Refining and Applying a Framework for Automatic Inference of Hyperproperties

Bachelor's Thesis Description

Mathias Blarer

Supervised by
Marco Eilers, Jérôme Dohrau, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zurich, Switzerland

March 20, 2019

1 Introduction

Frequent news about serious system vulnerabilities and expensive software defects demonstrate how hard it is to manually reason about the security and reliability of programs. A lot of effort is therefore put into the development of tools that automatically reason about program behaviour.

Hyperproperties are an interesting class of program properties to be looked at. These are properties that can make a statement not only about single execution traces but also about sets of execution traces. Examples of hyperproperties are determinism, non-interference, monotonicity, injectivity, and many more. From the perspective of program analysis, hyperproperties require reasoning about multiple program executions at the same time.

So-called *k-product programs* are constructions that self-compose a program k times, representing a set of k program executions with a single product program. Eilers et al. [2] have called their product program construction a *modular product program* because it allows hyperproperties to be modularly specified for procedures. A program is turned into a modular product program by interleaving multiple copies of the same program, renaming variables and using additional boolean *activation variables* to keep track for every execution under which conditions it is active. It is important to note that by construction, product programs reduce hyperproperties to properties of a single execution trace and thus, can be statically analyzed using existing tools and techniques like abstract interpretation and deductive verification.

Based on this concept of modular product programs, Knabenhans [4] developed a static analysis framework for hyperproperties which he implemented inside the static analyzer Sample[1], as part of the Viper[2] verification infrastructure [5]. His analysis uses abstract interpretation equipped with a binary decision tree (BDT) abstract domain, a heap domain and a numerical domain. The

[1] `http://www.pm.inf.ethz.ch/research/sample.html`
[2] `http://www.pm.inf.ethz.ch/research/viper.html`

BDT domain is needed for *trace partitioning*, a technique to perform abstract interpretation over a partition of the set of execution traces rather than over the set of traces itself. In our case, we are interested in distinguishing program traces in which every activation variable is respectively true or false, leading to more precise results. We also want the set of traces to be partitioned with respect to conditions that relate different input variables. Trace partitioning has proven suitable for the inference of hyperproperties and its implementation works well for many programs, however, there are still cases in which the analysis fails to infer the correct hyperproperties due to lack of precision.

The main goal of this thesis is to build upon the work of Knabenhans in order to make his framework more precise and to extend the framework to another application of hyperproperties. We are going to choose one of the two applications presented in the next section. In Sections 3 and 4, the goals of this thesis project are worked out in more detail.

2 Applications

Below, we introduce two applications for the automatic inference of hyperproperties: on one hand the analysis of execution paths and parallel memory accesses in GPU kernels, on the other hand the secure information flow of methods and programs.

2.1 GPU Kernels

In modern GPU design, SIMT (Single Instruction Multiple Threads) is a very common approach where multiple threads execute the same instruction in parallel. The following terminology stems from NVIDIA GPUs as it can be found in the CUDA programming guide [1] but the same concepts apply in a similar fashion to GPUs from other companies.

Warps are groups of 32 parallel threads that execute a single common instruction at a time. If within a warp the execution path differs, the warp will run the threads that have the same instruction and disable the others. Therefore, optimal performance can be achieved only if all threads within a warp have the same execution path.

Every thread has an ID that is consecutive within a warp. Inside a kernel definition, the programmer has access to the thread index `threadIdx`, a vector with three components x, y, and z, from which the ID can be computed.

For illustration purposes, we look at a simple kernel that reverses an array with CUDA. The code is taken and slightly modified from [3].

```
__global__ void reverse(int *data, int len)
{
    __shared__ int s[64];
    int i = threadIdx.x;
    int j = len-i-1;
    s[i] = data[i];
    __syncthreads(); // wait for all threads to finish
    data[i] = s[j];
}
```

When a warp accesses memory in global memory like `data` in the kernel

above, the warp initiates a 32-, 64-, or 128-byte memory transaction, depending on the size of the requested data and on the distribution of the accessed memory locations. If multiple memory addresses lie within the same chunk of data, only one memory transaction has to be performed, thus increasing overall throughput. To optimize the throughput of global memory accesses, it is therefore necessary that all memory locations, accessed simultaneously by a warp, are dense in global memory chunks of at least 32 bytes. In the given kernel code, threads with consecutive indices are accessing consecutive addresses in global memory, thereby minimizing the number of transactions needed.

Shared memory such as the array `s` has lower latency than global memory. It is stored on the multiprocessor and organized in memory banks. If memory locations in distinct banks are requested simultaneously by multiple threads, the requests can be served in parallel. Otherwise a *bank conflict* occurs and the affected requests are served sequentially. To optimize performance, bank conflicts should be avoided. Successive 32 bits in shared memory are assigned to successive memory banks; in the kernel code above, `s` in an array of 32-bit integers. The threads with consecutive indices will therefore access consecutive shared memory banks which optimizes the throughput of the instruction.

A useful application for our framework is to identify inefficient branching, sparse global memory accesses and shared memory bank conflicts in kernels written for SIMT GPUs.

2.2 Secure Information Flow

One way to test for security leaks in a program is to look at its *secure information flow*. Informally, the information flow of a method or program is secure if and only if its observable output is not influenced by any secret data.

To illustrate this property, we are looking at two simple examples. The first method, `foo`, leaks information about secret data and therefore does not have a secure information flow:

```
method foo(secret:Int) returns (res:Int) {
    x := 0
    if (secret >= 0) { x := 1 }
    res := x
}
```

The second method, `bar`, also uses secret data but only in encrypted form using a secure encryption scheme:

```
method bar(secret:Int, key:Int) returns (res:Int) {
    x := 0
    if (secret >= 0) { x := 1 }
    res := enc(x, key)
}
```

An existing analysis would typically infer that `bar` has a secure information flow. Still, this example shows how a *chosen-plaintext attack* can break the security of this method. By testing the method with many different values for `secret` and `key`, an attacker can observe for every `key` two different return values, one for `secret >= 0` and one for `secret < 0`. He will later be able to draw conclusions about `secret` when only looking at `key` and `res`.

We aim to devise an analysis that detects insecure information flow including vulnerabilities for chosen-plaintext attacks.

3 Core Goals

1. **Identify imprecisions in current framework**
   The existing analysis framework is imprecise for a number of programs. It is a challenge to optimize the precision of the BDT domain while keeping the abstract interpretation tractable. For instance, the BDT domain provides a mechanism for splitting the program traces at one point and for merging them back at another point. For different merge strategies, there is this tradeoff between precision and speed. A case where the analysis is particularly imprecise, is early loop termination through `break`, `continue` and `return` statements. In the context of checking asymmetry for comparator functions, we will test the current framework thoroughly on examples taken from [6], identify problems like the ones stated above and determine their cause.

2. **Resolve issues from previous goal**
   Having determined and examined issues related to precision and performance within the analysis framework, we will put forward options how to resolve these issues. Possible options might include, e.g., modifying the construction of the product program, adjusting the merge strategy for the BDT domain or making changes to the numerical domain. For each solution, we will implement it if feasible or explain why it is infeasible.

3. **Apply framework to a new problem**
   We will choose one application mentioned in Section 2, i.e., either GPU kernels or secure information flow. In our framework, we will develop an analysis that is able to reason about this hyperproperty and implement it in Sample.

4. **Evaluate framework performance**
   Finally, we will measure and evaluate the performance and precision of the existing analysis framework and compare it to other tools.

4 Extension Goals

- Building on top of the existing framework, we can create an interprocedural analysis for non-recursive methods. Such an analysis can be run in the topological order of the method dependencies.
- We can extend the interprocedural analysis to handle recursive programs.
- We can build a lightweight frontend that is suitable for the application we implemented as part of the core goals. Such a frontend takes source code and translates it to Viper, an intermediate language that is analyzable using our framework.

- We can use our analysis for more applications, an obvious choice is to take the application described in Section 2 that we do not choose as 3rd core goal.

References

[1] NVIDIA Corporation. CUDA C Programming Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, February 2019. Accessed: 2019-03-12.

[2] Marco Eilers, Peter Müller, and Samuel Hitz. Modular Product Programs. European Symposion on Programming (ESOP), 2018.

[3] Mark Harris. Using Shared Memory in CUDA C/C++. `https://devblogs.nvidia.com/using-shared-memory-cuda-cc/`, January 2013. Accessed: 2019-03-15.

[4] Christian Knabenhans. Automatic Inference of Hyperproperties. Bachelor's Thesis, ETH Zurich, August 2018.

[5] Peter Müller, Malte Schwerhoff, and Alexander 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)*, pages 41–62. Springer-Verlag, 2016.

[6] Marcelo Sousa and Isil Dillig. Cartesian Hoare Logic for Verifying k-Safety Properties. ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), 2016.