Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Adding Native Support for Havoc in Viper

Practical Work

Daniel Zhang

September 9, 2022

Advisors: Prof. Dr. Peter Müller, Dr. Malte Schwerhoff

Department of Computer Science, ETH Zürich

Abstract

To *havoc* a memory region means to assign it nondeterministic values. This notion is especially common when describing the semantics of concurrent code, e.g. to simulate a thread modifying a shared region to an unknown value. Viper is an Intermediate Verification Language with several front-ends, some of which verify concurrent code. However, Viper has no native havoc statement. Instead, developers encode a havoc using a sequence of existing statements. Unfortunately this encoding is overly complex, and has become a performance bottleneck for some front-ends. In this project, we explore adding havoc as a native feature to Viper, including a "havocall" version which operates on an unbounded set of regions. Finally, we demonstrate a performance improvement over the existing encoding.

Contents

Chapter 1

Introduction

Program verification has become an area of renewed interest due recent technological developments and its applications to safety-critical software. As a result, various program verification languages have arisen. These languages cover an assortment of domains, from verifying smart contracts to assembly language. However, in all of these situations, building the entire verification framework, from a high-level source encoding down to the verification conditions, is a painstaking task.

To ease their development, many verification languages instead compile to an *intermediate verification language* (IVL). The IVL will transform the input into *verification conditions*, which can be processed by an SMT solver, such as Z3. The verification result is then translated back up the toolchain. An IVL should have simple, predictable behavior, providing only a core set of features. On the other hand, it should share enough similarities with its front-ends, e.g. by allowing memory accesses and control-flow patterns, to facilitate an easy translation.

One such IVL is Viper, developed by the Programming Methodology Group at ETH Zürich [6]. Viper distinguishes itself from other IVLs (e.g. Why3 [1] and Boogie [4]) due to its permission model, based on a variant of separation logic [7]. In this logic, the heap is viewed as a collection of distinct *resources*, each of which has a *memory footprint* that may satisfy some invariant. This logic is useful for describing heap-dependent data structures, especially in concurrent settings. For this reason, Viper is an appealing target language for applications that verify multithreaded code.

Another technique for specifying concurrent programs is *rely-guarantee reasoning* [3]. This reasoning provides tools for specifying shared memory regions which may be manipulated by multiple threads, subject to constraints. These constraints may take the form of invariants on the region, or restrictions on acceptable state transitions. Combined with separation logic, these two tools allow users to specify complex properties involving atomic synchronization primitives.

In order to encode rely-guarantee reasoning, it is helpful to have a notion of "modifying a memory region to an unknown value". For example, this could be used to model a thread assigning unknown values to a shared region. This concept is referred to as *havocking* a memory region. Even better, we would like to know that the memory region still satisfies its invariant despite the other thread's modification. A natural extension of havocking a single memory region is havocking an unbounded set of memory regions, each of which satisfies some constraint.

It is possible to encode this operation using existing Viper primitives, specifically `exhale` and `inhale` statements. However, this encoding is circuitous — it achieves this behavior by operating on *permission amounts* instead of *snapshots*, a more direct and conceptually simple approach (see Section 3.1). As a consequence, its behavior is much slower than it could be. Specifically, its performance is a bottleneck for verifying programs written in Voila, a front-end for Viper that specializes in fine-grained concurrency [11]. As a temporary fix, a "havoc hack" was added to Viper. This improved performance significantly, and motivated the development of a more flexible, long-term solution.

The goal of this project is to add two new statements into Viper: `havoc` and `havocall`. These two statements will havoc a single resource and an unbounded set of resources, respectively. This project includes an implementation in Silicon, one of Viper's two backends. This implementation should handle all resources, specifically field access, predicates, magic wands, and their quantified versions. These two statements should outperform `exhale` and `inhale`, providing a more efficient encoding for Viper front-ends and allowing us to retire the "havoc hack".

The contents of this report is as follows. First we provide the necessary background on Viper. This includes a brief introduction to the language, its permission model, and its implementation with Silicon. Next, we discuss how to incorporate havoc into Viper. This involves summarizing the design goals of havoc and how they influence the symbolic execution rules in Silicon. We conclude the report with a discussion of the implementation, and we analyze performance improvements.

Chapter 2

Background

This section provides the background on Viper and Silicon needed to introduce havoc. First, we provide a language overview of Viper, including its permission model. We then discuss its implementation in Silicon, one of Viper's back-ends. In particular, we discuss the program state, and we introduce some symbolic execution helper functions.

2.1 Viper Language Overview

As mentioned earlier, Viper is an IVL that provides native support for a variation of separation logic [7]. Several front-ends that compile to Viper have already been developed. For example, Gobra, Prusti, and Nagini verify code written in Go, Rust, and Python, respectively. Viper has two back-ends: Carbon and Silicon. Carbon compiles Viper code into Boogie, a simpler IVL. Then, the Boogie program is verified via translation to Z3, and the results are reported back up the tool chain. On the other hand, Silicon symbolically executes Viper code. While stepping through each code path, Silicon builds up a list of verification conditions, and dispatches them directly to Z3. We do not mention Carbon for the remainder of this report, as this project's goal was focused on Silicon.

Viper shares similarities with other imperative languages. Programs are divided into several top-level declarations, including *methods* and *functions*. Method bodies contain a sequence of statements, while function bodies consist of a single (side-effect-free) expression. Both methods and functions have a contract, specifying the pre- and post-conditions. Viper has a simple, static type system. `Bools` and `Ints` represent Booleans and mathematical integers, respectively. In addition, the types `Set[T]` and `Seq[T]` denote the aggregate types of mathematical sets and sequences, respectively. Objects on the heap all have the type `Ref` (short for "reference"). Viper has no notion of classes. All fields are declared at the top level and apply to any referencetyped variable. The user can allocate a field for any reference using a `new` statement.

2.2 Viper's Permission Model

In this section, we provide an introduction into Viper's permission model. At any given point, the program state consists of a set of *resources*. Each resource has an underlying *memory footprint* — this consists of the set of memory locations that make up the resource. As a simple example, the memory at the location `x.f` forms the footprint of this *field resource*. It is also possible to combine field resources into *predicates*, which have larger footprints. For example, the predicate `Pair(x)` might be defined to contain both the `f` and `g` fields of `x`. More details follow shortly.

In order to access the memory in a resource's footprint, the programmer must have the appropriate *resource permission*. Given full permission to a resource, the programmer is guaranteed to have exclusive access to its memory footprint, and therefore the footprint is disjoint from every other resource's footprint. This allows the programmer to modify memory without worrying that it will alias with another resource's footprint.

Viper provides two primitive statements for manipulating the amount of permission to a resource. The `inhale` statement adds a resource to the heap. `inhale` statements must be used with caution, as they can cause unsoundness. For example, inhaling full permission to the same resource twice would violate the disjointness property. The `exhale` statement asserts that we have access to the resource, and then removes it from the heap. Any fact about this resource's memory is forgotten. These primitives can be combined to represent the exchange of resources, e.g. from a caller to callee, or between threads.

There are three basic kinds of resources: fields, predicates, and magic wands. We provide a brief introduction on each.

Fields *Field resources* allow users to read or write to a reference's field. To assert full permission to a field resource, a programmer uses the keyword `acc`, as in `acc(e.f)`. (It is possible to have *partial* permission to a resource, as explained later in this section.) `acc` is an *accessibility predicate*, and is Viper's version of separation logic's points-to predicate [7]. As mentioned above, full permission to a resource implies exclusive access. Thus, if we simultaneously have `acc(x.f)` and `acc(y.f)`, then they have different footprints, and we can conclude `x != y`.

Predicates *Predicates resources* combine field resources with logical connectives to create larger structures. Like fields, users can denote access to pred-

```
field f: Int
field g: Int
predicate Pair(x: Ref) {
    acc(x.f) && acc(x.g) && x.f < x.g
}

method foo(x: Ref)
{
    inhale acc(x.f)
    inhale acc(x.g)
    x.f := 1
    x.g := 2
    fold Pair(x)
    // ... code that does not modify Pair(x) ...
    unfold Pair(x)
    assert x.f < x.g    // guaranteed by predicate body
    assert x.f == 1     // information that is preserved.
}
```

Figure 2.1: A simple example of Viper with a predicate. First, we inhale two field resources. We assign 1 and 2 to them so that they satisfy the constraints in `Pair`. This allows us to fold the predicate, replacing the two field resources with `Pair(x)`. When we unfold the predicate, we replace `Pair(x)` with the two field resources. We further learn that `x.f < x.g`, as this is guaranteed from the predicate body. Finally, we can still assert that `x.f == 1`, as this was preserved across the fold-unfold pair.

icates using the `acc` keyword. In addition, if the user has access to a predicate resource, they can use the `unfold` statement. Internally, this exhales the predicate instance, and inhales the predicate body instantiated with its arguments. The `fold` statement works in the opposite direction — given the predicate body, the fold statement exchanges it for the predicate resource. One key feature of this logic is that information is preserved across fold-unfold pairs. An example is shown in Figure 2.1, where information about `Pair(x)` is preserved after it is folded and unfolded. This feature is achieved via *predicate snapshots*, which we discuss in Section 2.3.

Magic Wands Magic wands take the form `A --* B`, where `A` and `B` contain resources and logical connectives. If the user has `A` and the wand `A --* B`, they can exchange both for `B`. Wands represent separation logic's *separating implication* [8]. A thorough discussion of magic wands is beyond the scope of this report. However, they share many similarities with predicates. As suggested above, just like predicates, wands can be exchanged with other resources — instead of using `fold` and `unfold`, we use `package` and `apply`. Likewise, it is possible to preserve information across a `package-apply` pair using a *magic wand snapshot*. Because their behavior is so similar in the context of this report, for the remainder of this paper we only discuss predicates specifically, pointing out the differences with wands when applicable.

Fractional Permissions We can extend the syntax `acc(e.f)` by allowing `acc(e.f, p)`. The expression `p` has type `Perm` and denotes a *permission amount*. When `p` is 1 (also depicted with the keyword `write`), we have full permission to `x.f` — this is equivalent to `acc(x.f)`. If `p` is 0 (also depicted by `none`), we have no permission to the resource — this is equivalent to it being absent from our heap. For other values of `p` between 0 and 1 exclusive, we have a *fractional* permission amount. This provides enough permission to read, but not enough to modify the memory. Holding a permission amount greater than 1 to a memory location is unsound. Permission amounts can be split and combined using usual arithmetic rules. As a consequence, if we have read-only access to a resource, nowhere else can there exist full permission to the same resource. Fractional permissions extend to predicates with the syntax `acc(P(e), p)`. However, they do not extend to magic wands (yet).

Quantified Permissions Predicates are useful for representing data structures with recursive formats. However, they struggle to encode other data structures, such as arrays or graphs, with no inherent access patterns. To solve this problem, Viper uses *quantified permissions*, which correspond to separation logic's *iterated separating conjunction* [5]. Viper uses quantifiers to express permissions to an unbounded number of resources. Their syntax is

```
forall x: T :: c(x) ==> R(x)
```

where T is the type of x (the quantified variable), $c(x)$ is a Boolean expression, and $R(x)$ is any of the above resources. This permission indicates that we have access to any resource $R(x)$ which satisfies the condition $c(x)$. Viper supports quantifying over multiple variables, but to simplify their presentation in this report, we only include the case with one quantifier.

2.3 Implementation in Silicon

This section describes how Viper's permission model is realized in Silicon. Before discussing the encoding, we introduce the concept of a *symbolic value*. Then we discuss non-quantified permissions, before extending their implementation to quantified permissions. Next, we describe Silicon's program state. Finally, we introduce two of Silicon's symbolic execution primitives, and we describe how they manipulate the state.

Symbolic Values We use *symbolic values* when describing the semantics of a Viper program. Let V be the type of symbolic values. The symbolic execution rules operate on symbolic values, and so many program types and expressions have symbolic counterparts. For example, every program variable `v` $\in$ *Var* has a corresponding symbolic value $v \in V$. As in the preceding sentence, we use fonts to distinguish between program variables and symbolic values. The program types `Int` and `Bool` have symbolic types *Int*, *Bool* $\subset V$. In addition, all binary operators and functions have corresponding symbolic functions which operate on symbolic values. In particular, if-then-else expressions, whose concrete syntax in Viper is `c ? x : y`, are represented with the symbolic expression $ite(c, x, y)$.

We use the function `fresh` to introduce a fresh symbolic value, i.e. a symbolic value about which we have no knowledge. The type of the value is clear from context.

Heap Chunks Silicon represents the heap as a set of *heap chunks*. The category of heap chunks is subdivided into predicate chunks, field chunks, and magic wand chunks. Each heap chunk corresponds to a resource that we have access to.

Predicate chunks have the shape $id(\bar{r};\, s, p)$. The string *id* uniquely identifies the predicate type. Items to the left of the semicolon are in-arguments to the predicate. Throughout this report, we use an overline to denote multiple instances. Here, $\bar{r}$ represents the symbolic values of the arguments to *id*. The value p denotes the permission amount. Field chunks can be represented using the same syntax as predicate chunks. For example, field chunk corresponding to `e.f` is $f(e;\, s, p)$. Therefore, for the remainder of the report, we use “predicate chunk” to refer to field chunks as well.

Finally, s represents the *snapshot* of the predicate chunk. The snapshot represents the values in the resource’s memory footprint. Each snapshot has the type $Snap \subset V$. For field chunks, the snapshot contains the symbolic value at the field location. For other predicate chunks, snapshots represent trees which mirror the predicate body — the leaves of a snapshot tree contain field chunk snapshots. The exact details of how predicate snapshots are constructed is not relevant for this report (see [9], Section 3.1.2). The key takeaway is that snapshots represent the values in the heap chunk’s footprint. If a snapshot is replaced with a fresh one, the verifier loses information about its underlying memory.

Consider the brief example in Figure 2.2. The permissions `acc(x.f)` and `acc(x.g)` are represented by the chunks $f(x;\, s_1, 1)$ and $g(x;\, s_2, 1)$, respectively. The variable assignment on line 9 introduces the fact $s_1 = 42$ to our knowledge base. On line 10, we exchange our two chunks for the chunk $P(x;\, s, 1)$. Furthermore, we learn a fact about s that aggregates information about s_1 and s_2 (specifically, we learn $s = Pair(s_1, s_2)$ — no relation to the

```
field f: Int
field g: Int
predicate P(x: Ref) {
    acc(x.f) && acc(x.g)
}
method foo(x: Ref)
    requires acc(x.f) && acc(x.g)
{
    x.f := 42
    fold P(x)
}
```

Figure 2.2: A simple example where we exchange two field chunks for a predicate chunk

`Pair` predicate from Figure 2.1). The knowledge $s_1 = 42$ persists. Thus if `P(x)` were unfolded, we would retain the knowledge that `x.f == 42`.

Quantified Heap Chunks Quantified resources represent an unbounded number of resources and therefore have a different heap chunk representation. Consider the quantified permission.

`forall` x`:` T `::` $c(x)$ `==>` `acc(`$id(\overline{e(x)})$`,` $p(x)$`)`

Its quantified heap chunk has the shape $id(\bar{r}; sm, pm)$. As before, $\overline{r : E}$ are the arguments to the predicate (or field) id. Then, $sm : \overline{E} \rightarrow Snap$ is a *snapshot map*. The value at $sm(\bar{r})$ is the snapshot of the predicate instance $id(\bar{r})$. Next, $pm : \overline{E} \rightarrow Perm$ is a *permission map*. Analogous to snapshot maps, the value at $pm(\bar{r})$ contains the permission amount of $id(\bar{r})$. If the condition $c(x)$ is not satisfied, then the permission amount is 0.

Observe that pm maps $\overline{E} \rightarrow Perm$. However, the condition $c(x)$ is expressed in terms of $x \in T$. With this construction, it is not clear how to check that a predicate instances $id(\bar{r})$ satisfies the condition $c(x)$. In order for this to be possible, the argument expressions $\overline{e(x)}$ must satisfy an *injectivity property*. We go into further detail when discussing `havocall` in Section 3.4.2, which is subject to the same constraint.

Program State Now that we know how to encode heap chunks, we can describe how Silicon represents the program state. We provide formal definitions that are compatible with [9]. The program state has type Σ, with typical representative σ. The state contains several components — here we list only the relevant ones.

- A store $\gamma : Var \rightarrow V$. This maps programmatic variables Var to symbolic values V. The store is updated, e.g. after a variable assignment, or when we enter a new scope.

- A heap $h : H$. The heap contains a set of heap chunks.
- A *path conditions stack* $\pi : \Pi$. This contains a set of *path conditions*, i.e. Boolean expressions that must be true in the current execution path. The stack representation makes it easier to handle backtracking, e.g. when executing if statements, and is not relevant for this report.

A path condition $v \in V$ can be added to a path condition stack π with the helper function `pc-add`. This function has signature $\Pi \rightarrow V \rightarrow \Pi$. The final (return) argument is the new path condition stack, with v added. The implementation can be found in Figure 3.3 of [9]. We use this function when formulating havoc's execution rules in Section 3.4.1.

Symbolic Execution We now show Silicon processes program statements using symbolic execution. The execution primitives that we will introduce both use *continuation-passing style* (CPS) [2]. This paradigm is useful for representing unusual control-flow patterns, such as branching and backtracking. It provides no benefit for this report's contribution (namely `havoc` and `havocall`). However, we describe all functions using CPS to maintain consistency with [9].

Let S and E be the types of program statements and expressions, respectively. Furthermore, let R be the type of *verification results*, in particular `Success` or `Failure`. We introduce two symbolic execution primitives:

$$\begin{aligned}\texttt{exec}\colon & \Sigma \rightarrow S \rightarrow (\Sigma \rightarrow R) \rightarrow R\\ \texttt{eval}\colon & \Sigma \rightarrow E \rightarrow (\Sigma \rightarrow V \rightarrow R) \rightarrow R\end{aligned}$$

The function `exec` is used to process a statement $s \in S$. Doing so will likely modify the program state. For example, a variable assignment might update the store $\sigma.\gamma$, or an inhale statement might add a heap chunk to $\sigma.h$. Therefore, a new state $\sigma' \in \Sigma$ is computed. This state is passed to the continuation $Q : \Sigma \rightarrow R$, which is the final (non-return) argument to `exec` (as is canonical in CPS). Eventually, this continuation will yield a verification result, which we immediately return.

As a helper function, `exec` often calls `eval`. This function evaluates an expression into a symbolic value $v \in V$. As an example, evaluating the local variable `x` is simply the map look-up $\sigma.\gamma[\texttt{x}]$. In general, an expression evaluation may modify the state. Therefore, the new state and symbolic value are passed on to the continuation $Q : \Sigma \rightarrow V \rightarrow R$.

Chapter 3

Adding Havoc to Viper

In this section, we describe how havoc works as a native feature in Viper. First, we discuss issues with its encoding via exhale and inhale as motivation for introducing the new statements. Then, we introduce the syntax of the `havoc` and `havocall` statements. We then outline the semantics, with special consideration for when the heap contains fractional permissions. We conclude this section by showing the symbolic execution rules for havoc statements in Silicon.

3.1 Havocking with Exhale and Inhale

As mentioned in the introduction, it is possible to encode havoc via exhale and inhale. Now that we are familiar with how Silicon represents the heap, we provide more detail on this encoding and explain its shortcomings.

Suppose we want to havoc the resource `e.f`. An equivalent Viper encoding would be

```
label L
var p: Perm := perm(e.f)
exhale acc(e.f, p)
inhale acc(old[L](e.f), p)
```

We briefly introduce the new syntax. The expression `perm(R)` computes the amount of permission `p` to the resource `R`. The `label` statement allows users to refer to the heap at an earlier state. Let h_L denote the heap at the label `L`. Then, the expression `old[L](e)` evaluates the expression `e` in h_L. It is possible to extend this encoding to handle cases where (1) we havoc only under a certain condition or (2) we havoc a quantified resource.

We now show how these statements have the same behavior as havoc, starting with simple cases. During an exhale statement, Silicon scans through the heaping, looking for a matching chunk with enough permission, i.e. a

```
1 method foo(e1: Ref, e2: Ref)
2     requires acc(e1.f, 1/3) && acc(e2.f, 2/3)
3 {
4     // encode havoc e2.f:
5     var p: Perm := perm(e2.f)
6     exhale acc(e2.f, p)
7     inhale acc(e2.f, p)
8 }
```

Figure 3.1: A program that Silicon's default exhale semantics fails to verify. At the start of the function, we have two heap chunks. However, we don't know if `e1` and `e2` alias each other. Therefore, p might contain a full permission. Neither heap chunk definitively has at least p permissions, so the exhale fails.

chunk $f(r; s, q)$ where $r = e$ and $q \geq p$. It then removes p permissions, leaving the chunk with $q - p$ permissions remaining. Since p is defined to contain all the permission to `e.f`, we remove the entire heap chunk. In the inhale statement, we add a new heap chunk $f(r; s', p)$, where s' is a fresh snapshot. Since s' has no relation to s, all the original facts about the footprint are lost. Notice that the net effect of these statements is simply to replace s with a fresh snapshot s'. However, we achieved this result with exhale and inhale, which operate on permission amounts.

Although this implementation is sound, havocking would fail in several situations. Suppose permissions are split as fractions across multiple heap chunks. For example, our heap might contain the two chunks $f(e; s, 1/3)$ and $f(e; s, 2/3)$. With the above definition, the exhale statement would look for a heap chunk with permission $p = 1$, and it would fail. To solve this problem, Silicon performs *state consolidations* (see [9], Section 3.4.2). At certain points, Silicon identifies chunks that alias each other and consolidates them into a single chunk. Once these chunks are consolidated, the exhale succeeds.

However, the aliasing information may be ambiguous at the time of the exhale. Consider the example in Figure 3.1. At the function start, our heap contains the chunks $f(e_1; s_1, 1/3)$ and $f(e_2; s_2, 2/3)$. We are unsure if $e_1 = e_2$, and so p could be as high as 1. Thus, line 6 fails because neither heap chunk is a suitable candidate. State consolidation does not help because Silicon cannot prove that the chunks alias each other.

This situation can be remedied by adjusting the exhale algorithm as follows. We iterate over all heap chunks with the resource f. One by one, we remove permissions from chunks until we have accrued all p permissions. To handle the above issues, the permission amount might depend on aliasing facts. This method is much more nuanced because at each point, we have several considerations. If the resource is disjoint, we should remove no permissions. We should only remove permissions as needed, i.e. no more than p in total. Moreover, for any given heap chunk, we cannot remove so much permission as to yield a negative permission amount.

Observe this modified exhale algorithm on the above example. We first consider $f(e_1; s_1, 1/3)$. If $e_1 = e_2$, then these resources alias, and we should remove permissions. On the other hand, if e_1 and e_2 are disjoint, then we should skip this resource. This results in the new heap chunk $f(e_1; s_1, p'_1)$, where $p'_1 = ite(e_1 = e_2, \max(0, p - 1/3), 1/3)$.

The complication propagates to the next heap chunk, $f(e_2; s_2, 2/3)$. How much permission should we remove from this heap chunk? It depends on how much was consumed by the previous one. The remaining permission that needs to be exhaled is $p_{left} := p - p'_1$. Thus, we replace this chunk with a new one $f(e_2; s_2, p'_2)$, where $p'_2 = ite(e_2 = e_2, \max(0, p_{left} - 2/3), 2/3)$. In this case, the if-then-else expression trivially simplifies to $p_{left} - 2/3$.

As you can see, using these semantics for exhale incurs a significant time penalty. For this reason, it is not the default behavior in Silicon — it must be enabled using the flag `enableMoreCompleteExhale`. There are two sources of slowness. First of all, this implementation mixes lots of casework with permission arithmetic. In addition, this implementation does not scale well. The amount of permission that we remove from one heap chunk depends on how much we removed from the previous ones. Thus, as we have more and more heap chunks, the execution slows down super-linearly. This is confirmed with experimental results in Section 4. The slowness is even more evident in the context of quantified permissions.

Recall our original goal for encoding havoc: we want to replace the heap chunk $f(r; s, p)$ with $f(r; s', p)$, where s' is fresh. With exhale and inhale, we achieved this end, albeit with a complicated and indirect approach. It would be simpler and more efficient to just replace the snapshot s with s'. In other words, we should operate on the snapshot directly, rather than achieve our means by manipulating the permissions. This motivates the introduction of a new statement: `havoc`.

3.2 Syntax

Havoc statements take the following form:

```
havoc c ==> R
```

Here, c is a Boolean expression which we refer to as the *havoc condition*. The havoc condition can be syntactically omitted, in which case it is `true`. Then, the resource R is referred to as the *havocked resource*. This resource is only havocked if the havoc condition is satisfied. It can be any non-quantified resource, i.e. a field, predicate, or a magic wand. Note that havoc only acts on snapshot values and has no effect on permission amounts (unlike inhale and exhale). For this reason, surrounding the havocked resource with `acc` is not valid syntax.

To havoc an unbounded number of resources, we introduce the `havocall` statement. It has similar syntax but includes a quantified variable:

```
havocall x: T :: c(x) ==> R(x)
```

Both the condition and the resource can depend on the quantified variable. `havocall` with multiple quantified variables is allowed. However, it is not conceptually more challenging than `havocall` with one quantifier, so for the remainder of the report, we simplify their presentation by only considering one quantified variable.

3.3 Semantic Properties of Havocking Resources

Suppose we encounter the statement `havoc x.f`. If our heap contains the field chunk $f(x; s, \texttt{write})$, then our task is simple — we must replace s with a fresh snapshot. However, we must address the same concerns as in Figure 3.1. There could be several field chunks with fractional permissions that all correspond to `x.f`. We may have a field chunk $f(y; s, p)$, where x and y alias.

These possibilities lead to the following procedure. First, we gather the field chunks with the shape $f(r;\, s, p)$. Instead of directly replacing s with a fresh snapshot, we must handle the case where r and x alias. For this reason, we replace s the snapshot $s' := ite(x = y, \texttt{fresh}, s)$. Note that this term is not eagerly evaluated. Therefore, we handle situations like 3.2 properly. The heap chunk's permission value remains the same.

Havocking predicates behaves exactly the same as havocking fields. When a predicate instance's snapshot is replaced, we lose all information about the predicate's footprint. However, the predicate instance can still be unfolded. When the unfold occurs, we may learn information about the predicate's memory footprint mandated from the predicate definition. However, all other facts about the footprint are lost. For example, consider the program in Figure 3.3. After the havoc statement, we lose information that `x.f` contains the value 2. However, when we unfold, we still know that `x.f` satisfies the predicate's body.

Havocking Resources with Fractional Permissions

Previously, we motivated the havoc statement with the need to "change a memory region to an unknown value". When our heap contains a full

```
1  field f: Int
2  method foo(x: Ref, y: Ref)
3      requires acc(x.f) && x.f == 42
4  {
5      havoc y.f
6      assume x != y
7      assert x.f == 42
8  }
```

Figure 3.2: On line 5, we have no information about the reference `y`. In particular, we don't know if `y` is just an alias of `x`. Therefore, we cannot assert `x.f == 42` immediately after line 5. After line 6 we learn that `x` and `y` are different references, so the havoc statement has no effect on `x.f`. (In fact, it has no affect on the program state at all, since we have no heap chunks that match the havoc statement.) Therefore, we can assert `x.f == 42`. This example shows that havoc must not eagerly replace snapshots

```
1  field f: Int
2  predicate positive(x: Ref) {acc(x.f) && x.f > 0}
3  method foo(x: Ref)
4      requires acc(x.f) && x.f == 2
5  {
6      fold positive(x)
7      havoc positive(x)
8      unfold positive(x)
9      assert x.f > 0
10 }
```

Figure 3.3: Havocking a predicate. After the havoc statement, we lose information about the body of the predicate. Therefore, on line 9, we cannot assert that `x.f` is 2. However, from the predicate body definition, we know that it is still positive.

permission to the havocked resource, this interpretation is exactly correct. However, the behavior is more subtle when our heap contains only a fractional permission to the havocked resource. In this case, we do not have full permissions, so we cannot modify the resource's memory. Even though we replace the snapshot with a fresh one, this does not constitute a nondeterministic assignment.

Consider the example in Figure 3.4. At the start of the method, the heap contains a full permission to `x.f`. However, a fractional permission is hidden inside the predicate `P(x)`. When we havoc `x.f`, only the snapshot for the remaining fraction is replaced. Thus, by unfolding `P(x)`, Viper can relearn the facts about `x.f` that were havocked. In this situation, it would be incorrect to describe havoc as "assigning a nondeterministic value to `x.f`",

```
 1 field f: Int
 2 predicate P(x: Ref) {acc(x.f, 1/3)}
 3 method HavocFrac(x: Ref)
 4     requires acc(x.f) && x.f == 42
 5 {
 6     fold P(x)
 7     havoc x.f
 8     unfold P(x)
 9     assert x.f == 42
10 }
```

Figure 3.4: Havocking with fractional permissions. On line 6, we fold 1/3 of the permission into the predicate `P`. At this point, the heap contains `acc(P(x))` and `acc(x.f, 2/3)`. When we havoc `x.f` on line 7, only 2/3 of its permissions are available — the other 1/3 is stored in `P(x)`. To execute the havoc, we replace the snapshot of `acc(x.f, 2/3)` with a fresh snapshot. Therefore, immediately after line 7, it would be impossible to assert `x.f == 42`. However, we then unfold `P(x)` to reveal our 1/3 permission of `acc(x.f)`. This permission is combined with the 2/3 permission, and the information about the snapshot is consolidated. We relearn that `x.f == 42`. This examples shows that it is possible to retain information about a resource when only a fraction has been havocked.

since we didn't have full permissions in the first place.

This behavior might seem unintuitive. It would be possible to implement havoc such that it fails unless a full permission is havocked. However, the chosen definition matches existing constructs in Viper. For example, the user might think that the statement

```
exhale acc(x.f, perm(x.f))
```

removes all accesses to `x.f`. However, a fraction of `x.f` could be similarly hidden inside of a predicate. In addition, adding this restriction would require permission math, which would undercut our potential performance gains.

3.4 Symbolic Execution Rules

In this section, we introduce the execution rules for `havoc` and `havocall`. We begin with the simplest case, where we havoc a resource and the heap contains only non-quantified resources. Next, we address `havocall` statements with non-quantified resources. Finally, we introduce the execution rules for both statements in the context of quantified resources.

3.4.1 Havoc with Non-Quantified Permissions

We begin with the execution rules for havocking non-quantified resources. We provide pseudocode for three functions in Figure 3.5. Some of these helper function will be reused with `havocall` and quantified resources.

The function `exec` was introduced in Section 2.3 and is used to execute all Viper statements — we add case for havoc in Figure 3.5 line 2. First, we evaluate all involved expressions into terms. Next, we call the helper function `execHavoc`. To this function, we must provide an extra *data* argument. This argument provides extra information needed to calculate the snapshot replacement condition (more on this later). The exact contents of the *data* argument differs between the implementations of `havoc` and `havocall`. Therefore, we give it the type *HavocData*, defined by the following tagged union:

$$HavocData = \texttt{HavocOneData}\ V \mid \texttt{HavocAllData}\ V$$

In this case, we use the `HavocOneData` constructor. The function `execHavoc` calculates the new state, which contains our newly havocked heap chunks. We then call the continuation on this new state.

The function `execHavoc` replaces snapshots for the relevant resources. First, we gather the heap chunks of type *id* — these are the only ones which could be affected by the `havoc` statement. We process each of these heap chunks individually. We use the subroutine `replacementCond` to provide the *replacement condition* b, which determines under which condition the heap chunk should be havocked. In this case, `replacementCond` checks whether the arguments to the havocked resource are equal to the heap chunk, and if c' is true. For example, if we encounter the statement `havoc` c `==>` $P(e_1, e_2)$ and our heap contains the heap chunk $P(r_1, r_2; s, p)$, then the replacement condition is $b := (c' \wedge e'_1 = r_1 \wedge e'_2 = r_2)$. We emphasize that b is not evaluated eagerly — it is merely a symbolic expression.

We then construct a new snapshot s'. If b holds, then the heap chunk is havocked, and so s' will equal a fresh snapshot. Otherwise, it points to the original snapshot. After processing all the relevant heap chunks, we return the new state.

3.4.2 Havocall with Non-Quantified Permissions

Before discussing the execution rules for `havocall`, we must mention the injectivity requirement. Consider the generic form of `havocall`:

```
havocall x: T :: c(x) ==> P(e(x)).
```

where arguments of P have type $\overline{E}$. Let $\overline{e(x)}'$ be the symbolic expressions corresponding to the arguments $\overline{e(x)}$. Let $e^\star : T \to \overline{E}$ be the function that

```
1:  exec:  $\Sigma \to S \to (\Sigma \to R) \to R$
2:  exec($\sigma_1$,havoc $c$ ==> $id(\overline{e})$,$Q$) =
3:      eval($\sigma_1$,$c :: \overline{e}$,$(\lambda\, \sigma_2, c' :: \overline{e'} \cdot$
4:          $\sigma_2 :=$ execHavoc($\sigma_2$,$id$,$c'$,HavocOneData($\overline{e'}$))
5:          $Q(\sigma_2)))$
6:
7:  execHavoc:  $\Sigma \to Id \to V \to HavocData \to \Sigma$
8:  execHavoc($\sigma$,$id$,$c'$,$data$) =
9:      Let $h_{id} \subseteq \sigma.h$ be all the chunks with identifier $id$
10:     $h'_{id} := \emptyset$
11:     foreach $id(\overline{r};\, s, p) \in h_{id}$ do
12:         $b :=$ replacementCond($c'$, $\overline{r}$, $data$)
13:         $s' := ite(b,$fresh()$,s)$
14:         $h'_{id} := h'_{id} \cup \{id(\overline{r};\, s', p)\}$
15:     $\sigma\{h := (\sigma.h \setminus h_{id}) \cup h'_{id}\}$
16:
17: replacementCond:  $V \to \overline{V} \to HavocData \to V$
18: replacementCond($c'$,$\overline{r}$,HavocOneData($\overline{e'}$)) =
19:      $c' \wedge \overline{e' = r}$
```

Figure 3.5: Execution rules for havocking a non-quantified resource

maps x to $\overline{e(x)'}$. We require that $e^\star$ is injective. The justification is very similar to the reasoning for quantified permissions, which we summarize here. Suppose we have a heap chunk $P(\overline{r}; s, p)$. Under what situations should we replace s with a fresh snapshot? We should havoc s if there exists some y such that the conditions $c(y)'$ and $\bigwedge \overline{e(y)' = r}$ both hold. However, evaluating such existentials is very difficult for SMT solvers. We therefore impose the restriction that $e^\star$ is an invertible function, with inverse $e^{-1} : \overline{E} \to T$. With this condition, to determine if we should havoc the snapshot, it suffices to check $c(e^{-1}(\overline{r}))$.

We are now ready to discuss the execution rule for `havocall`, shown in Figure 3.6. First, we must evaluate all the relevant expressions. This is complicated since the expressions could contain quantified variables, so we cannot simply pass them to `eval`. Instead, we employ the same technique used to evaluate quantified permissions in [9], Section 4.2.2. Essentially, we leverage the execution rule for evaluating `forall` expressions. Then, we pattern match on the result to extract the condition and argument terms. The function $\mathcal{D}$ is a dummy function whose only purpose is to allow us to process the predicate arguments. This takes place on lines 2–3.

Lines 4–11 all relate to the injectivity requirement. First, we assert $e^\star$ is actu-

```
exec($\sigma_1$,havocall $x : T :: c(x)$ ==> $id(\overline{e(x)}), Q$) =
    eval($\sigma_1$,forall $x : T :: c(x)$ ==> $\mathcal{D}(\overline{e(x)})$,
          $(\lambda\, \sigma_2, (\forall x : T \cdot c(x)' \Rightarrow \mathcal{D}'(\overline{e(x)'})) \cdot$
        Let $y_1, y_2$ be fresh symbolic constants of type $T$
        assert($\sigma_2$,$c(y_1)' \wedge c(y_2)' \wedge \overline{e(y_1)' = e(y_2)'}$
                    $\Rightarrow y_1 = y_2$)
        Let $img_e$ be a fresh function of type $\overline{E} \to Bool$
        $img_{def} := \forall x : T \cdot c(x)' \Rightarrow img_e(\overline{e(x)'})$
        Let $e^{-1}$ be a fresh function of type $\overline{E} \to T$
        $inv_1 := \forall \overline{r : E} \cdot img_e(\overline{r}) \wedge c(e^{-1}(\overline{r}))' \Rightarrow \overline{\bigwedge e_i(e^{-1}(\overline{r}))' = r_i}$
        $inv_2 := \forall x : T \cdot c(x)' \Rightarrow e^{-1}(\overline{e(x)'}) = x$
        $\sigma_3 :=$ execHavoc($\sigma_2$,$id$,$c(x)'$,HavocAllData($e^{-1}$))
        $\pi_3 :=$ pc-add($\sigma_3.\pi$,$\{inv_1, inv_2, img_{def}\}$)
        $Q(\sigma_3\,\{\pi := \pi_3\}))$)

replacementCond($c(x)'$,$\overline{r}$,HavocAllData($e^{-1}$)) =
    $c(e^{-1}(\overline{r}))'$
```

Figure 3.6: Execution rules for havocall with non-quantified resources

ally injective. Then, we must ensure that e^{-1} only receives arguments within its domain. Therefore, we define a function img_e, which detects if a value is in the domain of e^{-1}, i.e. the range of $e^\star$. This function is axiomatized with img_{def}. We then define a fresh inverse function e^{-1} and axiomatize it appropriately. We must pass e^{-1} to `execHavoc`, as it is used to construct the replacement condition.

As before, `execHavoc` iterates through all the relevant heap chunks, replacing them with a fresh snapshot if the replacement condition holds. The replacement condition is calculated in `replacementCond`. Finally, we add the injectivity axioms to our path conditions and continue the computation.

3.4.3 Incorporating Quantified Permissions

We now consider `havoc` and `havocall` with quantified permissions. In general, mixing quantified permissions and non-quantified permissions is challenging. Silicon simplifies this by making the following design decision: either the heap consists entirely of quantified permissions or of non-quantified permissions. If the programmer mixes the two, Silicon performs a simple translation step from non-quantified to quantified permissions. Therefore, we only need to provide a new execution rule if quantified permissions are present.

```
1: execHavocQP($\sigma, id, c', data$) =
2:     Let $h_{id} \subseteq \sigma.h$ be all the chunks with identifier $id$
3:     $h'_{id} := \emptyset$
4:     $\pi' := \sigma.\pi$
5:     foreach $id(\bar{r}; sm(\bar{r}), p(\bar{r})) \in h_{id}$ do
6:         $b(\bar{r}) =$ replacementCond($c', \bar{r}, data$)
7:         Let $sm'$ be a fresh snapshot map of type $\overline{E} \to Snap$
8:         $sm'_{def} := \forall \overline{r : E} \cdot \neg b(\bar{r}) \Rightarrow sm(\bar{r}) = sm'(\bar{r})$
9:         $\pi' :=$ pc-add($\pi', sm'_{def}$)
10:        $h'_{id} := h'_{id} \cup \{id(\bar{r}, sm'(\bar{r}), p(\bar{r}))\}$
11:    $\sigma\{\pi := \pi', h := (h \setminus h_{id}) \cup h'_{id}\}$
```

Figure 3.7: Execution rules for havoc and havocall with quantified resources

The main difference with the previous implementation is that we must replace snapshot maps instead of snapshots. To this end, we only need to provide an alternative definition of `execHavoc`, which we call `execHavocQP`. When the state contains quantified permissions, the `havoc` and `havocall` cases of `exec` will call this function instead. (This detail is omitted from the pseudocode.) Apart from this, the implementations of the other two functions, `exec` and `replacementCond`, need no adjustments.

The function `execHavocQP` starts off similarly to `execHavoc`. We extract the relevant heap chunks of type *id*. For each heap chunk, we determine the snapshot's replacement condition.

At this point, the implementation diverges from `execHavoc`. On line 7, we declare a fresh snapshot map sm' for our new chunk. We axiomatize sm' as follows: for any input $\bar{r}$, the snapshot $sm(\bar{r})$ is preserved if the replacement condition is *not* satisfied. We provide no axiomatization for other inputs (i.e. where the replacement condition might be satisfied). This has the effect of replacing the snapshot values with unknown entities. Finally, we then add this axiom to our set of path conditions, and add the heap chunk to our heap.

Consider the example in Figure 3.8. In this case, our heap contains the quantified heap chunk $P(r; sm(r), pm(r))$. We would like to havoc `P(g(x))` under the condition `b`. Intuitively, we should end up with a snapshot map where if b holds, then only the snapshot $P(g(x))$ is replaced. We start by evaluating the expressions. Next, `exec` calls `execHavocQP`. We consider our heap chunk $P(r; sm(r), pm(r))$. We calculate the replacement condition to be $b \wedge r = g(x)$. We use this to axiomatize a new snapshot map, $sm'(r)$, with

$$\forall r : Snap \cdot \neg(b \wedge r = g(x)) \Rightarrow sm(r) = sm'(r)$$

```
predicate P(x: Ref) {...}
function g(x: Ref): Ref {...}
method foo(s: Set[Ref], x: Ref, b: Bool)
    requires forall z: Ref :: z in s ==> P(z)
{
    havoc b ==> g(x).f
}
```

Figure 3.8: An example where we havoc with quantified permissions.

Finally, we add the updated heap chunk $P(r; sm'(r), pm(r))$ to our state and return. The situation would be similar if we had performed a havocall instead of a havoc — we would have done the required injectivity steps, and we would have used a `HavocAllData` when calculating the replacement condition.

Chapter 4

Implementation and Evaluation

Implementation

As mentioned before, Viper contains two implementations: Silicon and Carbon. Both implementations share a common component called Silver, which handles parsing, typechecking, and generating the AST. For this project, we added the nodes `Havoc` and `Havocall` to Viper's AST. We implemented the corresponding parsing and typechecking rules, before handling their execution in Silicon.

The corresponding symbolic execution rules have stubs in `Executor.scala`, which dispatch functions in a new file: `HavocSupporter.scala`. This file, contains the vast majority of the required code changes in Silicon. The implementation is under 200 lines of code, and it closely follows the pseudocode in Section 3.4.1. For much of the implementation, existing helper functions could be employed. For example, all functions for checking injectivity and axiomatizing an inverse function already existed from quantified permissions. In addition, there is a new test suite which checks various combinations of havoc and havocall.

Benchmarks

To judge the performance of havoc, a few benchmarks were written. They were generated with a Python script, so as to parameterize them on an input size. For example, they might have n variables aliasing, or n invocations of havoc. There were two versions of each benchmark — one with havoc or havocall, and one with the equivalent behavior in exhale and inhale statements. All benchmarks were run with `enableMoreCompleteExhale`. In addition, Silicon was built in single-threaded mode to yield more consistent results. Only six benchmarks were written. Ideally, we would test the performance of havoc in more situations. However, these benchmarks are meant to stress the usage of havoc, and so they may not represent realistic

```
1  method foo(y: Ref, x_i: Ref)
2      requires acc(x_i.f, 1/n)   // for each i
3      requires x_n.f == 42
4  {
5      havoc y.f
6
7      // Learn each x_i is an alias of y
8      assume y == x_i    // for each i
9
10     // Therefore, x_n has been havocked
11     assert x_n.f == 42    // should fail
12 }
```

Figure 4.1: An example benchmark, parameterized on the test size n. Any expression with x_i should be instantiated with x_1 through x_n. Two versions were created. In the alternate version, line 5 is replaced with exhale and inhale statements.

code. Thus, their performance should be taken with a grain of salt anyways. Nevertheless, they demonstrate asymptotic improvement of havoc over exhale and inhale. An example benchmark is shown in Figure 4.1.

Of the six benchmarks created, all of them performed at least as well as their exhale-inhale counterpart, and three of them performed asymptotically better. The most striking example is shown in Figure 4.2. This plot corresponds to the benchmark in Figure 4.1. This benchmark was based off the example in 3.1, where we havoc an expression with several heap chunks that might alias. In that section, we argued that the performance was slow because the permission math scaled super-linearly. This plot confirms our suspicion. On the other hand, havocking a heap chunk is a relatively simple operation — it does not depend on operations with previous heap chunks.

Usage in Voila

To further test havoc, we incorporated it into one of Viper's front-ends: Voila. Voila is a verification language that focuses on programs with fine-grained concurrency. For this reason, it relies on an efficient implementation of havoc and havocall. In fact, the authors listed it as a major performance impediment [10].

To handle this issue, Silicon offers a "havoc hack" — when built without the flag `disableHavocHack407`, the user can invoke an unconditional havocall. For each predicate type `P` that a user wants to havoc, they must declare the abstract method (without a body) `___silicon_hack407_havoc_all_P`. This method takes no arguments (even if the predicate does). Then, when they want to perform the havocall, they call this method. There is no way

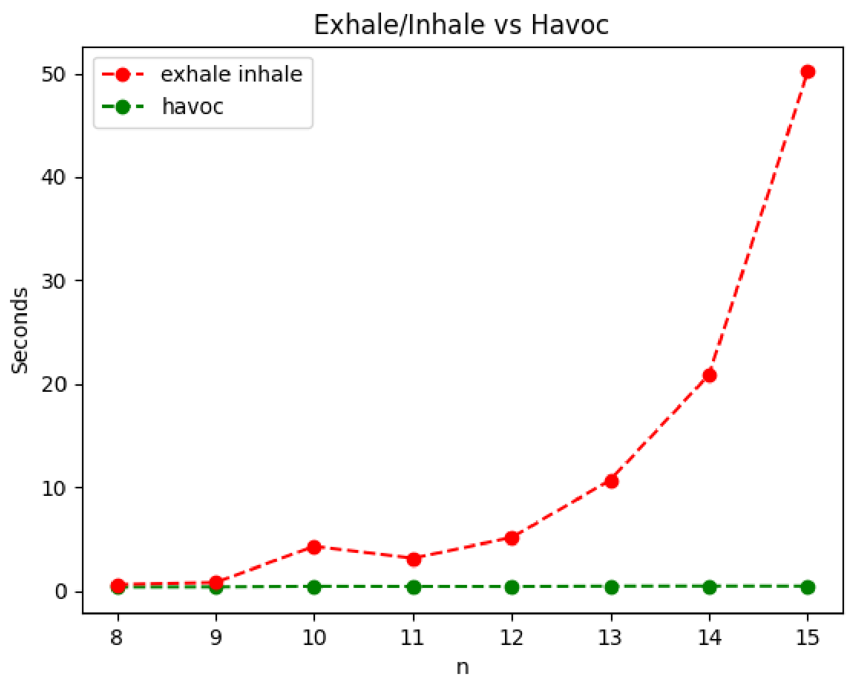

Figure 4.2

to constrain which predicate instances are havocked, either by providing a havoc condition or by providing expressions for the arguments. To execute this statement, Silicon identifies all the heap chunks with the predicate type *P* and simply gives them a fresh snapshot. A comparison between the three different ways of quantified havocking is shown in Figure 4.3.

Voila already provides a flag `disableSiliconSpecificHavockingCode`. This flag controls whether the exhale-inhale or the havoc hack version is used. I added another option which outputs the `havocall` version. The 49 tests in the folder `voila_evaluation_examples` were used, again in single-threaded mode. A comparison between the havoc hack and havocall version is shown in Figure 4.4. The two versions had comparable performances, but in most cases, `havocall` outperformed. This result is surprising because the havoc hack's implementation is targeted for this use case. However, both versions dramatically outperformed exhale-inhale. Of the 49 tests attempted, 39 failed outright. A further 6 timed out after three minutes, and the remaining 4 were much slower than either the havoc hack over havocall.

```
havocall x: Ref :: P(x)
```

(a) havocall implementation

```
label L
exhale forall z: Ref :: acc(P(x), perm(P(x)))
inhale forall z: Ref :: acc(P(x), old[L](perm(P(x))))
```

(b) Exhale-inhale implementation

```
___silicon_hack407_havoc_all_P()
```

(c) "Havoc hack" implementation

Figure 4.3: Comparison of three ways of havocking all occurrences of a the predicate `P(x)`

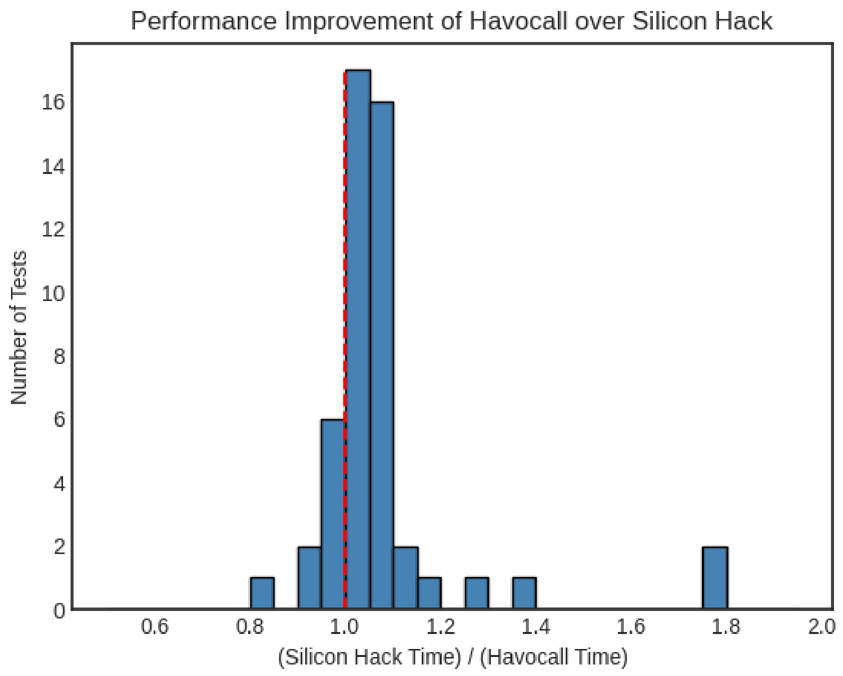

Figure 4.4: This histogram compares the performance of havocall with Silicon's havoc hack on Voila's evaluation examples. 49 test were run, and for each test, the relative slowdown of the Silicon hack was calculated. A line at $x = 1$ is drawn to indicate no improvement. Everything to the right indicates a performance improvement for havocall. The majority of tests show a slight improvement of havocall over the havoc hack.

Chapter 5

Conclusion

In this project, we introduced two new statements, havoc and havocall into Viper. These two statements mimic a nondeterministic assignment to a resource's memory by replacing its snapshot. Viper already had a way of havocking snapshots by combining `exhale` and `inhale` statements. In this report, we demonstrated that a native havoc statement outperforms the existing method, both in targeted benchmarks and in code generated from a Viper front-end. The implementation in Silicon is contained within its own package and is relatively concise.

Future Work

There are still a few outstanding tasks if the Programming Methodology Group decides that it wants to add havoc and havocall as core features to Viper. Most importantly, there should be a Carbon implementation of these statements. This must be done to maintain a consistent feature set between the two back-ends. As of this writing, it is unclear how difficult this would be because Carbon represents permissions very differently from Silicon.

It would also be useful to have more front-ends use havoc statements. Currently, the only "real-world" code that we've benchmarked uses an unconditional havocall. Benchmarking conditional versions of havoc and havocall would allow us to explore the performance differences in more situations.

Lastly, in a final meeting with the Programming Methodology Group, we discussed that `havoc` might be an unintuitive keyword. In most other situations, it corresponds to a true assignment of a nondeterministic value. However, as discussed in Section 3.3, havocking a resource with fractional permissions does not change its value, and in certain instances, the value can be re-asserted. Possible alternative names for havoc include `forget` or `refresh`.

Bibliography

[1] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[2] Daniel P. Friedman and Mitchell Wand. *Essentials of programming languages (3. ed.)*. MIT Press, Cambridge, MA, 2008.

[3] Cliff B Jones. *Development Methods for Computer Programs Including a Notion of Interference*. Oxford University Computing Laboratory, 1981.

[4] K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.

[5] P. Müller, M. Schwerhoff, and A. J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification (CAV)*, volume 9779 of *LNCS*, pages 405–425. Springer-Verlag, 2016.

[6] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[7] M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3:01):1–54, 2012.

[8] M. Schwerhoff and A. J. Summers. Lightweight Support for Magic Wands in an Automatic Verifier. In J. T. Boyland, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 37 of *LIPIcs*, pages 614–638. Schloss Dagstuhl, 2015.

[9] Malte Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. PhD thesis, ETH Zürich, 2016.

[10] Felix Wolf. Verifying fine-grained concurrent data structures. Master's thesis, Master thesis, ETH Zurich, 2018.

[11] Felix A. Wolf, Malte Schwerhoff, and Peter Müller. Concise outlines for a complex logic: A proof outline checker for tada (full paper). *CoRR*, abs/2010.07080, 2020.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Adding Native Support for Havoc in Viper

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Zhang	Daniel

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 9.9.2022	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.