ is not a resistable attack relative to P and $K_1, \ldots, K_n$. In other words, if σ_i is the single execution in $[\![P]\!]_I^{\tau_{addr}, K_i}(inps)$, we need to prove that $(\sigma_1, \ldots, \sigma_n) \in \mathcal{B}_n^{\tau_{addr}}(P, K_1, \ldots, K_n)$. By definition of $\mathcal{B}_n^{\tau_{addr}}(\cdot)$, 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 $(\mathbb{N}, V, \widehat{M})$, where $\widehat{M}$ is defined 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 $\ell \in \mathbb{N}$, if ℓ is in the range of some 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 for morph $\tau_{addr}(P, K_i)$. 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 for P. As in the proof of Theorem 4.1, we assume without loss of generality that programs occupy 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 function $h((\mathbb{N}, \overline{V_j}, \overline{M_j}), \cdot)$. If the reduction rule applied at step j is any reduction but (R11*) of Figure 7, then take function $h((\mathbb{N}, \overline{V_{j+1}}, \overline{M_{j+1}}), \cdot)$ 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 function

$h((\mathbb{N}, \overline{V_{j+1}}, \overline{M_{j+1}}), \cdot)$ 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. (It is easy to see that such a relation can be found.) 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 $E^i[\![ex]\!](V_j, M_J) = \langle i', \mathbf{int}\rangle$, then $E[\![ex]\!](\overline{V_j}, \overline{M_j}) = i'$;
- if $E^i[\![ex]\!](V_j, M_j) = \langle i', \mathbf{ptr}(-,-)\rangle$, 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 initial states $\sigma[1]$ and $\sigma_i[1]$ are the same, up to the types associated with the values in memory, meaning that states $\widehat{\sigma}[1]$ and $\sigma_i[1]$ are also the same, up to the tagging of the values required by high-level semantics $[\![\cdot]\!]_H$.

For the inductive case, assume the result holds 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.1.) 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, $V_{j+1}(\mathsf{outputs}) = V_j(\mathsf{outputs})$, 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 establish:

(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) $\overline{V_{j+1}}(\mathsf{outputs}) = V_{j+1}(\mathsf{outputs})$;

(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 $E^i[\![ex_k]\!](V_j, M_j) = \langle i_k, t_k\rangle$ 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 $\sigma_i[j+1] \precsim \widehat{\sigma}[j+1]$. ∎

D Summary of Notation

P	Program
$inps$	Inputs
$\tau(P, K)$	Morph of program P with key K under obfuscator τ
$[\![P]\!]_I(inps)$	Implementation semantics of P with input $inps$ (generic)
$[\![P]\!]_I^{\tau,K}(inps)$	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 $\tau(P, K_1), \ldots, \tau(P, K_n)$
KC_τ	Key classifier for obfuscator τ
$T^{mrph}_{K_1,\ldots,K_n}$	Exact type system corresponding to $K_1, \ldots, K_n$
$[\![P]\!]_I^{mrph,K_1,\ldots,K_n}(inps)$	Implementation semantics for $T^{mrph}_{K_1,\ldots,K_n}$
T^{rand}	Randomized exact type system
L	Memory locations
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
$[\![P]\!]_H(inps)$	High-level semantics of program P on input $inps$
$[\![P]\!]_I^{base}(inps)$	Base semantics of Toy-C program P on input $inps$
τ_{addr}	Address obfuscator for Toy-C
T^{strg}	Toy-C type system for strong typing
$[\![P]\!]_I^{strg}(inps)$	Implementation semantics for T^{strg}
T^{info}	Toy-C type system for integrity-based typing
$[\![P]\!]_I^{info}(inps)$	Implementation semantics for T^{info}

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. In *Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 158–168. ACM Press, 2006.

[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] P. G. Bishop. Software fault tolerance by design diversity. In M. Lyu, editor, *Software Fault Tolerance*, chapter 9. John Wiley & Sons, 1995.

[8] L. Chen and A. Avizienis. *N*-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 25th International Symposium on Fault-Tolerant Computing*, pages 113–119. IEEE Computer Science Press, 1995.

[9] 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.

[10] 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.

[11] 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.

[12] D. E. Eckhardt and L. D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, 11(12):1511–1517, 1985.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] B. Littlewood and D. R. Miller. Conceptual modeling of coincident failures in multiversion software. *IEEE Transactions on Software Engineering*, 15(12):1596–1614, 1989.

[18] 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.

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

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

[21] H. Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[22] U. Sammapun, I. Lee, O. Sokolsky, and J. Regehr. Statistical runtime checking of probabilistic properties. In *Proc. Runtime Verification (RV'07)*, pages 164–175, 2007.

[23] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[24] 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.

[25] 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.

[26] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *Proc. 9th ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, pages 115–125. ACM Press, 2004.

[27] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.

[28] Y. Weiss and E. G. Barrantes. Known/chosen key attacks against software instruction set randomization. In *Proc. 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 349–360. IEEE Computer Society, 2006.

[29] 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.

[30] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *Proc. 23rd International Conference on Machine Learning (ICML'06)*, pages 1105–1112, 2006.