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

Abstract

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

1. Introduction

Computers that execute the same program risk being vulnerable to the same attacks. This explains why the Internet, whose machines typically have much software in common, is so susceptible to malware. It is also a reason that replication of servers does not necessarily enhance the availability of a service in the presence of attacks—geographically-separated or not, server replicas, by definition, will all exhibit the same vulnerabilities and thus are unlikely to exhibit the independence required for 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, because attacks typically depend on memory layout and/or instruction sequence specifics. But building multiple distinct versions of a program is expensive, so researchers have turned to mechanical means for creating diverse sets of replicas.

Various approaches have been proposed, including relocation and/or padding the run-time stack by random amounts [10, 6, 21], rearranging basic blocks and code within basic blocks [10], randomly changing the names of system calls [7] or instruction opcodes [13, 4, 3], and randomizing the heap memory allocator [5]. Some of these approaches are more effective than others. For example, Sacham *et al.* [18] derive experimental limits on the address space randomization scheme proposed by Xu *et al.* [21], while Sovarel *et al.* [19] 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 to accomplish such transformations have been called *obfuscators* [8]. An obfuscator τ takes two inputs—a program P and a secret key K—and produces a *morph* $\tau(P, K)$, which is a program whose semantics is equivalent to P but whose implementation differs. Secret key K prescribes which exact transformations are applied in producing $\tau(P, K)$. Note that since P and τ is assumed to be public, knowledge of K would enable an attacker to learn implementation details for

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

morph $\tau(P, K)$ and perhaps even automate the generation of attacks for different morphs.

Barak *et al.* [2] and Goldwasser and Kalai [11] give theoretical limits on the effectiveness of obfuscators as a way to keep secret the details of an algorithm or its embodiment as a program. This work, however, says 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 this correspondence is important when designing a set of defenses for a given threat model, knowing the specific correspondences is not the same as knowing the overall power of mechanically-generated diversity as a defense. This paper explores that latter, broader, issue, by

- giving the definitions needed for proving results about the defensive power of obfuscation;
- giving a precise characterization of attacks, applicable to viewing diversity as a defense;
- developing the thesis that mechanically-generated diversity is comparable to type systems, and deriving an admittedly unusual type system equivalent to obfuscation in the presence of finitely many keys;
- exhibiting, for a C-like language and an obfuscator that rearranges memory, an increasingly tighter sequence of type systems for soundly approximating that obfuscator. The most accurate type system is based on information flow. We also show that no type system corresponds exactly to the obfuscator, and therefore approximations are the best we can achieve.

All proofs, as well as detailed semantics for the language and type systems we describe in the text, can be found in a companion technical report [17].

2. Attacks and Obfuscators: A Framework

We assume 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. A program's *behavior* defines a sequence of inputs and outputs.

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

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

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

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 $\tau(P, K_i)$ and that cannot be produced by some other morph $\tau(P, K_j)$—presumably because implementation details differ from morph to morph.[1] When morphs are deterministic, the definition of a resistable attack simplifies to being an input that produces different behaviors in some pair of morphs $\tau(P, K_i)$ and $\tau(P, K_j)$.

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

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

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

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

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. We do this below for a particular language, Toy-C.

When morphs are deterministic programs, and thus σ_i is a unique execution of morph $\tau(P, K_i)$ on input $inps$, then by definition $inps$ is 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 $(\sigma_1, \ldots, \sigma_n) \notin \mathcal{B}_n^\tau(P, K_1, \ldots, K_n)$.

$\mathcal{B}_n^\tau(\cdot)$ cannot in general be an equivalence relation on executions because our notion of 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. Execution Equivalence for C-like Languages

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

- L is the set of memory locations;
- V is a *variable map*, which associates relevant information with every variable. For variables available to programs, V associates the memory locations where the content of the variable is stored; and for variables used to model program execution, V associates information such as sequences of 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 manipulated by programmers explicitly, and each program variable is bound to a finite not necessarily contiguous sequence $\langle \ell_1, \ldots, \ell_k \rangle$ of memory locations in L:

- If $k = 1$, then the variable holds a single value; 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 not directly accessible to the programmer, being artifacts of the language implementation and execution environment. For our purposes, it suffices to assume that the following hidden variables exist:

- `pc` records the memory location of the next instruction to execute; it is always bound to an element of $L \cup \{\bullet\}$, where $\bullet$ indicates the program terminated;
- `outputs` records the finite sequence of outputs that 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, being capable of interpretation as a data value or as a memory location. This ambiguity reflects an unfortunate reality of system implementation languages, such as C, that do not distinguish between integers and pointers.

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

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

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

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

(4) $\sigma[1].\mathsf{outputs} = \langle\rangle$ 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; in other words, 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; in other words, input values only get consumed, and at most one input is consumed at every execution step.

Equivalence of Executions. The formal definition of $\mathcal{B}^{\tau}_n(P, K_1, \ldots, K_n)$ for a C-like language is based on relating executions of morphs to executions in a suitably chosen *high-level semantics* of the original program. 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 the inputs hidden variables and removing repetitions in the resulting sequence.[3] Such a relation for a family of obfuscations and a C-like language is given in §4.

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

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

4. Concrete Example: The Toy-C Language

4.1. The Language

In order to give a concrete example of how to use our framework to reason about diversity and attacks, we introduce a toy C-like language, Toy-C. The syntax and operational semantics of Toy-C programs should be self-explanatory. We only outline the language here, giving complete details in a companion technical report [17, App. 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. Observable variables are specified on a per-procedure basis. Whether a variable is observable does not affect execution of a program; the annotation

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

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

```
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

is used only for determining equivalence of executions (see §4.2).

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

Reference Semantics. Toy-C program execution is described by a *reference semantics* $[\![\cdot]\!]_I^{ref}$, which we use as a basis for other semantics defined in subsequent sections. Full details of the reference semantics appear in a companion technical report [17, App. A.2].

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

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

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

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

Let us consider a threat model in which attackers are allowed to invoke programs and supply inputs. These inputs are used as arguments to the main procedure of the program. For example, consider the program of Figure 1. According to reference semantics $[\![\cdot]\!]_I^{ref}$, on input 0,1, or 2, the program terminates in a final state where ret is bound to a memory location containing the integer 99. However, on input -1, the program terminates in a final state where ret is bound to a memory location containing the integer 42; the input -1 makes the variable tmp point to the memory location bound to variable ret, which (according to reference

semantics $[\![\cdot]\!]_I^{ref}$) precedes buf on the stack, so that the assignment $*\mathsf{tmp} := 42$ stores 42 in the location associated with ret. Presumably, this behavior is undesirable, and input -1 ought to be considered an attack.

4.2. 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.[4]

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

An implementation semantics $[\![P]\!]_I^{\tau_{addr}, K}$ specifying how to execute morph $\tau(P, K)$ is obtained by modifying reference semantics $[\![P]\!]_I^{ref}$ to take into account the transformations 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 $v_1, \dots, v_n$ are arguments to the procedure, then they are pushed in order $v_{\pi_n(1)}, \dots, v_{\pi_n(n)}$;
- 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 a companion technical report [17, App. B].

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

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

A high-level semantics $[\![\cdot]\!]_H$ that serves our purpose is a variant of reference semantics $[\![\cdot]\!]_I^{ref}$, but where 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 a value intended to be interpreted literally—for instance, an integer representing some count. In contrast, a pointer is intended to be interpreted as a stand-in for the value stored at the memory location pointed to; the actual memory location given by a pointer is typically irrelevant.[5]

Executions in high-level semantics $[\![\cdot]\!]_H$ are similar to executions described in §3, using states of the form $(\widehat{L}, \widehat{V}, \widehat{M})$, where 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, a 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 rationalize an implementation state in terms of a high-level state. More precisely, an execution $\sigma \in [\![P]\!]_I^{\tau_{addr}, K}(inps)$ in the implementation semantics of $\tau_{addr}(P, K)$ and an 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

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

[5] 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.

we describe below) such that for some stuttered sequence[6] $\widehat{\sigma}'$ 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 $\widehat{V}(\mathsf{pc}) = h((L, V, M), V(\mathsf{pc}))$;

(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 of executions for morphs using definition (2).

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

5. Obfuscation and Type Systems

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 was declared to accommodate.

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

(1) Assigning an inappropriate value to some variable.

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

There is no need to worry about (1) with Toy-C, because the same values serve both as integers and addresses. Thus, our focus here is on (2).

Eliminating vulnerabilities is clearly preferable to having them be difficult to exploit. So why bother with obfuscation? The answer is that strong type systems are not always an option with legacy code. The relative success of recent work [12, 14] in adding strong typing to languages like C not withstanding, obfuscation is applicable to any object code, independent of what 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.

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

We start our comparison by discussing how the kind of strong typing being advocated with programming languages, such as Java, compares to what can be achieved with obfuscation. To be concrete, we show that strong typing for Toy-C does eliminate all vulnerabilities targeted by

τ_{addr}, but strong typing also signals type errors for programs and inputs that are not considered resistable attacks relative to τ_{addr}. This discrepancy prompts us to investigate how to weaken strong typing to capture more accurately what τ_{addr} accomplishes for Toy-C.

All of the type systems we study are *dynamic* type systems—extra information is associated with values, and this information is checked during execution. When the check detects a type error, execution is halted. Admittedly, this is a very general notion of type system. It encompasses all type systems in the literature, but also includes approaches that are not typically viewed as type systems.

A type system, admittedly unusual, that 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 is the trivial type system, $T^{mrph}_{K_1,\ldots,K_n}$, instantiated by an implementation semantics $[\![P]\!]_I^{mrph,K_1,\ldots,K_n}(inps)$ that repeatedly runs all morphs in parallel, taking unanimous consensus before performing an observable action:

> Execute program P up to the next output statement (including updates to observable program variables), and also 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 P to produce its output, and repeats the procedure;
> - If not, then the type system signals a type error and aborts execution.

Theorem 5.1. *Let $K_1, \ldots, K_n$ be arbitrary keys for τ. For any program P and inputs $inps$, $inps$ is a resistable attack on P relative to τ and $K_1, \ldots, K_n$ if and only if $\sigma \in [\![P]\!]_I^{mrph,K_1,\ldots,K_n}(inps)$ signals a type error.*

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

5.1. Exact Type Systems for τ_{addr}

As defined in §4.2, obfuscator τ_{addr} admits infinitely many keys. Although for many applications we care only about a finite set of keys at any given time (e.g., when using morphs to implement server replicas, of which there are only finitely many), the exact set of keys might not be known in advance or may change during the lifetime of the application. Therefore, it is sensible to try to identify inputs that are resistable attacks relative to τ_{addr} and any finite subset of the possible keys, or equivalently, to recognize inputs that are not resistable attacks relative to τ_{addr} and every finite subset of the possible keys.

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

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

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

As we now show, it is impossible to design a type system that aborts executions for exactly those inputs for which there exists a finite set $K_1,\ldots,K_n$ of keys and the input is a resistable attack relative to τ_{addr} and $K_1,\ldots,K_n$.

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

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

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

```
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

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

Theorem 5.2. *Let $[\![\cdot]\!]_I$ be an implementation semantics for Toy-C such that:*

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

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

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

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

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

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

5.2. Strong Typing for Toy-C

Obfuscator τ_{addr} is intended to defend against attacks that involve accessing memory outside the extent of a buffer. Thus, to eliminate the vulnerabilities targeted by τ_{addr}, a type system only has to check that a memory read[7] or write through a pointer into a buffer allocated to a variable does not access memory outside that buffer.

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

To put strong typing into Toy-C, we associate information with values manipulated by programs. More precisely, values will be represented as pairs $\langle i, \mathbf{int}\rangle$—an *integer value* i—and $\langle i, \mathbf{ptr}(start, end)\rangle$—a *pointer value* i pointing to a buffer starting at address $start$ and ending at address end [12, 14]. Our type system[8] 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.

S1. The representation of a 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 $\langle i + i', \mathbf{ptr}(start, end)\rangle$.

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

[8] An alternate form of strong typing is to enforce the following, stronger, invariant: every pointer value $\langle i, \mathbf{ptr}(start, end)\rangle$ 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 &a + 10: it evaluates to a pointer value outside its allowed range (viz., extent of a). But although signalling a type error seems reasonable for program (a) in Figure 3, it seems inappropriate with program (b) because this problematic pointer value is never actually used. Note that morphs created by τ_{addr} will not differ in behavior when executing program (b).

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

(a)

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

(b)

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

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

To formalize how Toy-C programs execute under type system T^{strg}, we extend reference semantics $[\![\cdot]\!]_I^{ref}$ to track the types of values. Details of the resulting implementation semantics $[\![\cdot]\!]_I^{strg}$ appears in a companion technical report [17, App. C.1]. One modification to $[\![\cdot]\!]_I^{ref}$ is that $[\![\cdot]\!]_I^{strg}$ uses values of the form $\langle i, t\rangle$, where i is an integer and t is a type, as described above.

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

Theorem 5.3. *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 $K_1, \ldots, K_n$.*

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 further supports our thesis that there is a connection between type systems and obfuscation. Moreover, any type system that is more restrictive than T^{strg} and therefore causes more executions to signal a type error will also have the property given in Theorem 5.3.

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(c), 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} because morphs created by τ_{addr} do not differ in their behavior, since output statement $\mathsf{print}(0)$ is always going to be executed.

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

(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);
  print(0);
}
```

(c)

Figure 4. Sample programs

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

5.3. 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 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 [15].

We adapt T^{strg} and design a new type system T^{info} that takes integrity into account. Roughly speaking, a new type **low** is associated with any value having low integrity. Rather than signalling a type error when dereferencing a pointer to a memory location that lies outside its range, the type of the value extracted from the memory location is set to **low**. The resulting implementation semantics $[\![P]\!]_I^{info}$ appears in a companion technical report [17, App. 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(c). 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 it having type **low** will prevent the integer from having an observable effect. The value $\langle i, \mathbf{low}\rangle$ is never used in the rest of the program, so execution proceeds without signalling a type error (in contrast to T^{strg}, which does signal a type error).

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

Theorem 5.4. *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.*

Thus, just like T^{strg}, T^{info} is a sound approximation of τ_{addr}. 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}.

Consider the program of Figure 4(a). 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, as the guard always yields true. We believe that it would be quite difficult to develop a type system[9] that can identify guards that are validities, because doing so requires a way to decide when two expressions have the same value in all executions. Yet, if we had a more precise way to establish the integrity of the program counter (for instance, by being able to establish that two expressions affecting control flow have the same value in all executions), then we would have a type system that more closely correspond to τ_{addr}. The results of §5.1, however, indicate that this is impossible.

6. Concluding Remarks

This paper gives a reduction from defenses created by mechanically-generated diversity to probabilistic dynamic type checking. But we have ignored the probabilities. For practical application, these probabilities actually do matter, because if the dynamic type checking is performed with low probability, then checks are frequently skipped and attacks are likely to succeed. The probabilities, then, are the 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. Our framework is thus best seen as only a first step in characterizing the effectiveness of program obfuscation and other mechanically-introduced diversity.

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, either. This is because there is (to our knowledge) no non-trivial and complete characterization of the attacks that strong typing repels. Simply enumerating which known attacks are blocked and which are not does not give a satisfying basis for characterizing a defense in a world where new attacks are constantly being perpetrated. We should strive for characterizations that are more abstract—a threat model based on the

[9] There are static analyses, such as constant propagation with conditional branches [20], that achieve some, but not all, of what is needed.

resources or information available to the attacker, for example. In the absence of suitable abstract threat models, reductions from one defense to another, like what is being introduced in this paper, might well be the only way to get insight into the relative powers of defenses. Moreover, such reductions remain valuable even after suitable threat models have been developed.

We 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.

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. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. 10th ACM Conference on Computer and Communications Security (CCS'03)*, pages 281–289. ACM Press, 2003.

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

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

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

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

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

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

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

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

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

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

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

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

[17] R. Pucella and F. B. Schneider. Independence from obfuscation: A semantic framework for diversity. Technical Report 2006-2016, Computer Science Department, Cornell University, 2006.

[18] 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.

[19] 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.

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

[21] 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.